ici, on apprend tout à partir de zero
Le Site Du Zero

Apprenez à programmer en C / C++ !

Vous aimeriez apprendre à programmer, mais vous ne savez vraiment pas par où commencer ?
(autrement dit: vous en avez marre des cours trop compliqués que vous ne comprenez pas ?)

Auteur : M@teo21
Créé le : 29/07/2005 à 00h29
Modifié le : 26/11/2006 à 19h39
Avancement : 60%
Imprimer tout le tutorial

C'est votre jour de chance
Vous venez de tomber sur un cours de programmation pour débutants, vraiment pour débutants.

Il n'y a aucune honte à être débutant, tout le monde est passé par là, moi y compris
Ce qu'il vous faut est pourtant simple. Il faut qu'on vous explique tout, progressivement, depuis le début :

- Comment s'y prend-on pour créer des programmes comme des jeux, ou encore des programmes avec des fenêtres ?
- De quels logiciels a-t-on besoin pour programmer ?
- Dans quel langage commencer à programmer ? D'ailleurs, c'est quoi un langage ?

Je vous souhaite une agréable lecture
Puisse ce tutorial aider un maximum d'entre vous.

Ce cours est composé des parties suivantes :

I. [Langage C] Les bases du débutant
II. [Langage C] Techniques avancées
III. [Librairie C] Création de jeux 2D en SDL
IV. [Langage C++] La Programmation Orientée Objet
V. Annexes

PARTIE 1 : [LANGAGE C] LES BASES DU DÉBUTANT

Vous débutez ?
C'est par là qu'on commence

Les bases de la programmation sont expliquées à travers ces premiers chapitres, aussi soyez très attentifs ! Ce que vous allez apprendre ici sera nécessaire pour pouvoir comprendre la suite du cours

Prêts ? A l'assaut !

1) Vous avez dit "programmer" ?
- Programmer, c'est quoi ?
- Programmer, dans quel langage ?
- Programmer, c'est dur ?
  - *Q.C.M.*

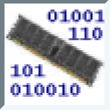
2) Ayez les bons outils !

- Les outils nécessaires au programmeur
- Vous pouvez choisir... Dev-C++
- Ou bien... Visual C++
- Ou encore... Code::Blocks
- Sous Mac... Xcode
  - *Q.C.M.*

3) Votre premier programme

- Console ou fenêtre ?
- Un minimum de code
- Ecrire un message à l'écran
- Les commentaires, c'est très utile !
  - *Q.C.M.*

4) Un monde de variables



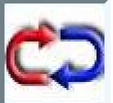
- Une affaire de mémoire
- Déclarer une variable
- Afficher le contenu d'une variable
- Récupérer une saisie
  - *Q.C.M.*

5) Une bête de calcul
- Les calculs de base
- Les raccourcis
- La librairie mathématique
  - *Q.C.M.*

6) Les conditions

- La condition "if... else"
- Les booléens, le coeur des conditions
- La condition "switch"
- Les ternaires : des conditions condensées
  - *Q.C.M.*

7) Les boucles

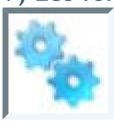


- Qu'est-ce qu'une boucle ?
- La boucle while
- La boucle do... while
- La boucle for
  - *Q.C.M.*

8) TP : Plus ou Moins, votre premier jeu

- Préparatifs et conseils
- Correction !
- Idées d'amélioration

9) Les fonctions



- Créer et appeler une fonction
- Plein d'exemples pour bien comprendre
  - *Q.C.M.*

Ainsi s'achève la première partie de ce cours de C / C++ pour débutants
Nous y avons appris **les principes de base** de la programmation en C, mais nous sommes encore très loin d'avoir tout vu !

Les choses sérieuses commenceront dans la partie II

Partie 2 : [Langage C] Techniques avancées

Cette seconde partie introduit une notion très importante du langage C : **les pointeurs**. Nous verrons ce que c'est et tout ce qui en découle, tout ce qu'on peut faire avec.

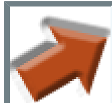
Je ne vous le cache pas, et vous vous en doutiez sûrement, la partie II est à un cran de difficulté supérieur.
Là encore, je fais mon maximum pour tout vous expliquer le plus simplement possible
Lorsque vous serez arrivés à la fin de cette partie, vous serez capables de vous débrouiller dans la plupart des programmes écrits en C. Dans la partie suivante nous verrons alors comment ouvrir une fenêtre, créer des jeux 2D, jouer du son etc.

Accrochez votre ceinture quand même, parce que ça va secouer un tantinet

1) La programmation modulaire

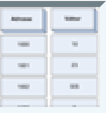
- Les prototypes
- Les headers
- La compilation séparée
- La portée des fonctions et variables
  - *Q.C.M.*

2) A l'assaut des pointeurs



- Un problème bien ennuyeux
- La mémoire, une question d'adresse
- Utiliser des pointeurs
- Envoyer un pointeur à une fonction
- Qui a dit : "Un problème bien ennuyeux" ?
  - *Q.C.M.*

3) Les tableaux



- Les tableaux dans la mémoire
- Définir un tableau
- Parcourir un tableau
- Passage de tableaux à une fonction
  - *Q.C.M.*

4) Les chaînes de caractères

- Le type char
- Les chaînes sont des tableaux de char !
- Fonctions de manipulation des chaînes
  - *Q.C.M.*

5) Le préprocesseur

- Les includes
- Les defines
- Les macros

- Les conditions
  - *Q.C.M.*

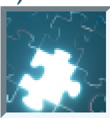
6) Créez vos propres types de variables !

- Définir une structure
- Utilisation d'une structure
- Pointeur de structure
- Les énumérations
  - *Q.C.M.*

7) Lire et écrire dans des fichiers

- Ouvrir et fermer un fichier
- Différentes méthodes de lecture / écriture
- Se déplacer dans un fichier
- Renommer et supprimer un fichier
  - *Q.C.M.*

8) L'allocation dynamique



- La taille des variables
- Allocation de mémoire dynamique
- Allocation dynamique d'un tableau
  - *Q.C.M.*

9) TP : Réalisation d'un pendu

- Les consignes
- La solution (1 : le code du jeu)
- La solution (2 : la gestion du dictionnaire)
- Idées d'amélioration

Si vous arrivez jusque-là, vous pouvez vous dire que le plus dur est fait !
Certes la partie II comporte son lot de difficultés, mais avec un peu de bonne volonté on arrive à tout !

Une récompense attend tous ceux qui seront parvenus à comprendre toute la partie II. Cette récompense... c'est la partie III !

PARTIE 3 : [LIBRAIRIE C] CRÉATION DE JEUX 2D EN SDL

Arrivés à ce stade, vous connaissez la plupart des bases du C. Vous avez donc la théorie nécessaire pour réaliser à peu près n'importe quel programme. Mais... pour le moment nous n'avons fait que des *printf* en console, ce qui fait que nos programmes sont encore bien monotones.

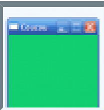
Dans la partie III, ça va changer ! Nous allons étudier une librairie qui a pour nom **SDL** (*Simple Directmedia Layer*).
Cette librairie, une fois installée, rajoute de nombreuses possibilités. Vous allez pouvoir en effet ouvrir des fenêtres, faire du plein écran, dessiner, gérer le contrôle du clavier, de la souris, du joystick...

Bref, à partir de maintenant nous allons vraiment pouvoir nous amuser !

1) Installation de la SDL

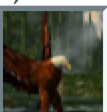
- Pourquoi avoir choisi la SDL ?
- Téléchargement de la SDL
- Créer un projet SDL

2) Création d'une fenêtre et de surfaces



- Charger et arrêter la SDL
- Ouverture d'une fenêtre
- Manipulation des surfaces
- (Exercice) Créer un dégradé
  - *Q.C.M.*

3) Afficher des images



- Charger une image BMP
- Gestion de la transparence
- Charger plus de formats d'image avec SDL_Image
  - *Q.C.M.*

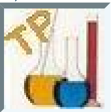
4) La gestion des évènements (Partie 1/2)

- Le principe des évènements
- Le clavier
- (Exercice) Diriger Zozor au clavier
- La souris
  - *Q.C.M.*

5) La gestion des évènements (Partie 2/2)

- Initialiser le joystick
- Les évènements du joystick
- Les évènements de la fenêtre
  - *Q.C.M.*

6) TP : Mario Sokoban

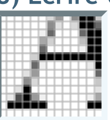


- Cahier des charges du Sokoban
- Le main et les constantes
- Le jeu
- Chargement et enregistrement de niveaux
- L'éditeur de niveaux
- Résumé et améliorations

7) Maîtrisez le temps !

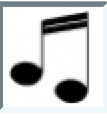
- Le Delay et les Ticks
- Les Timers
  - *Q.C.M.*

8) Ecrire du texte avec SDL_ttf



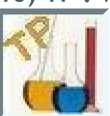
- Installer SDL_ttf
- Chargement de SDL_ttf
- Les différentes méthodes d'écriture
  - *Q.C.M.*

9) Jouer du son avec FMOD



- Installer FMOD
- Initialiser et libérer FMOD
- Les sons courts
- Les musiques (MP3, OGG, WMA...)
- Les musiques (MIDI)
  - *Q.C.M.*

10) TP : visualisation spectrale du son



- Les consignes
- La solution
- Idées d'amélioration

La partie sur la SDL est terminée, mais il est fort probable que des TP supplémentaires fassent leur apparition dans le futur.
Cette partie n'était qu'une *application pratique* de ce que vous avez appris dans les parties I et II. Vous n'avez en fait rien découvert de nouveau sur le langage C, mais vous avez vu comment *concrétiser* vos connaissances en travaillant sur une librairie intéressante, la SDL.

S'il y en a parmi vous qui sont intéressés par la 3D, je vous recommande vivement de lire le cours sur OpenGL rédigé par Kayl. C'est une librairie graphique 3D dont vous avez sûrement déjà entendu parler. Kayl a plus d'expérience que moi dans le domaine de la 3D, il sait de quoi il parle et vous apprendrez une foule de choses intéressantes avec lui !
Notez que pour suivre son cours il faut avoir lu tout mon cours de C / C++ jusqu'à la partie III sur la SDL incluse (Kayl utilise la SDL et OpenGL en même temps, vous verrez)

Bien, attaquons maintenant la partie IV sur le C++ si vous le voulez bien

PARTIE 4 : [LANGAGE C++] LA PROGRAMMATION ORIENTÉE OBJET

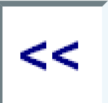
Après avoir découvert le langage C dans les parties précédentes, nous nous intéressons maintenant au C++
Le langage C++ est basé sur le C : ce que vous avez donc appris jusqu'ici va vous resservir, pour ne pas dire vous être indispensable !

Les modifications entre le C et le C++ sont nombreuses. La plus importante d'entre elles est l'introduction de la **Programmation Orientée Objet**, que l'on abrège couramment **POO**. On en entend souvent parler, mais qu'est-ce que c'est concrètement ?
La réponse se trouve dans cette partie du cours

1) Introduction au C++

- Pourquoi avoir créé le C++ ?
- La programmation orientée quoi ?
  - *Q.C.M.*

2) Premier programme C++ avec cout et cin



- Configurer l'IDE pour le C++
- Analyse du premier code source C++
- Le flux de sortie cout
- Le flux d'entrée cin
  - *Q.C.M.*

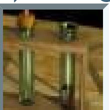
3) Nouveautés pour les variables

- Le type bool
- Les déclarations de variables
- Les allocations dynamiques
- Le typedef automatique
- Les références
  - *Q.C.M.*

4) Nouveautés pour les fonctions

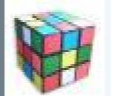
- Des valeurs par défaut pour les paramètres
- La surcharge des fonctions
- Les fonctions inline
  - *Q.C.M.*

5) La magie de la POO par l'exemple : string



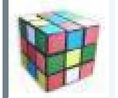
- Des objets... pour quoi faire ?
- Lire et écrire dans une chaîne via string
- Opérations sur les string
  - *Q.C.M.*

6) Les classes (Partie 1/2)



- Créer une classe
- Droits d'accès et encapsulation
- Séparer prototypes et définitions
  - *Q.C.M.*

7) Les classes (Partie 2/2)



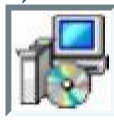
- Constructeur et destructeur
- Associer des classes entre elles
- Action !
  - *Q.C.M.*

Partie 5 : Annexes

Dans cette partie, vous trouverez des chapitres annexes au cours.
Ils ne sont pas à lire à la fin : vous pouvez les lire n'importe quand. Si certains demandent d'avoir lu au moins quelques chapitres du cours, cela sera indiqué dans l'introduction.

Ne négligez pas les annexes, vous y trouverez sûrement de nouvelles informations intéressantes !

1) Créer une installation



- Télécharger Inno Setup
- Créer une nouvelle installation

2) Créer une icône pour son programme

- Les logiciels d'édition d'icônes
- Associer une icône à son programme

Partie 1 : [Langage C] Les bases du débutant

Vous débutez ?
C'est par là qu'on commence

Les bases de la programmation sont expliquées à travers ces premiers chapitres, aussi soyez très attentifs ! Ce que vous allez apprendre ici sera nécessaire pour pouvoir comprendre la suite du cours

Prêts ? A l'assaut !

Vous avez dit "programmer" ?

Bonjour ! Soyez les bienvenus dans mon cours de programmation en C / C++ pour débutants !

Je serai votre guide (ou "professeur" si vous préférez) tout au long de ce cours. Qui je suis moi ? Mon nom, ou plutôt mon pseudonyme, est **M@teo21**. J'ai déjà réalisé pour le **Site du Zér0** plusieurs autres cours, notamment sur la création de sites web. Ce n'est donc pas la première fois que je rédige un cours pour débutants

Mais assez parlé de moi, parlons plutôt de **vous**.
Vous êtes là pour une raison précise : vous voulez apprendre à programmer. Vous ne connaissez rien à la programmation, vous n'êtes même pas sûrs de bien savoir ce que c'est et pourtant… Vous voulez apprendre à programmer, ça y'a pas de doute.

Mais programmer en C / C++… Ca veut dire quoi ? Est-ce que c'est bien pour commencer ? Est-ce que vous avez le niveau pour programmer ? Est-ce qu'on peut tout faire avec ?
Ce chapitre a pour but de répondre à toutes ces questions apparemment bêtes, et pourtant très importantes.
Grâce à ces questions simples, vous saurez à la fin de ce premier chapitre ce qui vous attend. C'est quand même mieux de savoir à quoi sert ce qu'on va apprendre, vous trouvez pas ?

Programmer, c'est quoi ?

On commence par la question la plus simple qui soit, la plus basique de toutes les questions basiques
Si vous avez l'impression de déjà savoir tout ça, je vous conseille de lire quand même, ça ne peut pas vous faire de mal Je pars de zéro pour ce cours, donc je vais devoir répondre à la question :

Bon, je vais éviter de vous faire comme mon prof de français : je ne vais pas vous donner l'origine du mot "programmer". Et puis de toute façon si je vous disais que ça vient du latin *programmeus* je crois que vous auriez un peu de mal à me croire
Simplement, programmer signifie réaliser des "programmes informatiques". Les programmes demandent à l'ordinateur d'effectuer des actions.

Votre ordinateur est rempli de programmes en tous genres :

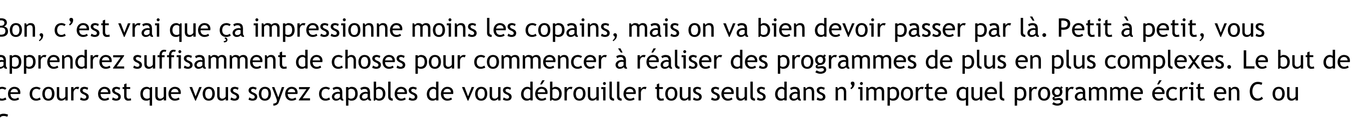
- La calculatrice est un programme
- Votre traitement de texte est un programme
- Votre logiciel de « Chat » est un programme
- Les jeux vidéo sont des programmes

En bref, les programmes sont partout et permettent de faire à priori tout et n'importe quoi sur un ordinateur. Vous pouvez inventer un logiciel de cryptage révolutionnaire si ça vous chante, ou réaliser un jeu de combat en 3D sur Internet, peu importe. Votre ordinateur peut tout faire (sauf le café, mais j'y travaille).

Le célèbre jeu Half-Life 2, programmé en C++

Attention ! Je n'ai pas dit que réaliser un jeu vidéo se faisait en claquant des doigts. J'ai simplement dit que tout cela était possible, mais soyez sûrs que ça demande beaucoup de travail

Comme vous débutez, nous n'allons pas commencer par voir comment réaliser un jeu 3D. Ce serait du pur suicide
Nous allons devoir passer par des choses très simples. Une des premières choses que nous verrons est *comment afficher un message à l'écran*. Oui, je sais ça n'a rien de très transcendant, mais rien que ça croyez-moi, c'est pas si facile que ça en a l'air



Bon, c'est vrai que ça impressionne moins les copains, mais on va bien devoir passer par là. Petit à petit, vous apprendrez suffisamment de choses pour commencer à réaliser des programmes de plus en plus complexes. Le but de ce cours est que vous soyez capables de vous débrouiller tous seuls dans n'importe quel programme écrit en C ou C++.

Mais tenez au fait, vous savez ce que c'est vous, cette histoire de "C / C++" ?

PROGRAMMER, DANS QUEL LANGAGE ?

Votre ordinateur est une machine bizarre, c'est le moins que l'on puisse dire. On ne peut s'adresser à lui qu'en lui envoyant des 0 et des 1. Ainsi, si je traduis "Fais le calcul 3 + 5" en langage informatique, ça pourrait donner quelque chose comme :

0010110110010011010011110

(j'invente hein, je ne connais pas la traduction informatique par cœur :p)
Ce que vous voyez là, c'est le langage informatique de votre ordinateur, appelé **langage binaire** (retenez bien ce mot !). Votre ordinateur ne connaît que ce langage-là et, comme vous pouvez le constater, c'est absolument incompréhensible, immonde et imbuvable

Donc voilà notre premier vrai problème :

Comment parler à l’ordinateur plus simplement qu’en binaire avec des 0 et des 1 ?

Votre ordinateur ne parle pas l’anglais et encore moins le français. Pourtant, il est inconcevable d’écrire un programme en langage binaire. Même les informaticiens les plus fous ne le font pas, c’est vous dire

Eh bien, l’idée que les informaticiens ont eue, c’est d’inventer de nouveaux langages qui seraient ensuite traduits en binaire pour l’ordinateur. Le plus dur à faire, c’est de réaliser le programme qui fait la "traduction". Heureusement, ce programme a déjà été écrit par des informaticiens et nous n’aurons pas à le refaire (ouf !). On va au contraire s’en servir pour écrire des phrases comme :
"Fais le calcul 3 + 5"
Qui seront traduites par le programme de "traduction" en quelque chose comme :
"0010110110010011010011110".

Si on fait un schéma de ce que je viens de dire, ça donne quelque chose comme ça :

Schéma (super-simplifié) de réalisation d’un programme

Un peu de vocabulaire

Là j’ai parlé avec des mots simples, mais il faut savoir qu’en informatique il existe un mot pour chacune de ces choses-là. Tout au long de ce cours, vous allez d’ailleurs apprendre pas mal de vocabulaire.
Non seulement vous aurez l’air de savoir de quoi vous parlez, mais si un jour (et ça arrivera) vous devez parler à un autre programmeur, vous saurez vous faire comprendre. Certes, les gens autour de vous vous regarderont comme des extra-terrestres, mais ça il faudra pas y faire attention

Reprenons le schéma qu’on vient de voir.
La première case est "Votre programme est écrit dans un langage simplifié". Ce fameux "langage simplifié" est appelé en fait "**langage de haut niveau**".
Il existe plusieurs "niveaux" de langages. Plus un langage est haut niveau, plus il est proche de votre vraie langue (comme le français). Un langage de haut niveau est donc facile à utiliser (chouette !), mais cela a aussi quelques petits défauts comme nous le verrons plus tard.

Il existe de nombreux langages de plus ou moins haut niveau en informatique dans lesquels vous pouvez écrire vos programmes. En voici quelques-uns par exemple :

- Le C
- Le C++
- Java
- Visual Basic
- Delphi
- Etc etc...

Notez que je ne les ai pas classés par "niveau de langage", donc n'allez pas vous imaginer que le premier de la liste est plus facile que le dernier ou l'inverse Ce sont juste quelques exemples en vrac qui me sont passés par la tête.

(et d’avance désolé pour tous les autres langages qui existent, mais faire une liste complète serait vraiment trop

long *)*

Certains de ces langages sont plus haut niveau que d'autres (donc en théorie un peu plus faciles à utiliser), on va voir un peu plus loin notamment ce qui différencie le langage C du langage C++.

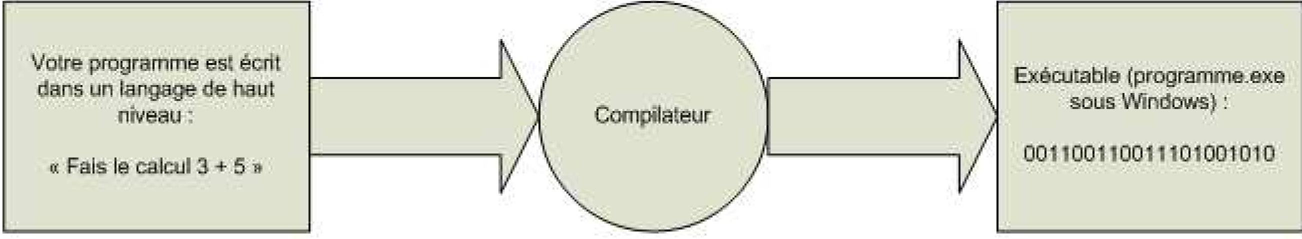
Un autre mot de vocabulaire à retenir est : **code source**. Ce qu'on appelle le code source, c'est tout simplement le code de votre programme écrit dans un langage de haut niveau. C'est donc vous qui écrivez le code source, qui sera ensuite traduit en binaire.

Venons-en justement au « programme de traduction » qui traduit notre langage de haut niveau (comme le C ou le C++) en binaire. Ce programme a un nom : on l'appelle **le compilateur**. La traduction, elle, s'appelle **la compilation**. Très important : il existe un compilateur différent pour chaque langage de haut niveau. C'est d'ailleurs tout à fait logique : les langages étant différents, on ne traduit pas le C++ de la même manière qu'on traduit le Delphi

Vous verrez par la suite que, pour les langages C / C++ par exemple, il existe même plusieurs compilateurs différents ! Il y a le compilateur écrit par Microsoft, le compilateur GNU etc. On verra tout ça dans le chapitre suivant.
Heureusement, ces compilateurs-là sont quasiment identiques (même s'il y a parfois quelques "légères" différences que nous apprendrons à reconnaître).

Enfin, le programme binaire créé par le compilateur est appelé : **l'exécutable**. C'est d'ailleurs pour cette raison que les programmes (tout du moins sous Windows) ont l'extension ".exe" comme EXEcutable.

Reprenons notre schéma de tout à l'heure, et utilisons cette fois des vrais mots tordus d'informaticien. Ca donne :



Le même schéma, avec le bon vocabulaire

Pourquoi choisir d'apprendre le C / C++ ?

Comme je vous l'ai dit plus haut, il existe de très nombreux langages de haut niveau. Doit-on commencer par l'un d'entre eux en particulier ? Grande question
Pourtant, il faut bien faire un choix, commencer la programmation à un moment ou à un autre. Et là, vous avez en fait le choix entre :

- **Un langage très haut niveau** : c'est facile à utiliser, plutôt "grand public", comme Visual Basic. Cependant, un langage comme celui-ci a plusieurs défauts : tout d'abord il est payant, coûte cher, mais il est aussi assez limité. Par exemple, votre programme ne fonctionnera que sous Windows : n'espérez pas le faire marcher sous Linux ou Macintosh ! Enfin, et surtout, vous ne pourrez pas faire tout ce que vous voulez avec ce type de langage, vous vous rendrez compte que vous êtes en fait assez limité.
- **Un langage un peu plus bas niveau** (mais pas trop quand même !) : c'est peut-être un peu plus difficile que Visual Basic certes, mais avec un langage comme le C (ou le C++) vous allez en apprendre beaucoup plus sur la programmation et sur le fonctionnement de votre ordinateur. Vous serez ensuite largement plus capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.
  Par ailleurs, le C et le C++ sont des langages très populaires. Ils sont utilisés pour programmer une grande partie des logiciels que vous connaissez.
  Enfin, pour programmer en C ou C++, vous n'êtes pas obligés d'acheter des logiciels hors de prix ! Nous verrons dans le second chapitre que programmer dans ces langages est tout à fait gratuit.

Voilà en gros les raisons qui m'incitent à vous apprendre le langage C plutôt qu'un autre. Je ne dis pas qu'il *faut* commencer par ça, mais je vous dis plutôt que c'est un bon choix qui va vous donner de solides connaissances.

Je vais supposer tout au long de ce cours que c'est votre premier langage de programmation, que vous n'avez jamais fait de programmation avant. Si, par hasard, vous avez déjà un peu programmé, ça ne vous fera pas de mal de reprendre à zéro

Stop, il y a quelque chose que je ne comprends pas... Je vais apprendre un langage appelé "C / C++" ou je vais apprendre 2 langages : l'un appelé "C" et l'autre appelé "C++" ?

La bonne réponse est que vous allez apprendre en fait 2 langages. Non, ça ne va pas faire 2 fois plus de travail
Je m'explique. Le langage C et le langage C++ sont très similaires. **Quand je désigne les 2 à la fois (comme je l'ai fait jusqu'ici), j'écris "C / C++".**
Voici ce qu'il faut savoir sur la différence entre les 2 avant de continuer :

- Au tout début, à l'époque où les ordinateurs pesaient des tonnes et faisaient la taille de votre maison, on a commencé à inventer un langage de programmation appelé **l'Algol**.
- Ensuite, les choses évoluant, on a créé un nouveau langage appelé le **CPL**, qui évolua lui-même en **BCPL**, puis qui pris le nom de **langage B** (euh si vous retenez pas tout ça c'est pas grave, j'écris juste pour faire semblant d'avoir de la culture là)
- Puis, un beau jour, on en est arrivés à créer encore un autre langage qu'on a appelé... le **langage C**. Ce langage, s'il a subi quelques modifications, reste encore un des langages les plus utilisés aujourd'hui.
- Un peu plus tard, on a proposé d'ajouter des choses au langage C. Une sorte d'amélioration si vous voulez. Ce nouveau langage, que l'on a appelé "C++", est entièrement basé sur le C. **Le langage C++** n'est en fait rien d'autre que le langage C avec des ajouts (quels ajouts ? On verra ça plus tard dans le cours).

Il y a plusieurs façons d'apprendre la programmation, je vous l'ai dit plus haut.
Certaines personnes pensent qu'il est bien d'enseigner directement le C++. Elles n'ont peut-être pas tort. Après tout, si le C++ c'est du langage C "avec des trucs en +", ça revient un peu au même.

Pourtant, moi (et cet avis n'engage que moi), je pense que ce serait mélanger les choses. Aussi j'ai décidé que j'allais séparer mon cours en 2 grosses parties :

- **Le langage C**
- **Le langage C++**

Vu que vous aurez déjà appris le langage C dans un premier temps, quand on en viendra au langage C++ ça ira bien plus vite. Je n'aurai pas à vous réapprendre toutes les bases du C, j'aurai juste besoin de vous indiquer quels ajouts ont été faits dans le C++ (enfin, y'a de quoi dire quand même)

Qu'il n'y ait pas de malentendus. Le langage C++ n'est pas "meilleur" que le langage C, il permet juste de programmer différemment. Il permet disons aussi au final de programmer un peu plus vite et de mieux organiser le code de son programme.

Ce n'est PAS parce que Half-Life 2 a été codé en C++ qu'il faut absolument faire du C++ pour réaliser des jeux ou des programmes complexes.
Le langage C n'est pas un "vieux langage oublié", au contraire il est encore très utilisé aujourd'hui. Il est à la base des plus grands systèmes d'exploitation tels Unix (et donc Linux et Mac OS), ou encore Windows.

Retenez donc : le C et le C++ ne sont pas des langages concurrents, on peut faire autant de choses avec l'un qu'avec l'autre. Ce sont juste 2 manières de programmer assez différentes.

L'avantage, c'est qu'à la fin de ce cours vous saurez aussi bien programmer en C qu'en C++ selon vos besoins

Programmer, c'est dur ?

Voilà une question qui doit bien vous torturer l'esprit
Alors : faut-il être un super mathématicien qui a fait 10 ans d'études supérieures pour pouvoir commencer la programmation ?

La réponse, que je vous rassure, est non
Non, un super niveau en maths n'est pas nécessaire. En fait tout ce que vous avez besoin de connaître, ce sont les 4 opérations de base :

- L'addition (bon j'espère que vous maîtrisez)
- La soustraction (ouille ouille ouille !)
- La multiplication (argh)
- La division (bah pourquoi y'a plus personne tout à coup ?)

J'espère que vous connaissez tout ça Et histoire d'en être sûr, je vous expliquerai dans un prochain chapitre comment l'ordinateur réalise ces opérations de base.

Bref, niveau maths, il n'y a pas de difficulté insurmontable
En fait, tout dépend du programme que vous allez faire : si vous devez faire un logiciel de cryptage, alors oui il vous faudra connaître des choses en maths. Si vous devez faire un programme qui fait de la 3D, oui il vous faudra quelques connaissances en géométrie de l'espace.

Chaque cas est particulier.
Pour apprendre le langage C / C++, vous n'avez pas besoin de connaissances pointues en quoi que ce soit.

Il faut savoir comment un ordinateur fonctionne pour comprendre ce qu'on fait. De ce point de vue-là, rassurez-vous, je vous apprendrai tout au fur et à mesure.

Un programmeur a aussi certaines qualités comme :

- **La patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
- **Le sens de la logique** : pas besoin d'être fort en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir (*ah ben zut alors !*)
- **Le calme** : on ne tape pas sur son ordinateur avec un marteau Ce n'est pas ça qui fera marcher votre programme

En bref, et pour faire simple, il n'y a pas de véritables connaissances requises pour programmer. Un nul en maths peut s'en sortir sans problème, le tout est d'avoir la patience de réfléchir. Il y en a beaucoup d'ailleurs qui découvrent qu'ils adorent ça ! Pfiou ! Nous voilà enfin arrivés à la fin de ce premier chapitre
Vous n'avez pas vu une seule ligne de code, certes. On a profité de ce premier chapitre pour voir ce qu'était la programmation et ce que signifiait le C / C++. Maintenant, vous avez une meilleure idée de ce qui vous attend mais vous êtes encore loin d'avoir tout vu !

Dans le prochain chapitre, vous commencerez vos premières manipulations. En effet, vous allez installer les logiciels nécessaires à tout bon programmeur qui se respecte

Ayez les bons outils !

Après un premier chapitre un peu "blabla" (mais nécessaire !), nous commençons à entrer dans le vif du sujet. Nous allons répondre à la question suivante :

Il n'y aura rien de difficile à faire dans ce chapitre, on va prendre le temps de se familiariser avec de nouveaux logiciels.

Profitez-en ! Dans le chapitre suivant, nous commencerons à vraiment programmer et il ne sera plus l'heure de faire la sieste

LES OUTILS NÉCESSAIRES AU PROGRAMMEUR

Alors à votre avis, de quels outils un programmeur a-t-il besoin ?
Si vous avez attentivement suivi le chapitre précédent, vous devez en connaître au moins un !

Vous voyez de quoi je parle ?
…
…
…
Vraiment pas ?

Eh oui, il s'agit du **compilateur**, ce fameux programme qui permet de traduire votre langage C en langage binaire !

Comme je vous l'avais un peu déjà dit dans le premier chapitre, il existe plusieurs compilateurs pour le langage C / C++. Nous allons voir que le choix du compilateur ne sera pas très compliqué dans notre cas

Bon, de quoi d'autre a-t-on besoin ?
Je ne vais pas vous laisser deviner plus longtemps Voici le strict minimum pour un programmeur :

- **Un éditeur de texte** pour écrire le code source du programme (en C ou C++). En théorie un logiciel comme le Bloc-Notes sous Windows, ou "*vi*" sous Linux fait l'affaire. L'idéal, c'est d'avoir un éditeur de texte intelligent qui colore tout seul le code, ce qui vous permet de vous repérer dedans bien plus facilement
- **Un compilateur** pour transformer ("compiler") votre source en binaire.
- **Un débugger** pour vous aider à traquer les erreurs dans votre programme (on n'a malheureusement pas encore inventé le "correcteur", un truc qui corrigerait tout seul nos erreurs)

A priori, si vous êtes un casse-cou de l'extrême, vous pouvez vous passer de débugger… Mais bon, je sais pertinemment que dans moins de 5 minutes vous reviendrez en pleurnichant me demander où on peut trouver un débugger qui marche bien

A partir de maintenant on a 2 possibilités :

- Soit on récupère chacun de ces 3 programmes **séparément**. C'est la méthode la plus compliquée, mais elle fonctionne Sous Linux en particulier, bon nombre de programmeurs préfèrent utiliser ces 3 programmes séparément. Je ne détaillerai pas cette méthode ici, je vais plutôt vous parler de la méthode simple.
- Soit on utilise un programme "3-en-1" (comme les liquides vaisselle, oui oui) qui combine éditeur de texte, compilateur et débugger. Ces programmes "3-en-1" sont appelés **IDE**, ou encore "Environnements de développement"

Il existe plusieurs environnements de développement. Vous aurez peut-être un peu de mal à choisir celui qui vous plaît au début. Une chose est sûre en tout cas: vous pouvez faire n'importe quel type de programme, quel que soit l'IDE que vous choisissez.

Choisissez votre IDE

Il m’a semblé intéressant de vous montrer 3 IDE parmi les plus connus. Tous sont disponibles gratuitement. Personnellement, je navigue un peu entre tous ceux-là et j’utilise l'IDE qui me plaît selon l'humeur du jour

- Vous avez par exemple **Dev C++** qui est très bien. Que son nom ne vous trompe pas : vous pouvez aussi bien faire du C que du C++ avec lui
- Plus récent que Dev C++, l'IDE **Code::Blocks** semble promis à un bel avenir. Il est aussi gratuit et plus tenu à jour que Dev. Il possède en outre quelques fonctionnalités intéressantes et fonctionne sous Windows et Linux. Je conseille d'utiliser celui-ci pour débuter.
- Un des IDE les plus connus, c'est celui de Microsoft : **Visual C++**. Il existe à la base en version payante (chère !), mais heureusement il existe une version gratuite intitulée **Visual C++ Express** qui est vraiment très bien (il y a peu de différences avec la version payante).

Personnellement, entre Dev C++, Code::Blocks et Visual C++ j’aurais tendance à préférer Visual. Je trouve son débugger plus puissant. Toutefois, comme vous débutez vous ne serez pas capables de profiter de toute sa puissance. Vous vous y mettrez sûrement, mais plus tard.

Cela nous laisse donc un choix entre Dev C++ et Code::Blocks pour commencer. Grosso modo, il faut retenir que Dev-C++ est un IDE qui a eu beaucoup de succès pendant un moment, mais il n'est plus mis à jour depuis bien trop longtemps à mon goût. Vous en entendrez quand même sûrement parler car beaucoup de personnes l'ont utilisé et l'utilisent encore.
Je vous recommande donc Code::Blocks pour commencer, mais ce n'est pas une obligation. Quel que soit l'IDE que vous choisissez vous serez capables de faire autant de choses. Vous n'êtes pas limités.

Dans la suite de ce chapitre, je vais présenter chacun de ces 3 IDE histoire que vous puissiez un peu les comparer, ne serait-ce que visuellement.

Au passage, notez que ces IDE sont disponibles sous Windows uniquement, sauf Code::Blocks qui fonctionne aussi sous Linux.

Si vous êtes sous Mac, sachez qu'il existe un IDE appelé "Xcode" et qu'il est présent sur le CD d'installation de Mac OS. Nous verrons son fonctionnement vers la fin de ce chapitre.

Vous pouvez choisir... Dev-C++

Dev C++ est un environnement de développement (IDE) gratuit. C’est probablement le plus connu de tous. Toutefois, il n'est plus trop mis à jour. Aujourd'hui, on recommande de plus en plus d'utiliser **Code::Blocks** (aussi gratuit, présenté plus bas).

Dev C++ est disponible en français et vous pouvez le télécharger sur Internet rapidement. C'est avec cet IDE que j'ai commencé à rédiger ce cours, mais cela fait un moment que j'utilise Code::Blocks et Visual C++. Ne soyez donc pas étonnés si je fais souvent référence à Dev-C++ au début du cours.

Dev-C++ est gratuit et a tout ce qu'il faut pour programmer !

Comme tout programme, il y en a eu plusieurs versions. Les captures d'écran que je fais sont sur la version 4.9.9.0 comme vous pouvez le voir.
Ce genre de programme évolue vite, mais si vous avez une version supérieure ne vous inquiétez pas. Le fonctionnement du programme ne change pas d'une version à l'autre. Peut-être avez-vous de nouvelles icônes, et encore…

Allez sur le site de Bloodshed (l'éditeur du programme) pour le récupérer. Prenez le premier lien de téléchargement que vous voyez (*Dev-C++ with Mingw/GCC*) :

Site web de Dev-C++

A l'installation, faites tout ce qu'on vous recommande de faire. Ca se passe normalement assez vite

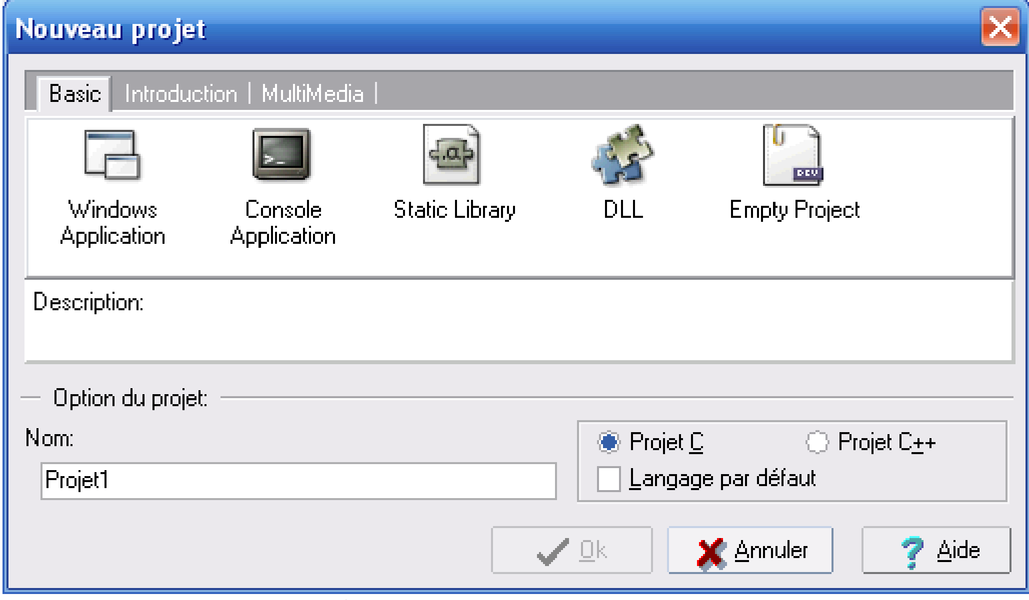
Le démarrage de Dev-C++

Lancez Dev C++.
La première fois, il vous demandera de le configurer. Il vous demandera votre langue et l'aspect du logiciel que vous voulez avoir.
Par ailleurs, il vous posera ensuite 2-3 questions sur la création de fichiers spéciaux pour vous aider. Je vous conseille de laisser les options par défaut, à savoir "*Oui, je le veux*"
Ca ne vous sera pas utile de suite, mais plus tard vous ne regretterez pas d'avoir répondu oui

Au départ, rien ne s'affiche. Il va falloir demander à Dev C++ de créer un nouveau projet.

Un **projet** c'est l'ensemble de tous les fichiers source du programme. En effet, quand on programme, on sépare souvent notre code dans plusieurs fichiers différents. Ces fichiers seront ensuite "combinés" par le compilateur qui en fera un exécutable (un ".exe").

Pour créer un nouveau projet c'est simple : allez dans le menu "Fichier / Nouveau / Projet". Vous devriez voir quelque chose qui ressemble à ça :



La fenêtre de création de projet de Dev C++

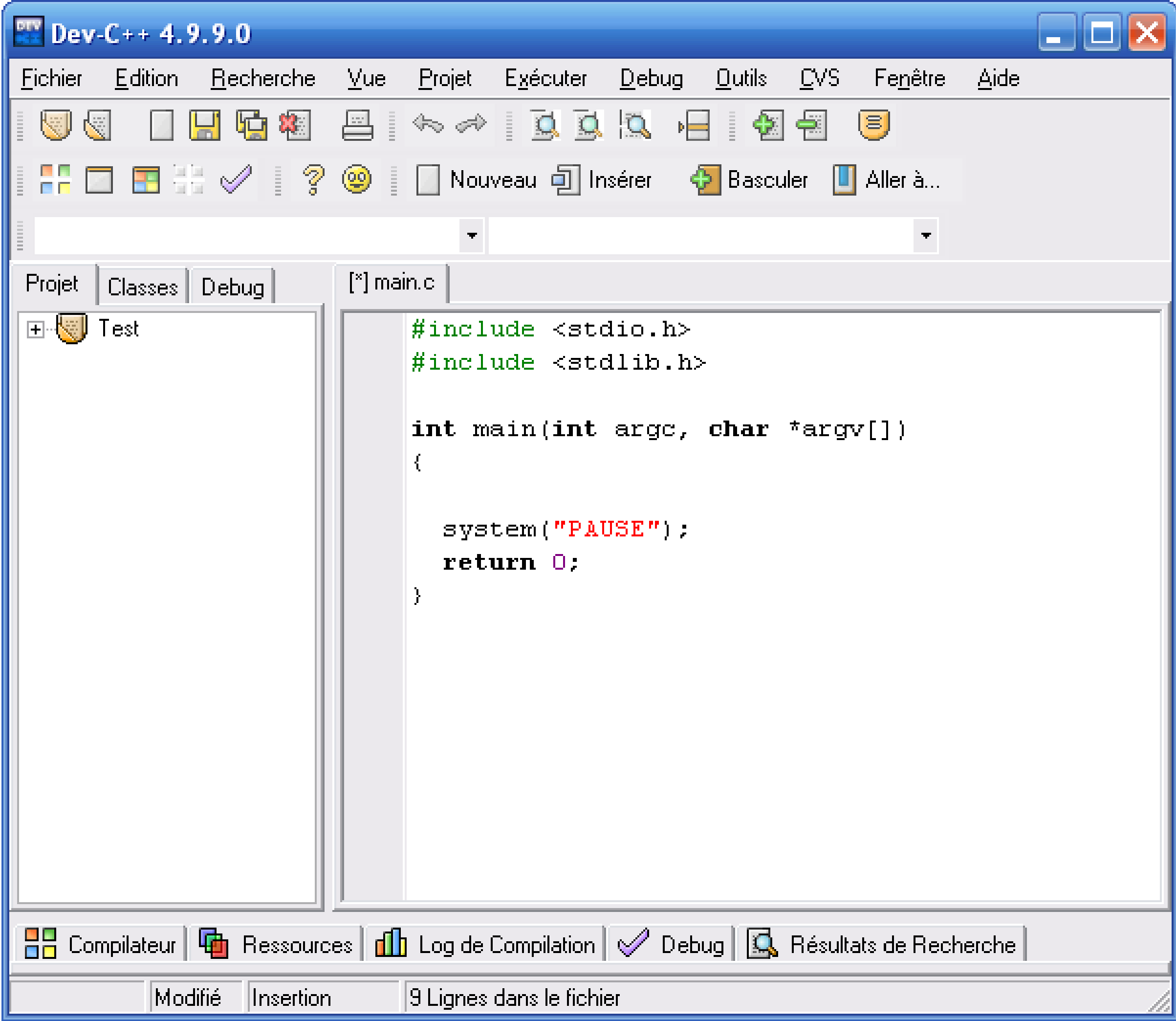
Là, on vous demande quel genre de programme vous voulez créer. Retenez bien la marche à suivre, car vous devrez faire cela la plupart du temps (surtout au début) :

- Cliquez sur "**Console Application**". Eh oui, il n'est pas possible de commencer par créer des fenêtres avec "Windows Application", il est vraiment trop tôt On va pour commencer se contenter de créer des programmes qui s'affichent dans une console, qui ressemble un peu à DOS.
- Sélectionnez "**Projet C**" si ce n'est déjà fait.
- Cochez "**Langage par défaut**"
- Donnez **un nom** à votre projet (autre que "Projet 1")

Faites OK.
On vous demande alors où placer le fichier " .dev ". Ce fichier, propre à Dev C++, est le fichier de votre projet. Il contient la liste des fichiers source de votre programme. Il vous faut enregistrer votre projet avant même d'avoir commencé à programmer ! Remarquez, avec Visual C++ c'est pareil, sauf que lui il n'utilise pas un fichier de projet mais plusieurs

Je vous conseille de créer un dossier pour votre projet.
Une fois que vous avez indiqué où enregistrer votre projet, Dev C++ crée alors un premier fichier source qui s'appelle " main.c ". Ce sera le fichier principal de notre programme, on aura l'occasion d'en reparler dans le prochain chapitre.

Normalement, Dev C++ écrit déjà un peu de code dedans (le strict minimum). N'essayez pas de deviner ce qu'il signifie, attendez plutôt le chapitre suivant qu'on analyse ça en détail



Un nouveau projet tout neuf !

Pour ceux qui seraient déjà un peu perdus, j'ai réalisé une vidéo vous montrant comment je crée un nouveau projet sous Dev-C+. Je suis exactement les mêmes étapes que je vous ai énoncées plus haut :

Créer un nouveau projet avec Dev-C++ (254 Ko)

Que dire de plus sur Dev C++ ?
Il est constitué de plusieurs parties qu'on a tout intérêt à regarder de plus près histoire de voir comment tout ce bazar fonctionne

Les principales fonctionnalités de Dev-C++

Voyons voir plus en détail comment Dev-C++ est organisé :

Les différentes parties de Dev C++

J'ai séparé Dev en 4 grandes parties :

1. En haut, vous avez les menus et la barre d'outils. Je peux vous conseiller d'aller modifier un peu les options si ça vous chante. C'est dans les menus Outils / Options d'environnement et Outils / Options de l'éditeur.
   Pour ce qui est des icônes de la barre d'outils, qu'on utilisera souvent, il y en a beaucoup que vous connaissez. Les premières servent notamment à créer un nouveau projet, un nouveau fichier, à enregistrer le fichier, à enregistrer tous les fichiers ouverts etc. Je souhaite attirer votre attention sur les boutons se situant au début de la deuxième ligne (du moins sur ma capture d'écran) :

   Les icônes lançant la compilation

   Ces 5 icônes sont sans aucun doute les plus utilisées, et pour cause : ce sont elles qui permettent d'appeler le compilateur pour créer un exécutable de votre projet
   Dans l'ordre, de gauche à droite, ces icônes signifient :
   - **Compiler** : tous les fichiers source de votre projet sont envoyés au compilateur qui va se charger de créer un exécutable. S'il y a des erreurs (ce qui a de fortes chances d'arriver), l'exécutable ne sera pas créé et on vous indiquera les erreurs en bas de DevC++ (dans la partie que j'ai numérotée 4)
   - **Exécuter** : cette icône lance juste le dernier exécutable que vous avez compilé. Cela vous permettra donc de tester votre programme et voir ainsi ce qu'il donne Dans l'ordre, si vous avez bien suivi, on doit d'abord compiler, puis exécuter pour tester ce que ça donne. On peut aussi utiliser le 3ème bouton…
   - **Compiler & Exécuter** : pas besoin d'être un génie pour comprendre que c'est la combinaison des 2 boutons précédents. C'est d'ailleurs ce bouton que vous utiliserez le plus souvent. Notez que s'il y a des erreurs pendant la compilation (pendant la génération de l'exécutable), le programme ne sera pas exécuté. A la place, vous aurez droit à une beeelle liste d'erreurs à corriger

- **Tout reconstruire** : quand vous faites " Compiler ", DevC++ ne recompile en fait que les fichiers que vous avez modifiés et pas les autres. Parfois, je dis bien parfois, vous aurez besoin de demander à Dev de vous recompiler tous les fichiers. On verra plus tard quand on a besoin de ce bouton, et vous verrez plus en détail le fonctionnement de la compilation dans un chapitre futur. Pour l'instant, on se contente de savoir le minimum nécessaire pour pas tout mélanger
  Ce bouton ne nous sera donc pas utile de suite.
- **Débugger** : ce bouton lance votre programme en mode débuggage. C'est un mode particulier qu'on apprendra à utiliser plus tard. Cela vous permet de traquer les erreurs de votre programme, de le mettre en " pause " lors de son exécution etc etc.

  Je vous conseille d'utiliser les raccourcis plutôt que de cliquer sur les boutons, parce que c'est quelque chose qu'on fait vraiment très très souvent. Vous pouvez connaître le raccourci en pointant sur le bouton qui vous intéresse. Chez moi par exemple, je tape F9 pour faire " Compiler & Exécuter "

2. Dans la section de gauche de Dev-C++ s'affichent en général tous les fichiers de votre projet (qui s'appelle " Test " sur ma capture d'écran). Cliquez sur le petit " + " à gauche pour dérouler la liste des fichiers ouverts. On se sert souvent de cette liste pour naviguer d'un fichier du projet à un autre.
   Les onglets en haut de cette section sont les suivants :
   - **Projet** : c'est là que vous avez la liste des fichiers du projet dont je viens de vous parler.
   - **Classes** : c'est un onglet que nous n'utiliserons pas en langage C. Cela ne sert que quand on fait du C++, et ne comptez pas sur moi pour vous expliquer maintenant ce que c'est
   - **Debug** : c'est l'onglet qui est utilisé pendant que vous débuggez votre programme. Il permet en particulier de voir ce qu'il y a dans votre mémoire vive. Vous ne savez pas ce que c'est une "mémoire vive" ? Ce n'est pas grave, on aura le temps de le découvrir plus tard

3. Ah, la partie principale C'est là que s'affiche le fichier source en C ou C++ que vous êtes en train de modifier. C'est dans cette zone de l'écran que vous passerez le plus clair de votre temps
   Notez qu'en haut de cette zone, tous les fichiers ouverts apparaissent sous forme d'onglets. Sur ma capture d'écran il n'y a que "main.c" pour le moment. Cliquez sur l'un d'eux pour afficher le fichier correspondant.

   Autre info utile : tous les fichiers modifiés et non enregistrés sont précédés d'une petite étoile [*]. Enregistrez souvent. Enregistrez tout le temps. On ne compte plus le nombre de tentatives de suicide de personnes qui avaient oublié d'enregistrer et qui ont eu une coupure de courant (je rigole hein, vous jetez pas par la fenêtre si ça vous arrive, ça résoudra rien). N'hésitez pas à utiliser le bouton "Sauvegarder tout" de la barre d'outils , il enregistre tous les fichiers ouverts d'un seul coup.

4. Le bas de l'écran... C'est la zone que détestent tous les programmeurs. En effet, lors d'une compilation qui "plante", les erreurs s'affichent dans la partie basse de l'écran. En général, vous ne naviguerez pas trop entre les onglets de cette partie, sauf peut-être l'onglet *Debug* pour débugger votre programme et l'onglet "log de compilation" qui indique si la compilation s'est bien passée ou non.

Pfiou ! On a fait à peu près le tour de Dev C++
On a vu les principales sections du programme qu'on utiliserait. Normalement ça devrait vous permettre de vous débrouiller la plupart du temps

Passons maintenant à Visual C++ !

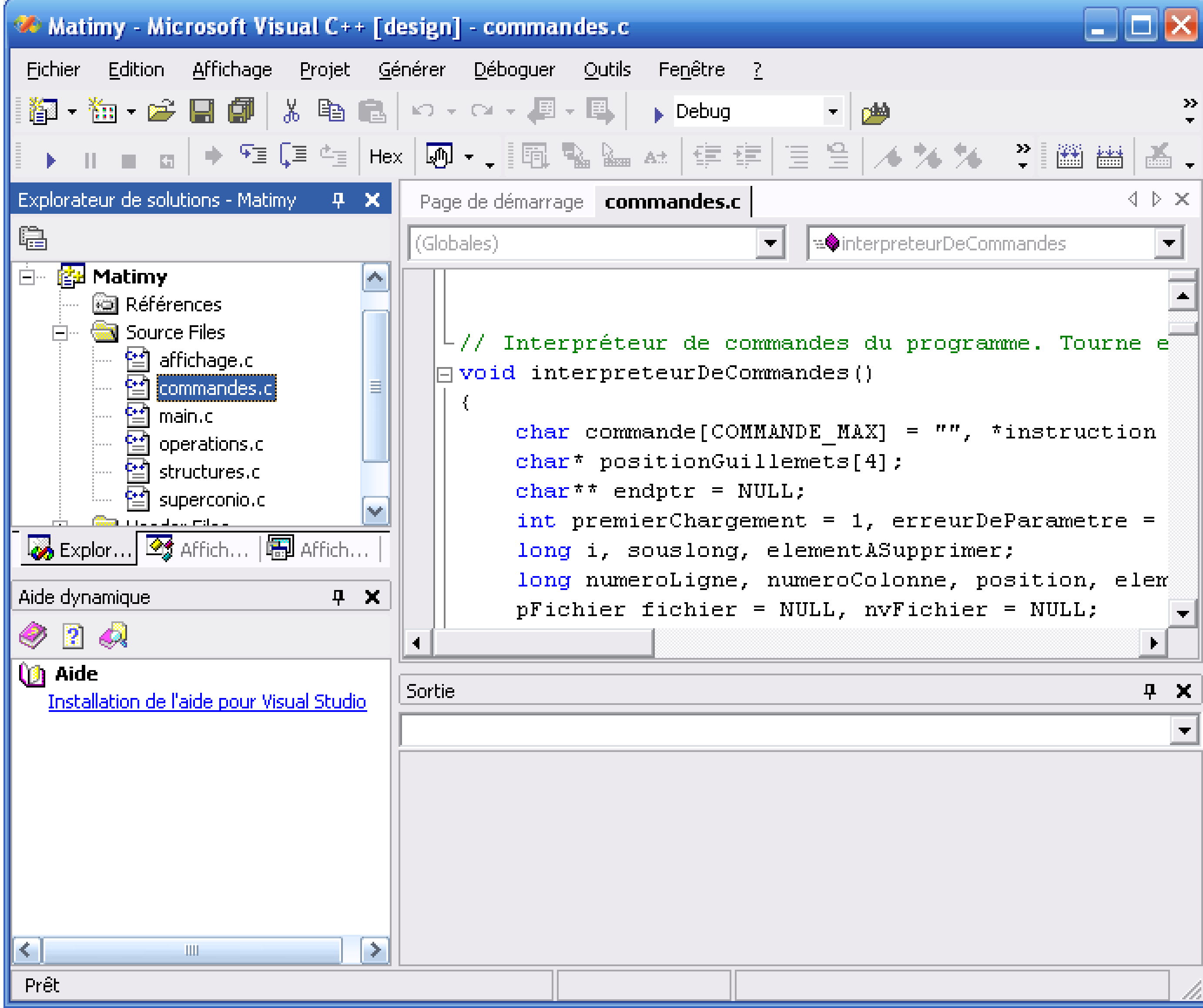
OU BIEN... VISUAL C++

Quelques petits rappels sur Visual C++ :

- C'est l'IDE de Microsoft
- Il est à la base payant, mais Microsoft a sorti une version gratuite intitulée Visual C++ Express.

Je vais dans un premier temps vous présenter la version payante de Visual C++ qui est celle que j'utilise.
Je vous parlerai ensuite de Visual C++ Express, qui ressemble beaucoup à Visual C++ (on s'en serait doutés !).

Voici une capture d'écran de Visual C++ .NET 2005 :



L'IDE Visual C++ de Microsoft

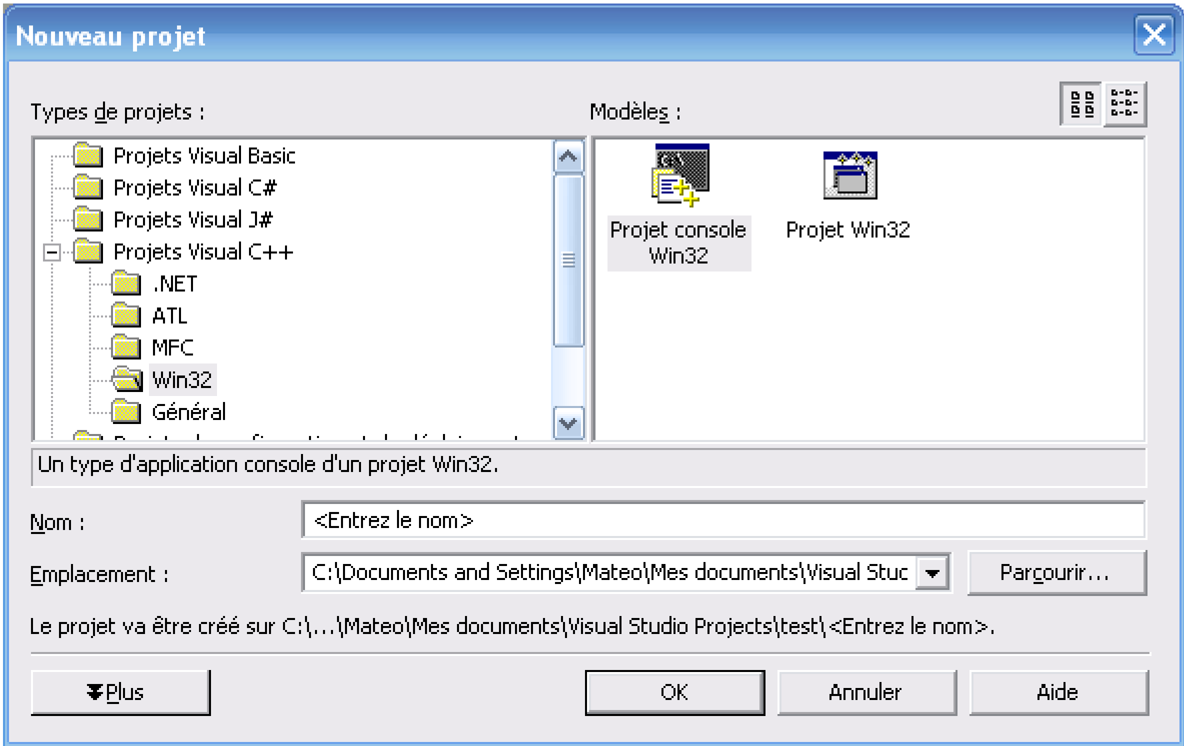
Je vais aller un peu plus vite que pour Dev C++ car dans l'ensemble les 2 IDE se ressemblent assez.

Comme pour Dev, il y a eu (et il y aura) plusieurs versions de Visual. Les dernières versions s'appellent Visual C++ .NET.

Microsoft a, il faut le savoir, rassemblé les IDE de différents langages dans une sorte de gros pack appelé " Visual Studio ". Cet ensemble de logiciels comprend Visual C++, Visual Basic ainsi que d'autres outils. Nous, nous n'avons besoin que de Visual C++ ici.

Un nouveau projet avec Visual C++

Pour créer un nouveau projet, direction le menu "Fichier / Nouveau / Projet" de Visual Studio. Vous allez voir quelque chose qui ressemble à ceci :

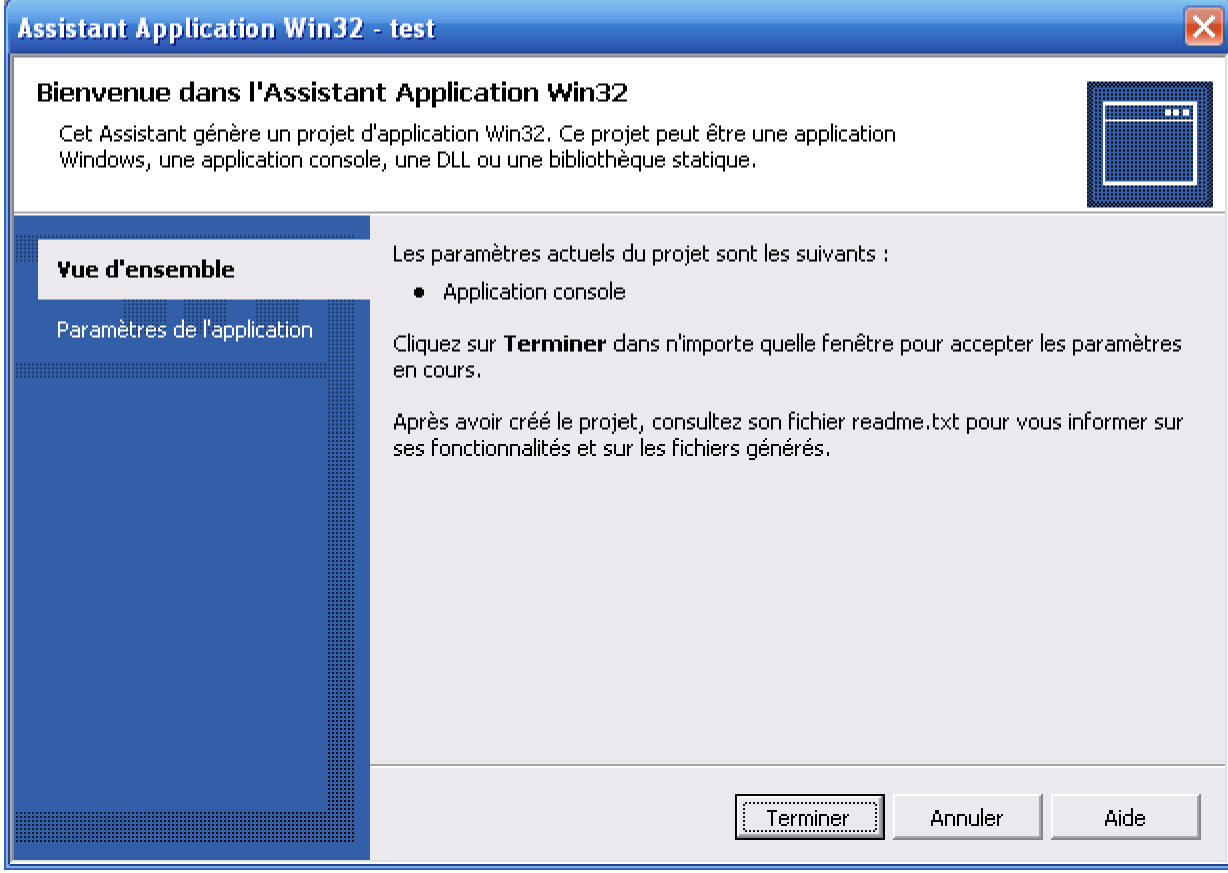


Création d'un nouveau projet avec Visual C++

Il y a beaucoup plus de choix que pour Dev-C++, notamment parce que, comme je vous l'ai dit, Visual Studio permet de créer des programmes dans d'autres langages.
En revanche, Visual ne propose pas aussi clairement que Dev le choix entre C et C++. En fait, si vous voulez faire du C, il faudra aller dans le dossier "Projets Visual C++" à gauche. Ca ne change pas grand chose pour nous.

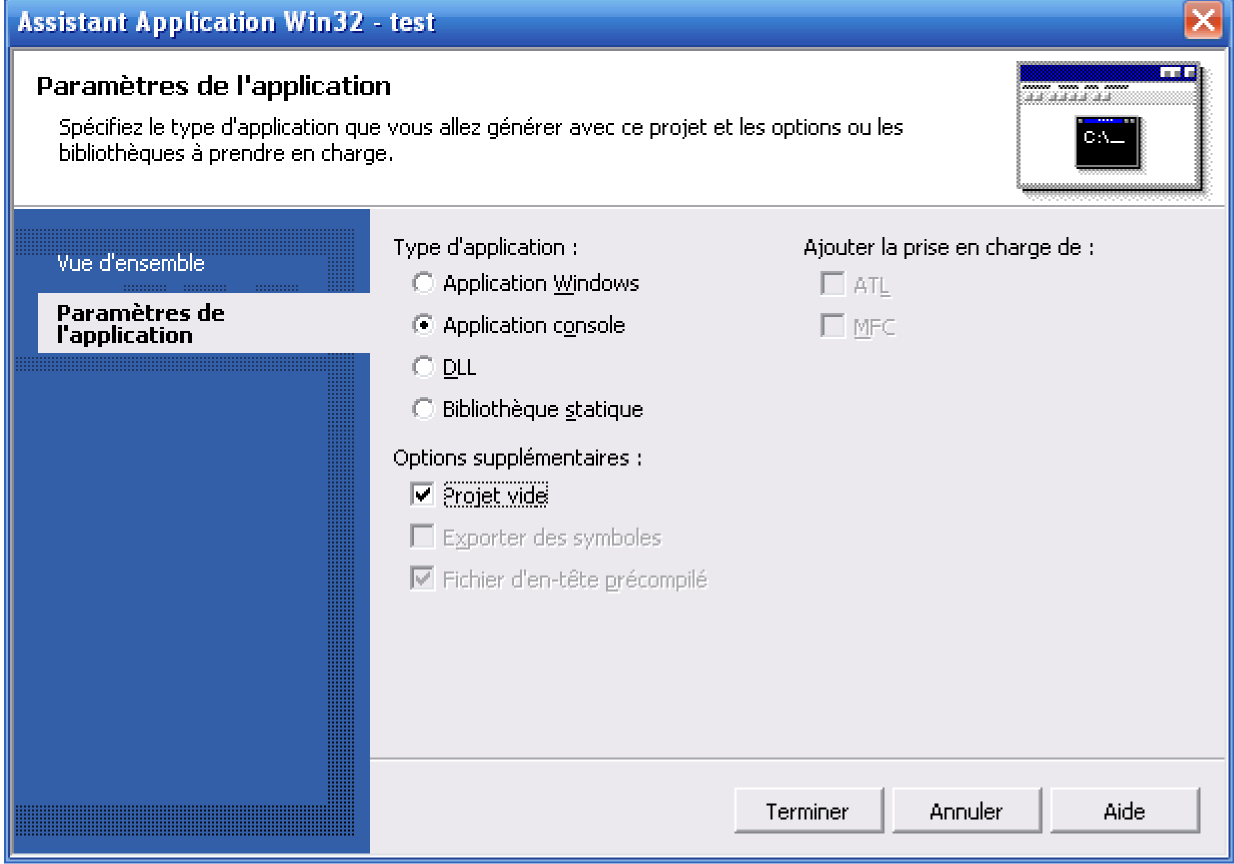
Dans la liste qui apparaît, choisissez le sous-dossier "Win32". Enfin, à droite cliquez sur "Projet console Win32", comme sur ma capture d'écran.

Indiquez un nom et un dossier où stocker votre projet, qu'on va appeler "test" pour le moment.
Une fenêtre d'assistant apparaît :



L'assistant de nouveau projet

Ok c'est bien gentil, mais ici il n'y a rien à faire. En fait, il faut aller dans le menu à gauche "Paramètre de l'application". Ca devient alors un peu plus intéressant déjà :



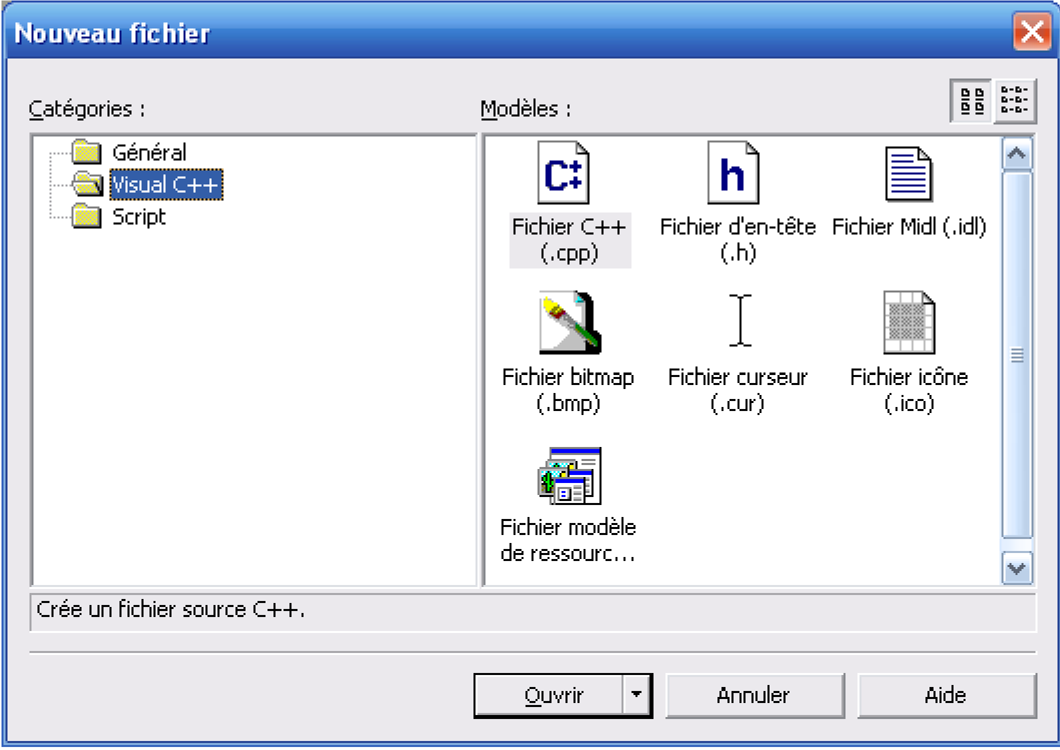
Un peu plus d'options

C'est là qu'il ne faut pas se tromper. Normalement, vous devriez avoir "Application console" de choisi pour "Type d'application".
Vous devez surtout cocher la case "Projet vide" (qui n'est pas cochée par défaut). En effet, si vous ne faites pas ça Visual va nous créer un peu trop de fichiers pour nous pauvres débutants que nous sommes

Vous y êtes ?
Alors cliquez sur "Terminer" en bas, et c'est bon !

Les principales fonctionnalités de Visual C++

Actuellement, le projet est vide. Il n'y a aucun fichier dedans. Je vous invite à aller dans le menu "Fichier / Nouveau / Fichier". Vous devriez alors voir cette fenêtre :



Création d'un nouveau fichier

Dans les petits dossiers à gauche, sélectionnez "Visual C++". Là encore, vous avez pas mal de choix.
Comme vous pouvez le voir, Visual possède des éditeurs de fichiers bitmap, de curseurs ou encore d'icônes pour votre programme. Certes, c'est vrai que c'est sympa, mais ce n'est pas non plus indispensable.
Dans la version gratuite (Visual C++ Express) vous devriez avoir moins de choix. En effet, l'éditeur d'icônes et de curseurs est réservée à la version payante... et franchement on s'en fout, parce qu'il existe plein d'outils gratuits pour ce genre de choses

2 types de fichiers seulement nous intéressent :

- Fichier C++ (.cpp)
- Fichier d'en-tête (.h)

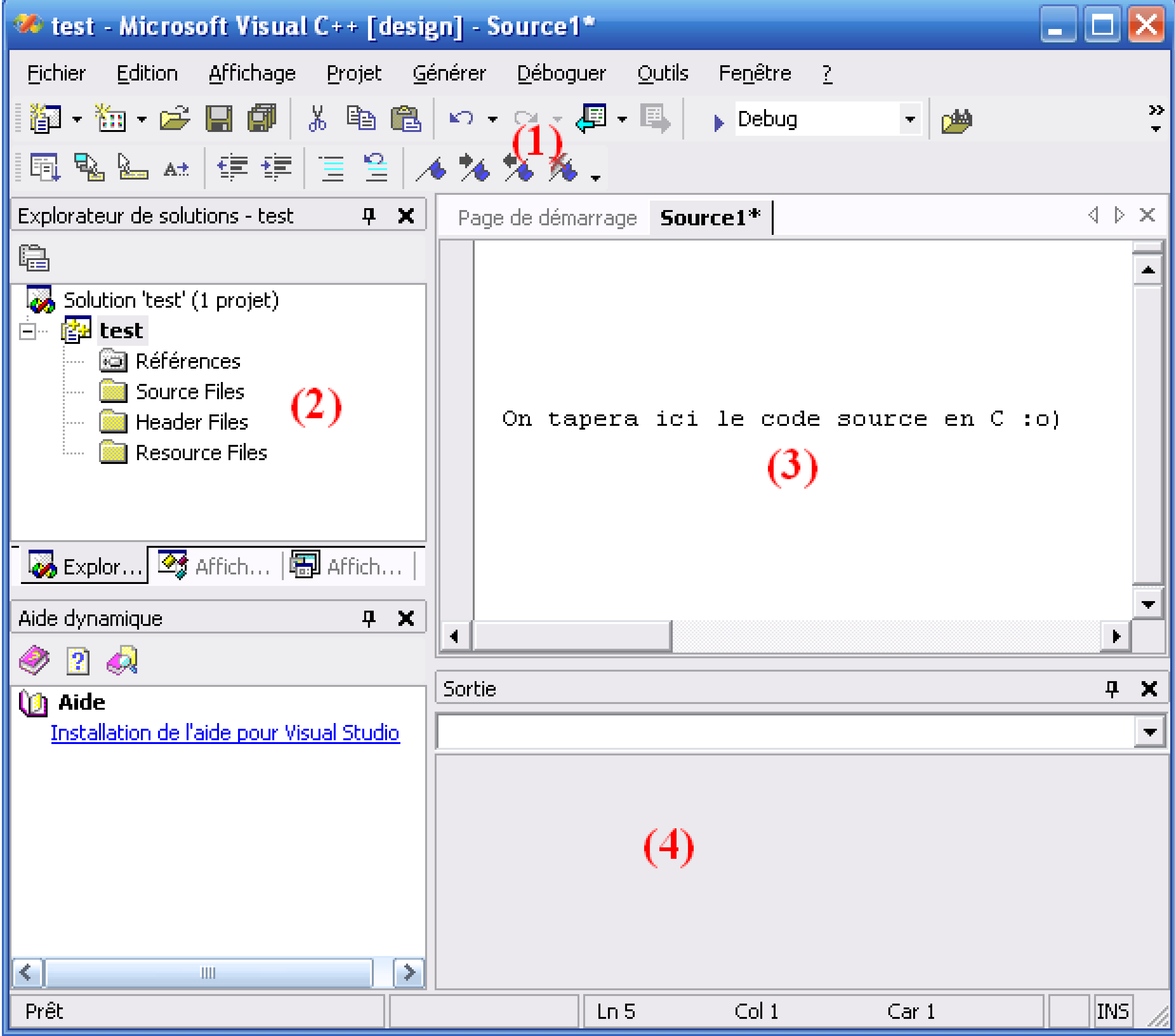
Nous verrons ce que sont les fichiers d'en-tête un peu plus tard.
Pour le moment, créez un fichier C++ (.cpp).

Contrairement à Dev, Visual ne nous demande pas si on veut faire du C ou du C++. Il met du C++ par défaut. Pour nous, ça ne changera pas grand chose : vous enregistrerez vos fichiers en .c (même si Visual vous propose .cpp par défaut) et tout ira bien

Résumons en vidéo ce qu'il faut faire sous Visual pour créer un nouveau projet et ajouter un fichier source :

Créer un nouveau projet sous Visual-C++ (317 Ko)

C'est fait ? Parfait
Voyons voir maintenant l'IDE de Visual en détail :



Visual C++ en détail

Comme vous pouvez le constater, il ressemble pas mal à Dev-C++.

On va rapidement (re)voir quand même ce que signifient chacune des parties :

1. La barre d'outils, tout ce qu'il y a de plus standard. Ouvrir, enregistrer, enregistrer tout, couper, copier, coller etc. Par défaut, il semble qu'il n'y ait pas de bouton de barre d'outils pour compiler. Vous pouvez les rajouter en faisant un clic droit sur la barre d'outils, puis en choisissant "Déboguer" et "Générer" dans la liste. Toutes ces icônes de compilation ont leur équivalent dans les menus "Générer" et "Déboguer". Si vous faites "Générer", cela créera l'exécutable (ça signifie "Compiler" pour Visual). Si vous faites "Déboguer / Exécuter", on devrait vous proposer de compiler avant d'exécuter le programme. F7 permet de générer le projet, et F5 de l'exécuter.

2. Dans cette zone très importante vous voyez normalement la liste des fichiers de votre projet. Cliquez sur l'onglet "Explorateur de solutions" en bas si ce n'est déjà fait. Vous devriez voir que Visual crée déjà des dossiers pour séparer les différents types de fichiers de votre projet (sources, en-tête et ressources). Nous

verrons un peu plus tard quels sont les différents types de fichiers qui constituent un projet

3. La partie principale. C'est là qu'on modifie les fichiers source.
4. C'est là encore la "zone de la mort", celle où on voit apparaître toutes les erreurs de compilation. C'est dans le bas de l'écran aussi que Visual affiche les informations de débuggage quand vous essayez de corriger un programme buggé. Je vous ai d'ailleurs dit tout à l'heure que j'aimais beaucoup le débugger de Visual, et je pense que je ne suis pas le seul On essaiera d'apprendre à l'utiliser un peu plus tard si on trouve le temps.

Voilà, on a fait le tour de Visual C++.
Vous pouvez aller jeter un œil dans les options (Outils / Options) si ça vous chante, mais n'y passez pas 3 heures. Il faut dire qu'il y a tellement de cases à cocher de partout qu'on ne sait plus trop où donner de la tête

Visual C++ Express : une version gratuite

Il existe une version gratuite de Visual C++ appelée **Visual C++ Express Edition** !

Il n'y a pas d'éditeur de ressources (vous permettant de dessiner des images, des icônes, ou des fenêtres). Mais bon, ça entre nous on s'en fout parce qu'on n'aura pas besoin de s'en servir dans ce tutorial Ce ne sont pas des fonctionnalités indispensables bien au contraire.

Vous trouverez les instructions pour télécharger Visual C++ Express à cette adresse :

Site de Visual C++ Express Edition

Visual C++ Express est en français et est totalement gratuit. Ce n'est donc pas une version d'essai limitée dans le temps.
Notez que cette version gratuite existe depuis relativement peu de temps, auparavant Visual était disponible uniquement en version payante.

C'est une chance d'avoir un IDE aussi puissant que celui de Microsoft disponible gratuitement, donc ne la laissez pas passer
Ca vous permettra par la suite de profiter de son très puissant et renommé débugger, que tous les programmeurs adorent

OU ENCORE... CODE::BLOCKS

Code::Blocks est un IDE libre et gratuit, plus récent que Dev-C++
Si Dev reste un des IDE gratuits les plus connus, il le doit surtout à son ancienneté.

Code::Blocks étant relativement nouveau, il n'est pas encore aussi connu. Pourtant, n'allez pas penser qu'il est peu avancé pour autant ! Bien au contraire, je dois avouer que j'ai été surpris par cet éditeur et je vous recommande de l'essayer

En outre, Code::Blocks est disponible **pour Windows et pour Linux**. En théorie il serait possible de le faire marcher sous Mac mais à l'heure où j'écris ces lignes on ne nous propose pas vraiment de version Mac malheureusement
Code::Blocks n'est disponible pour le moment qu'en anglais. Ca ne devrait PAS vous repousser à l'utiliser. Quand vous programmerez vous serez de toute façon confronté bien souvent à des documentations en anglais, donc raison de plus pour s'entraîner à utiliser cette langue. Ca ne complique pas l'utilisation du logiciel de toute manière.

Télécharger Code::Blocks

Rendez-vous sur la page de téléchargements de Code::Blocks.

Si vous êtes sous Windows, repérez la section "Windows" un peu plus bas sur cette page. Téléchargez le logiciel en prenant : "**Code::Blocks IDE, with MINGW compiler**" (l'autre version étant sans compilateur, vous auriez eu du mal à compiler vos programmes)
Si vous êtes sous Linux, il y a un lien en haut pour choisir le package RPM à télécharger.

L'installation est très simple et rapide. Laissez toutes les options par défaut et lancez le programme.

Le fonctionnement du programme est quasiment le même que pour Dev et Visual, vous ne serez pas perdus.
Vous trouverez là encore dans la barre d'outils les boutons (dans l'ordre) "Compiler", "Exécuter", "Compiler & Exécuter" et "Tout recompiler" (comme Dev)

Créer un nouveau projet

Pour créer un nouveau projet c'est très simple : allez dans le menu File / New Project.
Dans la fenêtre qui s'ouvre, choisissez "Console application", et sélectionnez "File Options : C Source" dans la liste déroulante en bas.

Comme vous pouvez le voir, Code::Blocks propose de réaliser pas mal de types de programmes différents qui utilisent des librairies connues comme la SDL (2D), OpenGL (3D), QT et wxWidgets (Fenêtres) etc etc... Pour l'instant, ces icônes servent plutôt à faire joli car <u>les librairies ne sont pas installés sur votre ordinateur</u>, vous ne pourrez donc pas les faire marcher.
Nous nous intéresserons à ces autres types de programmes bien plus tard. En attendant il faudra vous contenter de "Console", car vous n'avez pas encore le niveau nécessaire pour créer les autres types de programmes.

Cliquez sur "Create" pour créer le projet.
On vous demandera où enregistrer les fichiers (là encore, je vous recommande de créer un dossier spécial pour chaque projet que vous créez).

Code::Blocks vous créera un premier projet avec déjà un tout petit peu de code source dedans

Sous Mac... Xcode

Malheureusement pour ceux qui possèdent un Mac, tous les IDE présentés ci-dessus ne fonctionnent que sous Windows. N'y a-t-il donc aucun IDE pour Mac ?

Bien sûr que si, rassurez-vous Il en existe plusieurs sous Mac, et je vais vous présenter ici le plus célèbre d'entre eux : Xcode.

Cette section dédiée à Xcode est une adaptation d'un tuto paru sur LogicielMac.com, avec l'aimable autorisation de son auteur PsychoH13.

Xcode, où es-tu ?

Tous les utilisateurs de Mac OS ne sont pas des programmeurs. Apple l'a bien compris et n'installe pas par défaut d'IDE avec Mac OS.
Heureusement, pour ceux qui voudraient programmer, tout est prévu. En effet, Xcode est présent sur le CD d'installation de Mac OS.

Insérez donc le CD dans le lecteur et installez-le. Il se trouve dans les "Developer Tools".

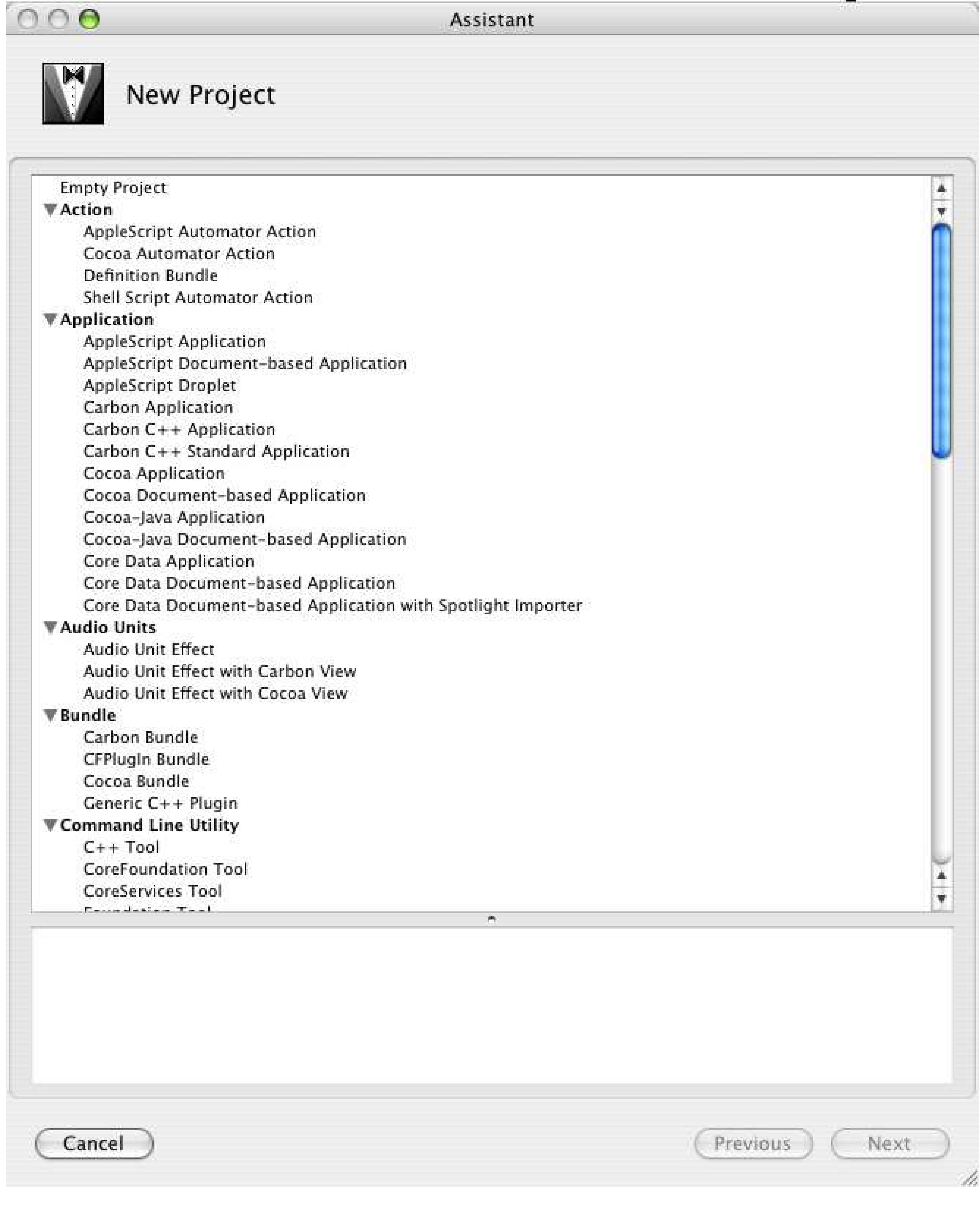
Par ailleurs, je vous conseille de mettre en favoris la page dédiée aux développeurs sur le site d'Apple. Vous y trouverez une foule d'informations utiles pour le développement sous Mac. Vous pourrez notamment y télécharger plusieurs logiciels pour développer.
N'hésitez pas à vous inscrire à l'ADC (Apple Development Connection), c'est gratuit et vous serez ainsi tenu au courant des nouveautés.

Lancement de Xcode

Lorsque vous lancez Xcode pour la première fois, vous serez probablement surpris. Et y'a de quoi
Contrairement à la plupart des logiciels Mac, il n'y a pas de fenêtre de bienvenue. En fait, la première fois, on trouve ça un peu vide... et pourtant, c'est un logiciel très puissant !

Xcode est l'IDE le plus utilisé sous Mac, créé par Apple lui-même. Les plus grands logiciels, comme iPhoto et Keynote, ont été codés à l'aide de Xcode. C'est réellement l'outil de développement de choix quand on a un Mac !

La première chose à faire est de créer un nouveau projet, alors commençons par ça
Allez dans le menu File / New Project. La fenêtre suivante s'ouvre :



Que de choix n'est-ce pas
Bon allez je vous aide : pour commencer, il faut que vous alliez dans la section "Command line utility" et que vous sélectionniez "Standard tool".

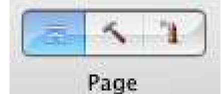
Cliquez ensuite sur Next. On vous demandera où vous voulez enregistrer votre projet (un projet doit toujours être enregistré dès le début). Placez-le dans le dossier que vous voulez.

Une fois créé, votre projet se présentera sous la forme d'un dossier contenant de multiples fichiers dans le Finder. Le fichier à l'extension `.xcodeproj` correspond au fichier du projet. C'est lui que vous devrez sélectionner la prochaine fois si vous souhaitez réouvrir votre projet.

La fenêtre de développement

Dans Xcode, si vous sélectionnez main.c, vous devriez avoir la fenêtre suivante :

La fenêtre est découpée en 4 parties, ici numérotées de 1 à 4 :

1. La première partie est la barre de boutons tout en haut. Vous pouvez la configurer comme bon vous semble, changer les boutons, etc. Voyons les plus importants d'entre eux :
   - 

     Ces 3 boutons vous permettent de naviguer entre, dans l'ordre :
     - "Project" : là où vous voyez vos fichiers et où vous les modifiez
     - "Build" : vous y voyez le résultat de la compilation de votre programme, et les erreurs s'il y en a eu.
     - "Debug" : la fenêtre de déboggage, où vous pouvez exécuter votre programme ligne par ligne pour trouver et comprendre les erreurs de vos programmes.
   -

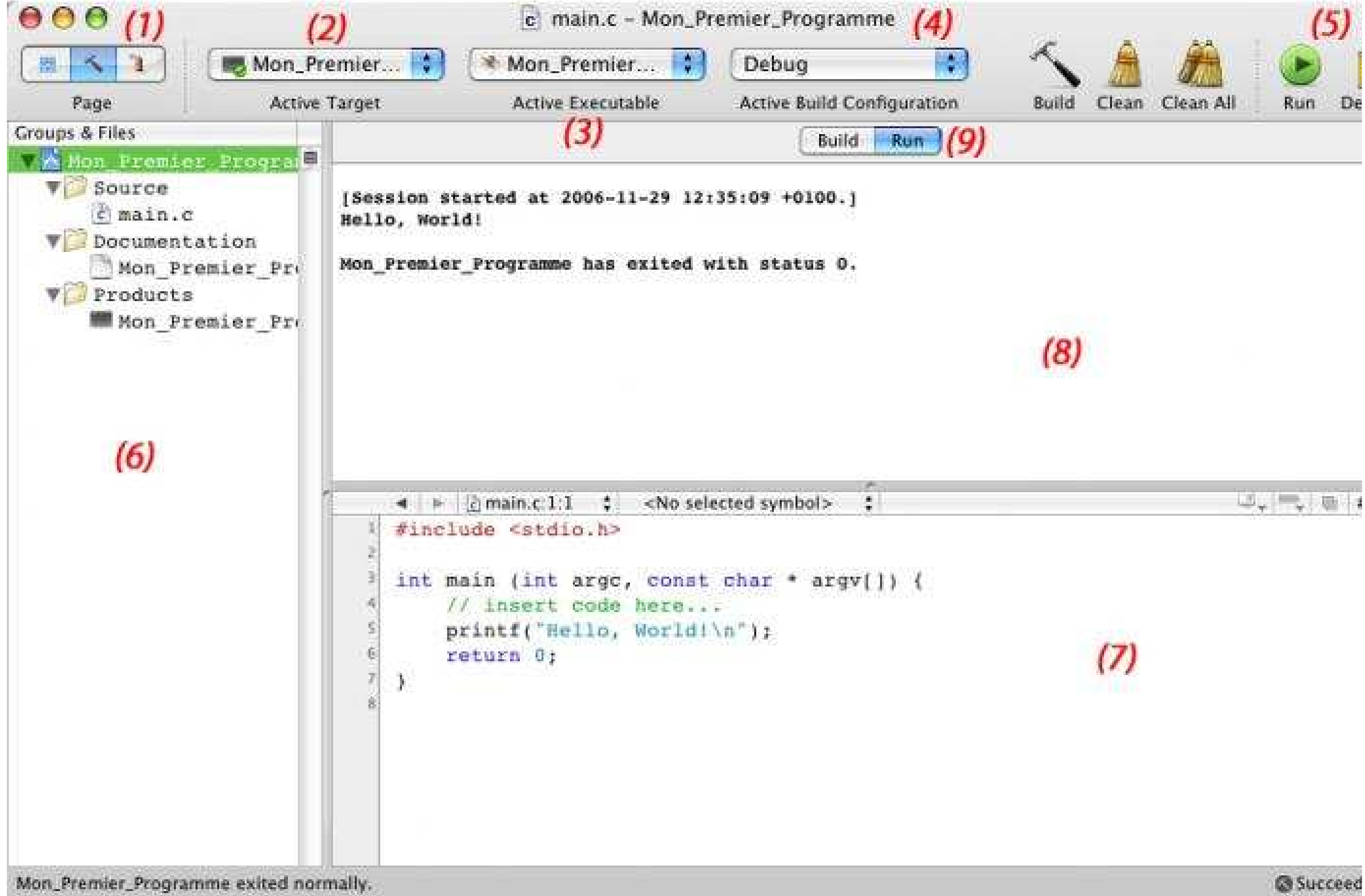
     Ces deux boutons signifient :
     - "Build" : compile votre projet, donc crée un exécutable à partir de vos sources.
     - "Build and Go" (le bouton que vous utiliserez le plus souvent) : compile votre projet et le lance pour le tester.
2. La partie de gauche correspond à l'arborescence de votre projet. Certaines sections regroupent les erreurs, les

avertissements, etc. Xcode vous place automatiquement dans la section la plus utile, celle qui porte le nom de votre projet.

3. La troisième partie change en fonction de ce que vous avez sélectionné dans la partie de gauche. Ici, on a la liste des fichiers de notre projet :
   - *main.c* : c'est le fichier source de votre programme (il peut y en avoir plusieurs dans les gros programmes)
   - *Mon_Premier_Programme* : c'est votre programme une fois compilé, donc l'exécutable que vous pouvez distribuer. Si le fichier est en rouge, c'est qu'il n'existe pas encore (vous n'avez donc pas encore compilé votre programme, mais Xcode le référence quand même).
   - *Mon_Premier_Programme.1* : c'est votre programme présenté en langage assembleur, un langage très proche du processeur. Cela ne nous intéressera pas, mais si vous voulez prendre peur n'hésitez pas à y jeter un oeil
4. Enfin, la 4ème partie, la plus intéressante : c'est celle dans laquelle vous pourrez écrire votre code source en langage C. Par défaut, Xcode met juste un petit code d'exemple qui affiche "Hello, world!" à l'écran.

Lancement du programme

Pour tester ce premier programme, cliquez sur le bouton "Build and Go" de la barre d'outils.
Votre écran devrait maintenant ressembler à cela :



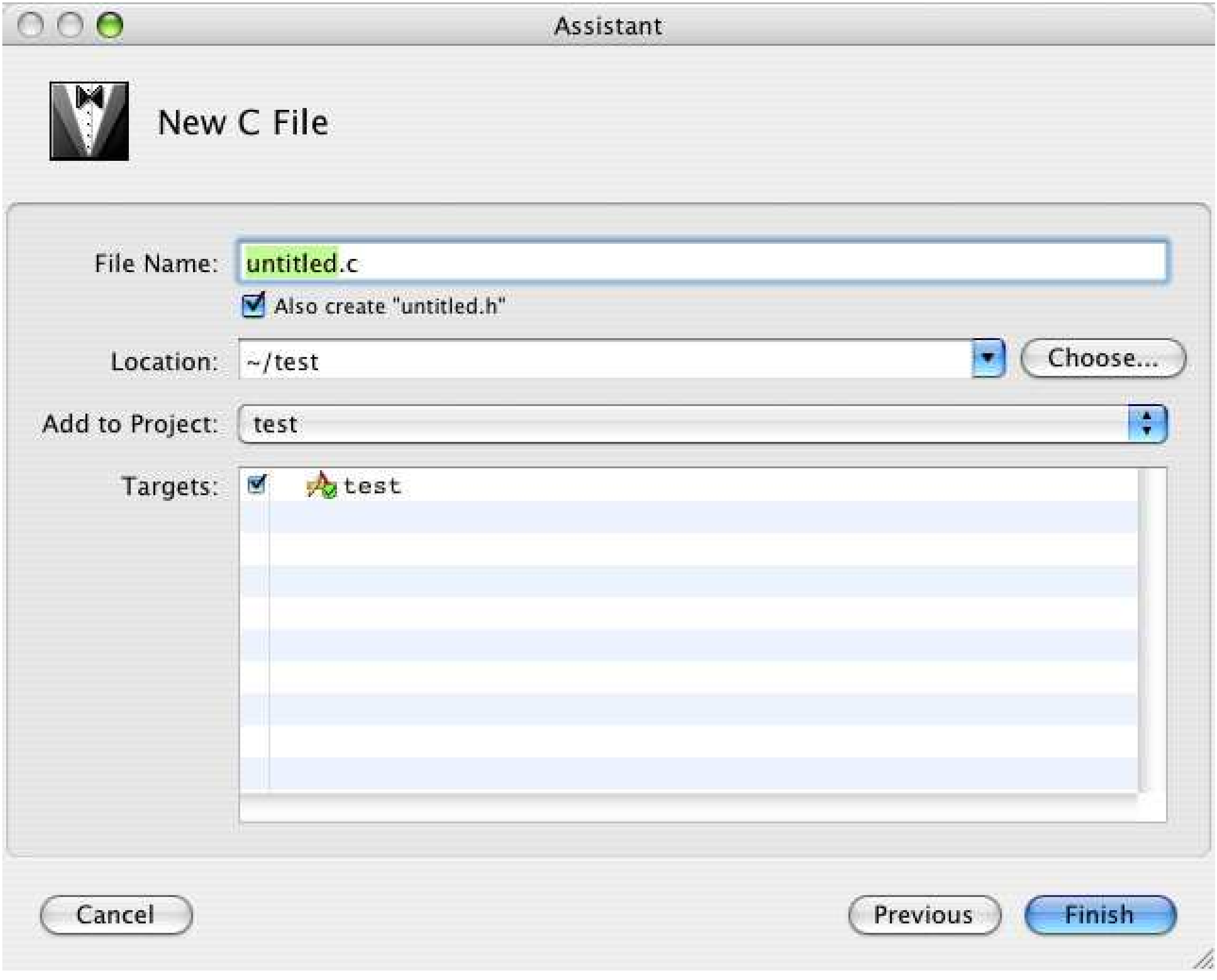
1. Ce sont les boutons qui permettent de changer de page, comme on l'a vu plus tôt. Sélectionnez "Project" si vous souhaitez revenir à la fenêtre précédente.
2. C'est la cible, le fichier qui réunit les sources compilées de votre programme.
3. L'exécutable de votre application.
4. Le mode de compilation. Il peut être :
   - Debug : l'exécutable reste dans Xcode et contient des information de débogage pour vous aider à résoudre vos erreurs éventuelles. C'est ce que vous utiliserez lorsque vous développerez votre application.
   - Release : à n'utiliser qu'à la fin. Xcode génère alors l'application définitive, faite pour être partagée et utilisée par d'autres ordinateurs.

5. Ces 2 boutons vous permettent de démarrer l'application directement (Run) ou de la démarrer en mode "Debug" pour exécuter le programme instruction par instruction, afin de résoudre les erreurs. N'utilisez "Debug" que lorsque vous avez des erreurs dans votre programme (ça ne devrait pas être votre cas pour l'instant).
6. La liste des fichiers de votre projet.
7. L'éditeur du code source, comme tout à l'heure.
8. La console de Xcode. C'est là que vous verrez votre programme s'exécuter.
9. Les boutons "Build" et "Run" vous permettent de passer du mode "Compilation" au mode "Exécution". En clair, avec le premier vous pouvez voir ce qui s'est passé pendant la compilation, tandis que dans le second vous pouvez voir ce que votre application a affiché une fois qu'elle a été démarrée.

Ajouter un nouveau fichier

Au début, vous n'aurez qu'un seul fichier source (main.c). Cependant, plus loin dans le cours, je vous demanderai de créer de nouveaux fichiers source lorsque nos programmes deviendront plus gros.

Pour créer un nouveau fichier source sous Xcode, rendez-vous dans le menu "File / New File".
Un assistant vous demande quel type de fichier vous voulez créer. Rendez-vous dans la section "BSD" et sélectionnez "C File" (Fichier C).



Vous devrez donner un nom à votre nouveau fichier (ce que vous voulez). L'extension, elle, doit rester `.c`.
Parfois, nous le verrons plus loin, il faudra aussi créer des fichiers `.h` (mais on en reparlera). La case à cocher "Also create fichier.h" est là pour ça. Pour le moment, elle ne nous intéresse pas.

Cliquez ensuite sur "Finish". C'est fait ! Votre fichier est créé et rajouté à votre projet, en plus de `main.c`

Vous êtes maintenant prêts à programmer sous Mac

Notez que je parlerai probablement d'une instruction `system("PAUSE")` dans les chapitres suivants. Cette instruction ne fonctionne pas sous Mac : il ne faudra donc pas la mettre dans vos codes source.

Nous avons fait le tour dans ce chapitre des IDE les plus connus. N'oubliez pas cependant qu'il en existe d'autres et que rien ne vous empêche de les utiliser si vous les préférez. Quel que soit l'IDE choisi, vous pourrez suivre sans problème la suite du cours.

Je sais par exemple que sous Linux il existe des IDE très bien (Linux n'est pas le système d'exploitation des programmeurs pour rien). Je n'ai malheureusement pas le temps et la place de vous présenter tous les IDE du monde

J'espère en tout cas que ce chapitre vous aura permis de vous familiariser avec votre futur environnement de travail. Regardez-le bien, vous risquez de passer pas mal de temps dessus

Votre premier programme

On a préparé le terrain jusqu'ici, maintenant il serait bien de commencer à programmer un peu qu'en dites-vous ?

C'est justement l'objectif de ce chapitre ! A la fin de celui-ci, vous aurez réussi à créer **votre premier programme** !

Bon d'accord, ce programme sera en noir et blanc et ne saura que vous dire bonjour, il sera donc complètement nul mais... Ce sera votre premier programme et je peux vous assurer que vous en serez fiers

On y va quand vous voulez

Console ou fenêtre ?

Console ou fenêtre ?

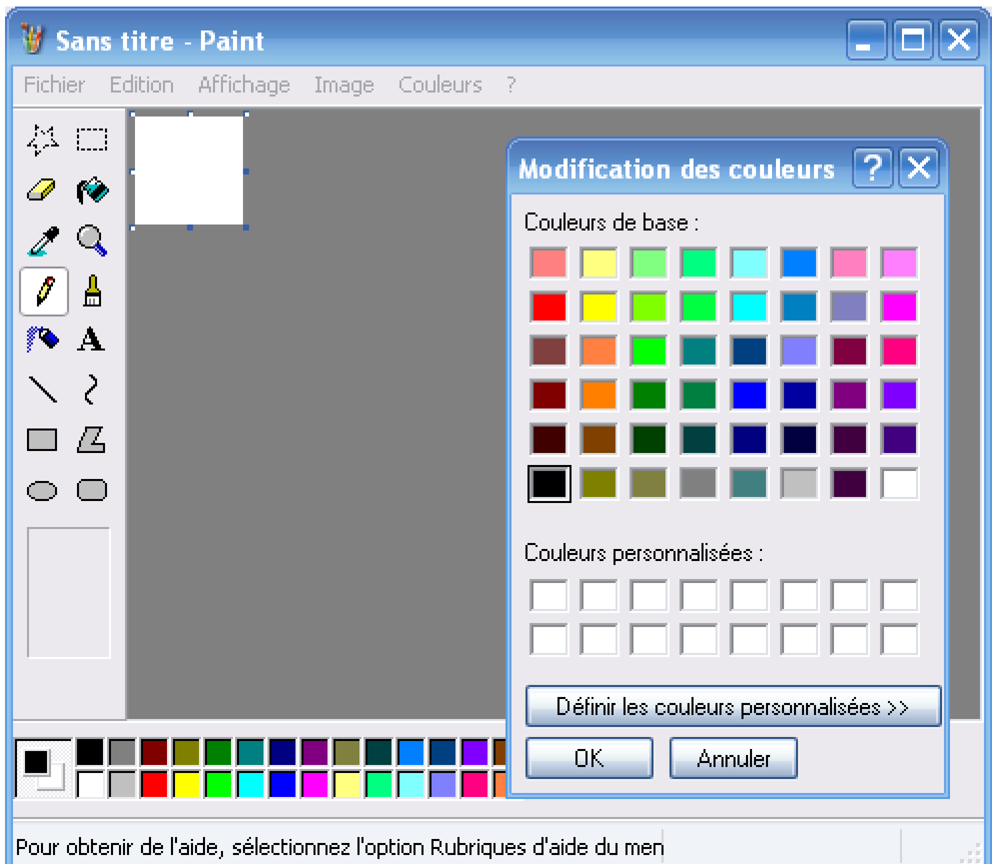
Nous en avons rapidement parlé dans le chapitre précédent. Notre IDE (Dev ou Visual) nous demandait quel type de programme nous voulions créer, et je vous avais dit de répondre *console*.

Il faut savoir qu'en fait il existe 2 types de programmes, pas plus :

- Les programmes avec fenêtres
- Les programmes en console

Les programmes en fenêtres

Ce sont les programmes que vous connaissez.
Voici un exemple de programme en fenêtres que vous connaissez sûrement :



Le programme Paint

Ca donc, c'est un programme avec des fenêtres.
Je suppose que vous aimeriez bien créer ce type de programmes, mmh ? Eh ben vous allez pas pouvoir de suite

En effet, créer des programmes avec des fenêtres en C / C++ c'est possible, mais... Quand on débute, c'est bien trop compliqué !
Pour débuter, il vaut mieux commencer par créer des programmes en console.

Mais au fait, à quoi ça ressemble un programme en console ?

Les programmes en console

Les programmes console ont été les premiers à apparaître. A cette époque, l'ordinateur ne gérait que le noir et blanc et il n'était pas assez puissant pour créer des fenêtres comme on le fait aujourd'hui.

Bien entendu, le temps a passé depuis. Windows a rendu l'ordinateur "grand public" principalement grâce à sa simplicité et au fait qu'il n'utilisait que des fenêtres. Windows est devenu tellement populaire qu'aujourd'hui presque tout le monde a oublié ce qu'était la console !
Oui vous là, ne regardez pas derrière vous, je sais que vous vous demandez ce que c'est

J'ai une grande nouvelle ! **La console n'est pas morte !**
En effet, Linux a remis au goût du jour l'utilisation de la console. Voici une capture d'écran d'une console sous Linux :

```
2.2.5_appli.html        3.2.7.css           3.6.8.html
2.2.5.css               3.2.8.css           3.6.9.html
2.2.6_appli.html        3.2.9_appli.html    ancres.html
2.2.6.css               3.2.9.css           base.php
2.3.10_appli.html       3.3.10.html         cible_formulaire.php
2.3.10.css              3.3.11.css          cible.html
2.3.11_appli.html       3.3.12.css          design1.css
2.3.11.css              3.3.13_appli.html   erreur_paragraphe.html
2.3.12.css              3.3.13.css          essai2.css
2.3.13.html             3.3.14_appli.html   essai.css
2.3.14.css              3.3.14.css          images
2.3.15.html             3.3.15.css          tests_design.html
2.3.16.css              3.3.1.css           traitement.php
2.3.17.css              3.3.2.html
2.3.18.css              3.3.3.css
[root@ns1 exemples]# cd ..
[root@ns1 xhtml-css]# ls
anims                css.php                 images       pseudoformats.php
annexes              design.php              images.php   qcm.php
autres               exemples                index.php    tableaux.php
boites_partiel.php   formatage_partiel.php   intro.php    texte.php
boites_partie2.php   formatage_partie2.php   liens.php    xhtml.php
conclusion.php       formulaires.php         listes.php
[root@ns1 xhtml-css]#
```

Un exemple de console, ici sous Linux

Brrr... Terrifiant hein ?
Voilà, vous avez maintenant une petite idée de ce à quoi ressemble une console

Plusieurs remarques ceci dit :

- Comme vous pouvez le voir, aujourd'hui on sait afficher de la couleur, tout n'est donc pas en noir et blanc ^^
- La console est assez peu accueillante pour un débutant
- C'est pourtant un outil puissant quand on sait le maîtriser

Comme je vous l'ai dit plus haut, créer des programmes en mode "console" comme ici, c'est très facile et idéal pour débuter (ce qui n'est pas le cas des programmes en mode fenêtres).

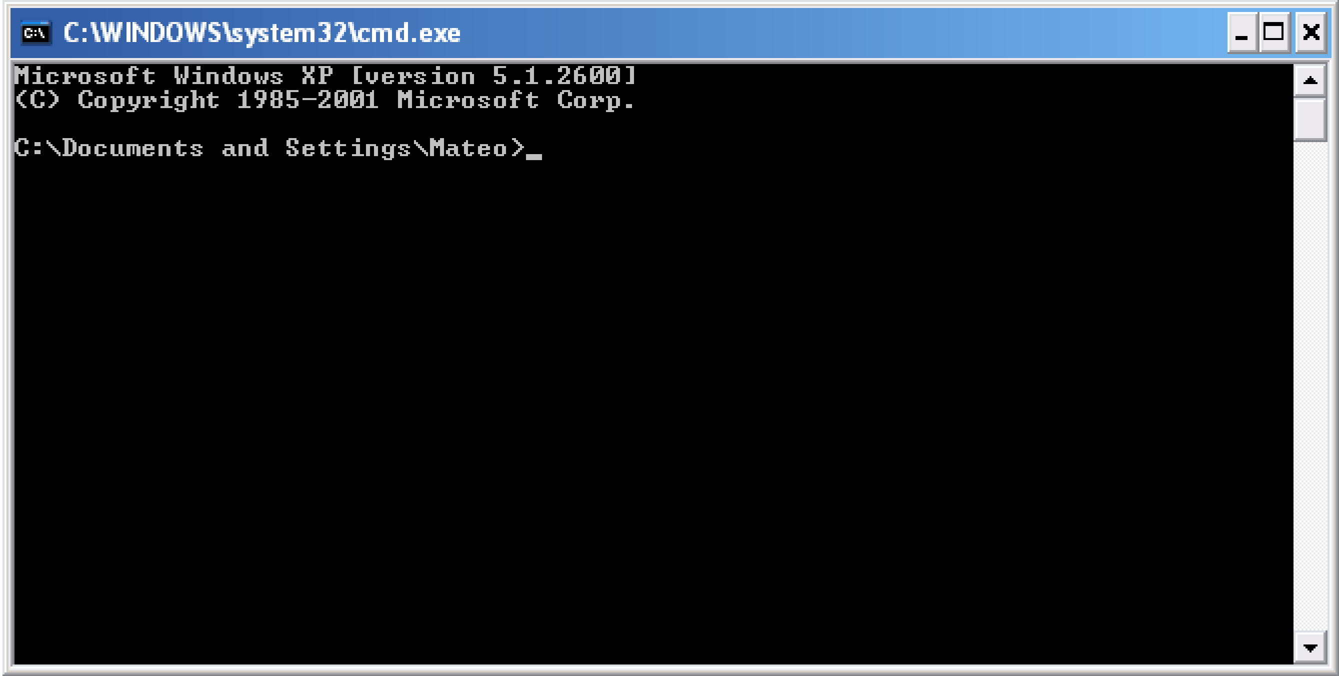
Notez que la console a évolué : elle peut afficher des couleurs, et rien ne vous empêche de mettre une image de fond. Voici une autre capture d'écran de console Linux honteusement pompée sur Internet

La console, ça peut aussi être joli

Et sous Windows ? Y'a pas de console ?

Si, mais elle est un peu... "cachée" on va dire
Vous pouvez avoir une console en faisant "Démarrer / Accessoires / Invite de commandes", ou bien encore en faisant "Démarrer / Exécuter", et en tapant ensuite "cmd".

Et voici la maaagnifique console de Windows :



La console de Windows

Si vous êtes sous Windows, sachez donc que c'est dans une fenêtre qui ressemble à ça que nous ferons nos premiers programmes. Si j'ai choisi de commencer par des petits programmes en console, ce n'est pas pour vous ennuyer, bien au contraire ! En commençant par faire des programmes en console, vous apprendrez les bases nécessaires pour ensuite pouvoir créer des fenêtres.

Soyez donc rassurés : dès que nous aurons le niveau pour créer des fenêtres, nous verrons comment en faire

UN MINIMUM DE CODE

Pour n'importe quel programme, il faudra taper un minimum de code. Ce code ne fera rien de particulier, mais il est indispensable.
C'est ce "code minimum" que nous allons découvrir maintenant. Il devrait servir de base pour la plupart de vos programmes en langage C.

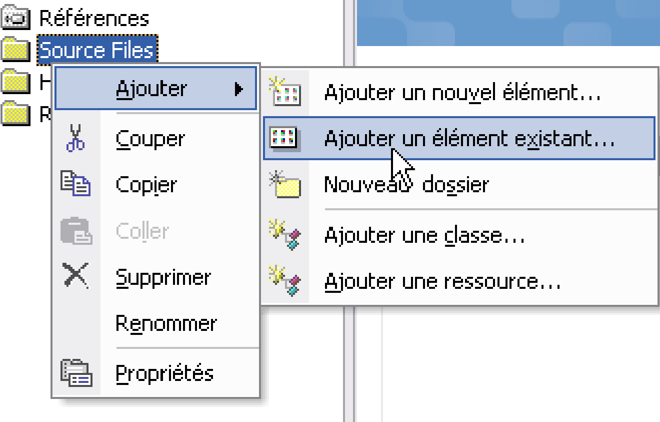
Ah oui, je le reprécise quand même au cas où : nous allons maintenant apprendre **le langage C**, comme je vous l'ai dit plus tôt. Tout ce que je vais vous apprendre maintenant, vous le réutiliserez lorsque nous verrons le C++, donc vous avez intérêt à être attentifs tout le temps

Demandez le code minimal à votre IDE

Selon l'IDE que vous avez choisi dans le chapitre précédent, la méthode pour créer un nouveau projet n'est pas la même. Reportez-vous à ce chapitre précédent si vous avez oublié comment faire.

- **Sous Dev-C++** : demandez une *console application* que vous appellerez "bonjour". Dev va vous créer le code minimal, vous n'avez rien de plus à faire.
- **Sous Visual C++** : demandez un *projet console Win32* vide que vous appellerez "bonjour".
  Ajoutez un nouveau fichier à votre projet de type "*Fichier C++ (.cpp)*". Demandez de suite à enregistrer votre fichier dans le répertoire de votre projet sous le nom main.c (et non main.cpp, car l'extension .cpp est plutôt utilisée pour le C++)

  Puis, dans l'onglet "Explorateur de solutions" faites un clic droit sur le dossier "Source Files" et cliquez choisissez "Ajouter / Ajouter un élément existant". On vous demandera d'indiquer des fichiers : vous devrez sélectionner le fichier main.c que vous venez d'enregistrer.



Ajouter un fichier à un projet sous Visual

Démonstration en images (ou plutôt en vidéo) :

Ajouter le fichier main.c au projet (200 Ko)

Bon pour cette fois, la procédure à suivre aura été bien plus simple avec Dev, mais ça ne veut pas dire que Visual est plus compliqué à utiliser hein C'est juste pour la première fois.

Dev a donc généré le minimum de code en langage C dont on a besoin. Le voici :

Code : C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

  system("PAUSE");
  return 0;
}
```

Si vous êtes sous Visual, copiez-collez ce code source dans votre fichier main.c qui est pour l'instant vide.

Enregistrez le tout. Oui je sais, on n'a encore rien fait, mais enregistrez quand même, c'est une bonne habitude à prendre
Normalement, vous n'avez qu'un seul fichier source appelé *main.c* (le reste c'est des fichiers de projet générés par votre IDE).

Analysons le code minimal

Ce code minimal qu'on vient de voir n'est, j'imagine, rien que du charabia pour vous. Et pourtant, moi je vois là un programme console qui s'affiche, qui se met en pause et qui s'éteint.
Il va falloir apprendre à lire tout ça

Commençons par les 2 premières lignes qui se ressemblent beaucoup :

Code : C

```
#include <stdio.h>
#include <stdlib.h>
```

Ce sont des lignes spéciales que l'on ne voit qu'en haut des fichiers source. Ces lignes sont facilement

reconnaissables car elles commencent par un dièse #. Ces lignes spéciales, on les appelle **directives de préprocesseur** (un nom compliqué n'est-ce pas ?). Ce sont des lignes qui seront lues par un programme appelé préprocesseur, un programme qui se lance au début de la compilation.

Oui, comme je vous l'ai dit plus tôt, ce qu'on a vu au début n'était qu'un schéma très simplifié de la compilation. Il se passe en réalité plusieurs choses pendant une compilation. On les détaillera plus tard, pour le moment vous avez juste besoin de mettre ces lignes en haut de chacun de vos fichiers.

Oui mais elles signifient quoi ces lignes ? J'aimerais bien savoir quand même !

Le mot "include" en anglais signifie "inclure" en français. Ces lignes demandent d'inclure des fichiers au projet, c'est-à-dire d'ajouter des fichiers pour la compilation.
Il y a 2 lignes, donc 2 fichiers inclus. Ces fichiers s'appellent *stdio.h* et *stdlib.h*. Ce sont des fichiers qui existent déjà, des fichiers sources tout prêts. On verra plus tard qu'on les appelle des **librairies** (ou aussi **bibliothèques**). En gros, ces fichiers contiennent du code tout prêt qui permet d'afficher du texte à l'écran.

A noter : le mot anglais est "library" et il se traduit par "bibliothèque". "Librairie" est donc un faux-ami.
En théorie, on devrait donc dire bibliothèque et non librairie. Mais pour ma part, j'ai pris l'habitude d'écrire librairie (et je ne suis pas le seul) donc je continuerai à utiliser ce terme. Retenez quand même que la traduction exacte est plutôt "bibliothèque".

Sans ces fichiers, écrire du texte à l'écran aurait été mission impossible. L'ordinateur à la base ne sait rien faire, il faut tout lui dire. Vous voyez la galère dans laquelle on est

Bref, les 2 premières lignes incluent les librairies qui vont nous permettre (entre autres) d'afficher du texte à l'écran assez "facilement"

Passons à la suite. La suite, c'est tout ça :

Code : C

```c
int main(int argc, char *argv[])
{

  system("PAUSE");
  return 0;
}
```

Ce que vous voyez là, c'est ce qu'on appelle **une fonction**. Un programme en langage C est constitué de fonctions, il ne contient quasiment que ça. Pour le moment, notre programme ne contient donc qu'une seule fonction.

Une fonction permet grosso modo de rassembler plusieurs commandes à l'ordinateur. Regroupées dans une fonction, les commandes permettent de faire quelque chose de précis. Par exemple, on peut créer une fonction "ouvrir_fichier" qui contiendra une suite d'instructions pour l'ordinateur lui expliquant comment ouvrir un fichier. L'avantage, c'est qu'une fois la fonction écrite, vous n'aurez plus qu'à dire "ouvrir_fichier", et votre ordinateur saura comment faire sans que vous ayez à tout répéter !
(c'est beau la technologie !)

Sans rentrer dans les détails de la construction d'une fonction (il est trop tôt, on reparlera des fonctions plus tard), analysons quand même ses grandes parties. La première ligne contient le nom de la fonction, c'est le deuxième mot.

Oui notre fonction s'appelle donc *main*. C'est un nom de fonction particulier qui signifie "principal". Main est la fonction principale de votre programme, c'est toujours par la fonction main que le programme commence.

Une fonction a un début et une fin, délimités par des accolades { et }. Toute la fonction main se trouve donc entre ces accolades. Si vous avez bien suivi, notre fonction main contient 2 lignes :

Code : C

```c
system("PAUSE");
return 0;
```

Ces lignes à l'intérieur d'une fonction ont un nom. On les appelle **instructions** (ça en fait du vocabulaire qu'il va falloir retenir).
Chaque instruction est une commande à l'ordinateur. Chacune de ces lignes demande à l'ordinateur de faire quelque chose de précis.

Comme je vous l'ai dit un peu plus haut, en regroupant intelligemment (c'est le travail du programmeur) les instructions dans des fonctions, on crée si on veut des "*bouts de programmes tout prêts*". En utilisant les bonnes instructions, rien ne nous empêcherait donc de créer une fonction "ouvrir_fichier" comme je vous l'ai expliqué tout à l'heure, ou encore une fonction "avancer_personnage" dans un jeu vidéo par exemple

Un programme, ce n'est en fait au bout du compte rien d'autre qu'une série d'instructions : "fais ceci" "fais cela". Vous donnez des ordres à votre ordinateur et il les exécute (du moins si vous l'avez bien dressé)

TRES IMPORTANT : toute instruction se termine O-BLI-GA-TOI-RE-MENT par un point-virgule " ; ". C'est d'ailleurs comme ça qu'on reconnaît ce qui est une instruction et ce qui n'en est pas une. Si vous oubliez de mettre un point-virgule à la fin d'une instruction, votre programme ne compilera pas !

La première ligne :
`system("PAUSE");`
demande à l'ordinateur de mettre en pause le programme (c'est fou, on l'aurait presque deviné tout seul).
Quand votre programme arrivera à cette ligne, il va afficher un message à l'écran : " *Appuyez sur une touche pour continuer* " et va attendre que vous appuyiez sur n'importe quelle touche avant de passer à l'instruction suivante.

L'instruction system("PAUSE") ne fonctionne que sous Windows. Elle est en fait utilisée par Dev C++ pour mettre en pause le programme juste avant qu'il ne se termine.

Les IDE un peu plus intelligents, comme Code::Blocks, rajoutent une instruction du même type automatiquement à la compilation. Dans ce cas, l'instruction `system("PAUSE");` est inutile : vous pouvez simplement l'enlever. Vu qu'au début de ce cours je travaillerai principalement sous Dev-C++, vous devriez voir souvent cette instruction. Plus loin dans le cours j'évoluerai vers Code::Blocks et Visual C++ qui sont des IDE un peu plus poussés.
Si vous êtes sous un autre système d'exploitation que Windows (Linux ou Mac OS), vous lancerez votre programme directement depuis la console et n'aurez pas besoin d'utiliser cette instruction. Si vous tenez à mettre en pause votre programme avant la fin, vous pouvez remplacer l'instruction `system("PAUSE");` par l'instruction `getchar();` (qu'il faudra peut-être écrire 2 fois pour que ça marche)

Passons à l'instruction suivante :
`return 0;`
Bon ben ça en gros, ça veut dire que c'est fini (eh oui déjà ^^). Cette ligne indique qu'on arrive à la fin de notre fonction *main* et demande de renvoyer la valeur 0.

Hein ? Pourquoi mon programme renverrait-il le nombre 0 ?

En fait, chaque programme une fois terminé renvoie une valeur, par exemple pour dire que tout s'est bien passé (*0 = tout s'est bien passé, n'importe quelle autre valeur = erreur*). La plupart du temps, cette valeur n'est pas vraiment utilisée, mais il faut quand même en renvoyer une.
Votre programme aurait marché sans le return 0, mais on va dire que c'est plus propre et plus sérieux de le mettre, donc on le met

Et voilà ! On vient de détailler un peu le fonctionnement du code minimal.

Certes, on n'a pas vraiment tout vu en profondeur, et vous devez avoir quelques questions en suspens. Soyez rassurés : toutes vos questions trouveront une réponse petit à petit. Je ne peux pas tout vous divulguer d'un coup, sinon c'est l'embrouille assurée

D'ailleurs, en parlant d'embrouille, ça va vous suivez toujours ?
Si tel n'est pas le cas, rien ne presse. Ne vous forcez pas à lire la suite. Faites une pause et relisez ce début de chapitre à tête reposée. Tout ce que je viens de vous apprendre est fondamental, surtout si vous voulez être sûrs de pouvoir suivre après

Tenez, d'ailleurs comme je suis de bonne humeur je vous fais un schéma qui récapitule le vocabulaire qu'on vient d'apprendre

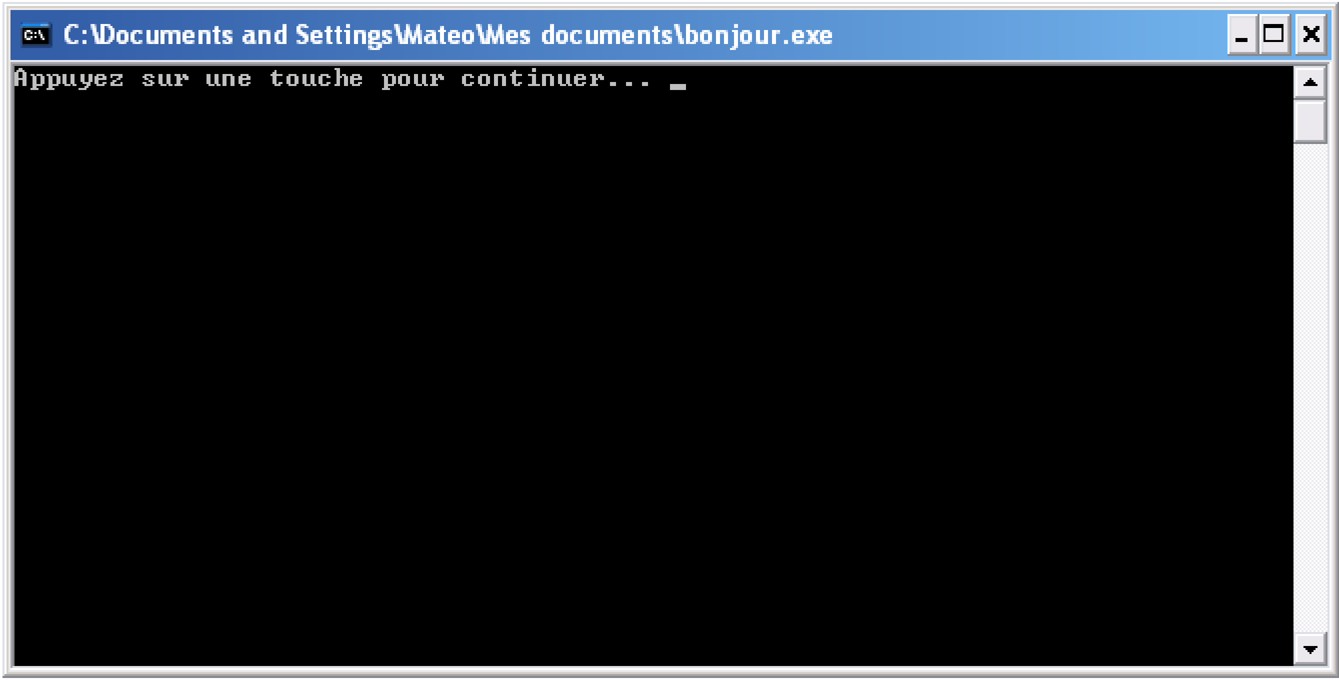
```
#include <stdio.h>    } Directives de préprocesseur
#include <stdlib.h>

int main(int argc, char *argv[])   }
{                                   }
                                    } Fonction
  system("PAUSE");  } Instructions  }
  return 0;                         }
}
```

Le vocabulaire du programme minimal

Testons notre programme

Tester devrait aller vite. Tout ce que vous avez à faire c'est compiler le projet, puis l'exécuter (cliquez sur "Compiler & Exécuter" sous Dev).
Si vous ne l'avez pas encore fait, on vous demandera d'enregistrer les fichiers. Faites-le.

Après un temps d'attente insupportable (la compilation), votre premier programme va apparaître sous vos yeux totalement envahis de bonheur



Votre premier programme !

Comme indiqué à l'écran, appuyez sur une touche. Votre programme s'arrête alors.

Oui je sais c'est nul, c'est moche, c'est tout ce que vous voulez
Mais bon, quand même ! C'est un premier programme, un instant dont vous vous souviendrez toute votre vie

... Non ?
...

Bon, avant que vous me fassiez déjà une première déprime, je propose qu'on passe à la suite sans plus tarder

Ecrire un message à l'écran

A partir de maintenant, on va ajouter nous-mêmes du code dans ce programme minimal.
Votre mission, si vous l'acceptez : afficher le message "Bonjour" à l'écran.

Comme tout à l'heure, une console doit s'ouvrir. Le message "Bonjour" doit s'afficher dans la console.

Comment fait-on pour écrire du texte dans la console ?

On va devoir rajouter une ligne dans la fonction main. Vous vous rappelez que ces lignes ont un nom particulier n'est-ce pas ?
On les appelle **des instructions**. On va donc rajouter l'instruction qui commande à l'ordinateur : "*Affiche-moi un message à l'écran*"

Cette instruction a un nom, elle s'appelle *printf* (retenez-le !).
En fait, printf est une fonction déjà écrite par d'autres programmeurs avant vous.

Cette fonction, où se trouve-t-elle ? Moi je ne vois que la fonction *main* !

Vous vous souvenez de ces 2 lignes ?

```
#include <stdio.h>
#include <stdlib.h>
```

Je vous avais dit qu'elles permettaient d'ajouter des librairies dans votre programme.
Les librairies sont en fait des fichiers avec pleins de fonctions toutes prêtes à l'intérieur. Ces fichiers-là (stdio.h et stdlib.h) contiennent la plupart des fonctions de base dont on a besoin dans un programme. stdio.h en particulier contient des fonctions permettant d'afficher des choses à l'écran (comme *printf*) mais aussi de demander à l'utilisateur de taper quelque chose (ce sont des fonctions que l'on verra plus tard).

Dis Bonjour au Monsieur

Dans notre fonction *main*, on va faire appel à la fonction *printf*.
C'est une fonction qui en appelle une autre (ici, main appelle *printf*). Vous allez voir que c'est tout le temps comme ça que ça se passe en langage C

Donc, pour faire appel à une fonction, c'est très simple : il suffit d'écrire son nom. Ecrivez donc *printf* sur une ligne au tout début de la fonction main (avant le `system("PAUSE")`).

C'est bien, mais on n'est pas encore tirés d'affaire Il faut indiquer quoi écrire à l'écran. Pour faire ça, il va falloir donner à la fonction *printf* le texte à afficher. Pour ce faire, ouvrez des parenthèses après le mot printf. Puis, ouvrez des guillemets à l'intérieur des parenthèses. Enfin, tapez le texte à afficher entre les guillemets.
Dans notre cas, on va donc taper très exactement :

```
printf("Bonjour");
```

J'espère que vous n'avez pas oublié le point-virgule à la fin, je vous rappelle que c'est très important ! Cela permet d'indiquer que l'instruction s'arrête là.

Voici le code source que vous devriez avoir sous les yeux :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("Bonjour");
  system("PAUSE");
  return 0;
}
```

On a donc 3 instructions qui commandent dans l'ordre à l'ordinateur :

1. Affiche "Bonjour" à l'écran.
2. Met le programme en pause, affiche le message "Appuyez sur une touche pour continuer" et attend qu'on appuie sur une touche avant de passer à l'instruction suivante.
3. La fonction *main* est terminée, renvoie 0. Le programme s'arrête alors.

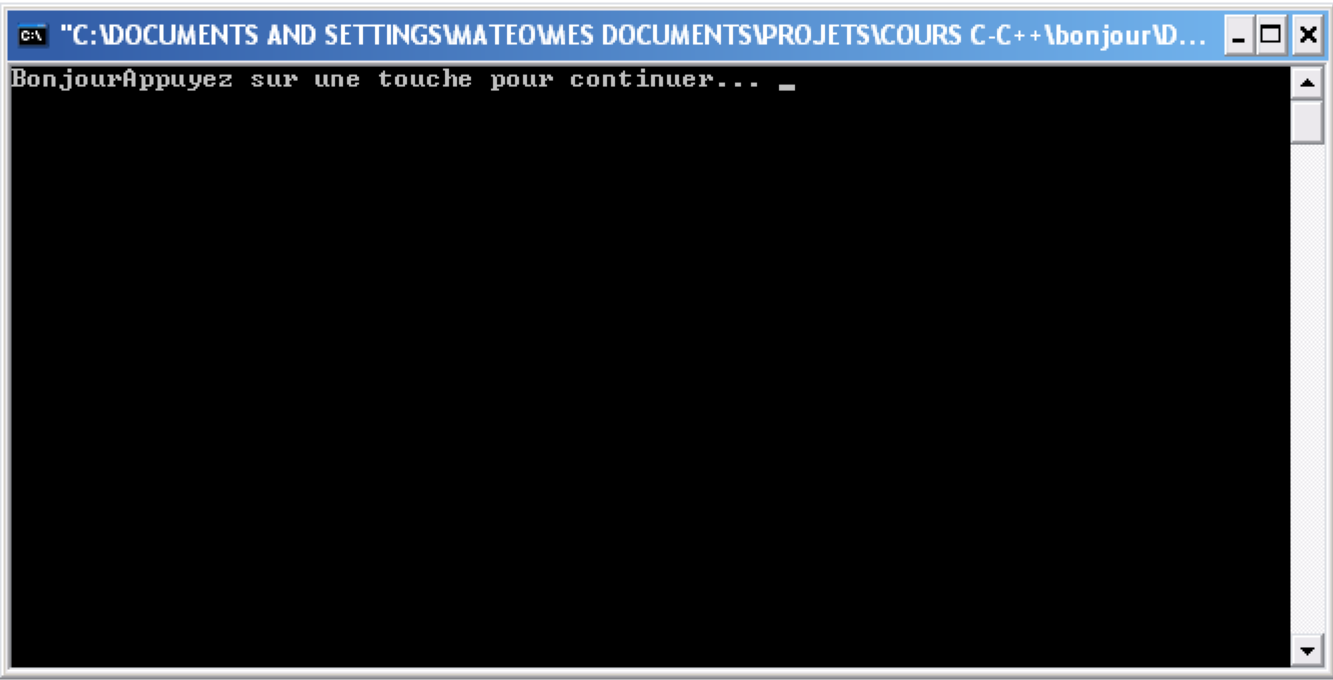
A quoi ça sert de mettre le programme en pause ? On ne pourrait pas enlever l'instruction `system("PAUSE")` ?

Si, bien sûr qu'on pourrait Testez sans cette instruction et vous verrez.
Le programme ne se met pas en pause. En clair, il affiche le message "Bonjour" et puis s'arrête. Du coup, la fenêtre de la console apparaît et disparaît à la vitesse de l'éclair, vous n'avez pas le temps de lire ce qui est écrit à l'écran.

Stupide, isn't it ?

Notez qu'avec certains IDE, comme je vous l'ai dit plus tôt, il se peut que la pause soit faite automatiquement. Dans ce cas, l'instruction `system("PAUSE")` est inutile et vous pouvez l'enlever

On va donc tester le programme avec une pause, ce qui devrait nous afficher :



Un programme poli qui dit Bonjour... Enfin presque

Allons allons, ce n'est pas bien grave, on va apprendre à corriger ça tout de suite
Une des solutions pour rendre notre programme plus présentable serait de faire un retour à la ligne après "Bonjour" (comme si on appuyait sur la touche "Entrée" quoi)

Mais bien sûr, ce serait trop simple de taper "Entrée" dans notre code source pour qu'une entrée soit effectuée à l'écran ! Il va falloir utiliser ce qu'on appelle des caractères spéciaux...

Les caractères spéciaux

Les caractères spéciaux sont des lettres spéciales qui permettent d'indiquer qu'on veut aller à la ligne, faire une tabulation etc...
Les caractères spéciaux sont faciles à reconnaître : c'est un ensemble de 2 caractères. Le premier d'entre eux est toujours un anti-slash (\ ), et le second un nombre ou une lettre. Voici 2 caractères spéciaux courants que vous aurez probablement besoin d'utiliser, ainsi que leur signification :

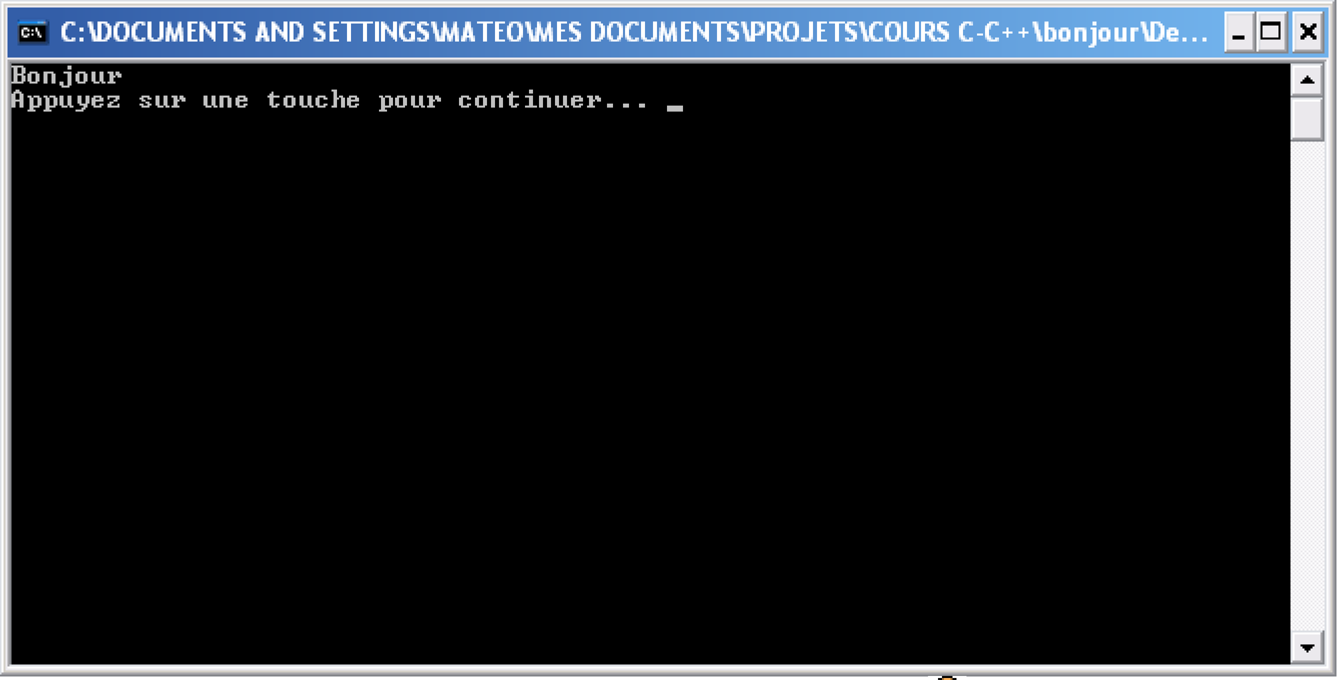
- **\n** : retour à la ligne (= "Entrée")
- **\t** : tabulation

Dans notre cas, pour faire une entrée, il suffit de taper \n pour créer un retour à la ligne.
Si je veux donc faire un retour à la ligne juste après le mot Bonjour, je devrai taper :

```
printf("Bonjour\n");
```

Votre ordinateur comprend qu'il doit afficher "Bonjour" suivi d'un retour à la ligne.

Votre programme va maintenant avoir une tête un peu plus présentable



Ah, voilà un Bonjour un peu plus présentable !

C'est mieux quand même

> Vous pouvez écrire à la suite du \n sans aucun problème. Tout ce que vous écrirez à la suite du \n sera placé sur la deuxième ligne. Vous pourriez donc vous entraîner à écrire :
> `printf("Bonjour\nAu Revoir\n");`
> Cela affichera "Bonjour" sur la première ligne et "Au revoir" sur la ligne suivante.

Le syndrome de Gérard

> Bonjour, je m'appelle Gérard et j'ai voulu essayer de modifier votre programme pour qu'il me dise "Bonjour Gérard". Seulement voilà, j'ai l'impression que l'accent de Gérard ne s'affiche pas correctement... Que faire ?

Tout d'abord, bonjour Gérard
C'est une question très intéressante que vous nous posez là. Je tiens en premier lieu à vous féliciter pour votre esprit d'initiative, c'est très bien d'avoir eu l'idée de modifier un peu le programme. C'est en "bidouillant" les programmes que je vous donne que vous allez en apprendre le plus. Ne vous contentez pas de ce que vous lisez, essayez un peu vos propres modifications des programmes que nous voyons ensemble !

Bien, maintenant pour répondre à la question de notre ami Gérard, j'ai une bien triste nouvelle à vous annoncer : la console de Windows ne gère pas les accents
Par contre la console de Linux oui

A partir de là vous avez 2 solutions :

- **Passer à Linux**. C'est une solution un peu radicale et il me faudrait tout un cours entier pour vous expliquer comment vous servir de Linux. Si vous n'avez pas le niveau, oubliez cette possibilité pour le moment
- **Ne pas utiliser d'accents**. C'est malheureusement la solution que vous risquez de choisir. La console de

Windows a ses défauts que voulez-vous. Il va vous falloir prendre l'habitude d'écrire sans accents. Bien entendu, comme plus tard vous ferez probablement des programmes avec des fenêtres, vous ne devriez pas avoir ce problème-là

Vous devrez donc écrire :

```
printf("Bonjour Gerard\n");
```

On remercie notre ami Gérard pour nous avoir soulevé ce problème

ps : si d'aventure vous vous appeliez Gérard, sachez que je n'ai rien contre ce prénom C'est simplement le premier prénom avec un accent qui m'est passé par la tête
Et puis bon, il faut toujours que quelqu'un prenne pour les autres, que voulez-vous

Les commentaires, c'est très utile !

Avant de terminer ce premier chapitre de "véritable" programmation, je dois absolument vous montrer un truc génial qu'on appelle **les commentaires**. Quel que soit le langage de programmation, on a la possibilité d'ajouter des commentaires à son code. Le langage C n'échappe pas à la règle.

Qu'est-ce que ça veut dire "commenter" ?
Cela signifie : taper du texte au milieu de votre programme pour indiquer ce qu'il fait, à quoi sert telle ligne de code etc. C'est vraiment quelque chose d'indispensable car, même en étant un génie de la programmation, on a besoin de faire quelques annotations par-ci par-là. Cela permet :

- De vous retrouver au milieu d'un de vos codes sources plus tard. On ne dirait pas comme ça, mais on oublie vite comment fonctionnent les programmes qu'on a écrit Si vous faites une pause ne serait-ce que de quelques jours, vous aurez besoin de vous aider de vos propres commentaires pour vous retrouver dans un gros code.
- Si vous donnez votre projet à quelqu'un d'autre (qui ne connaît pas à priori votre code source), cela lui permettra de se familiariser avec bien plus rapidement.
- Enfin, ça va me permettre à moi de rajouter des annotations dans les codes sources de ce cours. Cela me permettra de mieux vous expliquer à quoi peut servir telle ou telle ligne de code.

Il y a plusieurs manières de rajouter un commentaire. Tout dépend de la longueur du commentaire que vous voulez écrire :

- Votre commentaire est **court** : il tient sur une seule ligne, il ne fait que quelques mots. Dans ce cas, vous devez taper un double slash (//) suivi de votre commentaire. Par exemple :

  Code : C
  ```c
  // Ceci est un commentaire
  ```

  Vous pouvez aussi bien écrire un commentaire seul sur sa ligne, ou bien à droite d'une instruction. C'est d'ailleurs quelque chose de très pratique car ainsi on sait que le commentaire sert à indiquer à quoi sert la ligne sur laquelle il est. Exemple :

  Code : C
  ```c
  printf("Bonjour"); // Cette instruction affiche Bonjour à l'écran
  ```

  Notez que ce type de commentaire a normalement été introduit par le langage C++, mais vous n'aurez pas de problème en l'utilisant pour un programme en langage C (sauf si vous êtes un puriste)

- Votre commentaire est **long** : vous avez plein de choses à raconter, vous avez besoin d'écrire plusieurs phrases

qui tiennent sur plusieurs lignes. Dans ce cas, vous devez taper un code qui signifie "début de commentaire" et un autre code qui signifie "fin de commentaire" :

- Pour indiquer le *début du commentaire* : tapez un slash suivi d'une étoile (/*)
- Pour indiquer la *fin du commentaire* : tapez une étoile suivie d'un slash (*/)

Vous écrirez donc par exemple :

Code : C

```
/* Ceci est
Un commentaire
Sur plusieurs lignes */
```

Reprenons notre code source qui écrit "Bonjour", et ajoutons-lui quelques commentaires juste pour s'entraîner :

Code : C

```
/*
Ci-dessous, ce sont des directives de préprocesseur.
Ces lignes permettent d'ajouter des fichiers au projet, fichiers que l'on appelle
"librairies".
Grâce à ces librairies, on disposera de fonctions toutes prêtes pour afficher par exemple
un message à l'écran
*/

#include <stdio.h>
#include <stdlib.h>

/*
Ci-dessous, vous avez la fonction principale du programme, appelée "main". C'est par
cette fonction que tous les programmes commencent.
Ici, ma fonction se contente d'afficher "Bonjour" à l'écran, met en pause le programme
puis s'arrête
*/

int main(int argc, char *argv[])
{
  printf("Bonjour"); // Cette instruction affiche Bonjour à l'écran
  system("PAUSE");   // Le programme se met en pause
  return 0;          // Le programme renvoie le nombre 0 puis s'arrête
}
```

Voilà ce que donnerait notre programme avec quelques commentaires
Oui, il a l'air d'être plus gros, mais en fait c'est le même que tout à l'heure. Lors de la compilation, tous les commentaires seront ignorés. Ces commentaires n'apparaîtront pas dans le programme final, ils servent seulement aux programmeurs.

Normalement, on ne commente pas chaque ligne du programme. J'ai dit (et je le redirai) que c'était important de mettre des commentaires dans un code source, mais il faut savoir doser : commenter chaque ligne ne servira la plupart du temps à rien. A force, vous saurez que le *printf* permet d'afficher un message à l'écran, pas besoin de l'indiquer à chaque fois

Le mieux est de commenter plusieurs lignes à la fois, c'est-à-dire d'indiquer à quoi sert une série d'instructions histoire d'avoir une idée. Après, si le programmeur veut se pencher plus en détail dans ces instructions, il est assez intelligent pour y arriver tout seul.

Retenez donc : les commentaires doivent guider le programmeur dans son code source, lui permettre de se repérer. Essayez de commenter un ensemble de lignes plutôt que toutes les lignes une par une.

Et pour finir sur une petite touche culturelle, voici une citation tirée de chez IBM :

Citation : Règle de la maison IBM

> Si après avoir lu uniquement les commentaires d'un programme vous n'en comprenez pas le fonctionnement, jetez le tout !

Comme vous pouvez le constater, on n'a pas chômé dans ce chapitre. C'est la première fois que nous voyons du "vrai" code source de "vraie" programmation, et toutes ces lettres et ces symboles doivent vous faire tourner un peu la tête...
C'est normal, ça fait toujours ça la première fois

Plutôt que de foncer tête baissée sur la suite, je vous invite à prendre votre temps : relisez ce chapitre, faites quelques tests avec ce que vous savez déjà. Je préfère éviter que vous appreniez trop de nouvelles choses à la fois, tout simplement parce que vous ne retiendrez rien si vous allez trop vite.

Et puis, je ne veux pas briser le suspense, mais je tiens à vous avertir que les chapitres qui vont suivre seront tous aussi riches en nouveautés

Un monde de variables

Nous entrons maintenant dans un chapitre Ô combien important pour la suite, un chapitre à ne négliger sous aucun prétexte (en d'autres termes, c'est pas le moment de regarder les mouches voler)

Résumé des épisodes précédents :
Vous avez appris dans le chapitre précédent comment faire pour créer un nouveau projet en console avec votre IDE (Dev-C++, Visual C++ ou un autre). Je vous ai notamment expliqué qu'il était trop compliqué, pour un débutant, de commencer par réaliser des fenêtres graphiques (et je ne vous parle même pas de créer un super jeu vidéo 3D en réseau). Nous allons donc, pour nos débuts, travailler dans une console faisant penser à DOS. Dès que nous aurons le niveau bien entendu, on verra comment faire des choses plus intéressantes.

Vous savez donc afficher un texte à l'écran. Super.
Je sais, vous allez me dire que ça ne vole pas très haut pour le moment, mais c'est justement parce que vous ne connaissez pas encore ce qu'on appelle **les variables** en programmation.

Ah les variables, parlons-en ! C'est quelque chose d'incontournable, quel que soit votre langage de programmation. Le langage C n'échappe pas à la règle.

J'ai tout ce chapitre pour vous l'expliquer. Je ne veux pas gâcher le suspens, mais sachez qu'en gros on va apprendre à faire retenir des nombres à l'ordinateur. On va apprendre à stocker des nombres dans la mémoire.

Je souhaite que nous commencions par quelques explications sur la mémoire de votre ordinateur. Comment fonctionne une mémoire ? Combien un ordinateur possède-t-il de mémoires différentes ?

Ca pourra paraître un peu simpliste à certains d'entre vous, mais dites-vous bien qu'il y en a peut-être ici qui ne savent pas ce qu'est une mémoire

Une affaire de mémoire

Ce que je vais vous apprendre dans ce chapitre a donc un rapport direct avec la mémoire de votre ordinateur.

Tout être humain normalement constitué a une mémoire. Eh bien c'est pareil pour un ordinateur... à un détail près : un ordinateur a plusieurs types de mémoire !

Pourquoi un ordinateur aurait-il plusieurs types de mémoire ? Une seule mémoire aurait suffit, non ?

Non, en fait le problème c'est qu'on a besoin d'avoir une mémoire à la fois **rapide** (pour récupérer une information très vite) et **importante** (pour stocker beaucoup de choses). Or, vous allez rire, mais jusqu'ici nous avons été infichus de créer une mémoire qui soit à la fois très rapide et importante. Plus exactement, la mémoire rapide coûte cher, donc on n'en fait qu'en petites quantités.
Du coup, pour nous arranger, nous avons dû doter les ordinateurs de mémoires très rapides mais pas importantes, et de mémoires importantes mais pas très rapides (vous suivez toujours ?)

Les différents types de mémoire

Pour vous donner une idée, voici les différents types de mémoire existant dans un ordinateur, de la plus rapide à la plus lente :

1. Les registres : une mémoire ultrarapide située directement dans le processeur.
2. La mémoire cache : elle fait le lien entre les registres et la mémoire vive.
3. La mémoire vive : c'est la mémoire avec laquelle nous allons travailler le plus souvent.
4. Le disque dur : que vous connaissez sûrement, c'est là qu'on enregistre les fichiers.

Comme je vous l'ai dit, j'ai classé les mémoires de la plus rapide (les registres) à la plus lente (le disque dur). Si vous avez bien suivi, vous avez compris aussi que la mémoire la plus rapide était la plus petite, et la plus lente la plus grosse.
Les registres sont donc à peine capables de retenir quelques nombres, tandis que le disque dur peut stocker de très gros fichiers.

Quand je dis qu'une mémoire est "lente", c'est à l'échelle de votre ordinateur bien sûr. Eh oui, pour un ordinateur 8 millisecondes pour accéder au disque dur c'est déjà trop long !

Que faut-il retenir dans tout ça ?
En fait, c'est pour vous situer un peu. Vous savez désormais qu'en programmation, on va surtout travailler avec la mémoire vive. On verra aussi comment lire et écrire sur le disque dur, pour lire et créer des fichiers (mais on ne le fera que plus tard). Quant à la mémoire cache et aux registres, on n'y touchera pas du tout ! C'est votre ordinateur qui s'en occupe.

Dans des langages très bas niveau, comme l'assembleur (abrégé "ASM"), on travaille au contraire plutôt directement avec les registres. Je l'ai fait, et je peux vous dire que faire une simple multiplication dans ce langage est un véritable parcours du combattant ! Heureusement, en langage C (et dans la plupart des autres langages de programmation), c'est beaucoup plus facile.

Il faut ajouter une dernière chose très importante : seul le disque dur retient tout le temps les informations qu'il contient. Toutes les autres mémoires (registres, mémoire cache, mémoire vive) sont des mémoires temporaires : lorsque vous éteignez votre ordinateur ces mémoires se vident !

Heureusement, lorsque vous rallumerez l'ordinateur, votre disque dur sera toujours là pour rappeler à votre ordinateur qui il est

La mémoire vive en photos

Vu qu'on va travailler pendant un moment avec la mémoire vive, je pense qu'il serait bien de vous la présenter

On va y aller par zooms successifs. Ca, c'est votre ordinateur :

Vous reconnaissez le clavier, la souris, l'écran et l'unité centrale (la tour).
Intéressons-nous maintenant à l'unité centrale, le cœur de votre ordinateur qui contient toutes les mémoires :

Ce qui nous intéresse, c'est ce qu'il y a à l'intérieur de l'unité centrale, si on l'ouvre :

C'est un joyeux petit bazar
Rassurez-vous, je ne vous demanderai pas de savoir comment tout cela fonctionne. Je veux juste que vous sachiez où se trouve la mémoire vive là-dedans. Je vous l'ai encadrée en rouge.
Je n'ai pas indiqué les autres mémoires (registres et mémoire cache) car de toute façon elles sont bien trop petites pour être visibles à l'oeil nu

Voici à quoi ressemble une barrette de mémoire vive de plus près :

Cliquez sur l'image si vous voulez voir en plus grand

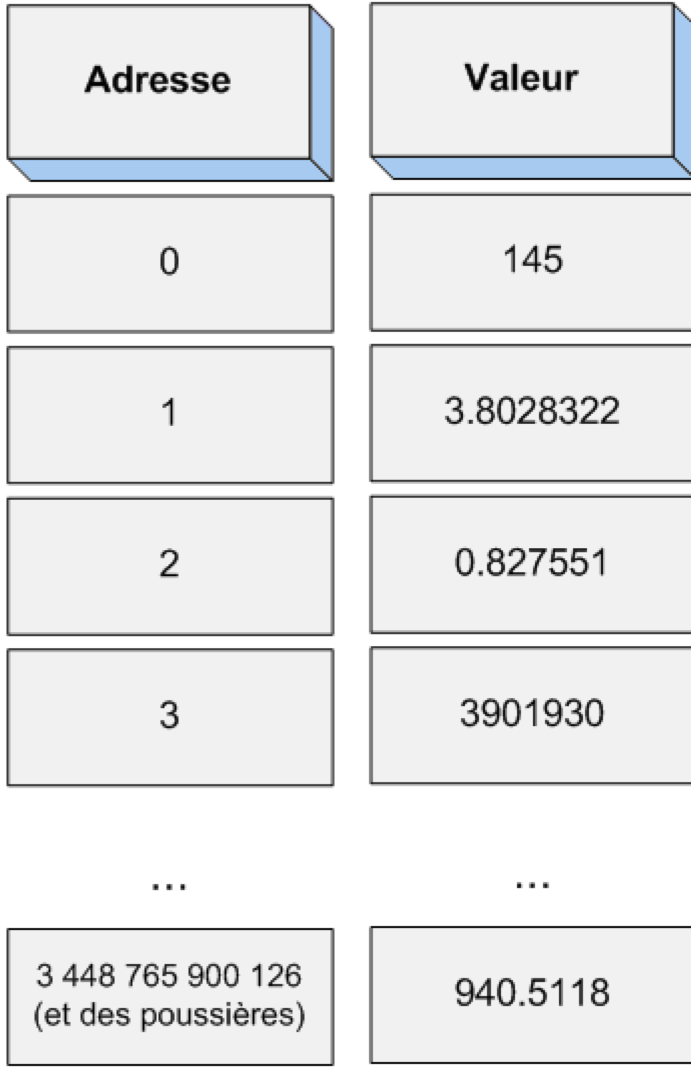
La mémoire vive est aussi appelée **RAM**, ne vous étonnez donc pas si par la suite j'utilise plutôt le mot RAM qui est un peu plus court.

Le schéma de la mémoire vive

En photographiant de plus près la mémoire vive, on n'y verrait pas grand-chose. Pourtant, il est très important de

savoir comment ça fonctionne à l'intérieur. C'est d'ailleurs là que je veux en venir depuis tout à l'heure

Je vais vous faire un schéma du fonctionnement de la mémoire vive. Il est ultra-simplifié (comme mes schémas de compilation), mais c'est parce que nous n'avons pas besoin de trop de détails. Si vous retenez ce schéma déjà, ça sera très bien



Comme vous le voyez, il faut en gros distinguer 2 colonnes :

- Il y a les **adresses** : une adresse est un nombre qui permet à l'ordinateur de se repérer dans la mémoire vive. On commence à l'adresse 0 (au tout début de la mémoire) et on finit à l'adresse 3 448 765 900 126 et des poussières... Euh, en fait je ne connais pas le nombre d'adresses qu'il y a dans la RAM, je sais juste qu'il y en a beaucoup.
  En plus ça dépend de la quantité de mémoire vive que vous avez. Plus vous avez de mémoire vive, plus il y a d'adresses, donc plus on peut stocker de choses
- A chaque adresse, on peut stocker une **valeur** (un nombre) : votre ordinateur stocke dans la mémoire vive ces nombres pour pouvoir s'en souvenir par la suite. On ne peut stocker qu'un nombre par adresse !

Notre RAM ne peut stocker que des nombres.

Mais alors, comment fait-on pour retenir des mots ?

Bonne question. En fait, même les lettres ne sont que des nombres pour l'ordinateur ! Une phrase est une simple succession de nombres !
Il existe un tableau qui fait la correspondance entre les nombres et les lettres. C'est un tableau qui dit par exemple : *le nombre 67 correspond a lettre Y*. Je ne rentre pas dans les détails, on aura l'occasion de reparler de cela plus loin dans le cours.

Revenons à notre schéma. Les choses sont en fait très simples : si l'ordinateur veut retenir le nombre 5 (qui pourrait être le nombre de vies qu'il reste au joueur), il le met quelque part en mémoire où il y a de la place et note l'adresse correspondante (par exemple 3 062 199 902)
Plus tard, lorsqu'il veut savoir à nouveau quel est ce nombre, il va chercher à la "case" mémoire n°3 062 199 902 ce qu'il y a, et il trouve la valeur... 5 !

Voilà en gros comment ça fonctionne. C'est peut-être un peu flou pour le moment (quel intérêt de stocker un nombre s'il faut à la place retenir l'adresse ?) mais tout va rapidement prendre du sens dans la suite de ce chapitre je vous le promets

DÉCLARER UNE VARIABLE

Croyez-moi, cette petite introduction sur la mémoire va nous être plus utile que vous ne le pensez.
Maintenant que vous savez ce qu'il faut, on peut retourner programmer

Alors une variable, c'est quoi ?
Eh bien c'est une petite information temporaire qu'on stocke dans la RAM. . Tout simplement.
On dit qu'elle est "variable" car c'est une valeur qui peut changer pendant le déroulement du programme. Par exemple, notre nombre 5 de tout à l'heure (le nombre de vies restant au joueur) risque de diminuer au fil du temps. Si ce nombre atteint 0, on saura que le joueur a perdu.

Nos programmes, vous allez le voir, sont remplis de variables. Vous allez en voir partout, à toutes les sauces

En langage C, une variable est constituée de 2 choses :

- Elle a une **valeur** : c'est le nombre qu'elle stocke, par exemple 5.
- Elle a un **nom** : c'est ce qui permet de la reconnaître. En programmant en C, on n'aura pas à retenir l'adresse mémoire (ouf !), on va juste indiquer des noms de variables à la place. C'est le compilateur qui fera la conversion entre le nom et l'adresse. Voilà déjà un souci en moins.

Donner un nom à ses variables

En langage C, chaque variable doit donc avoir un nom. Pour notre fameuse variable qui retient le nombre de vies, on aimerait bien l'appeler "Nombre de vies" ou quelque chose du genre.

Hélas, il y a quelques contraintes. Vous ne pouvez pas appeler une variable n'importe comment :

- Il ne peut y avoir que des lettres minuscules et majuscules et des chiffres (abcABC012...).
- Votre nom de variable doit commencer par une lettre.
- Les espaces sont interdits. A la place, on peut utiliser le caractère "underscore" _ (qui ressemble à un trait de soulignement). C'est le seul caractère différent des lettres et chiffres autorisé.
- Vous n'avez pas le droit d'utiliser des accents (éàê etc).

Enfin, et c'est très important à savoir, le langage C (comme le C++) fait la différence entre les majuscules et les minuscules. Pour votre culture, sachez qu'on dit que c'est un langage qui "respecte la casse".
Donc, du coup, les variables largeur, LARGEUR ou encore LArgEuR sont 3 variables différentes en langage C, même si pour nous ça a l'air de signifier la même chose !

Voici quelques exemples de noms de variable corrects : nombreDeVies, nombre_de_vies, prenom, nom, numero_de_telephone, numeroDeTelephone.

Chaque programmeur a sa propre façon de nommer des variables. Pendant ce cours, je vais vous montrer ma manière de faire :

- Je commence tous mes noms de variables par une lettre minuscule.
- S'il y a plusieurs mots dans mon nom de variable, je mets une lettre majuscule au début de chaque nouveau mot.

Je vais vous demander de faire de la même manière que moi, ça nous permettra d'être sur la même longueur d'ondes

Quoi que vous fassiez, faites en sorte de donner des noms clairs à vos variables. On aurait pu abréger *nombreDeVies*, en l'écrivant par exemple *ndv*. C'est peut-être plus court, mais c'est beaucoup moins clair pour vous quand vous relisez votre code. N'ayez donc pas peur de donner des noms un peu plus longs pour que ça reste compréhensible.

Les types de variables

Notre ordinateur, vous pourrez le constater, n'est en fait rien d'autre qu'une (très grosse) machine à calculer. Il ne sait traiter que des nombres.

Oui mais voilà, j'ai un scoop ! Il existe **plusieurs types de nombres** !

Par exemple, il y a les nombres entiers positifs :

- 45
- 398
- 7650

Mais il y a aussi des nombres décimaux, c'est-à-dire des nombres à virgules :

- 75,909
- 1,7741
- 9810,7

En plus de ça, il y a aussi des nombres entiers négatifs :

- -87
- -916

... Et des nombres négatifs décimaux !

- -76,9
- -100,11

Votre pauvre ordinateur a besoin d'aide ! Lorsque vous lui demandez de stocker un nombre, vous devez dire de quel type il est. Ce n'est pas vraiment qu'il ne soit pas capable de le reconnaître tout seul, mais... Ca l'aide beaucoup à s'organiser, et à faire en sorte de ne pas prendre trop de mémoire pour rien.

Lorsque vous créez une variable, vous allez donc devoir **indiquer son type**.
Voici les principaux types de variables existant en langage C (il y en aura un autre qui fera son apparition en C++) :

Nom du type	Nombres stockables
char	-128 à 127
int	-2 147 483 648 à 2 147 483 647
long	-2 147 483 648 à 2 147 483 647
float	-3.4 x 10 puissance 38 à 3.4 x 10 puissance 38
double	-1.7 x 10 puissance 308 à 1.7 x 10 puissance 308

(Je suis loin d'avoir mis tous les types, mais j'ai conservé les principaux *)*

Les 3 premiers types permettent de stocker des nombres entiers (1, 2, 3, 4...)
Les 2 derniers permettent de stocker des nombres décimaux (13.8, 16.911...)

Les types float et double permettent de stocker des nombres décimaux extrêmement grands.
Si vous ne connaissez pas les puissances de 10, dites-vous par exemple que le type double permet de stocker le nombre 1 suivi de 308 zéros derrière !
C'est-à-dire : 10000000000000000000000000000000000000000000000000..... (je ne vais quand même pas écrire 308 zéros pour vous)

Vous remarquerez qu'int et un long ont l'air identiques. Avant ce n'était pas le cas (un int était plus petit qu'un long), mais aujourd'hui les mémoires ont évolué et on a assez de place pour stocker des grands nombres, donc on se moque un peu de la différence entre un int et un long. Le langage C "conserve" tous ces types pour des raisons de compatibilité, même si certains sont un peu de trop.
En pratique, j'utilise principalement **char, long et double**.

Vous verrez que la plupart du temps on manipule des nombres entiers (tant mieux, parce que c'est plus facile à utiliser)

Attention avec les nombres décimaux ! Votre ordinateur ne connaît pas la virgule, il utilise le point.
Vous ne devez donc pas écrire 54,9 mais plutôt 54.9 !

Ce n'est pas tout ! Pour les types stockant des entiers (char, int, long...), il existe d'autres types dits "unsigned" (non signés) qui eux ne peuvent stocker que des nombres positifs. Pour les utiliser, il suffit d'écrire le mot "unsigned" devant le type :

unsigned char	0 à 255
unsigned int	0 à 4 294 967 295
unsigned long	0 à 4 294 967 295

Comme vous le voyez, les unsigned sont des types qui ont le défaut de ne pas pouvoir stocker de nombre négatifs, mais qui ont l'avantage de pouvoir stocker des nombres 2 fois plus grands (char s'arrête à 128, tandis que unsigned char s'arrête à 255 par exemple).

Pourquoi avoir créé 3 types pour les nombres entiers ? Un seul type aurait été suffisant non ?

Oui, mais on a créé plusieurs types à l'origine pour économiser de la mémoire. Ainsi, quand on dit à l'ordinateur qu'on a besoin d'une variable de type "char", on prend moins d'espace en mémoire que si on avait demandé une variable de type "long".

Toutefois, c'était utile surtout à l'époque où la mémoire était limitée. Aujourd'hui, nos ordinateurs ont largement assez de mémoire vive pour que ça ne soit plus vraiment un problème. Il ne sera donc pas utile de se prendre la tête pendant des heures sur le choix d'un type. Si vous ne savez pas si votre variable risque de prendre une grosse valeur, mettez long.
Et je dis ça sérieusement. Ne vous prenez pas trop la tête sur le choix d'un type pour le moment

En résumé, on fera surtout la distinction entre nombres entiers et décimaux :

- Pour un nombre entier, on utilisera le plus souvent long.
- Pour un nombre décimal, on utilisera généralement double.

Déclarer une variable

On y arrive. Maintenant, créez un nouveau projet console que vous appellerez "variables".
On va voir comment déclarer une variable, c'est-à-dire **demander à l'ordinateur la permission d'utiliser un peu de mémoire**.

Une déclaration de variable, c'est très simple maintenant que vous savez tout ce qu'il faut
Il suffit d'indiquer dans l'ordre :

1. Le type de la variable que l'on veut créer
2. Tapez espace
3. Indiquez le nom que vous voulez donner à la variable
4. Et enfin n'oubliez pas le point-virgule

Par exemple, si je veux créer ma variable nombreDeVies de type long, je dois taper la ligne suivante :

Code : C

```c
long nombreDeVies;
```

Et c'est tout !

Quelques autres exemples stupides pour la forme :

Code : C

```c
long noteDeMaths;
double sommeArgentRecue;
unsigned long nombreDeZerosEnTrainDeLireUnNomDeVariableUnPeuLong;
```

Bon bref, vous avez compris le principe je pense

Ce qu'on fait là s'appelle une **déclaration de variable** (un vocabulaire à retenir là)
Vous devez faire les déclarations de variables au début des fonctions. Comme pour le moment on n'a qu'une seule fonction (la fonction *main*), vous allez déclarer la variable comme ceci :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  // Début de la fonction
  long nombreDeVies;

  system("PAUSE");
  return 0;
  // Fin de la fonction
}
```

Si vous lancez ce programme, vous constaterez avec stupeur... qu'il ne fait rien

Quelques explications

Alors, avant que vous ne m'étrangliez en croyant que je vous mène en bateau depuis tout à l'heure, laissez-moi juste dire une chose pour ma défense

En fait, il se passe des choses, mais vous ne les voyez pas. Lorsque le programme arrive à la ligne de la déclaration de variable, il demande bien gentiment à l'ordinateur s'il peut utiliser un peu d'espace dans la mémoire vive.
Si tout va bien, l'ordinateur répond "Oui bien sûr, fais comme chez toi". Généralement, cela se passe sans problème.

> Le seul problème qu'il pourrait y avoir, c'est qu'il n'y ait plus de place en mémoire... Mais cela arrive rarement heureusement, car pour remplir toute la mémoire rien qu'avec des long il faut vraiment être un bourrin de première

Soyez sans craintes donc, vos variables devraient normalement être créées sans souci.

> Une petite astuce à connaître : si vous avez plusieurs variables du même type à déclarer, inutile de faire une ligne pour chaque variable. Il vous suffit de séparer les différents noms de variable par des virgules sur la même ligne :
>
> **Code : C**
>
> ```c
> long nombreDeVies, niveau, ageDuJoueur;
> ```
>
> Cela créera 3 variables long appelées nombreDeVies, niveau et ageDuJoueur.

Et maintenant ?
Maintenant qu'on a créé notre variable, on va pouvoir lui donner une valeur

Affecter une valeur à une variable

C'est tout ce qu'il y a de plus bête. Si vous voulez donner une valeur à la variable nombreDeVies, il suffit de procéder comme ceci :

Code : C

```c
nombreDeVies = 5;
```

Rien de plus à faire. Vous indiquez le nom de la variable, un signe égal, puis la valeur que vous voulez mettre

dedans.
Ici, on vient de donner la valeur 5 à la variable nombreDeVies.
Notre programme complet ressemble donc à ceci :

Code : C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  long nombreDeVies;
  nombreDeVies = 5;

  system("PAUSE");
  return 0;
}
```

Là encore, rien ne s'affiche à l'écran, tout se passe dans la mémoire.
Quelque part dans les tréfonds de votre ordinateur, une petite case de mémoire vient de prendre la valeur 5. C'est pas magnifique ça ?
Pour un peu on en pleurerait

On peut s'amuser si on veut à changer la valeur par la suite :

Code : C

```
long nombreDeVies;
nombreDeVies = 5;
nombreDeVies = 4;
nombreDeVies = 3;
```

Dans cet exemple, la variable va prendre d'abord la valeur 5, puis 4, et enfin 3. Comme votre ordinateur est très rapide, tout cela se passe extrêmement vite. Vous n'avez pas le temps de cligner des yeux que votre variable vient de prendre les valeurs 5, 4 et 3... et ça y est votre programme est fini

La valeur d'une nouvelle variable

Voici une question très importante que je veux vous soumettre :

Quand on déclare une variable, quelle valeur a-t-elle au départ ?

En effet, quand l'ordinateur lit cette ligne :

Code : C

```
long nombreDeVies;
```

Il réserve un petit emplacement en mémoire, d'accord. Mais quelle est la valeur de la variable à ce moment-là ? Y a-t-il une valeur par défaut (par exemple 0) ?

Eh bien, accrochez-vous : la réponse est non. Non non et non, il n'y a pas de valeur par défaut. En fait, l'emplacement est réservé mais la valeur ne change pas. On n'efface pas ce qui se trouve dans la "case mémoire". Du coup, votre variable prend la valeur qui se trouvait là avant dans la mémoire, et cette valeur peut être n'importe quoi !

Si cette zone de la mémoire n'a jamais été modifiée, la valeur est peut-être 0. Mais vous n'en êtes pas sûrs, il

pourrait très bien y avoir le nombre 363 ou 18 à la place, c'est-à-dire un reste d'un vieux programme qui est passé par là avant !
Il faut donc faire très attention à ça si on veut éviter des problèmes par la suite. Le mieux est d'initialiser la variable dès qu'on la déclare. En C, c'est tout à fait possible. En gros, ça consiste à combiner la déclaration et l'affectation d'une variable dans la même instruction :

Code : C

```c
long nombreDeVies = 5;
```

Ici, la variable nombreDeVies est déclarée et elle prend tout de suite la valeur 5.
L'avantage, c'est que vous êtes sûrs après que cette variable contient une valeur correcte, et pas du n'importe quoi

Les constantes

Il arrive parfois que l'on ait besoin d'utiliser une variable dont on voudrait qu'elle garde la même valeur pendant toute la durée du programme. C'est-à-dire qu'une fois déclarée, vous voudriez que votre variable conserve sa valeur et que personne n'ait le droit de changer ce qu'elle contient.

Ces variables particulières sont appelées **constantes**, justement parce que leur valeur reste constante.

Pour déclarer une constante, c'est en fait très simple : il faut utiliser le mot "const" juste devant le type quand vous déclarez votre variable.
Par ailleurs, il faut obligatoirement lui donner une valeur au moment de sa déclaration comme on vient d'apprendre à le faire. Après, il sera trop tard : vous ne pourrez plus changer la valeur de la constante.

Exemple de déclaration de constante :

Code : C

```c
const long NOMBRE_DE_VIES_INITIALES = 5;
```

Ce n'est pas une obligation, mais par convention on écrit les noms des constantes entièrement en **majuscules** comme je viens de le faire là. Cela nous permet ainsi de distinguer facilement les constantes des variables. Notez qu'on utilise l'underscore _ à la place de l'espace.

A part ça, une constante s'utilise comme une variable normale, vous pouvez afficher sa valeur si vous le désirez.
La seule chose qui change, c'est que si vous essayez de modifier la valeur de la constante plus loin dans le programme, le compilateur vous indiquera qu'il y a une erreur avec cette constante.

Les erreurs de compilation sont affichées en bas de l'écran, dans ce que j'appelle la "zone de la mort", vous vous souvenez ?
Dans un tel cas, le compilateur vous afficherait un mot doux du genre : *[Warning] assignment of read-only variable 'NOMBRE_DE_VIES_INITIALES'* (traduction : "mais t'es vraiment idiot, pourquoi t'essaies de modifier la valeur d'une constante ? ")

AFFICHER LE CONTENU D'UNE VARIABLE

On sait afficher du texte à l'écran avec la fonction *printf*.
Maintenant, on va voir comment afficher la valeur d'une variable avec cette même fonction.

On utilise en fait printf de la même manière, sauf que l'on rajoute un symbole spécial à l'endroit où on veut afficher la valeur de la variable.
Par exemple :

Code : C

```
printf("Il vous reste %ld vies");
```

Ce "symbole spécial" dont je viens de vous parler est en fait un % suivi des lettres "ld". Ces lettres permettent d'indiquer ce que l'on doit afficher. "ld" signifie que c'est un nombre entier.
Il existe plusieurs autres possibilités, mais pour des raisons de simplicité on va se contenter de retenir ces deux-là :

Symbole	Signification
%ld	Nombre entier (ex. : 4)
%lf	Nombre décimal (ex. : 5.18)

Je vous parlerai des autres symboles en temps voulu. Pour le moment, sachez que si vous voulez afficher une variable entière (char, int, long...), vous devez utiliser %ld, et pour un nombre décimal (float, double), vous utiliserez %lf.

On a presque fini. On a indiqué qu'à un endroit précis on voulait afficher un nombre entier, mais on n'a pas précisé lequel ! Il faut donc indiquer à la fonction *printf* quelle est la variable dont on veut afficher la valeur.
Pour ce faire, vous devez taper le nom de la variable après les guillemets et après avoir rajouté une virgule, comme ceci :

Code : C

```
printf("Il vous reste %ld vies", nombreDeVies);
```

Le %ld sera remplacé par la variable indiquée après la virgule, à savoir nombreDeVies.
On se teste ça un petit coup dans un programme ?

Code : C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  long nombreDeVies = 5; // Au départ, le joueur a 5 vies

  printf("Vous avez %ld vies\n", nombreDeVies);
  printf("**** B A M ****\n"); // Là il se prend un grand coup sur la tête
  nombreDeVies = 4; // Il vient de perdre une vie !
  printf("Ah desole, il ne vous reste plus que %ld vies maintenant !\n\n", nombreDeVies);

  system("PAUSE");
  return 0;
}
```

Ca pourrait presque être un jeu vidéo (il faut juste beaucoup d'imagination).
Ce programme affiche ceci à l'écran :

Code : Console

```
Vous avez 5 vies
**** B A M ****
Ah desole, il ne vous reste plus que 4 vies maintenant !

Appuyez sur une touche pour continuer...
```

Vous devriez reconnaître ce qui se passe dans votre programme :

1. Au départ le joueur a 5 vies, on affiche ça dans un printf
2. Ensuite le joueur prend un coup sur la tête (d'où le BAM)
3. Finalement il n'a plus que 4 vies, on affiche ça aussi avec un printf

Bref, c'est plutôt simple

Afficher plusieurs variables dans un même printf

Il est possible d'afficher la valeur de plusieurs variables dans un seul printf. Il vous suffit pour cela d'indiquer des %ld ou des %lf là où vous voulez, puis d'indiquer les variables correspondantes dans le même ordre, séparées par des virgules.

Par exemple :

Code : C

```
printf("Vous avez %ld vies et vous etes au niveau n°%ld", nombreDeVies, niveau);
```

Veillez à bien indiquer vos variables dans le bon ordre. Le premier %ld sera remplacé par la première variable (nombreDeVies), et le second %ld par la seconde variable (niveau). Si vous vous trompez d'ordre, votre phrase ne voudra plus rien dire

Allez un petit test maintenant. Notez que j'enlève les lignes tout en haut (les directives de préprocesseur commençant par un #), je vais supposer que vous les mettez à chaque fois maintenant :

Code : C

```
int main(int argc, char *argv[])
{
  long nombreDeVies = 5, niveau = 1;

  printf("Vous avez %ld vies et vous etes au niveau n°%ld\n", nombreDeVies, niveau);

  system("PAUSE");
  return 0;
}
```

Ce qui affichera :

Code : Console

```
Vous avez 5 vies et vous etes au niveau n°1

Appuyez sur une touche pour continuer...
```

RÉCUPÉRER UNE SAISIE

Les variables vont en fait commencer à devenir intéressantes maintenant. On va apprendre à demander à l'utilisateur de taper un nombre dans la console. Ce nombre, on va le récupérer et le stocker dans une variable.
Une fois que ça sera fait, on pourra faire tout un tas de choses avec, vous verrez

Pour demander à l'utilisateur de rentrer quelque chose dans la console, on va utiliser une autre fonction toute prête : *scanf*
Cette fonction ressemble beaucoup à *printf*. Vous devez mettre un %ld ou un %lf entre guillemets pour indiquer si vous voulez que l'utilisateur rentre un entier ou un décimal. Puis vous devez indiquer après le nom de la variable qui

va recevoir le nombre.

Voici comment faire par exemple

Code : C

```
scanf("%ld", &age);
```

On ne doit mettre que le %ld (ou le %lf) entre les guillemets.
Par ailleurs, il faut mettre le symbole & devant le nom de la variable qui va recevoir la valeur.

Là, il va falloir que vous me fassiez confiance. Si je dois vous expliquer ça tout de suite, on n'est pas sortis de l'auberge croyez-moi
Que je vous rassure quand même : je vous expliquerai un peu plus tard ce que signifie ce symbole. Pour le moment, je choisis de ne pas vous l'expliquer pour ne pas vous embrouiller, c'est donc plutôt un service que je vous rends là

Lorsque le programme arrive à un *scanf*, il se met en pause et attend que l'utilisateur rentre un nombre. Ce nombre sera stocké dans la variable "age".

Voici un petit programme simple qui demande l'âge de l'utilisateur et qui le lui affiche ensuite :

Code : C

```
int main(int argc, char *argv[])
{
  long age = 0; // On initialise la variable à 0

  printf("Quel age avez-vous ? ");
  scanf("%ld", &age); // On demande d'entrer l'age avec scanf
  printf("Ah ! Vous avez donc %ld ans !\n\n", age);

  system("PAUSE");
  return 0;
}
```

Code : Console

```
Quel age avez-vous ? 20
Ah ! Vous avez donc 20 ans !

Appuyez sur une touche pour continuer...
```

Le programme se met donc en pause après avoir affiché la question "Quel age avez-vous ?". Le curseur apparaît à l'écran, vous devez taper un nombre entier (votre âge). Tapez ensuite sur Entrée pour valider, et le programme continuera à s'exécuter.
Ici, tout ce qu'il fait après c'est afficher la valeur de la variable "age" à l'écran (*"Ah ! Vous avez donc 20 ans !"*).

Voilà, vous avez compris le principe
Grâce à la fonction *scanf*, on peut donc commencer à interagir avec l'utilisateur, histoire de lui demander 2-3 informations privées

Notez que rien ne vous empêche de taper autre chose qu'un nombre entier :

- Si vous rentrez un nombre décimal, comme 2.9, il sera automatiquement tronqué, c'est-à-dire que seule la

partie entière sera conservée. Dans ce cas, c'est le nombre 2 qui aurait été stocké dans la variable.
- Si vous tapez des lettres au hasard ("éèydf"), la variable ne changera pas de valeur. Ce qui est bien ici, c'est qu'on avait initialisé notre variable à 0 au début. Du coup, le programme affichera "0 ans" si ça n'a pas marché. Si on n'avait pas initialisé la variable, le programme aurait pu afficher n'importe quoi !

On va s'arrêter là pour le chapitre sur les variables
Comme je n'ai de cesse de vous le répéter, les variables sont utilisées tout le temps en programmation. Si vous avez compris qu'une variable était une petite information stockée temporairement en mémoire, vous avez tout compris. Il n'y a rien à savoir de plus... à part peut-être connaître quand même les types de variables (char, int, long...).

Entraînez-vous aussi à afficher la valeur d'une variable à l'écran et à récupérer un nombre saisi au clavier avec *scanf*
Dans le prochain chapitre, nous verrons comment faire des calculs en langage C. Il faut donc impérativement que vous soyez à l'aise avec *scanf* et *printf* si vous voulez suivre

Une bête de calcul

Je vous l'ai dit dans le chapitre précédent : votre ordinateur n'est en fait qu'une grosse machine à calculer. Que vous soyez en train d'écouter de la musique, regarder un film ou jouer à un jeu vidéo, votre ordinateur ne fait que des calculs.

Ce chapitre va vous apprendre à réaliser la plupart des calculs qu'un ordinateur sait faire. Nous réutiliserons ce que nous venons tout juste d'apprendre, à savoir les variables. L'idée, c'est justement de faire des calculs avec vos variables : ajouter des variables entre elles, les multiplier, enregistrer le résultat dans une autre variable etc.

Même si vous n'êtes pas fan des maths, ce chapitre est totalement indispensable. Et puis, parlons franchement : si vous ne savez pas faire une addition, vous n'êtes pas fait pour la programmation (et toc)

Les calculs de base

Il faut savoir qu'en plus de n'être qu'une vulgaire calculatrice, votre ordinateur est une calculatrice très basique puisqu'on ne peut faire que des opérations très simples :

- Addition
- Soustraction
- Multiplication
- Division
- Modulo (je vous expliquerai ce que c'est si vous ne savez pas)

Si vous voulez faire des opérations plus compliquées (des carrés, des puissances, des logarithmes et autres joyeusetés) il vous faudra les programmer, c'est-à-dire **expliquer à l'ordinateur comment le faire**.
Fort heureusement, nous verrons plus loin dans ce chapitre qu'il existe une librairie mathématique livrée avec le langage C qui contient plein de fonctions mathématiques toutes prêtes. Vous n'aurez pas à les réécrire donc, à moins que vous soyez maso (ou prof de maths, ça marche aussi)

Voyons voir donc l'addition pour commencer.
Pour faire une addition, on utilise le signe + (non, sans blague ?).
Vous devez mettre le résultat de votre calcul dans une variable. On va donc par exemple créer une variable "resultat" de type long et faire un calcul :

Code : C

```
long resultat = 0;

resultat = 5 + 3;
```

Pas besoin d'être un pro du calcul mental pour deviner que la variable "resultat" contiendra la valeur 8 après exécution
Bien sûr, rien ne s'affiche à l'écran avec ce code. Si vous voulez voir la valeur de la variable, rajoutez un printf comme vous savez maintenant si bien le faire :

Code : C

```
printf("5 + 3 =  %ld", resultat);
```

A l'écran, cela donnera :

Code : Console

```
5 + 3 = 8
```

Voilà pour l'addition.
Pour les autres opérations, c'est pareil, seul le signe utilisé change :

- Addition : +
- Soustraction : -
- Multiplication : *
- Division : /
- Modulo : %

Si vous avez déjà utilisé la calculatrice sur votre ordinateur, vous devriez connaître ces signes. Le signe "moins" est en fait le tiret, le signe "multiplié" est une étoile, et le signe "divisé" est le slash (la barre oblique).

Il n'y a pas de difficulté particulière pour ces opérations, à part pour les deux dernières (la division et le modulo). Nous allons donc parler un peu plus en détail de chacune d'elles.

La division

Les divisions fonctionnent normalement sur un ordinateur quand il n'y a pas de reste. Par exemple, 6 / 3 ça fait 2, votre ordinateur vous donnera la réponse juste. Jusque-là pas de souci.

Prenons maintenant une division avec reste comme 5 / 2.
5 / 2, si vous calculez bien, ça fait 2.5

Et pourtant ! Regardez ce que fait ce code :

Code : C

```
long resultat = 0;

resultat = 5 / 2;
printf ("5 / 2 = %ld", resultat);
```

Code : Console

```
5 / 2 = 2
```

Il y a un gros problème. On a demandé 5 / 2, on s'attend à avoir 2.5, et l'ordinateur nous dit que ça fait 2 !

Il y a anguille sous roche. Nos ordinateurs seraient-ils stupides à ce point ?
En fait, quand il voit les chiffres 5 et 2, votre ordinateur fait une division de nombres entiers. Cela veut dire qu'il tronque le résultat, il ne garde que la partie entière (le 2).

Eh mais je sais ! C'est parce que *resultat* est un long ! Si ça avait été un double, il aurait pu stocker un nombre décimal à l'intérieur !

Même pas

Essayez le même code en transformant juste *resultat* en double, et vous verrez qu'on vous affiche quand même 2.

Si on veut que l'ordinateur affiche le bon résultat, il va falloir transformer les nombres 5 et 2 de l'opération en nombres décimaux, c'est-à-dire écrire 5.0 et 2.0 (ce sont les mêmes nombres, mais pour l'ordinateur ce sont des nombres décimaux, donc il fait une division de nombres décimaux) :

Code : C

```
double resultat = 0;

resultat = 5.0 / 2.0;
printf ("5 / 2 = %lf", resultat);
```

Code : Console

```
5 / 2 = 2.500000
```

Là le nombre est correct. Bon il affiche plein de zéros derrière si ça lui chante, mais le résultat reste quand même correct.

Cette propriété de la division de nombres entiers est super importante. Il faut que vous reteniez que pour un ordinateur :
5 / 2 = 2
10 / 3 = 3
4 / 5 = 0
Si vous voulez avoir un résultat décimal, il faut que les nombres de l'opération soient décimaux :
5.0 / 2.0 = 2.5
10.0 / 3.0 = 3.33333
4.0 / 5.0 = 0.8

En fait, en faisant une division d'entiers comme "5 / 2", votre ordinateur répond à la question "Combien y a-t-il de fois 2 dans le nombre 5 ?". La réponse est 2 fois. De même, combien de fois y a-t-il le nombre 3 dans 10 ? 3 fois".

Mais alors me direz-vous, comment on fait pour récupérer le reste de la division ?
C'est là que super-modulo intervient

Le modulo

Le modulo est une opération mathématique qui permet d'obtenir **le reste d'une division**. C'est peut-être une opération moins connue que les 4 autres, mais pour votre ordinateur ça reste une opération de base... probablement justement pour combler le problème de la "division d'entiers" qu'on vient de voir.

Le modulo, je vous l'ai dit tout à l'heure, se représente par le signe %.
Voici quelques exemples de modulos :

- 5 % 2 = 1
- 14 % 3 = 2
- 4 % 2 = 0

Le modulo 5 % 2 est le reste de la division 5 / 2, c'est-à-dire 1. L'ordinateur calcule que 5 = 2 * 2 + 1 (c'est ce 1, le reste, que le modulo renvoie)
De même, 14 % 3, le calcul est 14 = 3 * 4 + 2 (modulo renvoie le 2)
Enfin, pour 4 % 2, la division tombe juste, il n'y a pas de reste, donc modulo renvoie 0.

Voilà, je ne peux pas dire grand-chose d'autre de plus au sujet des modulos. Je tenais juste à l'expliquer pour ceux qui ne connaîtraient pas

En plus j'ai une bonne nouvelle : on a vu toutes les opérations de base. Finis les cours de maths

Des calculs entre variables

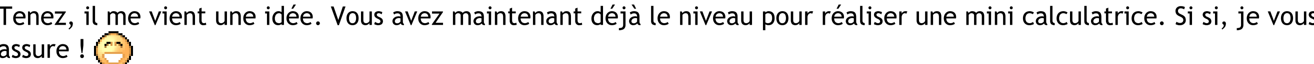
Ce qui serait intéressant, maintenant que vous savez faire les 5 opérations de base, ce serait de s'entraîner à faire des calculs entre plusieurs variables.
En effet, rien ne vous empêche de faire :

Code : C

```
resultat = nombre1 + nombre2;
```

Cette ligne fait la somme des variables nombre1 et nombre2, et stocke le résultat dans la variable resultat.

Et c'est là que les choses commencent à devenir très intéressantes



Tenez, il me vient une idée. Vous avez maintenant déjà le niveau pour réaliser une mini calculatrice. Si si, je vous assure !

Imaginez un programme qui demande 2 nombres à l'utilisateur. Ces deux nombres, vous les stockez dans des variables.
Ensuite, vous faites la somme de ces variables, et vous stockez le résultat dans une variable appelée "resultat".

Vous n'avez plus qu'à afficher le résultat du calcul à l'écran, sous les yeux ébahis de l'utilisateur qui n'aurait jamais été capable de calculer cela de tête aussi vite

Essayez de coder vous-même ce petit programme, c'est facile et ça vous entraînera

La réponse est ci-dessous :

Code : C

```
int main(int argc, char *argv[])
{
  long resultat = 0, nombre1 = 0, nombre2 = 0;

  // On demande les nombres 1 et 2 à l'utilisateur :

  printf("Entrez le nombre 1 : ");
  scanf("%ld", &nombre1);
  printf("Entrez le nombre 2 : ");
  scanf("%ld", &nombre2);

  // On fait le calcul :

  resultat = nombre1 + nombre2;

  // Et on affiche l'addition à l'écran :

  printf ("%ld + %ld = %ld\n", nombre1, nombre2, resultat);

  system("PAUSE");
  return 0;
}
```

Code : Console

```
Entrez le nombre 1 : 30
Entrez le nombre 2 : 25
30 + 25 = 55
```

Mine de rien, on vient de faire là notre premier programme qui a un intérêt. Notre programme est capable d'additionner 2 nombres et d'afficher le résultat de l'opération

Vous pouvez essayez avec n'importe quel nombre (du temps que vous ne dépassez pas les limites d'un type long), votre ordinateur effectuera le calcul en un éclair (encore heureux, parce que des opérations comme ça il doit en faire des milliards dans une même seconde)

Je vous conseille de faire la même chose avec les autres opérations pour vous entraîner (soustraction, multiplication...). En plus, vous ne devriez pas avoir trop de mal vu qu'il y a juste un ou deux signes à changer
Vous pouvez aussi ajouter une troisième variable et faire l'addition de 3 variables à la fois, ça fonctionne sans problème :

Code : C

```c
resultat = nombre1 + nombre2 + nombre3;
```

LES RACCOURCIS

Comme promis, nous n'avons pas de nouvelles opérations à voir. Et pour cause ! On les a déjà toutes vues
C'est avec ces simples opérations de base que vous pouvez tout créer. Il n'y a pas besoin d'autres opérations. Je reconnais que c'est difficile à avaler, se dire qu'un jeu 3D ne fait rien d'autre au final que des additions et des soustractions, pourtant c'est la stricte vérité

Ceci étant, il existe en C des techniques permettant de raccourcir l'écriture des opérations.
Pourquoi utiliser des raccourcis ? Parce que, souvent, on fait des opérations répétitives. Vous allez voir ce que je veux dire par là tout de suite, avec ce qu'on appelle l'incrémentation.

L'incrémentation

Vous verrez que vous serez souvent amenés à ajouter 1 à une variable. Au fur et à mesure du programme, vous aurez des variables qui augmentent de 1 en 1.

Imaginons que votre variable s'appelle "nombre" (nom très original n'est-ce pas). Sauriez-vous comment faire pour ajouter 1 à cette variable, sans savoir quel est le nombre qu'elle contient ?

Voici comment on doit faire :

Code : C

```c
nombre = nombre + 1;
```

Que se passe-t-il ici ? On fait le calcul nombre + 1, et on range ce résultat dans la variable... nombre ! Du coup, si notre variable nombre valait 4, elle vaut maintenant 5. Si elle valait 8, elle vaut maintenant 9 etc...

Cette opération est justement répétitive. Les informaticiens étant des gens particulièrement fainéants, ils n'avaient guère envie de taper 2 fois le même nom de variable (ben oui quoi, c'est fatigant !).
Ils ont donc inventé un raccourci pour cette opération qu'on appelle **l'incrémentation**. L'instruction ci-dessous fait exactement la même chose que le code qu'on vient de voir :

Code : C

```c
nombre++;
```

Cette ligne, bien plus courte que celle de tout à l'heure, signifie "Ajoute 1 à la variable nombre". Il suffit d'écrire le nom de la variable à incrémenter, de mettre 2 signes +, et de ne pas oublier le point-virgule bien entendu.

Mine de rien, cela nous sera bien pratique par la suite car, comme je vous l'ai dit, on sera souvent amenés à faire des incrémentations (c'est-à-dire ajouter 1 à une variable).

Si vous êtes perspicaces, vous avez d'ailleurs remarqué que ce signe ++ se trouve dans le nom du langage "C++". C'est en fait un clin d'oeil des programmeurs, et vous êtes maintenant capables de le comprendre ! C++ signifie que c'est du langage C "incrémenté", c'est-à-dire si on veut "du langage C à 1 niveau supérieur"

La décrémentation

C'est tout bêtement l'inverse de l'incrémentation : on enlève 1 à une variable.
Même si on fait plus souvent des incrémentations que des décrémentations, cela reste une opération pratique que vous utiliserez de temps en temps.

La décrémentation, si on l'écrit en forme "longue" :

Code : C

```
nombre = nombre - 1;
```

Et maintenant en forme "raccourcie" :

Code : C

```
nombre--;
```

On l'aurait presque deviné tout seul ça
Au lieu de mettre un ++, vous mettez un --. Si votre variable vaut 6, elle vaudra 5 après l'instruction de décrémentation.

Les autres raccourcis

Il existe d'autres raccourcis qui fonctionnent sur le même principe. Cette fois, ces raccourcis fonctionnent pour toutes les opérations de base : + - * / %

Cela permet là encore d'éviter une répétition du nom d'une variable sur une même ligne.
Ainsi, si vous voulez multiplier par 2 une variable :

Code : C

```
nombre = nombre * 2;
```

Vous pouvez l'écrire d'une façon raccourcie comme ceci :

Code : C

```
nombre *= 2;
```

Si le nombre vaut 5 au départ, il vaudra 10 après cette instruction.
Pour les autres opérations de base, cela fonctionne de la même manière. Voici un petit programme d'exemple :

Code : C

```
long nombre = 2;

nombre += 4; // nombre vaut 6...
nombre -= 3; // ... nombre vaut maintenant 3
nombre *= 5; // ... nombre vaut 15
nombre /= 3; // ... nombre vaut 5
nombre %= 3; // ... nombre vaut 2 (car 5 = 1 * 3 + 2)
```

(allez boudez pas, un peu de calcul mental n'a jamais tué personne *)*

L'avantage ici est qu'on peut utiliser toutes les opérations de base, et qu'on peut ajouter, soustraire, multiplier par n'importe quel nombre.
Ce sont des raccourcis à connaître si vous avez des lignes répétitives à taper un jour dans un programme

Retenez quand même que l'incrémentation reste de loin le raccourci le plus utilisé

La librairie mathématique

En langage C, il existe ce qu'on appelle des librairies "standard", c'est-à-dire des librairies toujours utilisables. Ce sont en quelque sorte des librairies "de base" qu'on utilise très souvent.

Les librairies sont, je vous le rappelle, des ensembles de fonctions toutes prêtes. Ces fonctions ont été écrites par des programmeurs avant vous, elles vous évitent en quelque sorte d'avoir à réinventer la roue à chaque nouveau programme

Vous avez déjà utilisé les fonctions *printf* et *scanf* de la librairie stdio.h.
Il faut savoir qu'il existe une autre librairie, appelée math.h, qui contient de nombreuses fonctions mathématiques toutes prêtes.

> En effet, les 5 opérations de base que l'on a vu sont loin d'être suffisantes ! Bon, il se peut que vous n'ayez jamais besoin de certaines opérations complexes comme les exponentielles (si vous ne savez pas ce que c'est, c'est que vous êtes peut-être un peu trop jeune ou que vous n'avez pas assez fait de maths dans votre vie). Toutefois, la librairie mathématique contient de nombreuses autres fonctions dont vous aurez très probablement besoin.

Tenez par exemple, on ne sait pas faire des puissances en C ! Comment calculer un simple carré ? Vous pouvez toujours essayer de taper 5^2 dans votre programme, mais votre ordinateur ne le comprendra jamais car il ne sait pas ce que c'est... A moins que vous le lui expliquiez en lui indiquant la librairie mathématique !

Pour pouvoir utiliser les fonctions de la librairie mathématique, il est indispensable de mettre la directive de préprocesseur suivante en haut de votre programme :

Code : C

```
#include <math.h>
```

Une fois que c'est fait, vous pouvez utiliser toutes les fonctions de cette librairie.

J'ai justement l'intention de vous les présenter
Bon, comme il y a beaucoup de fonctions je ne peux pas faire la liste complète ici. D'une part ça vous ferait trop à assimiler, et d'autre part mes pauvres petits doigts auraient fondu avant la fin de l'écriture du chapitre
Je vais donc me contenter des principales fonctions, c'est-à-dire celles qui me semblent les plus importantes.

Vous n'avez peut-être pas tous le niveau en maths pour comprendre ce que font ces fonctions. Si c'est votre cas, pas d'inquiétude. Lisez juste, cela ne vous pénalisera pas pour la suite.
Ceci étant, je vous offre un petit conseil gratuit : soyez attentifs en cours de maths, on dirait pas comme ça mais en fait ça finit par servir

fabs

Cette fonction retourne la valeur absolue d'un nombre, c'est-à-dire |x| (c'est la notation mathématique).
La valeur absolue d'un nombre est sa valeur positive :

- Si vous donnez -53 à la fonction, elle vous renvoie 53.
- Si vous donnez 53 à la fonction, elle vous renvoie 53.

En bref, elle renvoie toujours l'équivalent positif du nombre que vous lui donnez.

Code : C

```c
double absolu = 0, nombre=-27;

absolu = fabs(nombre); // absolu vaudra 27
```

Cette fonction renvoie un double, donc votre variable "absolu" doit être de type double.

Il existe aussi une fonction similaire appelée "abs", située dans "stdlib.h" cette fois.
La fonction "abs" marche de la même manière, sauf qu'elle utilise des entiers (int). Elle renvoie donc un nombre entier de type int et non un double comme fabs.

ceil

Cette fonction renvoie le premier nombre entier après le nombre décimal qu'on lui donne.

C'est une sorte d'arrondi. On arrondit en fait toujours au nombre entier supérieur.
Par exemple, si on lui donne 26.512, la fonction renvoie 27.

Cette fonction s'utilise de la même manière, et renvoie un double :

Code : C

```c
double dessus = 0, nombre = 52.71;

dessus = ceil(nombre); // dessus vaudra 53
```

floor

C'est l'inverse de la fonction précédente, cette fois elle renvoie le nombre directement en dessous.
Si vous lui donnez 37.91, la fonction *floor* vous renverra donc 37

pow

Cette fonction permet de calculer la puissance d'un nombre. Vous devez lui indiquer 2 valeurs : le nombre, et la puissance à laquelle vous voulez l'élever. Voici le schéma de la fonction :

Code : C

```c
pow(nombre, puissance);
```

Par exemple, "2 puissance 3" (que l'on écrit habituellement 2^3 sur un ordinateur), c'est le calcul 2 * 2 * 2, ce qui fait 8 :

Code : C

```c
double resultat = 0, nombre = 2;

resultat = pow(nombre, 3); // resultat vaudra 2^3 = 8
```

Vous pouvez donc utiliser cette fonction pour calculer des carrés. Il suffit d'indiquer une puissance de 2.

sqrt

Cette fonction calcule la racine carrée d'un nombre. Elle renvoie un double.

Code : C

```c
double resultat = 0, nombre = 100;

resultat = sqrt(nombre); // resultat vaudra 10
```

sin, cos, tan

Ce sont les 3 fameuses fonctions utilisées en trigonométrie.
Le fonctionnement est le même, ces fonctions renvoient un double.

Ces fonctions attendent une valeur en **radians**.

asin, acos, atan

Ce sont les fonctions arc sinus, arc cosinus et arc tangente, d'autres fonctions de trigonométrie.
Elles s'utilisent de la même manière et renvoient un double.

exp

Cette fonction calcule l'exponentielle d'un nombre. Elle renvoie un double (oui oui, elle aussi)

log

Cette fonction calcule le logarithme népérien d'un nombre (que l'on note aussi "ln")

log10

Cette fonction calcule le logarithme base 10 d'un nombre.

Conclusion

Conclusion, ben heureusement que je n'ai pas parlé des autres fonctions (en fait, je ne comprends même pas à quoi elle servent)
Déjà, avec ces fonctions-là vous avez de quoi faire si vous vous ennuyez

Encore une fois, si vous n'avez pas compris un mot de ce que j'ai dit, ce n'est pas bien grave car on n'en a pas absolument besoin. Tout dépend en fait du programme que vous allez faire : si vous programmez une calculatrice scientifique, c'est sûr que vous vous en servirez

Retenez quand même les fonctions **floor, ceil, et pow**, elles vous seront probablement utiles (même si vous ne programmez pas une calculatrice oui oui) Et voilà pour la minute mathématique du Site du Zér0

Ce chapitre est dédicacé à tous les profs de Maths (une profession mal reconnue, je vous le dis)
Si vous êtes encore étudiant, je vous offre ce petit conseil gratuit : la programmation, c'est souvent des maths. Suivre en maths, ça permet de bien mieux se débrouiller ensuite en programmation

Bien sûr, on ne calcule pas tout le temps des exponentielles et des tangentes quand on programme. Je l'ai dit et je le redis, ça dépend du programme qu'on fait. Par exemple, si certains d'entre vous envisagent de faire de la 3D (ce que je compte vous enseigner, mais bien plus tard dans le cours) il vous faudra quelques connaissances en géométrie de l'espace (vous savez, les vecteurs et tout ça)

Les conditions

Nous avons vu dans le premier chapitre qu'il existait de nombreux langages de programmation. Certains d'entre eux se ressemblent d'ailleurs : par exemple le PHP est très inspiré du langage C, bien qu'il serve plutôt à créer des sites web qu'à créer des programmes

En fait le langage C a été créé il y a assez longtemps, ce qui fait qu'il a servi de modèle à de nombreux langages plus récents.
La plupart des langages de programmation ont finalement des ressemblances, ils reprennent **les principes de base** de leurs aînés.

En parlant de principe de base : on est en plein dedans. On a vu comment créer des variables, faire des calculs avec (concept commun à tous les langages de programmation !), nous allons maintenant nous intéresser aux **conditions**.
Sans conditions, nos programmes informatiques feraient un peu toujours la même chose, ce qui serait carrément barbant à la fin

La condition "if... else"

Les conditions servent à "tester" des variables. On peut par exemple dire "Si la variable machin est égale à 50, fais ceci"... Mais ce serait dommage de ne pouvoir tester que l'égalité ! Il faudrait aussi pouvoir tester si la variable est inférieure à 50, inférieure ou égale à 50, supérieure, supérieure ou égale...

Ne vous inquiétez pas, le C a tout prévu
(mais vous n'en doutiez pas hein)

Pour étudier les conditions "if... else", nous allons suivre le plan suivant :

I. Quelques symboles à connaître avant de commencer

II. Le test if
III. Le test else
IV. Le test "else if"
V. Plusieurs conditions à la fois

VI. Quelques erreurs courantes à éviter

Avant de voir comment on écrit une condition de type "if... else" en C, il faut donc que vous connaissiez 2-3 symboles de base. Ces symboles sont indispensables pour réaliser des conditions.

Quelques symboles à connaître

Voici un petit tableau de symboles du langage C à connaître par coeur

Symbole	Signification
==	Est égal à
>	Est supérieur à
<	Est inférieur à
>=	Est supérieur ou égal à
<=	Est inférieur ou égal à
!=	Est différent de

Faites très attention, il y a bien 2 symboles "==" pour tester l'égalité. Une erreur courante que font les débutants et de ne mettre qu'un symbole =, ce qui n'a pas la même signification en C. Je vous en reparlerai un peu plus bas.

Un if simple

Attaquons maintenant sans plus tarder
Nous allons faire un test simple, qui va dire à l'ordinateur :

Citation : Test simple

> SI la variable vaut ça
> ALORS fais ceci

En anglais, le mot "si" se traduit par "if". C'est celui qu'on utilise en langage C pour introduire une condition. Ecrivez donc un if. Ouvrez ensuite des parenthèses : à l'intérieur de ces parenthèses vous devrez écrire votre condition.

Ensuite, ouvrez une accolade { et fermez-la un peu plus loin }. Tout ce qui se trouve à l'intérieur des accolades sera exécuté uniquement si la condition est vérifiée.

Cela nous donne donc à écrire :

Code : C

```c
if (/* Votre condition */)
{
  // Instructions à exécuter si la condition est vraie
}
```

A la place de mon commentaire "Votre condition", on va écrire une condition pour tester une variable.
Par exemple, on pourrait tester une variable "age" qui contient votre âge. Tenez pour s'entraîner, on va tester si vous êtes majeur, c'est-à-dire **si votre âge est supérieur ou égal à 18** :

Code : C

```
if (age >= 18)
{
  printf ("Vous etes majeur !");
}
```

Le symbole >= signifie "Supérieur ou égal", comme on l'a vu dans le tableau tout à l'heure.

S'il n'y a qu'une instruction entre les accolades, alors celles-ci deviennent facultatives. Vous pouvez donc écrire :

Code : C

```
if (age >= 18)
  printf ("Vous etes majeur !");
```

Tester ce code

Si vous voulez tester les codes précédents pour voir comment le if fonctionne, il faudra placer le if à l'intérieur d'une fonction main et ne pas oublier de déclarer une variable age à laquelle on donnera la valeur de notre choix.
Ca peut paraître évident pour certains, mais apparemment ça ne l'était pas pour tout le monde aussi ai-je rajouté cette explication

Voici un code complet **que vous pouvez tester** :

Code : C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    long age = 20;

    if (age >= 18)
    {
      printf ("Vous etes majeur !\n");
    }

    system("PAUSE");
    return 0;
}
```

Ici, la variable age vaut 20, donc le "Vous êtes majeur !" s'affichera.
Essayez de changer la valeur initiale de la variable pour voir. Mettez par exemple 15 : la condition sera fausse, et donc "Vous êtes majeur !" ne s'affichera pas cette fois

Servez-vous de ce code de base pour tester les prochains exemples du chapitre

Une question de propreté

La façon dont vous ouvrez les accolades n'est pas importante, votre programme marchera aussi bien si vous écrivez

tout sur une même ligne. Par exemple :

Code : C

```
if (age >= 18) {  printf ("Vous etes majeur !"); }
```

Pourtant, même si c'est possible d'écrire comme ça, c'est **ultra déconseillé** (notez que quand j'écris plus gros, en gras rouge souligné, c'est généralement parce que c'est vraiment important)
En effet, tout écrire sur une même ligne rend votre code **difficilement lisible**. Si vous ne prenez pas dès maintenant l'habitude d'aérer votre code, plus tard quand vous écrirez de plus gros programmes vous ne vous y retrouverez plus !

Essayez donc de présenter votre code source de la même façon que moi : une accolade sur une ligne, puis vos instructions (précédées d'une tabulation pour les "décaler vers la droite"), puis l'accolade de fermeture sur une ligne.

Il existe plusieurs bonnes façons de présenter son code source. Ca ne change rien au fonctionnement du programme final, mais c'est une question de "style informatique" si vous voulez
Si vous voyez un code de quelqu'un d'autre présenté un peu différemment, c'est qu'il code avec un style différent. Le principal, c'est que son code reste aéré et lisible.

Le "else" pour dire "sinon"

Maintenant que nous savons faire un test simple, allons un peu plus loin : si le test n'a pas marché (il est faux), on va dire à l'ordinateur d'exécuter d'autres instructions.

En français, nous allons donc écrire quelque chose qui ressemble à cela :

Citation : Test avec sinon

> SI la variable vaut ça
> ALORS fais ceci
> SINON fais cela

Il suffit de rajouter le mot **else** après l'accolade fermante du **if**.
Petit exemple :

Code : C

```
if (age >= 18) // Si l'âge est supérieur ou égal à 18
{
  printf ("Vous etes majeur !");
}
else // Sinon...
{
  printf ("Ah c'est bete, vous etes mineur !");
}
```

Les choses sont assez simples : si la variable age est supérieure ou égale à 18, on affiche le message "Vous êtes majeur !", sinon on affiche "Vous êtes mineur".

Le "else if" pour dire "sinon si"

On a vu comment faire un "si" et un "sinon". Il est possible aussi de faire un "sinon si" pour faire un autre test si le

premier test n'a pas marché. Le "sinon si" se met entre le if et le else.

On dit dans ce cas à l'ordinateur :

Citation : Avec un sinon si

SI la variable vaut ça ALORS fais ceci
SINON SI la variable vaut ça ALORS fais ça
SINON fais cela

Traduction en langage C :

Code : C

```
if (age >= 18) // Si l'âge est supérieur ou égal à 18
{
  printf ("Vous etes majeur !");
}
else if ( age > 4 ) // Sinon, si l'âge est au moins supérieur à 4
{
  printf ("Bon t'es pas trop jeune quand meme...");
}
else // Sinon...
{
  printf ("Aga gaa aga gaaa gaaa"); // Langage Bébé, vous pouvez pas comprendre ;o)
}
```

L'ordinateur fait les tests dans l'ordre :

1. D'abord il teste le premier if : si la condition est vraie, alors il exécute ce qui se trouve entre les premières accolades.
2. Sinon, il va au "sinon si" et fait à nouveau un test : si ce test est vrai, alors il exécute les instructions correspondantes entre accolades.
3. Enfin, si aucun des tests précédents n'a marché, il exécute les instructions du "sinon".

Le "else" et le "else if" ne sont pas obligatoires. Pour faire une condition, il faut juste au moins un "if" (logique me direz-vous, sinon il n'y a pas de condition !)
Notez qu'on peut mettre autant de "else if" que l'on veut. On peut donc écrire :

Citation : Plusieurs else if

SI la variable vaut ça
ALORS fais ceci
SINON SI la variable vaut ça ALORS fais ça
SINON SI la variable vaut ça ALORS fais ça
SINON SI la variable vaut ça ALORS fais ça
SINON fais cela

Plusieurs conditions à la fois

Il peut aussi être utile de faire plusieurs tests à la fois dans votre if. Par exemple, vous voudriez tester si l'âge est supérieur à 18 ET si l'âge est inférieur à 25.
Pour faire cela, il va falloir utiliser de nouveaux symboles :

&&	ET
||	OU
!	NON

Test ET

Si on veut faire le test que j'ai mentionné plus haut, il faudra écrire :

Code : C

```c
if (age > 18 && age < 25)
```

Les deux symboles "&&" signifient ET. Notre condition se dirait en français : "*Si l'âge est supérieur à 18 ET si l'âge est inférieur à 25*"

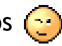
Test OU

Pour faire un OU, on utilise les 2 signes ||. Je dois avouer que ce signe n'est pas facilement accessible sur nos claviers. Pour le taper sur un clavier AZERTY français, il faudra faire **Alt Gr + 6**. Sur un clavier belge, il faudra faire Alt Gr + &.

Imaginons un programme débile qui décide si une personne a le droit d'ouvrir un compte en banque. C'est bien connu, pour ouvrir un compte en banque il vaut mieux ne pas être trop jeune (on va dire arbitrairement qu'il faut avoir au moins 30 ans) ou bien avoir plein d'argent (parce que là même à 10 ans on vous acceptera à bras ouverts)
Notre test pour savoir si le client a le droit d'ouvrir un compte en banque pourrait être :

Code : C

```c
if (age > 30 || argent > 100000)
{
  printf("Bienvenue chez PicsouBanque !");
}
else
{
  printf("Hors de ma vue, miserable !");
}
```

Ce test n'est valide que si la personne a plus de 30 ans ou si elle possède plus de 100 000 euros 

Test NON

Le dernier symbole qu'il nous reste à tester est le point d'exclamation. En informatique, le point d'exclamation signifie "Non".
Vous devez mettre ce signe avant votre condition pour dire "Si cela n'est pas vrai" :

Code : C

```c
if (!(age < 18))
```

Cela pourrait se traduire par "*Si la personne n'est pas mineure*".
Si on avait enlevé le "!" devant, cela aurait signifié l'inverse : "*Si la personne est mineure*".

Quelques erreurs courantes de débutant

N'oubliez pas les 2 signes ==

Si on veut tester si la personne a tout juste 18 ans, il faudra écrire :

Code : C

```
if (age == 18)
{
  printf ("Vous venez de devenir majeur !");
}
```

N'oubliez pas de mettre 2 signes "égal" dans un if, comme ceci : ==
Si vous ne mettez qu'un seul signe =, alors votre variable *prendra* la valeur 18 (comme on l'a appris dans le chapitre sur les variables). Nous ce qu'on veut faire ici, c'est tester la valeur de la variable, pas la changer ! Faites très attention à cela, beaucoup d'entre vous n'en mettent qu'un quand ils débutent et forcément... leur programme ne marche pas comme ils voudraient

Le point-virgule de trop

Une autre erreur courante de débutant : vous mettez parfois un point-virgule à la fin de la ligne d'un if. Or, un if est une condition, et on ne met de point-virgule qu'à la fin d'une instruction et non d'une condition.

Le code suivant ne marchera pas comme prévu car il y a un point-virgule à la fin du if :

Code : C

```
if (age == 18); // Notez le point-virgule ici qui ne devrait PAS être là
{
  printf ("Tu es tout juste majeur");
}
```

LES BOOLÉENS, LE COEUR DES CONDITIONS

Nous allons maintenant rentrer plus en détail dans le fonctionnement d'une condition de type if... else.
En effet, les conditions font intervenir quelque chose qu'on appelle **les booléens** en informatique.

C'est un concept très important, donc ouvrez grand vos oreilles (euh vos yeux plutôt)

Quelques petits tests pour bien comprendre

En cours de Physique-Chimie, mon prof avait l'habitude de nous faire commencer par quelques petites expériences avant d'introduire une nouvelle notion.
Je vais l'imiter un peu aujourd'hui

Voici un code source très simple que je vous demande de tester :

Code : C

```
if (1)
{
    printf("C'est vrai");
}
else
{
    printf("C'est faux");
}
```

Résultat :

Code : Console

```
C'est vrai
```

Mais ??? On n'a pas mis de condition dans le if, juste un nombre. Qu'est-ce que ça veut dire ça n'a pas de sens ?

Si ça en a, vous allez comprendre
Faites un autre test maintenant en remplaçant le 1 par un 0 :

Code : C

```
if (0)
{
    printf("C'est vrai");
}
else
{
    printf("C'est faux");
}
```

Résultat :

Code : Console

```
C'est faux
```

Faites maintenant d'autres tests en remplaçant le 0 par n'importe quel autre nombre entier, comme 4, 15, 226, -10, -36 etc...
Qu'est-ce qu'on vous répond à chaque fois ? On vous répond : "C'est vrai".

Résumé de nos tests : si on met un 0, le test est considéré comme faux, et si on met un 1 ou n'importe quel autre nombre, le test est vrai.

Des explications s'imposent

En fait, à chaque fois que vous faites un test dans un if, ce test renvoie la valeur 1 s'il est vrai, et 0 s'il est faux.

Par exemple :

Code : C

```
if (age >= 18)
```

Ici, le test que vous faites est "age >= 18".

Supposons que age vaille 23. Alors le test est vrai, et l'ordinateur "remplace" en quelque sorte "age >= 18" par 1.
Ensuite, l'ordinateur obtient (dans sa tête) un "if (1)". Quand le nombre est 1, comme on l'a vu, l'ordinateur dit que la condition est vraie, donc il affiche "C'est vrai" !

De même, si la condition est fausse, il remplace age >= 18 par le nombre 0, et du coup la condition est fausse : l'ordinateur va lire les instructions du "else".

Un test avec une variable

Testez maintenant un autre truc : envoyez le résultat de votre condition dans une variable, comme si c'était une opération (car pour l'ordinateur, **c'est** une opération !).

Code : C

```
long age = 20;
int majeur = 0;

majeur = age >= 18;
printf("Majeur vaut : %ld\n", majeur);
```

Comme vous le voyez, la condition age >= 18 a renvoyé le nombre 1 **car elle est vraie**. Du coup, notre variable majeur vaut 1, on vérifie d'ailleurs ça en faisant un printf qui montre bien qu'elle a changé de valeur.

Faites le même test en mettant age = 10 par exemple. Cette fois, majeur vaudra 0.

Cette variable "majeur" est un booléen

Retenez bien ceci :

On dit qu'une variable à laquelle on fait prendre les valeurs 0 et 1 est **un booléen.**

Et aussi ceci :

0 = Faux
1 = Vrai

Pour être tout à fait exact, 0 = faux et tous les autres nombres valent vrai (on a eu l'occasion de le tester plus tôt). Ceci dit, pour simplifier les choses on va se contenter de n'utiliser que les chiffres 0 et 1, pour dire si "quelque chose est faux ou vrai".

En langage C, il n'existe pas de type de variable "booléen". Il n'y a pas de type comme "double", "char"...
En fait, le type booléen n'a été rajouté qu'en C++. En effet, en C++ vous avez un nouveau type "bool" qui a été créé spécialement pour ces variables booléennes.

Comme pour l'instant on fait du C, on n'a donc pas de type spécial. Du coup, on est obligé d'utiliser un autre type. Pour ma part, afin de bien différencier dans mon code les variables qui contiennent des nombres de celles qui contiennent un booléen, j'utilise le type "int". Dans la suite de ce cours, tous mes int seront donc des booléens, ce qui les rendra faciles à identifier

Les booléens dans les conditions

Souvent, on fera un test "if" sur une variable booléenne :

Code : C

```
int majeur = 1;

if (majeur)
{
  printf("Tu es majeur !");
}
else
{
  printf("Tu es mineur");
}
```

Comme majeur vaut 1, la condition est vraie, donc on affiche "Tu es majeur !".

Ce qui est très pratique, c'est que la condition se lit facilement par un être humain. On voit "if (majeur)", ce que peut traduire par "*Si tu es majeur*"
Les tests sur des booléens sont donc faciles à lire et à comprendre, pour peu que vous ayez donné des noms clairs à vos variables comme je vous ai dit de le faire depuis le début

Tenez, voici un autre test imaginaire :

Code : C

```
if (majeur && garcon)
```

Ce test signifie "Si tu es majeur ET que tu es un garçon".
garcon est ici une autre variable booléenne qui vaut 1 si vous êtes un garçon, et 0 si vous êtes... une fille ! Bravo, vous avez tout compris !

Les booléens servent donc à exprimer si quelque chose est vrai ou faux.
C'est très utile, et ce que je viens de vous expliquer vous permettra de comprendre bon nombre de choses par la suite

Petite question : si on fait le test "if (majeur == 1)", ça marche aussi non ?

Tout à fait. Mais le principe des booléens c'est justement de raccourcir l'expression du if et de la rendre plus facilement lisible. Avouez que "if (majeur)" ça se comprend très bien non ?

Retenez donc : si votre variable est censée contenir un nombre, faites un test sous la forme "if (variable == 1)".
Si au contraire votre variable est censée contenir un booléen (c'est-à-dire soit 1 soit 0 pour dire vrai ou faux), faites un test sous la forme "if (variable)".

La condition "switch"

La condition "if... else" que l'on vient de voir est le type de condition le plus souvent utilisé.
En fait, il n'y a pas 36 façons de faire une condition en C. Le "if... else" permet de gérer tous les cas.

Toutefois, le "if... else" peut s'avérer quelque peu... répétitif. Prenons cet exemple :

Code : C

```c
if (age == 2)
{
  printf("Salut bebe !");
}
else if (age == 6)
{
  printf("Salut gamin !");
}
else if (age == 12)
{
  printf("Salut jeune !");
}
else if (age == 16)
{
  printf("Salut ado !");
}
else if (age == 18)
{
  printf("Salut adulte !");
}
else if (age == 68)
{
  printf("Salut papy !");
}
else
{
  printf("Je n'ai aucune phrase de prete pour ton age ");
}
```

Construire un switch

Les informaticiens détestent faire des choses répétitives, on a eu l'occasion de le vérifier plus tôt

Alors, pour éviter d'avoir à faire des répétitions comme ça quand on teste la valeur d'une seule et même variable, ils ont inventé une autre structure que le "if... else"
Cette structure particulière s'appelle "switch". Voici un switch basé sur l'exemple qu'on vient de voir :

Code : C

```c
switch (age)
{
case 2:
  printf("Salut bebe !");
  break;
case 6:
  printf("Salut gamin !");
  break;
case 12:
  printf("Salut jeune !");
  break;
case 16:
  printf("Salut ado !");
  break;
case 18:
  printf("Salut adulte !");
  break;
case 68:
  printf("Salut papy !");
  break;
default:
  printf("Je n'ai aucune phrase de prete pour ton age ");
  break;
}
```

Imprégnez-vous de mon exemple pour créer vos propres switch. On les utilise plus rarement, mais c'est vrai que c'est pratique car ça fait (un peu) moins de code à taper

L'idée c'est donc d'écrire "switch (maVariable)" pour dire "*Je vais tester la valeur de la variable maVariable*".
Vous ouvrez ensuite des accolades que vous refermez tout en bas.

Ensuite, à l'intérieur de ces accolades, vous gérez tous les "cas" : case 2, case 4, case 5, case 45...

Vous devez mettre une instruction break; obligatoirement à la fin de chaque cas. Si vous ne le faites pas, alors l'ordinateur ira lire les instructions en-dessous censées être réservées aux autres cas !
L'instruction break; commande en fait à l'ordinateur de "sortir" des accolades.

Enfin, le cas "default" correspond en fait au "else" qu'on connaît bien maintenant. Si la variable ne vaut aucune des valeurs précédentes, l'ordinateur ira lire le default.

Gérer un menu avec un switch

Le switch est très souvent utilisé pour faire des menus en console.
Je crois que le moment est venu de pratiquer un peu

Au boulot !

En console, pour faire un menu, on fait des printf qui affichent les différentes options possibles. Chaque option est numérotée, et l'utilisateur doit rentrer le numéro du menu qui l'intéresse.
Voici par exemple ce que la console devra afficher :

Code : Console

```
=== Menu ===

1. Royal Cheese
2. Mc Deluxe
3. Mc Bacon
4. Big Mac

Votre choix ?
```

(Vous aurez compris que j'avais un peu faim lorsque j'étais en train de rédiger ces lignes)

Voici votre mission (si vous l'acceptez) : reproduisez ce menu à l'aide de printf (facile), ajoutez un scanf pour enregistrer le choix de l'utilisateur dans une variable choixMenu (trop facile), et enfin faites un switch pour dire à l'utilisateur "Tu as choisi le menu Royal Cheese" par exemple.

Allez, au travail

Correction

Voici la solution que j'espère que vous avez trouvée

Code : C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  long choixMenu;

  printf("=== Menu ===\n\n");
  printf("1. Royal Cheese\n");
  printf("2. Mc Deluxe\n");
  printf("3. Mc Bacon\n");
  printf("4. Big Mac\n");
  printf("\nVotre choix ? ");
  scanf("%ld", &choixMenu);

  printf("\n");

  switch (choixMenu)
  {
    case 1:
       printf("Vous avez choisi le Royal Cheese. Bon choix !");
       break;
    case 2:
       printf("Vous avez choisi le Mc Deluxe. Berk, trop de sauce...");
       break;
    case 3:
       printf("Vous avez choisi le Mc Bacon. Bon, ca passe encore ca ;o)");
       break;
    case 4:
       printf("Vous avez choisi le Big Mac. Vous devez avoir tres faim !");
       break;
    default:
       printf("Vous n'avez pas rentre un nombre correct. Vous ne mangerez rien du tout
!");
       break;
  }

  printf("\n\n");

  system("PAUSE");
}
```

Et voilà le travail

J'espère que vous n'avez pas oublié le "default" à la fin du switch !
En effet, quand vous programmez vous devez toujours penser à tous les cas. Vous avez beau dire de taper un nombre entre 1 et 4, vous trouverez toujours un imbécile qui ira taper "10" ou encore "Salut" alors que ce n'est pas ce que vous attendez

Bref, soyez toujours vigilants de ce côté-ci : ne faites pas confiance à l'utilisateur, il peut parfois rentrer n'importe quoi. Prévoyez toujours un cas "default" ou un "else" si vous faites ça avec des if.

Je vous conseille de vous familiariser avec le fonctionnement des menus en console, car on en fait souvent dans des programmes console et vous en aurez sûrement besoin

LES TERNAIRES : DES CONDITIONS CONDENSÉES

Il existe une troisième façon de faire des conditions, plus rare.

On appelle cela des **expressions ternaires**.
Concrètement, c'est comme un "if... else", sauf qu'on fait tout tenir sur une seule ligne !

Comme un exemple vaut mieux qu'un long discours, je vais vous donner 2 fois la même condition : la première avec un "if... else", et la seconde, identique, mais sous forme de ternaire.

Une condition if... else bien connue

Supposons qu'on ait une variable booléenne "majeur" qui vaut vrai (1) si on est majeur, et faux (0) si on est mineur. On veut changer la valeur de la variable age en fonction du booléen, pour mettre "18" si on est majeur, "17" si on est mineur. C'est un exemple complètement débile je suis d'accord, mais ça me permet de vous montrer comment on peut se servir des ternaires.

Voici comment faire cela avec un if... else :

Code : C

```
if (majeur)
  age = 18;
else
  age = 17;
```

Notez que j'ai enlevé dans cet exemple les accolades car elles sont **facultatives** s'il n'y a qu'une instruction, comme je vous l'ai expliqué plus tôt.

La même condition en ternaire

Voici un code qui fait exactement la même chose que le code précédent, mais écrit cette fois sous forme de ternaire :

Code : C

```
age = (majeur) ? 18 : 17;
```

Les ternaires permettent, sur une seule ligne, de changer la valeur d'une variable en fonction d'une condition. Ici la condition est tout simplement "majeur", mais ça pourrait être n'importe quelle condition plus longue hein

Le point d'interrogation permet de dire "Est-ce que tu es majeur ?". Si oui, alors on met la valeur 18 dans age. Sinon (le ":" signifie "else" ici), on met la valeur 17.

Les ternaires ne sont pas du tout indispensables, personnellement je ne les utilise pas trop car ils peuvent rendre la lecture d'un code source un peu difficile.
Ceci étant, il vaut mieux que vous les connaissiez si, un jour, vous tombez sur un code plein de ternaires dans tous les sens Nous venons de voir plusieurs choses fondamentales de la programmation.
En effet, à partir de maintenant vous allez effectuer des conditions partout dans vos programmes, donc mieux vaut vous entraîner

Tenez j'ai une idée pour vous entraîner (mais je vous fournis pas la correction cette fois) : réalisez une calculatrice en console. Affichez d'abord un menu qui demande l'opération (addition, soustraction, multiplication, division... et pourquoi pas une autre comme racine carrée, issue de la librairie mathématique !).
Une fois que l'utilisateur a fait son choix, demandez-lui les valeurs et... affichez le résultat !

Pour en revenir à ce que nous avons appris dans ce chapitre, je voudrais insister sur un point en particulier : **les booléens**. Il est vraiment super-capital-ultra-important de retenir que les booléens sont des variables qui signifient vrai ou faux selon leur valeur (0 valant faux, et 1 valant vrai).
Le prochain chapitre réutilisera les booléens et le principe des conditions, donc mieux vaut être prêt avant de s'y lancer

Allez courage, on avance à pas de géant !

Les boucles

Après avoir vu comment réaliser des conditions en C, nous allons apprendre à réaliser **des boucles**

Qu'est-ce qu'une boucle ?
C'est une technique permettant de répéter les mêmes instructions plusieurs fois. Cela nous sera bien utile par la suite, notamment pour le premier TP qui vous attend après ce chapitre

Relaxez-vous : ce chapitre sera simple. Nous avons vu ce qu'étaient les conditions et les booléens dans le chapitre précédent, c'était un gros morceau à avaler. Maintenant ça va couler de source, et le TP ne devrait pas vous poser trop de problèmes
Enfin, profitez-en, parce qu'ensuite nous ne tarderons pas à entrer dans la partie II du tutorial, et là vous aurez intérêt à être sacrément armés

QU'EST-CE QU'UNE BOUCLE ?

Tout comme pour les conditions, il y a plusieurs façons de réaliser des boucles. Au bout du compte, cela revient à faire la même chose : répéter les mêmes instructions un certain nombre de fois.
Nous allons voir 3 types de boucles courantes en C :

- while
- do... while
- for

Dans tous les cas, le schéma est le même :

Voici ce qu'il se passe dans l'ordre :

1. L'ordinateur lit les instructions de haut en bas (comme d'habitude)
2. Puis, une fois arrivé à la fin de la boucle, il repart à la première instruction
3. Il recommence alors à lire les instructions de haut en bas...
4. ... Et il repart au début de la boucle.

Le problème dans ce système c'est que si on ne l'arrête pas, l'ordinateur est capable de répéter les instructions à l'infini ! Il est pas du genre à se plaindre vous savez, il fait ce qu'on lui dit de faire

Et c'est là qu'on retrouve... **des conditions** !
Quand on crée une boucle, on indique toujours une condition. Cette condition signifiera " *Répète la boucle tant que cette condition est vraie.* ".

Il y a plusieurs manières de s'y prendre comme je vous l'ai dit. Voyons voir sans plus tarder comment on réalise une boucle de type while en C

LA BOUCLE WHILE

Voici comment on construit une boucle while :

Code : C

```
while (/* Condition */)
{
    // Instructions à répéter
}
```

C'est aussi simple que cela
While signifie "Tant que". On dit donc à l'ordinateur "*Tant que la condition est vraie : répète les instructions entre accolades*".

Je vous propose de faire un test simple : on va demander à l'utilisateur de taper le nombre 47. Tant qu'il n'a pas tapé le nombre 47, on recommence à lui demander le nombre. Le programme ne pourra s'arrêter que si l'utilisateur tape le nombre 47 (je sais je sais, je suis diabolique) :

Code : C

```
long nombreEntre = 0;

while (nombreEntre != 47)
{
    printf("Tapez le nombre 47 ! ");
    scanf("%ld", &nombreEntre);
}
```

Voici maintenant le test que j'ai fait. Notez que j'ai fait exprès de me planter 2-3 fois avant de taper le bon nombre

Code : Console

```
Tapez le nombre 47 ! 10
Tapez le nombre 47 ! 27
Tapez le nombre 47 ! 40
Tapez le nombre 47 ! 47
```

Le programme s'est arrêté après avoir tapé le nombre 47.
Cette boucle while se répète donc tant que l'utilisateur n'a pas tapé 47, c'est assez simple.

Maintenant, essayons de faire quelque chose d'un peu plus intéressant : on veut que notre boucle se répète un certain nombre de fois.
On va pour cela créer une variable "compteur" qui vaudra 0 au début du programme et que l'on va **incrémenter** au fur et à mesure. Vous vous souvenez de l'incrémentation ? Ca consiste à ajouter 1 à la variable en faisant "*variable++;*".

Regardez attentivement ce bout de code et, surtout, essayez de le comprendre :

Code : C

```
long compteur = 0;

while (compteur < 10)
{
    printf("Salut les Zeros !\n");
    compteur++;
}
```

Résultat :

Code : Console

```
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
Salut les Zeros !
```

Ce code répète 10 fois l'affichage de "Salut les Zeros !".

Comment ça marche exactement ?

1. Au départ, on a une variable compteur initialisée à 0. Elle vaut donc 0 au début du programme.
2. La boucle while ordonne la répétition TANT QUE compteur est inférieur à 10. Comme compteur vaut 0 au départ, on rentre dans la boucle.
3. On affiche la phrase "Salut les Zeros !" via un printf.
4. On **incrémente** la valeur de la variable compteur, grâce à l'instruction "*compteur++;*". Compteur valait 0, il vaut maintenant 1.
5. On arrive à la fin de la boucle (accolade fermante), on repart donc au début, au niveau du while. On refait le test du while : "*Est-ce que compteur est toujours inférieur à 10 ?*". Ben oui, compteur vaut 1 Donc on recommence les instructions de la boucle.

Et ainsi de suite... Compteur va valoir progressivement 0, 1, 2, 3, ..., 8, 9, et 10. Lorsque compteur vaut 10, la condition "compteur < 10" est fausse. Comme l'instruction est fausse, on sort de la boucle.

On pourrait voir d'ailleurs que la variable compteur augmente au fur et à mesure dans la boucle, en l'affichant dans le printf :

Code : C

```c
long compteur = 0;

while (compteur < 10)
{
    printf("La variable compteur vaut %ld\n", compteur);
    compteur++;
}
```

Code : Console

```
La variable compteur vaut 0
La variable compteur vaut 1
La variable compteur vaut 2
La variable compteur vaut 3
La variable compteur vaut 4
La variable compteur vaut 5
La variable compteur vaut 6
La variable compteur vaut 7
La variable compteur vaut 8
La variable compteur vaut 9
```

Voilà, si vous avez compris ça, vous avez tout compris
Vous pouvez vous amuser à augmenter la limite du nombre de boucles ("< 100" au lieu de "< 10"). Cela m'aurait été d'ailleurs très pratique plus jeune pour rédiger les punitions que je devais réécrire 100 fois

Attention aux boucles infinies

Lorsque vous créez une boucle, assurez-vous toujours qu'elle peut s'arrêter à un moment ! Si la condition est toujours vraie, votre programme ne s'arrêtera jamais !
Voici un exemple de boucle infinie :

Code : C

```
while (1)
{
    printf("Boucle infinie\n");
}
```

Souvenez-vous des booléens : 1 = vrai, 0 = faux. Ici, la condition est toujours vraie, donc ce programme affichera "Boucle infinie" sans arrêt !

Pour arrêter un tel programme sous Windows, vous n'avez pas d'autre choix que de fermer la console en cliquant sur la croix en haut à droite. Sous Linux faites Ctrl + C.

Faites donc très attention : évitez à tout prix de tomber dans une boucle infinie.

Notez toutefois que les boucles infinies peuvent s'avérer utiles, notamment, nous le verrons plus tard, lorsque nous réaliserons des jeux.

La boucle do... while

Ce type de boucle est très similaire à while, bien qu'un peu moins utilisé en général.
La seule chose qui change en fait par rapport à while, c'est la position de la condition. Au lieu d'être au début de la boucle, **la condition est à la fin** :

Code : C

```
long compteur = 0;

do
{
    printf("Salut les Zeros !\n", compteur);
    compteur++;
} while (compteur < 10);
```

Qu'est-ce que ça change ?
C'est très simple : la boucle while pourrait très bien ne jamais être exécutée si la condition est fausse dès le départ. Par exemple, si on avait initialisé le compteur à 50, la condition aurait été fausse dès le début et on ne serait jamais rentré dans la boucle.
Pour la boucle do... while, c'est différent : cette boucle s'exécutera toujours au moins une fois. En effet, le test se fait à la fin comme vous pouvez le voir. Si on initialise compteur à 50, la boucle s'exécutera une fois.

Il est donc parfois utile de faire des boucles de ce type, pour s'assurer que l'on rentre au moins une fois dans la boucle. C'est quand même plus rare

Il y a une particularité dans la boucle do... while qu'on a tendance à oublier quand on débute : il y a un point-virgule tout à la fin ! N'oubliez pas d'en mettre un après le while, ou sinon votre programme plantera à la compilation

LA BOUCLE FOR

En théorie, la boucle while permet de réaliser toutes les boucles que l'on veut.
Toutefois, tout comme le switch pour les conditions, il est dans certains cas utiles d'avoir un autre système de boucle plus "condensé", plus rapide à écrire.

Les boucles for sont très très utilisées en programmation. Je n'ai pas de statistiques sous la main, mais sachez que vous utiliserez certainement autant de for que de while, donc il vous faudra savoir manipuler ces deux types de boucles.

Comme je vous le disais, les boucles for sont juste une autre façon de faire une boucle while.
Voici un exemple de boucle while que nous avons vu tout à l'heure :

Code : C

```
long compteur = 0;

while (compteur < 10)
{
    printf("Salut les Zeros !\n");
    compteur++;
}
```

Voici maintenant l'équivalent en boucle for :

Code : C

```
long compteur;

for (compteur = 0 ; compteur < 10 ; compteur++)
{
    printf("Salut les Zeros !\n");
}
```

Quelles différences ?

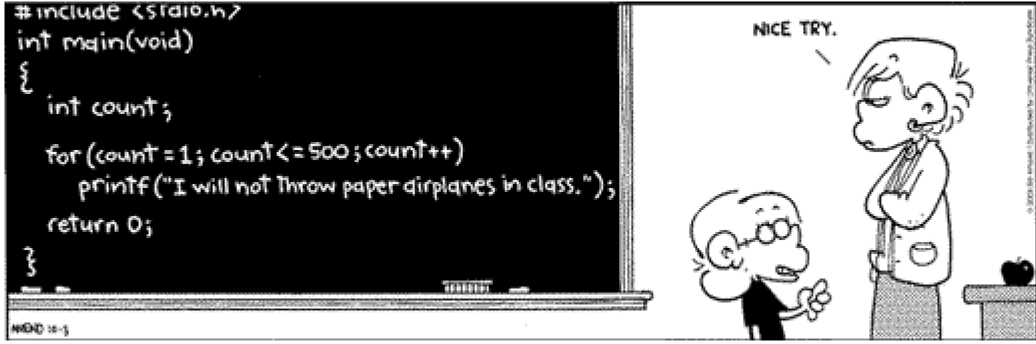
- Vous noterez qu'on n'a pas initialisé la variable compteur à 0 dès sa déclaration (mais on aurait pu le faire)
- Il y a beaucoup de choses entre les parenthèses après le for (nous allons détailler ça après)
- Il n'y a plus de compteur++; dans la boucle

Intéressons-nous à ce qui se trouve entre les parenthèses, car c'est là que réside tout l'intérêt de la boucle for. Il y a 3 instructions condensées, **séparée chacune par un point-virgule** :

- La première est **l'initialisation** : cette première instruction est utilisée pour préparer notre variable compteur. Dans notre cas, on initialise la variable à 0.
- La seconde est **la condition** : comme pour la boucle while, c'est la condition qui dit si la boucle doit être répétée ou pas. Tant que la condition est vraie, la boucle for continue.
- Enfin, il y a **l'incrémentation** : cette dernière instruction est exécutée à la fin de chaque tour de boucle pour mettre à jour la variable compteur. La quasi-totalité du temps on fera une incrémentation, mais on peut aussi faire une décrémentation (*variable--;*) ou encore n'importe quelle autre opération (*variable += 2;* pour avancer de 2 en 2 par exemple)

Bref, comme vous le voyez la boucle for n'est rien d'autre qu'un condensé
Sachez vous en servir, vous en aurez besoin plus d'une fois ! Dans le prochain chapitre, nous allons souffler un peu en faisant un TP. Les TP, vous allez le voir, sont des chapitres où vous n'apprenez rien de théorique, vous ne faites que de l'application pratique de ce que vous avez appris
Ce sera l'occasion de s'entraîner à réutiliser tout ce que vous avez assimilé jusqu'ici !

Bon allez, et pour terminer ce chapitre sur une touche d'humour, voici une petite blague de programmeurs que vous pouvez maintenant comprendre



Le texte du printf est "I will not throw paper airplanes in class", ce qui, pour ceux d'entre vous qui ne comprendraient pas l'anglais, signifie "Je ne jetterai plus d'avions en papier en classe".
"Nice try", répond le professeur, c'est-à-dire : "Bien essayé."

TP : Plus ou Moins, votre premier jeu

Nous arrivons maintenant dans le premier TP.
T.P. est l'accronyme de "Travaux pratiques". Ca veut dire... qu'on va pratiquer oui oui
Quel est le but des TP ?

Le but est de vous montrer que vous savez faire des choses avec ce que je vous ai appris. Car en effet, la théorie c'est bien, mais si on ne sait pas mettre tout ça en pratique de manière concrète... ben ça sert à rien d'avoir passé du temps à apprendre

Et, croyez-le ou non, mais vous avez déjà le niveau pour réaliser un premier programme amusant. C'est un petit jeu en mode console (les programmes en fenêtres arriveront plus tard je vous le rappelle).
Le principe du jeu est simple, et le jeu est facile à programmer. C'est pour cela que j'ai choisi d'en faire le premier TP du cours

PRÉPARATIFS ET CONSEILS

Le principe du programme

Avant toute chose, il faut que je vous explique en quoi va consister notre programme.
C'est un petit jeu que j'appelle "Plus ou moins".

Le principe est le suivant :

- L'ordinateur tire au sort un nombre entre 1 et 100
- Il vous demande de deviner le nombre. Vous rentrez donc un nombre entre 1 et 100
- L'ordinateur compare le nombre que vous avez rentré avec le nombre "mystère" qu'il a tiré au sort. Il vous dit

si le nombre mystère est supérieur ou inférieur à celui que vous avez entré
- Puis, l'ordinateur vous redemande le nombre.
- ... Et il vous indique si le nombre mystère est supérieur ou inférieur.
- Et ainsi de suite, jusqu'à ce que vous ayez trouvé le nombre mystère.

Le but du jeu, bien sûr, est de trouver le nombre mystère en un minimum de coups

Voici une capture d'écran d'une partie, c'est ce que vous devez arriver à faire :

Code : Console

```
Quel est le nombre ? 50
C'est plus !

Quel est le nombre ? 75
C'est plus !

Quel est le nombre ? 85
C'est moins !

Quel est le nombre ? 80
C'est moins !

Quel est le nombre ? 78
C'est plus !

Quel est le nombre ? 79
Bravo, vous avez trouve le nombre mystere !!!
```

Tirer un nombre au sort

Mais comment tirer un nombre au hasard ? Je ne sais pas le faire !

Certes
Nous ne savons pas générer un nombre aléatoire. Il faut dire que demander cela à l'ordinateur n'est pas simple : il sait bien faire des calculs, mais lui demander de choisir un nombre au hasard ça il sait pas faire !
En fait, pour "essayer" d'obtenir un nombre aléatoire, on doit faire faire des calculs complexes à l'ordinateur... ce qui revient au bout du compte à faire des calculs

Bon, on a donc 2 solutions :

- Soit on demande à l'utilisateur à rentrer le nombre mystère via un scanf d'abord. Ca implique qu'il y ait 2 joueurs : l'un rentre un nombre au hasard, et l'autre essaie de le deviner ensuite.
- Soit on tente le tout pour le tout, et on essaie quand même de générer un nombre aléatoire automatiquement. L'avantage est qu'on peut jouer tout seul du coup. Le défaut... est qu'il va falloir que je vous explique comment faire

Nous allons tenter la seconde solution, mais rien ne vous empêche de coder la première si vous voulez après

Pour générer un nombre aléatoire, on utilise la fonction rand().
Cette fonction génère un nombre au hasard. Mais nous, on veut que ce nombre soit compris entre 1 et 100 par exemple (si on ne connaît pas les limites ça va devenir trop compliqué)

Pour ce faire, on va utiliser la formule suivante :

Code : C

```c
srand(time(NULL));
nombreMystere = (rand() % (MAX - MIN + 1)) + MIN;
```

(je ne pouvais pas trop vous demander de la deviner)

La première ligne (avec srand) permet d'initialiser le générateur de nombre aléatoires. Oui c'est un peu compliqué je vous avais prévenu
nombreMystere est une variable qui contiendra le nombre au hasard.

L'instruction **srand** ne doit être exécutée qu'une seule fois (au début du programme). Il faut obligatoirement faire un srand une fois, et seulement une fois.
Vous pouvez ensuite faire autant de rand() que vous voulez pour générer des nombres aléatoires. Mais il ne faut PAS que l'ordinateur lise l'instruction **srand** 2 fois par programme, ne l'oubliez pas.

MAX et MIN sont des constantes, le premier est le nombre maximal (100) et le second le nombre minimal (1). Je vous recommande de définir ces constantes au début du programme, comme ceci :

Code : C

```c
const long MAX = 100, MIN = 1;
```

(pour un nombre aléatoire entre 1 et 100)

Les librairies à inclure

Pour que votre programme marche correctement, vous aurez besoin d'inclure 3 librairies : stdlib, stdio et time (la dernière sert pour les nombres aléatoires).
Votre programme devra donc commencer par :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

J'en ai assez dit !

Bon allez, j'arrête là parce que sinon je vais vous donner tout le code du programme si ça continue

Pour vous faire générer des nombres aléatoires, j'ai été obligé de vous donner des codes "tous prêts", sans vous expliquer totalement comment ils fonctionnaient. En général je n'aime pas faire ça mais là je n'ai pas vraiment le choix car ça compliquerait trop les choses pour le moment.
Soyez sûrs toutefois que par la suite vous apprendrez de nouvelles choses qui vous permettront de comprendre cela

Bref, vous en savez assez. Je vous ai expliqué le principe du programme, je vous ai fait une capture d'écran du programme au cours d'une partie.

Avec tout ça, vous êtes tout à fait capables d'écrire le programme

A vous de jouer !
Bonne chance !

CORRECTION !

Stop !
Je ramasse les copies

Alors, avez-vous réussi à coder le programme ?
Je veux pas vous mettre la pression mais... vous devriez
Cela n'est pas bien compliqué.

Je vais vous donner une correction (la mienne), mais il y a plusieurs bonnes façons de faire le programme. Si votre code source n'est pas identique au mien et que vous avez trouvé une autre façon de le faire, c'est bien aussi hein

La correction de "Plus ou Moins"

Code : C

```
/*

Plus ou Moins
-------------

Réalisé par M@teo21, pour les cours du Site du Zér0
www.siteduzero.com (cours de programmation en C / C++ pour débutants)

Création le : 20/12/2005

*/


#include <stdio.h>
#include <stdlib.h>
#include <time.h>


int main ( int argc, char** argv )
{
    long nombreMystere = 0, nombreEntre = 0;
    const long MAX = 100, MIN = 1;

    // Génération du nombre aléatoire

    srand(time(NULL));
    nombreMystere = (rand() % (MAX - MIN + 1)) + MIN;

    /* La boucle du programme. Elle se répète tant que l'utilisateur
    n'a pas trouvé le nombre mystère */

    do
    {
        // On demande le nombre
        printf("Quel est le nombre ? ");
        scanf("%ld", &nombreEntre);

        // On compare le nombre entré avec le nombre mystère

        if (nombreMystere > nombreEntre)
            printf("C'est plus !\n\n");
        else if (nombreMystere < nombreEntre)
            printf("C'est moins !\n\n");
        else
            printf ("Bravo, vous avez trouve le nombre mystere !!!\n\n");
    } while (nombreEntre != nombreMystere);

    system("PAUSE");
}
```

Exécutable et sources

Pour ceux qui le désirent, je mets à votre disposition en téléchargement l'exécutable du programme ainsi que les sources.

L'exécutable (.exe) est compilé pour Windows, donc si vous êtes sous un autre système d'exploitation il faudra recompiler le programme pour qu'il marche chez vous

Télécharger l'exécutable et les sources de "Plus ou Moins" (7 Ko)

Il y a deux dossiers, l'un avec l'exécutable (compilé sous Windows je le rappelle), et l'autre avec les sources.

Dans le cas de "Plus ou moins", les sources sont très simples : il y a juste un fichier main.c.
N'ouvrez pas le fichier main.c directement. Ouvrez d'abord votre ide favori (Dev, Visual...) et créez un nouveau projet de type console vide. Une fois que c'est fait, demander à **ajouter au projet le fichier main.c**.
Vous pourrez alors compiler le programme pour tester, et le modifier si vous le désirez

Explications

Je vais maintenant vous expliquer mon code, en commençant par le début.

Les directives de précompilateur

Ce sont les lignes commençant par # tout en haut. Elles incluent les librairies dont on a besoin.
Je vous les ai données tout à l'heure, donc si vous vous êtes plantés là c'est que vous êtes vraiment euh... pas doués

Les variables

On n'en a pas eu besoin de beaucoup.
Juste une pour le nombre entré par l'utilisateur (nombreEntre) et une autre qui retient le nombre aléatoire généré par l'ordinateur (nombreMystere).

J'ai aussi défini les constantes comme je vous l'ai dit au début de ce chapitre. L'avantage de définir les constantes en haut du programme, c'est que comme ça si vous voulez changer la difficulté (en mettant 1000 pour MAX par exemple) il suffit juste d'éditer cette ligne et de recompiler.

La boucle

J'ai choisi de faire une boucle do... while. En théorie, une boucle while simple aurait pu fonctionner aussi, mais j'ai trouvé qu'utiliser do... while était plus logique.

Pourquoi ?
Parce que, souvenez-vous, do... while est une boucle qui s'exécute **au moins une fois**. Et nous, on sait qu'on veut demander le nombre à l'utilisateur au moins une fois (il ne peut pas trouver le résultat en moins d'un coup, ou alors c'est qu'il est super fort)

A chaque passage dans la boucle, on redemande à l'utilisateur le nombre. On stocke le nombre qu'il propose dans nombreEntre.
Puis, on compare ce nombreEntre au nombreMystere. Il y a 3 possibilités :

- Le nombre mystère est supérieur au nombre entré, on indique donc l'indice "C'est plus !"
- Le nombre mystère est inférieur au nombre entré, on indique l'indice "C'est moins !"
- Et si le nombre mystère n'est ni supérieur ni inférieur ? Ben... c'est qu'il est égal forcément ! D'où le else. Dans ce cas, on affiche la phrase "Bravo vous avez trouvé !"

Il faut une condition pour la boucle. Celle-ci était facile à trouver : on continue la boucle **TANT QUE le nombre entré n'est pas égal au nombre mystère**.
La fois où le nombre est égal (c'est-à-dire quand on a trouvé), la boucle s'arrête. Le programme est alors terminé

IDÉES D'AMÉLIORATION

Non, vous ne croyiez tout de même pas que j'allais m'arrêter là ?
Les cours du Site du Zér0 c'est comme la mousse au chocolat, quand y'en a plus y'en a encore !

Je veux vous inciter à continuer à améliorer ce programme, pour vous entraîner. N'oubliez pas que c'est en vous entraînant comme ceci que vous progresserez ! Ceux qui lisent les cours d'une traite sans jamais faire de tests font une grosse erreur, je l'ai dit et je le redirai

Figurez-vous que j'ai une imagination débordante, et même sur un petit programme comme celui-ci je vois plein d'idées pour l'améliorer !
Attention : cette fois je ne vous fournis pas de correction, il faudra vous débrouiller tous seuls ! Si vous avez vraiment des problèmes, n'hésitez pas à aller faire un tour sur les forums du site pour poser vos questions

- **Faites un compteur de "coups"**. Ce compteur devra être une variable que vous incrémenterez à chaque fois que vous passez dans la boucle. Lorsque l'utilisateur a trouvé le nombre mystère, vous lui direz "*Bravo, vous avez trouvé le nombre mystère en 8 coups*" par exemple.
- Lorsque l'utilisateur aura trouvé le nombre mystère, le programme s'arrête. Pourquoi ne pas demander s'il veut faire **une autre partie** ?
  Si vous faites ça, il vous faudra faire une boucle qui englobera la quasi-totalité de votre programme. Cette boucle devra se répéter TANT QUE l'utilisateur n'a pas demandé à arrêter le programme. Je vous conseille de rajouter une variable booléenne "continuerPartie" initialisée à 1 au départ. Si l'utilisateur demande à arrêter le programme, vous mettrez la variable à 0 et le programme s'arrêtera.
- Implémentez **un mode 2 joueurs** ! Attention, je veux qu'on ait le choix entre un mode 1 joueur et un mode 2 joueurs !
  Vous devrez donc faire un menu au début de votre programme qui demande à l'utilisateur le mode de jeu qu'il veut faire.
  La seule chose qui changera entre les deux modes de jeu, c'est la génération du nombre mystère. Dans un cas ce sera un rand() comme on a vu, dans l'autre cas ça sera... un scanf
- Créez **plusieurs niveaux de difficulté**. Au début, faites un menu qui demande le niveau de difficulté. Par exemple :
  - 1 = entre 1 et 100
  - 2 = entre 1 et 1000
  - 3 = entre 1 et 10000

  Si vous faites ça, vous devrez changer votre constante MAX... Ben oui, ça ne peut plus être une constante si la valeur doit changer au cours du programme ! Renommez donc cette variable en nombreMaximum (vous prendrez soin d'enlever le mot-clé "const" sinon ça sera toujours une constante !). La valeur de cette variable dépendra du niveau qu'on aura choisi.

Voilà, ça devrait vous occuper un petit bout de temps

Amusez-vous bien et n'hésitez pas à chercher d'autres idées pour améliorer ce "Plus ou Moins", je suis sûr qu'il y en a !

N'oubliez pas que les forums sont à votre disposition si vous avez des questions Voilà notre premier TP s'achève ici
J'espère que vous l'avez apprécié et que vous allez tenter de faire un maximum de modifications tous seuls comme des grands, car c'est réellement ce qui vous fera progresser.

Au fur et à mesure du cours, les TP deviendront bien sûr de plus en plus intéressants, et vous vous étonnerez dans quelques temps de ce que vous arriverez à faire !

Les fonctions

Nous terminerons la partie I du cours ("Les bases") par cette notion fondamentale que sont les fonctions en langage C.

Tous les programmes en C se basent sur le principe que je vais vous expliquer dans ce chapitre.
Nous allons apprendre à structurer nos programmes en petits bouts... un peu comme si on faisait des legos

Tous les gros programmes en C sont en fait des assemblages de petits bouts de code, et ces petits bouts de code sont

justement ce qu'on appelle... **des fonctions**

Créer et appeler une fonction

Nous avons vu dans les tous premiers chapitres qu'un programme en C commençait par une fonction appelée "main".
Je vous avais d'ailleurs même fait un schéma récapitulatif, pour vous rappeler quelques mots de vocabulaire.
Attendez que je retrouve ce schéma

** va fouiller dans les archives poussièreuses **

Ah je l'ai
Souvenirs souvenirs :

```
#include <stdio.h>      } Directives de préprocesseur
#include <stdlib.h>

int main(int argc, char *argv[])
{
                                          } Fonction
  system("PAUSE");      } Instructions
  return 0;
}
```

C'était au tout début hein
En haut, les directives de préprocesseur (un nom barbare sur lequel on reviendra d'ailleurs). Ces directives sont faciles à identifier : elles commencent par un # et sont généralement mises tout en haut des fichiers sources.

Puis en-dessous, il y avait ce que j'avais déjà appelé "une fonction". Ici, sur mon schéma, vous voyez une fonction "main" (pas trop remplie il faut le reconnaître)
Je vous avais dit qu'un programme en langage C **commençait par la fonction main**. Que je vous rassure, c'est toujours vrai

Seulement, jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis. Revoyez vos codes sources et vous le verrez : nous sommes toujours restés à l'intérieur des accolades de la fonction main.

Eh bien, c'est mal d'avoir fait ça ?

Non ce n'est pas "mal", mais ce n'est pas ce que les programmeurs en C font dans la réalité.
Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction "main". Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes, mais imaginez des plus gros programmes qui font des milliers de lignes de code ! Si tout était concentré dans la fonction main, bonjour le bordel

Nous allons donc maintenant **apprendre à nous organiser**. Nous allons en fait découper nos programmes en petits bouts (souvenez-vous de l'image des legos que je vous ai donnée tout à l'heure).
Chaque "petit bout de programme" sera ce qu'on appelle une fonction.

Quel est le but d'une fonction ?

Une fonction exécute des actions et renvoie un résultat. C'est un **morceau de code** qui sert à faire quelque chose de précis.
On dit qu'une fonction possède une entrée et une sortie. Schématiquement, ça donne quelque chose comme ça :

Lorsqu'on appelle une fonction, il y a 3 étapes :

1. **L'entrée**: on fait "rentrer" des informations dans la fonction (en lui donnant des informations avec lesquelles travailler)
2. **Les calculs** : grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie** : une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore *le retour*.

Concrètement, on peut imaginer par exemple une fonction appelée "triple" qui calcule le triple du nombre qu'on lui donne (en le multipliant par 3) :

La fonction « triple » multiplie le nombre en entrée par 3

Bien entendu, les fonctions seront en général plus compliquées
Le but des fonctions est donc de simplifier le code source, pour ne pas avoir à retaper le même code plusieurs fois d'affilée.

Rêvez un peu : plus tard, nous créerons par exemple une fonction "afficherFenetre" qui ouvrira une fenêtre à l'écran.
Une fois la fonction écrite (c'est l'étape la plus difficile), on n'aura plus qu'à dire "Hep toi la fonction afficherFenetre, ouvre-moi une fenêtre !"
On pourra aussi écrire une fonction "deplacerPersonnage" dont le but sera de déplacer le personnage d'un jeu à l'écran etc etc

Schéma d'une fonction

Vous avez déjà eu un aperçu de comment est faite une fonction avec la fonction main.
Cependant pour bien que vous compreniez il va falloir que je vous montre quand même **comment on construit une fonction**.

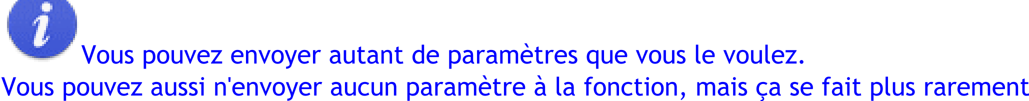
Voici le schéma d'une fonction, à retenir par coeur :

Code : C

```
type nomFonction(parametres)
{
    // Insérez vos instructions ici
}
```

Vous reconnaissez là un peu la forme de la fonction main.
Voici ce qu'il faut savoir sur ce schéma :

- **type *(correspond à la sortie)*** : c'est le type de la fonction. Comme les variables, les fonctions ont un type. Ce type dépend du résultat que la fonction renvoie : si la fonction renvoie un nombre décimal, vous mettrez sûrement double, si elle renvoie un entier vous mettrez int ou long par exemple. Mais il est aussi possible de créer des fonctions qui ne renvoient rien !
  Il y a donc 2 sortes de fonctions :
  - Les fonctions qui **renvoient une valeur** : on leur met un des types que l'on connaît (char, int, double...)
  - Les fonctions qui **ne renvoient pas de valeur** : on leur met un type spécial "void" (qui signifie "vide").
- **nomFonction** : c'est le nom de votre fonction. Vous pouvez appeler votre fonction comme vous voulez, du temps que vous respectez les mêmes règles que pour les variables (pas d'accents, pas d'espaces etc).
- **parametres *(correspond à l'entrée)*** : entre parenthèses, vous pouvez envoyer des paramètres à la fonction. Ce sont des valeurs avec lesquelles la fonction va travailler.
  Par exemple, pour une fonction "triple", vous envoyez un nombre en paramètre. La fonction "récupère" ce nombre et en calcule le triple, en le multipliant par 3. Elle renvoie ensuite le résultat de ses calculs.


Vous pouvez envoyer autant de paramètres que vous le voulez.
Vous pouvez aussi n'envoyer aucun paramètre à la fonction, mais ça se fait plus rarement.

- Ensuite vous avez les **accolades** qui indiquent le début et la fin de la fonction. A l'intérieur de ces accolades vous mettrez les instructions que vous voulez. Pour la fonction triple, il faudra taper des instructions qui multiplient par 3 le nombre reçu en entrée.

Une fonction, c'est donc un mécanisme qui reçoit des valeurs en **entrée** (les paramètres) et qui renvoie un résultat en **sortie**.

Créer une fonction

Voyons voir un exemple pratique sans plus tarder : la fameuse fonction "triple" dont je vous parle depuis tout à l'heure. On va dire que cette fonction reçoit un nombre entier de type long et qu'elle renvoie un nombre entier aussi de type long. Cette fonction calcule le triple du nombre qu'on lui donne :

Code : C

```
long triple(long nombre)
{
    long resultat = 0;

    resultat = 3 * nombre;  // On multiplie le nombre qu'on nous a transmis par 3
    return resultat;       // On retourne la variable resultat qui vaut le triple de
nombre
}
```

Voilà notre première fonction
Une première chose importante : comme vous le voyez, la fonction est de type long. Elle doit donc renvoyer une valeur de type long.

Entre les parenthèses, vous avez les variables que la fonction **reçoit**. Ici, notre fonction triple reçoit une variable de type long appelée "nombre".

La ligne qui indique de "renvoyer une valeur" est celle qui contient le "return". Cette ligne se trouve généralement à la fin de la fonction, après les calculs.
return resultat; dit à la fonction : arrête-toi là et renvoie le nombre "resultat". Cette variable "resultat" DOIT être de type long, car la fonction renvoie un long comme on l'a dit plus haut

La variable resultat est déclarée (= créée) dans la fonction "triple". Cela signifie qu'elle n'est utilisable que dans cette fonction, et pas dans une autre comme la fonction "main" par exemple. C'est donc une variable propre à la fonction "triple".

Mais est-ce la façon la plus courte de faire notre fonction triple ?
Non, on peut faire ça en une ligne en fait

Code : C

```
long triple(long nombre)
{
    return 3 * nombre;
}
```

Cette fonction fait exactement la même chose que la fonction de tout à l'heure, elle est juste plus rapide à écrire

Généralement, vos fonctions contiendront plusieurs variables pour effectuer leurs calculs et leurs opérations, rares seront les fonctions aussi courtes que "triple"

Plusieurs paramètres, aucun paramètre

Plusieurs paramètres

Notre fonction "triple" contient un paramètre, mais il est possible de créer des fonctions prenant plusieurs paramètres.
Par exemple, une fonction addition qui additionne deux nombres a et b :

Code : C

```
long addition(long a, long b)
{
    return a + b;
}
```

Il suffit de séparer les différents paramètres par une virgule comme vous le voyez

Aucun paramètre

Certaines fonctions, plus rares, ne prennent aucun paramètre en entrée. Ces fonctions feront généralement toujours la même chose. En effet, si elles n'ont pas de nombres sur lesquels travailler, vos fonctions serviront juste à effectuer certaines actions, comme afficher du texte à l'écran (et encore, ce sera toujours le même texte !)

Imaginons une fonction "bonjour" qui affiche juste bonjour à l'écran :

Code : C

```
void bonjour()
{
    printf("Bonjour");
}
```

Je n'ai rien mis entre parenthèses car la fonction ne prend aucun paramètre.
De plus, j'ai mis le type "void" dont je vous ai parlé plus haut.

En effet, comme vous le voyez ma fonction n'a pas non plus de "return". Elle ne retourne rien. Une fonction qui ne retourne rien est de type void.

Appeler une fonction

On va maintenant tester un code source pour pratiquer un peu avec ce qu'on vient d'apprendre.
Nous allons utiliser notre fonction "triple" (décidemment je l'aime bien) pour calculer le triple d'un nombre.

Pour le moment, je vous demande d'écrire la fonction "triple" AVANT la fonction main. Si vous la mettez après, ça ne marchera pas. Je vous expliquerai pourquoi par la suite

Code : C

```
#include <stdio.h>
#include <stdlib.h>

long triple(long nombre)
{
    return 3 * nombre;
}

int main(int argc, char *argv[])
{
    long nombreEntre = 0, nombreTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%ld", &nombreEntre);

    nombreTriple = triple(nombreEntre);
    printf("Le triple de ce nombre est %ld\n", nombreTriple);

    system("PAUSE");
    return 0;
}
```

Notre programme commence par la fonction main comme vous le savez.
On demande à l'utilisateur de rentrer un nombre. On envoie ce nombre qu'il a rentré à la fonction triple, et on récupère le résultat dans la variable nombreTriple. Regardez en particulier cette ligne, c'est la plus intéressante car c'est l'*appel de la fonction* :

Code : C

```
nombreTriple = triple(nombreEntre);
```

Entre parenthèses, on envoie une variable en **entrée** à la fonction triple, c'est le nombre sur lequel elle va travailler. Cette fonction renvoie une valeur, valeur qu'on récupère dans la variable nombreTriple. On demande donc à l'ordinateur dans cette ligne : "Demande à la fonction triple de me calculer le triple de nombreEntre, et stocke le résultat dans la variable nombreTriple".

Les mêmes explications sous forme de schéma

Vous avez encore du mal à comprendre comment ça fonctionne concrètement ?
Pas de panique ! Je suis sûr que vous allez comprendre avec mes schémas

Ce premier schéma vous explique dans quel ordre le code est lu. Commencez donc par lire la ligne numérotée "1", puis "2", puis "3" (bon vous avez compris je crois)

```c
#include <stdio.h>
#include <stdlib.h>

long triple(long nombre) 6/ On saute à la fonction triple et on récupère un paramètre (nombre)
{
    return 3 * nombre; 7/ On fait des calculs sur le nombre et on termine la fonction (c'est ce que return veut dire : il renvoie un résultat et termine la fonction).
}

int main(int argc, char *argv[]) 1/ Le programme commence par la fonction main (cette ligne)
{
    long nombreEntre = 0, nombreTriple = 0; 2/ Il lit les instructions dans la fonctions une par dans l'ordre

    printf("Entrez un nombre... "); 3/ Il lit l'instruction suivante et fait ce qui est demandé (pri
    scanf("%ld", &nombreEntre); 4/ Pareil, il lit l'instruction suivante et fait ce qui est demandé (

    nombreTriple = triple(nombreEntre); 5/ Il lit l'instruction. Ah ! On appelle la fonction triple, doit donc "sauter" à la ligne de la fonction triple plus
    printf("Le triple de ce nombre est %ld\n", nombreTriple); 8/ On retourne dans le main l'instruction suivante

    system("PAUSE"); 9/ Encore une instruction...
    return 0; 10/ Un return ! La fonction main se termine, et donc le programme est terminé.
}
```

Si vous avez compris dans quel ordre l'ordinateur lisait les instructions, vous avez déjà compris le principal

Maintenant, il faut bien comprendre qu'une fonction reçoit des paramètres en entrée et renvoie une valeur en sortie.

2/ La fonction triple retourne (return) une valeur. Cette valeur, c'est 3x le nombre qu'on lui a envoyé.

Cette valeur de retour est stockée dans la variable nombreTriple de la fonction main. Le signe "=" permet donc de dire : "Envoie le résultat de la fonction dans cette variable".

```c
#include <stdio.h>
#include <stdlib.h>

long triple(long nombre)
{
    return 3 * nombre;
}

int main(int argc, char *argv[])
{
    long nombreEntre = 0, nombreTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%ld", &nombreEntre);

    nombreTriple = triple(nombreEntre);
    printf("Le triple de ce nombre est %ld\n", nombreTriple);

    system("PAUSE");
    return 0;
}
```

1/ La variable r est envoyée er à la fonction tr fonction triple cette variable autre variable nom "nombre"

Note : on aura mettre le mêm variable dans l fonctions, ça a fonctionné. Ma pas obligé de même nom.

Note : ce n'est pas tout le temps le cas comme ça pour toutes les fonctions. Parfois, une fonction ne prend aucun paramètre en entrée, ou au contraire elle en prend plusieurs (je vous ai expliqué ça un peu plus haut).
De même, parfois une fonction renvoie une valeur, parfois elle ne renvoie rien (dans ce cas il n'y a pas de *return*).

Testons ce programme

Voici un exemple d'utilisation du programme (y'a rien de bien extraordinaire car c'est une fonction toute bête hein) :

Code : Console

```
Entrez un nombre... 10
Le triple de ce nombre est 30
```

Vous n'êtes pas obligés de stocker le résultat d'une fonction dans une variable ! Vous pouvez directement envoyer le résultat de la fonction triple à une autre fonction, comme si `triple(nombreEntre)` était une variable.

Regardez bien ceci, c'est le même code mais y'a un changement au niveau du dernier printf, et on n'a pas déclaré de variable nombreTriple car on ne s'en sert plus :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

long triple(long nombre)
{
    return 3 * nombre;
}

int main(int argc, char *argv[])
{
    long nombreEntre = 0;

    printf("Entrez un nombre... ");
    scanf("%ld", &nombreEntre);

    // Le résultat de la fonction est directement envoyé au printf et n'est pas stocké
dans une variable
    printf("Le triple de ce nombre est %ld\n", triple(nombreEntre));

    system("PAUSE");
    return 0;
}
```

Comme vous le voyez, `triple(nombreEntre)` est directement envoyé au printf.
Que fait l'ordinateur quand il tombe sur cette ligne ?

C'est très simple. Il voit que la ligne commence par printf, il va donc appeler la fonction printf.
Il envoie à la fonction printf tous les paramètres qu'on lui donne. Le premier paramètre est le texte à afficher, et le second est un nombre.
Votre ordinateur voit que pour envoyer ce nombre à la fonction printf il doit d'abord appeler la fonction triple. C'est ce qu'il fait : il appelle triple, il effectue les calculs de triple, et une fois qu'il a le résultat il l'envoie directement dans la fonction printf !

C'est un peu une imbrication de fonctions

Et le top, c'est qu'une fonction peut en appeler une autre à son tour !
Notre fonction triple pourrait appeler une autre fonction, qui elle-même appellerait une autre fonction etc... C'est ça le principe de la programmation en C ! Tout est combiné, comme dans un jeu de Legos

Au final, le plus dur sera d'écrire vos fonctions. Une fois que vous les aurez écrites, vous n'aurez plus qu'à appeler les fonctions sans vous soucier des calculs qu'elles peuvent bien faire à l'intérieur. Ca va permettre de simplifier considérablement l'écriture de nos programmes, et ça croyez-moi on en aura bien besoin !

Plein d'exemples pour bien comprendre

Vous avez dû vous en rendre compte : je suis un maniaque des exemples.
La théorie c'est bien, mais si on ne fait que ça on risque de ne pas retenir grand chose, et surtout ne pas comprendre

comment s'en servir, ce qui serait un peu dommage

Je vais donc maintenant vous montrer plusieurs exemples d'utilisation de fonctions, pour que vous ayez une idée de leur intérêt. Je vais m'efforcer de faire des cas différents à chaque fois, pour que vous puissiez avoir des exemples de tous les types de fonctions qui peuvent exister.

Je ne vous apprendrai pas grand chose de nouveau, mais ça sera l'occasion de voir des exemples pratiques. Si vous avez déjà compris tout ce que j'ai expliqué avant, c'est très bien et normalement aucun des exemples qui vont suivre ne devrait vous surprendre

Conversion euros / francs

On commence par une fonction très similaire à "triple", qui a quand même un minimum d'intérêt cette fois : une fonction qui convertit les euros en francs.
Pour ceux d'entre vous qui ne connaîtraient pas ces monnaies (il n'y a pas que des français sur le Site du Zér0 hein) sachez que :

1 euro = 6.55957 francs

On va créer une fonction appelée conversion.
Cette fonction prend une variable en entrée de type double et retourne une sortie de type double (on va forcément manipuler des chiffres décimaux !).

Code : C

```
double conversion(double euros)
{
    double francs = 0;

    francs = 6.55957 * euros;
    return francs;
}

int main(int argc, char *argv[])
{
    printf("10 euros = %lfF\n", conversion(10));
    printf("50 euros = %lfF\n", conversion(50));
    printf("100 euros = %lfF\n", conversion(100));
    printf("200 euros = %lfF\n", conversion(200));

    system("PAUSE");
    return 0;
}
```

Code : Console

```
10 euros = 65.595700F
50 euros = 327.978500F
100 euros = 655.957000F
200 euros = 1311.914000F
```

Il n'y a pas grand chose de différent par rapport à la fonction triple je vous avais prévenu
D'ailleurs, ma fonction conversion est un peu longue et pourrait être raccourcie en une ligne, je vous laisse le faire je vous ai déjà expliqué comment faire plus haut.

Dans la fonction main, j'ai fait exprès de faire plusieurs printf pour vous montrer l'intérêt d'avoir une fonction. Pour obtenir la valeur de 50 euros, je n'ai qu'à écrire `conversion(50)`. Et si je veux avoir la conversion en francs de 100 euros, j'ai juste besoin de changer le paramètre que j'envoie à la fonction (100 au lieu de 50).

A vous de jouer ! Ecrivez une seconde fonction (toujours avant la fonction main) qui fera elle la conversion inverse : Francs => Euros. Pas bien difficile hein, y'a juste un signe d'opération à changer

La punition

On va maintenant s'intéresser à une fonction qui ne renvoie rien (pas de sortie).
C'est une fonction qui affiche le même message à l'écran autant de fois qu'on lui demande. Cette fonction prend un paramètre en entrée : le nombre de fois où il faut afficher la punition.

Code : C

```
void punition(long nombreDeLignes)
{
    long i;

    for (i = 0 ; i < nombreDeLignes ; i++)
    {
        printf("Je ne dois pas recopier mon voisin\n");
    }
}

int main(int argc, char *argv[])
{
    punition(10);

    system("PAUSE");
    return 0;
}
```

Code : Console

```
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
```

On a ici affaire à une fonction qui ne renvoie aucune valeur. Cette fonction se contente juste d'effectuer des actions (ici, elle affiche des messages à l'écran).
Une fonction qui ne renvoie aucune valeur est de type "void", c'est pour cela qu'on a écrit void.

A part ça, rien de bien différent

Il aurait été bien plus intéressant de créer une fonction "punition" qui s'adapte à n'importe quelle punition. On lui aurait envoyé 2 paramètres : le texte à répéter, et le nombre de fois qu'il doit être répété. Le problème, c'est qu'on ne sait pas encore gérer le texte en C (au cas où vous auriez pas les yeux très ouverts, je vous rappelle qu'on fait que manipuler des variables contenant des nombres depuis le début)
D'ailleurs à ce sujet, je vous annonce que nous ne tarderons pas à apprendre à utiliser des variables qui retiennent du texte. C'est que c'est plus compliqué qu'il n'y paraît, et on ne pouvait pas l'apprendre dès le début du cours

Aire d'un rectangle

L'aire d'un rectangle est facile à calculer : largeur * hauteur.
Notre fonction aireRectangle va prendre 2 paramètres, la largeur et la hauteur. Elle renverra l'aire :

Code : C

```
double aireRectangle(double largeur, double hauteur)
{
    return largeur * hauteur;
}

int main(int argc, char *argv[])
{
    printf("Rectangle de largeur 5 et hauteur 10. Aire = %lf\n", aireRectangle(5, 10));
    printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %lf\n", aireRectangle(2.5,
3.5));
    printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %lf\n", aireRectangle(4.2,
9.7));

    system("PAUSE");
    return 0;
}
```

Code : Console

```
Rectangle de largeur 5 et hauteur 10. Aire = 50.000000
Rectangle de largeur 2.5 et hauteur 3.5. Aire = 8.750000
Rectangle de largeur 4.2 et hauteur 9.7. Aire = 40.740000
```

Pourrait-on afficher directement la largeur, la hauteur et l'aire dans la fonction ?

Bien sûr !
Dans ce cas, la fonction ne renverrait plus rien, elle se contenterait de calculer l'aire et de l'afficher immédiatement.

Code : C

```
void aireRectangle(double largeur, double hauteur)
{
    double aire = 0;

    aire = largeur * hauteur;
    printf("Rectangle de largeur %lf et hauteur %lf. Aire = %lf\n", largeur, hauteur,
aire);
}

int main(int argc, char *argv[])
{
    aireRectangle(5, 10);
    aireRectangle(2.5, 3.5);
    aireRectangle(4.2, 9.7);

    system("PAUSE");
    return 0;
}
```

Comme vous le voyez, le printf est *à l'intérieur* de la fonction aireRectangle et fait le même affichage que tout à l'heure. C'est juste une façon différente de procéder

Un menu

Ce code est plus intéressant et concret. On crée une fonction menu() qui ne prend aucun paramètre en entrée. Cette fonction se contente d'afficher le menu, et demande à l'utilisateur de faire un choix.
La fonction renvoie le choix de l'utilisateur.

Code : C

```
long menu()
{
    long choix = 0;

    while (choix < 1 || choix > 4)
    {
        printf("Menu :\n");
        printf("1 : Poulet de dinde aux escargots rotis a la sauce bearnaise\n");
        printf("2 : Concombres sucres a la sauce de myrtilles enrobee de chocolat\n");
        printf("3 : Escalope de kangourou saignante et sa gelee aux fraises poivree\n");
        printf("4 : La surprise du Chef (j'en salive d'avance...)\n");
        printf("Votre choix ? ");
        scanf("%ld", &choix);
    }

    return choix;
}

int main(int argc, char *argv[])
{
    switch (menu())
    {
        case 1:
            printf("Vous avez pris le poulet\n");
            break;
        case 2:
            printf("Vous avez pris les concombres\n");
            break;
        case 3:
            printf("Vous avez pris l'escalope\n");
            break;
        case 4:
            printf("Vous avez pris la surprise du Chef. Vous etes un sacre aventurier
dites donc !\n");
            break;
    }

    system("PAUSE");
    return 0;
}
```

J'en ai profité pour améliorer le menu (par rapport à ce qu'on faisait habituellement) : la fonction menu réaffiche le menu tant que l'utilisateur n'a pas rentré un nombre compris entre 1 et 4. Comme ça, aucun risque que la fonction renvoie un nombre qui ne figure pas au menu !

Dans le main, vous avez vu qu'on fait un `switch(menu())`. Une fois que la fonction menu() est terminée, elle renvoie le choix de l'utilisateur directement dans le switch. C'est pratique et rapide comme méthode

A vous de jouer ! Le code est encore améliorable : on pourrait afficher un message d'erreur si l'utilisateur rentre un mauvais nombre plutôt que de bêtement réafficher le menu Les fonctions deviendront particulièrement intéressantes lorsque nous ferons un TP qui en utilise. Nous profiterons alors de cette possibilité qu'on a de découper un programme en plusieurs fonctions.

En attendant, vous devez vous entraîner à créer des programmes avec des fonctions. Même si leur intérêt reste limité, même si ça sert à rien et c'est tout nul pour le moment, ça vous sera bénéfique par la suite et vous ne le regretterez pas

Un petit exercice avant de finir

Vous vous souvenez du TP "Plus ou Moins" ? J'espère que vous avez pas déjà oublié
Vous allez le modifier pour utiliser des fonctions. Voici la fonction main à utiliser :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main ( int argc, char** argv )
{
    long nombreMystere = 0, nombreEntre = 0;
    const long MAX = 100, MIN = 1;

    // Génération du nombre aléatoire
    nombreMystere = genereNombre(MIN, MAX);

    /* La boucle du programme. Elle se répète tant que l'utilisateur
    n'a pas trouvé le nombre mystère */

    do
    {
        // On demande le nombre
        printf("Quel est le nombre ? ");
        scanf("%ld", &nombreEntre);

        // On compare le nombre entré avec le nombre mystère
        compareNombres(nombreEntre, nombreMystere);

    } while (nombreEntre != nombreMystere);


    system("PAUSE");
}
```

A vous de créer les 2 fonctions qu'elle utilise : genereNombre (qui génère un nombre aléatoire compris entre MIN et MAX) et compareNombres qui compare le nombre entré au nombre mystère et affiche si c'est plus, si c'est moins, ou si c'est le bon résultat

Accrochez-vous !

Ne vous pressez pas trop pour aller dans la partie II. Je vous y expliquerai (entre autres) un concept un peu difficile : les pointeurs. Mieux vaut être à l'aise avec les fonctions avant d'y aller
Cependant, comme c'est un passage obligé, il faudra bien que vous le lisiez à un moment ou à un autre

Alors n'abandonnez pas, vous passerez vos plus durs moments avec le C dans la partie II, mais je vous promets que la récompense sera à la hauteur ensuite ! En effet, nous apprendrons dans la partie III à créer des jeux, à ouvrir des fenêtres, gérer le clavier, la souris, le joystick, le son etc etc

Ah ça motive un peu plus d'un coup hein ? Ainsi s'achève la première partie de ce cours de C / C++ pour débutants
Nous y avons appris **les principes de base** de la programmation en C, mais nous sommes encore très loin d'avoir tout vu !

Les choses sérieuses commenceront dans la partie II

PARTIE 2 : [LANGAGE C] TECHNIQUES AVANCÉES

Cette seconde partie introduit une notion très importante du langage C : **les pointeurs**. Nous verrons ce que c'est et tout ce qui en découle, tout ce qu'on peut faire avec.

Je ne vous le cache pas, et vous vous en doutiez sûrement, la partie II est à un cran de difficulté supérieur.
Là encore, je fais mon maximum pour tout vous expliquer le plus simplement possible
Lorsque vous serez arrivés à la fin de cette partie, vous serez capables de vous débrouiller dans la plupart des programmes écrits en C. Dans la partie suivante nous verrons alors comment ouvrir une fenêtre, créer des jeux 2D,

jouer du son etc.

Accrochez votre ceinture quand même, parce que ça va secouer un tantinet

La programmation modulaire

Ce premier chapitre de la partie II est la suite directe du chapitre sur les fonctions qu'on a vu dans la partie I.

Vous savez désormais qu'un vrai programme en C est composé de plein de fonctions. Chaque fonction sert à faire un travail précis et renvoie généralement un résultat. C'est en assemblant toutes ces fonctions entre elles que l'on parvient à créer n'importe quel programme

Seulement jusqu'ici nous n'avons travaillé que dans un seul fichier appelé main.c. Pour le moment c'était acceptable car nos programmes étaient tous petits, mais bientôt vos programmes vont être composés de dizaines, que dis-je de centaines de fonctions, et si vous les mettez tous dans un même fichier celui-ci va finir par être super long !
C'est pour cela que l'on a inventé ce qu'on appelle la **programmation modulaire**. Le principe est tout bête : plutôt que de mettre tout le code de votre programme dans un seul fichier (main.c), nous le "séparons" en plusieurs petits fichiers.

ATTENTION ATTENTION : à partir de la partie II, je ne mets plus l'instruction `system("PAUSE");` à la fin du main(). Rajoutez-la si vous en avez besoin, c'est-à-dire si votre programme s'ouvre et se ferme à la vitesse de l'éclair. Cela devrait être votre cas si vous utilisez Dev-C++.
A partir de ce niveau, je recommande de passer **à l'IDE Code::Blocks** plutôt que Dev-C++ (revoyez le chapitre 2 du cours au besoin). Code::Blocks est plus à jour que Dev-C++ et est plus intelligent notamment car il ne nécessite pas de mettre l'instruction `system("PAUSE");` à la fin du main().

Les prototypes

Jusqu'ici, je vous ai demandé de placer votre fonction *avant* la fonction main.
Pourquoi ?

Parce que l'ordre a une réelle importance ici : si vous mettez votre fonction avant le main dans votre code source, votre ordinateur l'aura lue et la connaîtra. Lorsque vous ferez un appel à la fonction dans le main, l'ordinateur connaîtra la fonction et saura où aller la chercher.
Si vous mettez votre fonction après le main, ça ne marchera pas car l'ordinateur ne connaîtra pas encore la fonction.
Essayez vous verrez

Mais... C'est un peu nul non ?

Tout à fait d'accord avec vous
Mais rassurez-vous, les programmeurs s'en sont rendu compte avant vous et ont prévu le coup

Grâce à ce que je vais vous apprendre maintenant, vous pourrez mettre vos fonctions dans n'importe quel ordre dans le code source. C'est mieux de ne pas avoir à s'en soucier, croyez-moi

Le prototype pour annoncer une fonction

Nous allons "annoncer" nos fonctions à l'ordinateur en écrivant ce qu'on appelle des **prototypes**. Ne soyez pas intimidés par ce nom high-tech, ça cache en fait quelque chose de tout bête

Regardez la première ligne de notre fonction aireRectangle :

Code : C

```
double aireRectangle(double largeur, double hauteur)
{
    return largeur * hauteur;
}
```

Copiez la première ligne (`double aireRectangle...`) tout en haut de votre fichier source (juste après les #include).
Rajoutez un point-virgule à la fin de cette nouvelle ligne.
Et voilà ! Maintenant vous pouvez mettre votre fonction aireRectangle après la fonction main si vous le voulez

Vous devriez avoir le code suivant sous les yeux :

Code : C

```
#include <stdio.h>
#include <stdlib.h>

// La ligne suivante est le prototype de la fonction aireRectangle :
double aireRectangle(double largeur, double hauteur);

int main(int argc, char *argv[])
{
    printf("Rectangle de largeur 5 et hauteur 10. Aire = %lf\n", aireRectangle(5, 10));
    printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %lf\n", aireRectangle(2.5,
3.5));
    printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %lf\n", aireRectangle(4.2,
9.7));

    return 0;
}

// Notre fonction aireRectangle peut maintenant être mise n'importe où dans le code
source :
double aireRectangle(double largeur, double hauteur)
{
    return largeur * hauteur;
}
```

Ce qui a changé ici, c'est l'ajout du prototype en haut du code source.
Un prototype, c'est en fait une indication pour l'ordinateur. Cela lui indique qu'il existe une fonction appelée aireRectangle qui prend tels paramètres en entrée et renvoie une sortie du type que vous indiquez.
Ca permet à l'ordinateur de s'organiser.

Grâce à cette ligne, vous pouvez maintenant mettre vos fonctions dans n'importe quel ordre sans vous prendre la tête

Ecrivez toujours le prototype de vos fonctions. Vos programmes ne vont pas tarder à se complexifier et à utiliser plein de fonctions : mieux vaut prendre dès maintenant la bonne habitude de mettre un prototype pour chacune de vos fonctions

Comme vous le voyez, la fonction main n'a pas de prototype. En fait, c'est <u>la seule</u> qui n'en nécessite pas, parce que l'ordinateur la connaît (c'est toujours la même pour tous les programmes, alors il peut bien la connaître à force)

Pour être tout à fait exact, il faut savoir que dans la ligne du prototype il est *facultatif* d'écrire les noms de variables en entrée. L'ordinateur a juste besoin de connaître les types des variables.

On aurait donc pu simplement écrire :

Code : C

```c
double aireRectangle(double, double);
```

Toutefois, l'autre méthode que je vous ai montrée tout à l'heure fonctionne aussi bien. L'avantage avec ma méthode c'est que vous avez juste besoin de copier-coller la première ligne de la fonction et de rajouter un point-virgule. Ca va plus vite

N'oubliez JAMAIS de mettre un point-virgule à la fin d'un prototype. C'est ce qui permet à l'ordinateur de différencier un prototype du véritable début d'une fonction.
Si vous ne le faites pas, vous risquez d'avoir des erreurs incompréhensibles lors de la compilation

Les headers

Jusqu'ici, nous n'avions qu'un seul fichier source dans notre projet. Ce fichier source, je vous avais demandé de l'appeler main.c

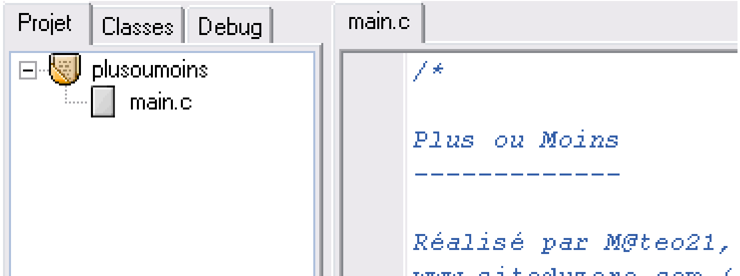
Plusieurs fichiers par projet

Dans la pratique, vos programmes ne seront pas tous écrits dans ce même fichier main.c. Bien sûr, c'est *possible* de le faire, mais ce n'est jamais très pratique se ballader dans un fichier de 10000 lignes (enfin personnellement je trouve).
C'est pour cela qu'en général on crée plusieurs fichiers par projet.

Euh c'est quoi un projet ?

Non, vous avez pas déjà oublié ?
Bon allez je vous le réexplique, parce qu'il est important qu'on soit bien d'accord entre nous là

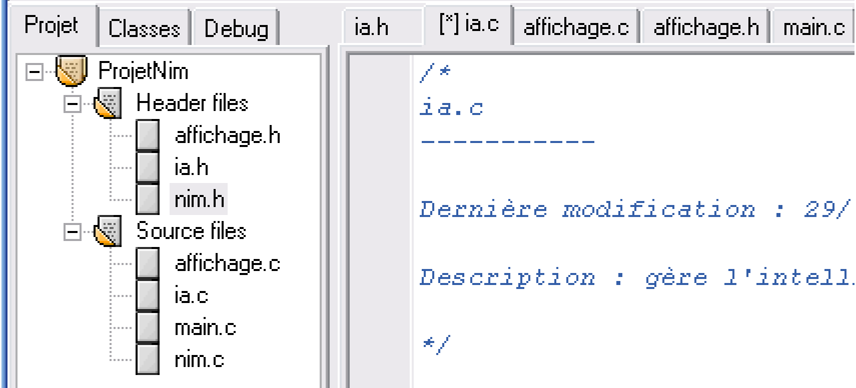
Un projet, c'est l'ensemble des fichiers source de votre programme.
Pour le moment, nos projets n'étaient composés que d'un fichier source. Regardez dans votre IDE (généralement c'est sur la gauche) :



Comme vous pouvez le voir sur cette capture d'écran à gauche, notre projet "plusoumoins" n'était composé que d'un fichier main.c.
Ca c'était parce que c'était notre premier TP et qu'il était tout bête

Laissez-moi maintenant vous montrer un vrai projet que j'avais réalisé il y a quelques temps pour un devoir d'école

(un jeu d'allumettes) :



Comme vous le voyez, il y a plusieurs fichiers. Un vrai projet ressemblera à ça : vous verrez plusieurs fichiers dans la colonne de gauche.
En fait, c'était un "assez petit" projet ça encore. Les gros programmes du commerce ont sûrement beaucoup plus de fichiers que ça, mais c'était pour vous donner une idée

Vous reconnaissez dans la liste le fichier main.c : c'est celui qui contient la fonction main. En général dans mes programmes, je ne mets que le main dans main.c (mais ce n'est pas du tout une obligation, chacun s'organise comme il veut !)

Mais pourquoi avoir créé plusieurs fichiers ? Et comment je sais combien de fichiers je dois créer pour mon projet ?

Ca c'est vous qui choisissez
En général, on regroupe dans un même fichier des fonctions par thème. Ainsi, dans le fichier affichage.c j'ai regroupé toutes les fonctions gérant l'affichage à l'écran, dans le fichier ia.c j'ai regroupé toutes les fonctions gérant l'intelligence artificielle de l'ordinateur etc etc...

Pour la petite histoire, j'ai connu un professeur qui voulait qu'on mette UNE seule fonction par fichier
Sans aller jusqu'à de tels extrêmes, sachez que tout est une question de dosage. Essayez de faire des fichiers avec plusieurs fonctions, sans qu'il y en ait trop à la fois (sinon on s'y perd) ou pas assez (sinon vous risquez d'avoir trop de fichiers par projet, et là aussi vous vous y perdrez)

Fichiers .h et .c

Comme vous le voyez, il y a 2 types de fichiers différents sur la capture d'écran que je vous ai montrée :

- **Les .h** : appelés fichiers headers. Ces fichiers contiennent les prototypes des fonctions.
- **Les .c** : les fichiers sources. Ces fichiers contiennent les fonctions elles-mêmes.

En général, on met donc rarement les prototypes dans les fichiers .c comme on l'a fait tout à l'heure dans le main.c (sauf si votre programme est tout petit).

Pour chaque fichier .c, il y a son équivalent .h qui contient les prototypes des fonctions.
Rejetez un oeil à ma capture d'écran :

- Il y a ia.c (le code des fonctions) et ia.h (les prototypes des fonctions)
- Il y a affichage.c et affichage.h
- etc.

Mais comment faire pour que l'ordinateur sache que les prototypes sont dans un autre fichier que le .c ?

Il faut **inclure** le fichier .h grâce à une directive de préprocesseur.
Attention, préparez-vous à comprendre plein de trucs tout d'un coup

Comment inclure un fichier header ?
Vous savez le faire, vous l'avez déjà fait !

Regardez par exemple le début de mon fichier affichage.c :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include "affichage.h" // On inclut affichage.h


// Affiche une ligne d'allumettes à l'écran, de la couleur demandée
void afficherLigneDAllumettes (long numeroDuTas, long nombreTotalDAllumettes, long
nombreDAllumettesAColorier, long couleur, int effacerLigne)
{
        long ordonneeActuelleDansLAllumette, abscisseAllumetteActuelle,
allumetteActuelle,
         ordonneeHautAllumette, abscisseGaucheAllumette, abscisseActuelle;

        ordonneeHautAllumette = ((numeroDuTas - 1) * 5) + 2;
        abscisseGaucheAllumette = 20 + (60 - (nombreTotalDAllumettes * 4)) / 2;



        if (effacerLigne)
        {
        // Reste du code...
```

L'inclusion se fait grâce à la directive de préprocesseur #include que vous connaissez bien maintenant
Regardez les premières lignes du code source ci-dessus :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include "affichage.h" // On inclut affichage.h
```

On inclut 3 fichiers .h : stdio, stdlib et affichage.
Notez une différence : les fichiers que vous avez créés et placés dans le répertoire de votre projet doivent être inclus avec des guillemets ("affichage.h") tandis que les fichiers correspondants aux librairies (qui sont installés, eux, dans le répertoire de votre IDE généralement) sont inclus entre chevrons (<stdio.h>).

Vous utiliserez donc :

- **Les chevrons < >** pour inclure un fichier se trouvant dans le répertoire "include" de votre IDE
- **Les guillemets " "** pour inclure un fichier se trouvant dans le répertoire de votre projet (à côté des .c généralement)

La commande #include demande d'insérer le contenu du fichier dans le .c. C'est donc une commande qui dit "Insère ici le fichier affichage.h" par exemple.

Et dans le fichier affichage.h que trouve-t-on ?
On trouve juste les prototypes des fonctions du fichier affichage.c !

Code : C

```c
/*
affichage.h
-----------

Par Mathieu et Jonathan
Dernière modification : 29/02/04

Description : gère l'affichage graphique du jeu

*/


// Affiche une ligne d'allumettes à l'écran, de la couleur demandée
void afficherLigneDAllumettes (long numeroDuTas, long nombreTotalDAllumettes,
             long nombreDAllumettesAColorier, long couleur, int effacerLigne);

// Place le curseur lors de la partie à la position définie
void afficherCurseurDuJeu(long position);

void afficherInfosJeu(long nombreDAllumettes, long numeroDuTas, int tourDuPremierJoueur,
long couleur, int jeuContreLOrdinateur);

// Affiche le menu d'accueil
void afficherMenu();
```

Voilà comment fonctionne un vrai projet
Quel intérêt de mettre les prototypes dans des fichiers .h ?

La raison est en fait assez simple. Quand dans votre code vous faites *appel* à une fonction, votre ordinateur doit déjà la connaître, savoir combien de paramètres elle prend etc. C'est à ça que sert un prototype : c'est le mode d'emploi de la fonction pour l'ordinateur.
Tout est une question d'ordre : si vous mettez vos prototypes dans des .h (headers) inclus en haut des fichiers .c, votre ordinateur connaîtra le mode d'emploi de toutes vos fonctions dès le début de la lecture du fichier.

En faisant cela, vous n'aurez ainsi pas à vous soucier de l'ordre dans lesquelles les fonctions se trouvent dans vos fichiers .c
Si vous faites un petit programme maintenant contenant 2-3 fonctions, vous vous rendrez compte que les prototypes semblent facultatifs (ça marche sans). Mais ça ne durera pas longtemps ! Dès que vous aurez un peu plus de fonctions, si vous ne mettez pas vos prototypes de fonctions dans des .h la compilation plantera lamentablement

Lorsque vous appelez une fonction située dans fonctions.c depuis le fichier main.c, vous aurez besoin d'inclure les prototypes de fonctions.c dans main.c. Il faudra donc mettre un #include "fonctions.h" en haut de main.c
Souvenez-vous de ceci : à chaque fois que vous faites appel à une fonction X dans un fichier, il faut que vous ayez inclus les prototypes de cette fonction dans votre fichier. Cela permet au compilateur de vérifier

si vous l'avez correctement appelée.

Comment j'ajoute des fichiers .c et .h à mon projet ?

Ca dépend de l'IDE que vous utilisez, mais globalement la procédure est la même : Fichier / Nouveau / Fichier source.
Cela crée un nouveau fichier vide. Ce fichier n'est pas encore de type .c ou .h, il faut que vous l'enregistriez pour le dire. Enregistrez donc ce nouveau fichier (même s'il est encore vide !). On vous demandera alors quel nom vous voulez donner au fichier. C'est là que vous choisissez si c'est un .c ou un .h :

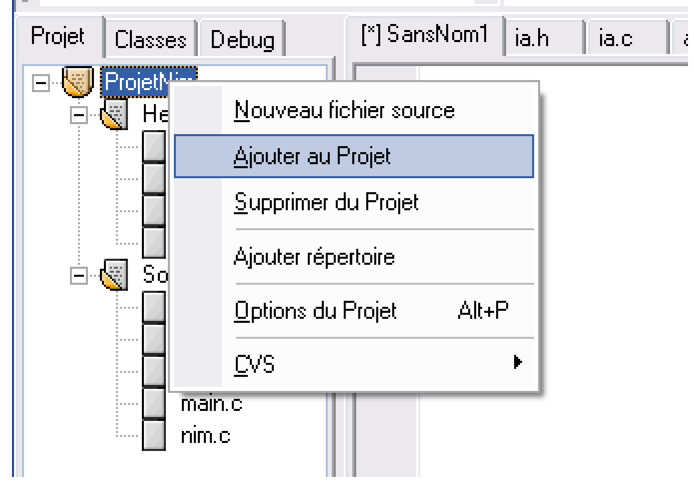
- Si vous l'appelez fichier.c, ce sera un .c
- Si vous l'appelez fichier.h, ce sera un .h

C'est aussi simple que cela
Enregistrez votre fichier dans le répertoire où se trouvent les autres fichiers de votre projet (le même dossier que main.c). Généralement, **vous enregistrerez tous vos fichiers dans le même répertoire, les .c comme les .h**.

Le dossier de mon jeu d'allumettes ressemble donc au final à ça :

Vous y voyez des .c et des .h ensemble.
Vous notez aussi qu'il y a des .o, nous allons voir juste après ce que c'est.

Bref, maintenant votre fichier est enregistré, mais il n'est pas encore vraiment ajouté au projet !
Pour l'ajouter au projet, faites un clic droit dans la partie à gauche de l'écran (où il y a la liste des fichiers du projet) et choisissez "Ajouter au projet" :



Une fenêtre s'ouvre et vous demande quels fichiers ajouter au projet. Sélectionnez le fichier que vous venez de créer, et c'est fait
Le fichier fait maintenant partie du projet et apparaît dans la liste à gauche !

Certains IDE, comme Code::Blocks, séparent automatiquement les .c et les .h dans la liste à gauche. D'autres IDE, comme Dev-C++, permettent de le faire mais ne les séparent pas automatiquement. Pour faire cela, vous pouvez ajouter un "répertoire" en faisant clic droit / ajouter répertoire (regardez sur ma capture d'écran ci-dessus).
Cela est surtout utile quand vous avez beaucoup de fichiers source et que vous voulez les organiser pour pas vous perdre dans votre projet. Ca ne change rien au programme final.

Les includes des librairies standard

Une question devrait vous trotter dans la tête...
Si on inclut les fichiers stdio.h et stdlib.h, c'est donc qu'ils existent quelque part et qu'on peut aller les chercher non ?

Oui bien sûr !
Ils sont installés normalement là où se trouve votre IDE. Dans mon cas sous Dev C++, je les trouve là :

```
C:\Program Files\Dev-Cpp\include
```

Il faut généralement chercher un dossier include.
Là-dedans, vous allez trouver plein plein de fichiers. Ce sont des headers (.h) des librairies standard, c'est-à-dire des librairies disponibles partout (que ce soit sous Windows, Mac, Linux...). Vous y retrouverez donc stdio.h et stdlib.h entre autres.

Vous pouvez les ouvrir si vous voulez, mais prévoyez une bassine à côté on sait jamais
En effet, c'est un peu compliqué et ça peut donner la nausée (il y a pas mal de choses qu'on n'a pas encore vues, notamment pas mal de directives de préprocesseur). Si vous cherchez bien, vous verrez que ce fichier est rempli de prototypes de fonctions standard, comme printf par exemple.

Ok, je sais maintenant où se trouvent les prototypes des fonctions standard. Mais je pourrai pas aussi voir le code source de ces fonctions ? Où sont les .c ?!

Ils n'existent pas
En fait, les fichiers .c sont déjà compilés (en code binaire, c'est-à-dire en code machine). Il est donc totalement impossible de les lire.

Vous pouvez retrouver les fichiers compilés dans un répertoire appelé "lib" généralement (pour "library"). Chez moi ils se trouvent dans :

```
C:\Program Files\Dev-Cpp\lib
```

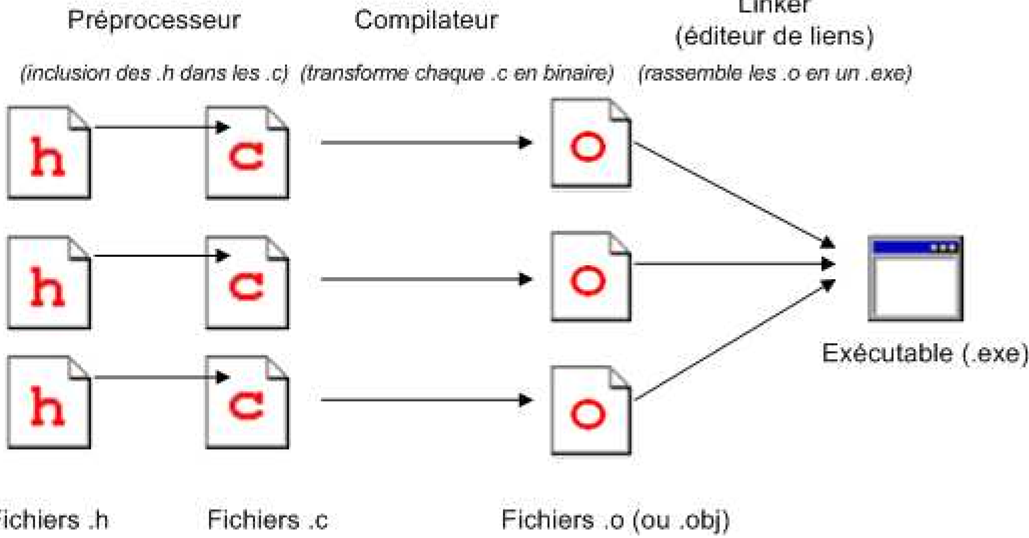
Les fichiers compilés des librairies ont l'extension .a sous Dev (qui utilise le compilateur appelé mingw), et ont l'extension .lib sous Visual C++ (qui utilise le compilateur Visual).
N'essayez pas de les lire c'est totalement pas comestible

Voilà vous savez maintenant un peu mieux comment ça fonctionne j'espère
Dans vos fichiers .c, vous incluez les .h des librairies standard pour pouvoir utiliser des fonctions standard comme printf. Votre ordinateur a ainsi les prototypes sous les yeux et peut vérifier si vous appelez les fonctions correctement (si vous n'oubliez pas de paramètres par exemple).

LA COMPILATION SÉPARÉE

Maintenant que vous savez qu'un projet est composé de plusieurs fichiers sources, nous pouvons rentrer plus en détail dans le fonctionnement de la compilation. Jusqu'ici, nous avions vu un schéma très simplifié.

Voici un schéma plus précis de la compilation. Croyez-moi, celui-là il vaut mieux le connaître par coeur !



Ca c'est un vrai schéma de ce qu'il se passe à la compilation.
Allez, je vous détaille ça dans l'ordre

1. **Préprocesseur** : le préprocesseur est un programme **qui démarre avant la compilation**. Son rôle est d'exécuter les instructions spéciales qu'on lui a données dans des directives de préprocesseur, ces fameuses lignes qui commencent par un #.
   Pour l'instant, la seule directive de préprocesseur que l'on connaît est #include, qui permet d'inclure un fichier dans un autre. Le préprocesseur sait faire d'autres choses, mais ça nous le verrons plus tard. Le #include est quand même ce qu'il y a de plus important
   Le préprocesseur "remplace" donc les lignes #include par le fichier indiqué. Il met à l'intérieur de chaque fichier .c les fichiers .h qu'on a demandé d'inclure.
   A ce moment-ci de la compilation, votre fichier .c est complet et contient tous les prototypes des fonctions que vous utilisez (votre fichier .c est donc un peu plus gros que la normale).

2. **Compilation** : cette étape très importante consiste à transformer vos fichiers sources en code binaire compréhensible par l'ordinateur. Le compilateur compile chaque fichier .c un à un. Il compile tous les fichiers source de votre projet, d'où l'importance d'avoir bien ajouté tous vos fichiers au projet (ils doivent tous apparaître dans la fameuse liste à gauche)
   Le compilateur génère un fichier .o (ou .obj, ça dépend du compilateur) par fichier .c compilé. Ce sont des fichiers binaires temporaires. Généralement, ces fichiers sont supprimés à la fin de la compilation, mais selon les options que vous mettez vous pouvez choisir de les garder (mais ça sert à rien)

3. **Edition de liens** : le linker (ou "éditeur de liens" en français) est un programme dont le rôle est d'assembler les fichiers binaires .o. Il les assemble en un seul gros fichier : l'exécutable final ! Cet exécutable a l'extension .exe sous Windows. Si vous êtes sous un autre OS, il devrait prendre l'extension adéquate

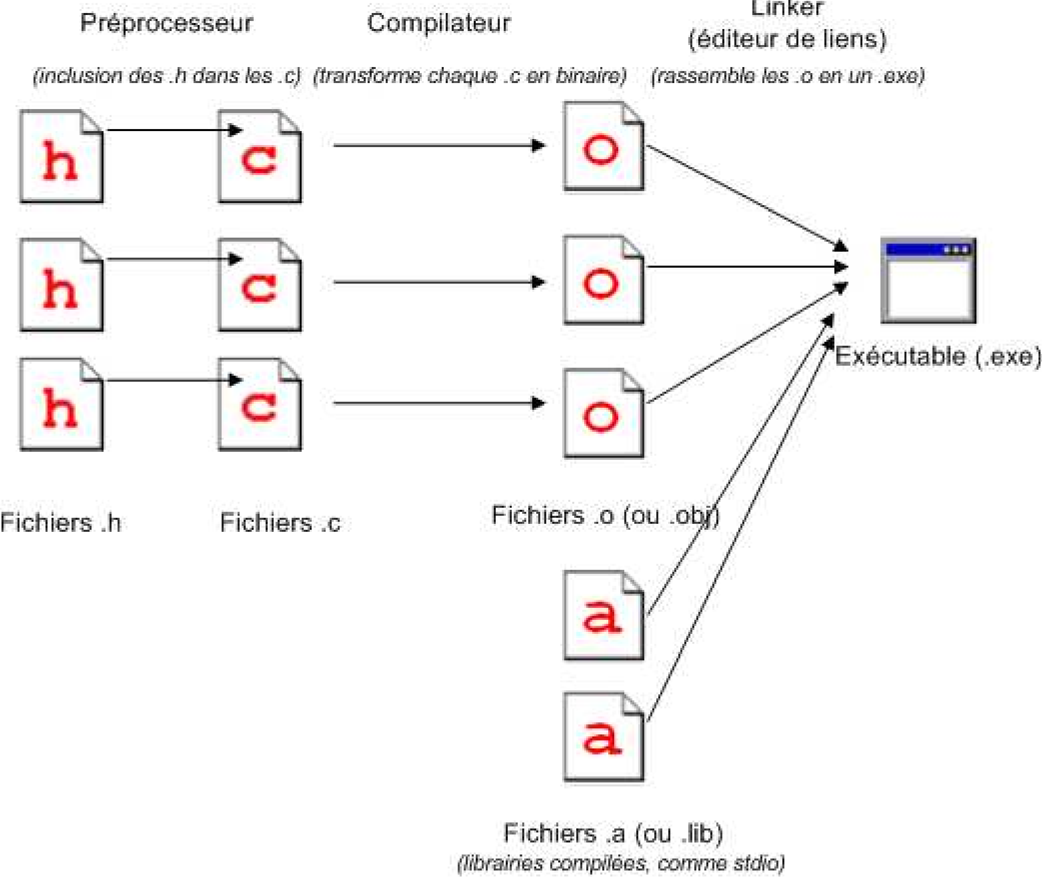
Et voilà, maintenant vous savez comment ça se passe à l'intérieur
Je le dis et je le répète, ce schéma est super important. Il fait la différence entre un programmeur du dimanche qui copie à l'arrache des codes sources et un programmeur qui sait et comprend ce qu'il fait

La plupart des erreurs surviennent à la compilation, mais il m'est arrivé aussi d'avoir des erreurs de linker. Cela signifie que le linker n'est pas arrivé à assembler tous les .o (il en manquait peut-être).

Lorsque vous utilisez des librairies

Notre schéma est par contre encore un peu incomplet. En effet, les librairies n'apparaissent pas dedans !
Comment cela se passe-t-il quand on utilise des librairies ?

En fait le début du schéma reste le même, c'est seulement le linker qui va avoir un peu plus de travail. Il va assembler vos .o (temporaires) avec les librairies compilées dont vous avez besoin (.a ou .lib selon le compilateur) :



Nous y sommes, le schéma est cette fois complet complet
Vos fichiers de librairies .a (ou .lib) sont rassemblés dans l'exécutable avec vos .o

C'est comme cela qu'on peut obtenir au final un programme 100% complet, qui contient toutes les instructions nécessaires à l'ordinateur, même celles *qui lui expliquent comment afficher du texte* !
Par exemple la fonction printf se trouve dans un .a, et donc sera rassemblée avec votre code source dans l'exécutable.

Dans quelques temps, nous apprendrons à utiliser des librairies graphiques. Celles-ci seront là aussi dans des .a et contiendront des instructions pour indiquer à l'ordinateur comment ouvrir une fenêtre à l'écran par exemple. Mais, patience, car tout vient à point à qui sait attendre c'est bien connu

LA PORTÉE DES FONCTIONS ET VARIABLES

Pour terminer ce chapitre, je vais vous parler de ce qu'on appelle **la portée** des fonctions et des variables.

Nous allons voir quand les variables et les fonctions sont accessibles, c'est-à-dire quand on peut faire appel à elles.

Les variables propres aux fonctions

Lorsque vous déclarez une variable dans une fonction, celle-ci est supprimée de la mémoire à la fin de la fonction :

Code : C

```
long triple(long nombre)
{
    long resultat = 0; // La variable resultat est créée en mémoire

    resultat = 3 * nombre;
    return resultat;
} // La fonction est terminée, la variable resultat est supprimée de la mémoire
```

Une variable déclarée dans une fonction n'existe donc que pendant que la fonction est exécutée.
Qu'est-ce que ça veut dire concrètement ? Que vous ne pouvez pas y accéder depuis une autre fonction !

Code : C

```
long triple(long nombre);

int main(int argc, char *argv[])
{
    printf("Le triple de 15 est %ld\n", triple(15));

    printf("Le triple de 15 est %ld", resultat); // Cette ligne plantera à la compilation

    return 0;
}

long triple(long nombre)
{
    long resultat = 0;

    resultat = 3 * nombre;
    return resultat;
}
```

Dans le main, j'essaie d'accéder à la variable résultat. Or, comme cette variable résultat a été créée dans la fonction triple, elle n'est pas accessible dans la fonction main !

Retenez : une variable déclarée dans une fonction n'est accessible qu'à l'intérieur de cette fonction.
On dit que c'est une variable locale.

Les variables globales : à éviter

Variable globale accessible dans tous les fichiers

Il est possible de déclarer des variables qui sont accessibles dans toutes les fonctions de tous les fichiers du projet.
Je vais vous montrer comment faire pour que vous sachiez que ça existe, mais généralement il faut éviter de le faire.
Ca aura l'air de simplifier votre code au début, mais après vous risquez de vous retrouver avec plein de variables accessibles partout, ce qui risquera de vous poser des soucis.

Pour déclarer une variable "globale" accessible partout, vous devez faire la déclaration de la variable **en-dehors** des fonctions. Vous ferez la déclaration tout en haut du fichier, après les #include généralement.

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

long resultat = 0; // Déclaration de variable globale

void triple(long nombre); // Prototype de fonction

int main(int argc, char *argv[])
{
    triple(15); // On appelle la fonction triple, qui modifie la variable globale
resultat
    printf("Le triple de 15 est %ld\n", resultat); // On a accès à resultat

    return 0;
}

void triple(long nombre)
{
    resultat = 3 * nombre;
}
```

Sur cet exemple, ma fonction triple ne renvoie plus rien (void). Elle se contente de modifier la variable globale resultat que la fonction main peut récupérer.

Ma variable resultat sera accessible dans tous les fichiers du projet, donc on pourra faire appel à elle dans TOUTES les fonctions du programme.

Ce type de choses est généralement à bannir dans un programme en C. Utilisez plutôt le retour de la fonction (return) pour renvoyer un résultat.

Variable globale accessible uniquement dans un fichier

La variable globale de tout à l'heure était accessible dans tous les fichiers du projet.
Il est possible de la rendre accessible uniquement dans le fichier où elle se trouve. Ca reste une variable globale quand même, mais disons qu'elle n'est globale qu'aux fonctions de ce fichier et non à toutes les fonctions du programme.

Pour créer une variable globale accessible uniquement dans un fichier, rajoutez juste le mot-clé static devant :

Code : C

```c
static long resultat = 0;
```

Variable static à une fonction

Si vous rajoutez le mot-clé "static" devant la déclaration d'une variable à l'intérieur d'une fonction, ça n'a pas le même sens que pour les variables globales.
En fait, la variable static n'est plus supprimée à la fin de la fonction. La prochaine fois qu'on appellera la fonction, **la variable aura conservé sa valeur**.

Par exemple :

Code : C

```
long triple(long nombre)
{
    static long resultat = 0; // La variable resultat est créée la première fois que la
fonction est appelée

    resultat = 3 * nombre;
    return resultat;
} // La variable resultat n'est PAS supprimée lorsque la fonction est terminée.
```

Qu'est-ce que ça signifie concrètement ?
Qu'on pourra rappeler la fonction plus tard et la variable resultat contiendra toujours la valeur de la dernière fois.

Voici un petit exemple pour bien comprendre :

Code : C

```
long incremente();

int main(int argc, char *argv[])
{
    printf("%ld\n", incremente());
    printf("%ld\n", incremente());
    printf("%ld\n", incremente());
    printf("%ld\n", incremente());

    return 0;
}

long incremente()
{
    static long nombre = 0;

    nombre++;
    return nombre;
}
```

Code : Console

```
1
2
3
4
```

Ici, la première fois qu'on appelle la fonction incremente, la variable "nombre" est créée.
Elle est incrémentée à 1, et une fois la fonction terminée la variable n'est pas supprimée.

Lorsque la fonction et appelée une seconde fois, la ligne de la déclaration de variable est tout simplement "sautée".
On ne recrée pas la variable, on réutilise la variable qu'on avait déjà créée.
Comme la variable valait 1, elle vaudra maintenant 2, puis 3, puis 4 etc...

Ce type de variable est assez rarement utilisé, mais ça peut vous servir à l'occasion donc je tenais à vous le présenter

Les fonctions locales à un fichier

Pour en finir avec les portées, nous allons nous intéresser à la portée des fonctions.
Normalement, quand vous créez une fonction, celle-ci est globale à tout le programme. Elle est accessible depuis n'importe quel autre fichier .c.
Il se peut que vous ayez besoin de créer des fonctions qui ne seront accessibles que dans le fichier où se trouve la fonction.

Pour faire cela, rajoutez le mot-clé static (encore lui) devant la fonction :

Code : C

```
static long triple(long nombre)
{
    // Instructions
}
```

Pensez à mettre à jour le prototype aussi :

Code : C

```
static long triple(long nombre);
```

Et voilà ! Votre fonction "static" triple ne peut être appelée que depuis une autre fonction du même fichier (par exemple main.c).
Si vous essayez d'appeler la fonction triple depuis une fonction d'un autre fichier (par exemple affichage.c), ça ne marchera pas car "triple" n'y sera pas accessible

On résume !

Portée des variables

- Une variable déclarée dans une fonction est supprimée à la fin de la fonction, elle n'est **accessible que dans cette fonction**.
- Une variable déclarée dans une fonction avec le mot-clé static devant n'est pas supprimée à la fin de la fonction, **elle conserve sa valeur au fur et à mesure de l'exécution du programme**
- Une variable déclarée en-dehors des fonctions est une variable globale, **accessible depuis toutes les fonctions de tous les fichiers source du projet**
- Une variable globale avec le mot-clé static devant **est globale uniquement dans le fichier où elle se trouve**, elle n'est pas accessible depuis les fonctions des autres fichiers.

Portée des fonctions

- **Une fonction est par défaut accessible depuis tous les fichiers du projet**, on peut donc l'appeler depuis n'importe quel autre fichier.
- Si on veut qu'une fonction ne soit **accessible que dans le fichier où elle se trouve**, il faut rajouter le mot-clé static devant.

Voilà, vous savez maintenant un peu mieux ce qu'on appelle la "programmation modulaire".

Plutôt que de mettre tout le code de son programme dans un seul énorme fichier, on le sépare intelligemment en plusieurs fichiers.
Il n'y a pas de "règle" qui dit comment vous devez séparer vos fonctions. Le mieux est de regrouper les fonctions ayant un même thème dans un même fichier .c (et de faire le fichier .h correspondant qui contiendra les prototypes bien sûr !)

Souvenez-vous de mon projet de jeu d'allumettes : il y avait un fichier pour gérer l'affichage à l'écran, un fichier pour l'IA (Intelligence Artificielle) de l'ordinateur, etc etc.
Nos prochains TP seront certainement séparés en plusieurs fichiers, donc essayez de vous entraîner chez vous à créer un projet utilisant au moins un autre fichier .c que le main.c. Pour le moment, vos projets sont sûrement très petits, donc vous aurez peut-être un peu de mal à "inventer" plein de fonctions à séparer en plusieurs fichiers : c'est normal.
Mais profitez-en parce que bientôt vous aurez des fonctions de partout dans tous les sens et vous regretterez ce bon vieux temps

A l'assaut des pointeurs

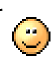
Les pointeurs. Nous y voici enfin.
Autant vous prévenir tout de suite : ce chapitre ne sera pas une ballade de plaisance. Oh que non

Nous sommes encore bien loin de la fin du cours de programmation, et pourtant je peux vous dire que c'est ce chapitre qui sera votre plus grand obstacle. C'est un véritable tournant que nous allons prendre dès maintenant en découvrant ce qu'on appelle **les pointeurs** en C.

A titre purement informatif (et ce n'est pas parce que j'aime bien raconter ma vie), il faut savoir que, plus jeune, j'avais essayé d'apprendre la programmation en C / C++ en lisant des livres. Quel que soit le livre, c'était toujours la même chose : arrivé au chapitre des pointeurs, je ne comprenais plus. Je ne comprenais pas :

- Comment ça fonctionnait
- A quoi ça pouvait bien servir

Aujourd'hui le temps a passé et je sais enfin de quoi il s'agit. Je sais aujourd'hui qu'on ne peut pas faire de programme en C sans se servir de pointeurs. Même dans "Plus ou Moins", vous en avez utilisé sans le savoir

Je vais faire mon maximum pour vous expliquer de ce dont il s'agit, doucement et sûrement. N'allez pas trop vite, vous pourriez vous brûler les ailes en un temps record
Restez attentifs et accrochez-vous : c'est maintenant ou jamais qu'il faut quadrupler d'attention. Ceux qui seront toujours en vie à la fin de ce chapitre auront gagné un pass pour la pluie de bonnes choses qui vous attend après 

(Ca va je vous ai pas trop fait peur là ?)

Un problème bien ennuyeux

Un des plus gros problèmes avec les pointeurs, en plus d'être assez difficiles à assimiler pour un débutant, c'est qu'on a du mal à comprendre à quoi ça peut bien servir.
Alors bien sûr, je pourrais vous dire : "Les pointeurs c'est totalement indispensable on s'en sert tout le temps, croyez-moi c'est super utile !", mais je vois de là vos mines sceptiques

Alors, afin d'éviter cela, je vais vous poser un problème que vous ne pourrez pas résoudre sans utiliser de pointeurs. Ce sera en quelque sorte le fil rouge du chapitre. Nous en reparlerons à la fin de ce chapitre et verrons quelle est la solution en utilisant ce que vous aurez appris.

Voici le problème : je veux écrire une fonction qui renvoie 2 valeurs.
"Impossible" me direz-vous !

En effet, on ne peut renvoyer qu'une valeur par fonction :

Code : C

```
long fonction()
{
    return machin;
}
```

Si on indique long, on renverra un nombre de type long (grâce à l'instruction return).

On peut aussi écrire une fonction qui ne renvoie aucune valeur avec le mot-clé void :

Code : C

```
void fonction()
{

}
```

Mais renvoyer 2 valeurs à la fois... c'est pas possible. On ne peut pas faire 2 "return" ni rien.

Alors supposons que je veuille écrire une fonction à laquelle on envoie un nombre de minutes, et celle-ci renverrait le nombre d'heures et minutes correspondantes :

Si on envoie 45, la fonction renvoie 0 heures et 45 minutes.
Si on envoie 60, la fonction renvoie 1 heure et 0 minutes.
Si on envoie 90, la fonction renvoie 1 heure et 30 minutes.

Allez, soyons fous, tentons le coup :

Code : C

```
#include <stdio.h>
#include <stdlib.h>

/* Je mets le prototype en haut. Comme c'est un tout
petit programme je ne le mets pas dans un .h, mais
en temps normal (dans un vrai programme) j'aurais placé
le prototype dans un fichier .h bien entendu ;o) */

void decoupeMinutes(long heures, long minutes);

int main(int argc, char *argv[])
{
    long heures = 0, minutes = 90;

    /* On a une variable minutes qui vaut 90.
    Après appel de la fonction, je veux que ma variable
    "heures" vaille 1 et que ma variable "minutes" vaille 30 */

    decoupeMinutes(heures, minutes);

    printf("%ld heures et %ld minutes", heures, minutes);

    return 0;
}

void decoupeMinutes(long heures, long minutes)
{
    heures = minutes / 60;  // 90 / 60 = 1
    minutes = minutes % 60; // 90 % 60 = 30 (rappelez-vous : modulo = reste de la
division, "90 divisés par 60 font 1, et il reste 30";)
}
```

Résultat :

Code : Console

```
0 heures et 90 minutes
```

Rhaa, zut zut zut et rezut, ça n'a pas marché. Remarquez, je n'avais guère d'espoir

Que s'est-il passé ?
En fait, quand vous "envoyez" une variable à une fonction, une copie de la variable est réalisée. Ainsi, la variable heures dans la fonction "decoupeMinutes" n'est pas la même que celle de la fonction main ! C'est juste **une copie** !

Votre fonction decoupeMinutes fait son job (d'ailleurs j'ose espérer que vous auriez su l'écrire cette fonction, y'a un bon exemple d'utilisation de la division et du modulo). A l'intérieur de la fonction decoupeMinutes, la variable

heures et la variable minutes valent les bonnes valeurs : 1 et 30.
Mais ensuite, la fonction s'arrête lorsqu'on arrive à l'accolade fermante. Comme on l'a appris dans les chapitres précédents, toutes les variables créées dans une fonction sont détruites à la fin de cette fonction. Votre copie de heures et votre copie de minutes sont donc supprimées.
On retourne ensuite à la fonction main, dans laquelle vos variables heures et minutes valent toujours 0 et 90. Echec !

I : notez que, comme une fonction fait une copie des variables qu'on lui envoie, vous n'êtes pas du tout obligés d'appeler vos variables de la même façon que dans le main. Ainsi, vous pourriez très bien écrire :

Code : C

```
void decoupeMinutes(long h, long m)
```

h pour heures et m pour minutes.
Si vos variables ne s'appellent pas de la même façon que dans le main, ça ne pose donc aucun problème !

Bref, vous aurez beau retourner le problème dans tous les sens... Vous pouvez essayer de renvoyer une valeur avec la fonction (en utilisant un return et en mettant le type long à la fonction), mais vous n'arriveriez à renvoyer qu'une des 2 valeurs. Vous ne pouvez pas renvoyer les 2 valeurs à la fois.

Voilà le problème est posé
Ce n'est qu'un exemple parmi tant d'autres qui va vous montrer l'utilité des pointeurs. J'ai choisi celui-ci parce qu'il me paraissait intéressant.

Allez, maintenant on peut attaquer le chapitre !

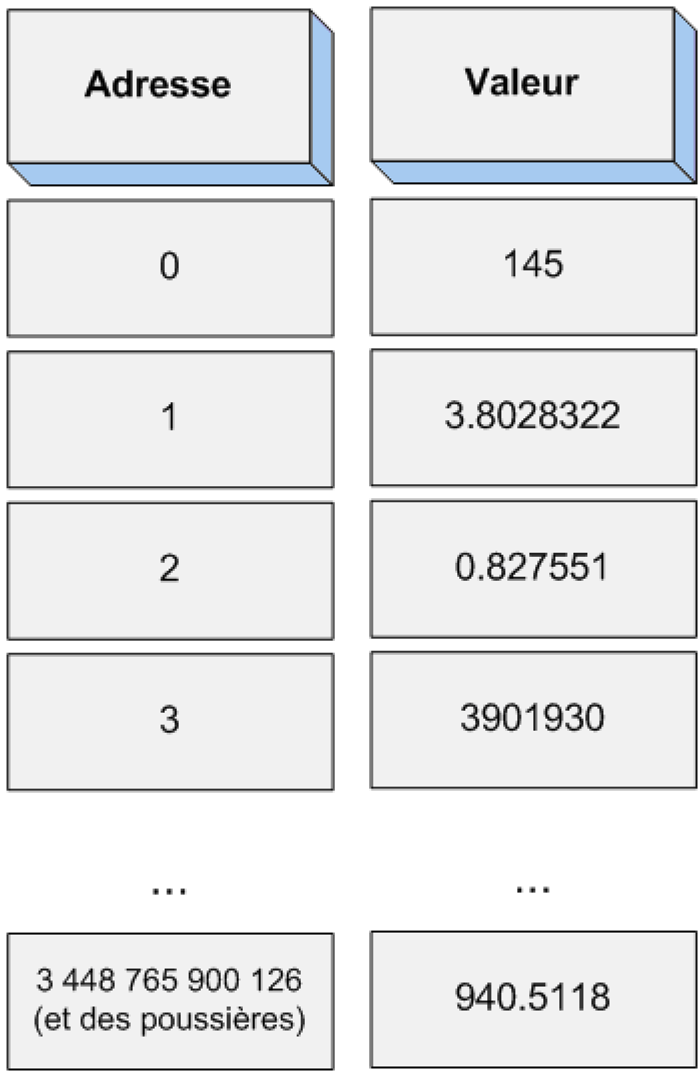
La mémoire, une question d'adresse

Rappel des faits

Petit flash-back.
Vous souvenez-vous du chapitre sur les variables ?

Quelle que soit la réponse, je vous recommande **très vivement** d'aller relire la première partie de ce chapitre, intitulée "Une affaire de mémoire". Bien entendu, je ne peux pas vous obliger à le faire, mais ne venez pas pleurnicher ensuite en me disant que vous ne comprenez rien

Il y avait un schéma très important dans ce chapitre, je vous le ressors pour l'occasion. C'était le schéma de la mémoire :



C'est un peu comme ça qu'on peut représenter la mémoire vive (RAM) de votre ordinateur.
Il faut lire ce schéma ligne par ligne.

La première ligne représente la "cellule" du tout début de la mémoire vive. Chaque cellule a un numéro, c'est **son adresse** (hyper important le vocabulaire là !). La mémoire comporte un grand nombre d'adresses, commençant à l'adresse numéro 0 et se terminant à l'adresse numéro *(insérez un très grand nombre ici)*.

A chaque adresse, on peut stocker un nombre. Un et UN SEUL nombre.

On ne peut donc pas stocker 2 nombres par adresse.

Votre mémoire n'est faite que pour stocker des nombres. Elle ne peut pas stocker de lettres, de phrases. Pour contourner ce problème, on a inventé une table qui fait la liaison entre les nombres et les lettres. Cette table dit par exemple : "*le nombre 89 représente la lettre Y*".
Mais bon, la gestion de texte en C n'est pas encore pour tout de suite. Nous en parlerons dans quelques chapitres. Avant de pouvoir comprendre ça, il faut d'abord comprendre ce que sont les pointeurs.

Adresse et valeur

Revenons-y justement car c'est le sujet.
Quand vous créez une variable "age" de type long par exemple, en tapant ça :

Code : C

```
long age = 10;
```

... votre programme demande au système d'exploitation (Windows par exemple) la permission d'utiliser un peu de mémoire. Le système d'exploitation répond en indiquant à quelle adresse en mémoire il vous laisse le droit d'inscrire votre nombre.
C'est d'ailleurs justement un des rôles principaux d'un système d'exploitation : **on dit qu'il alloue de la mémoire aux programmes**. C'est un peu lui le chef, il contrôle chaque programme et vérifie qu'il se sert de la mémoire à l'endroit qu'il lui a autorisé.

> C'est d'ailleurs là la cause n°1 des plantages de programmes : si votre programme essaie d'accéder à une zone de la mémoire qui ne lui appartient pas, le système d'exploitation (abrégez "OS") refuse cela et coupe brutalement le programme en guise de punition (*"C'est qui le chef ici ?"*).
> L'utilisateur, lui, voit une jolie boîte de dialogue *"Ce programme va être arrêté parce qu'il a effectué une opération non conforme"* (quand c'est pas trop trop grave), ou, pire : un terrible écran-bleu-de-la-mort
> Mais généralement, si l'OS est bien codé votre ordinateur ne devrait pas complètement se bloquer à cause d'un simple "dépassement de mémoire". Enfin, moi j'dis ça, j'dis rien

Je m'égare. Où en étions-nous déjà ?
Ah oui, notre variable age. La valeur 10 a été inscrite quelque part en mémoire, disons par exemple à l'adresse n°4655.
Ce qu'il se passe (et c'est le rôle du compilateur) c'est que le mot "age" dans votre programme est remplacé par l'adresse 4655 à l'exécution. Cela fait que, à chaque fois que vous avez tapé le mot age dans votre code source, cela est remplacé par 4655, et votre ordinateur voit ainsi à quelle adresse il doit aller chercher en mémoire ! Du coup, l'ordinateur se rend en mémoire à l'adresse 4655 et répond fièrement : "Ca vaut 10 !".

On sait donc comment récupérer la valeur de la variable : il suffit tout bêtement de taper "age" dans son code source. Si on veut afficher l'âge, on peut utiliser la fonction *printf* :

Code : C

```c
printf("La variable age vaut : %ld", age);
```

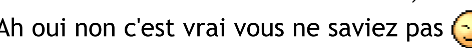
Résultat à l'écran :

Code : Console

```
La variable age vaut : 10
```

Rien de bien nouveau jusque-là.

Le scoop du jour

On sait afficher la valeur de la variable, mais saviez-vous que l'on peut aussi afficher l'adresse correspondante ?
... Ah oui non c'est vrai vous ne saviez pas


Pour afficher l'adresse de la variable, on doit utiliser le symbole %p (le p du mot "pointeur") dans le printf. En outre, on doit envoyer à la fonction printf non pas la variable age, mais son adresse... Et pour faire cela, vous devez mettre le symbole & devant la variable age, comme je vous avais demandé de le faire pour les scanf il y a quelques temps sans vous expliquer pourquoi

Tapez donc :

Code : C

```c
printf("L'adresse de la variable age est : %p", &age);
```

Résultat :

Code : Console

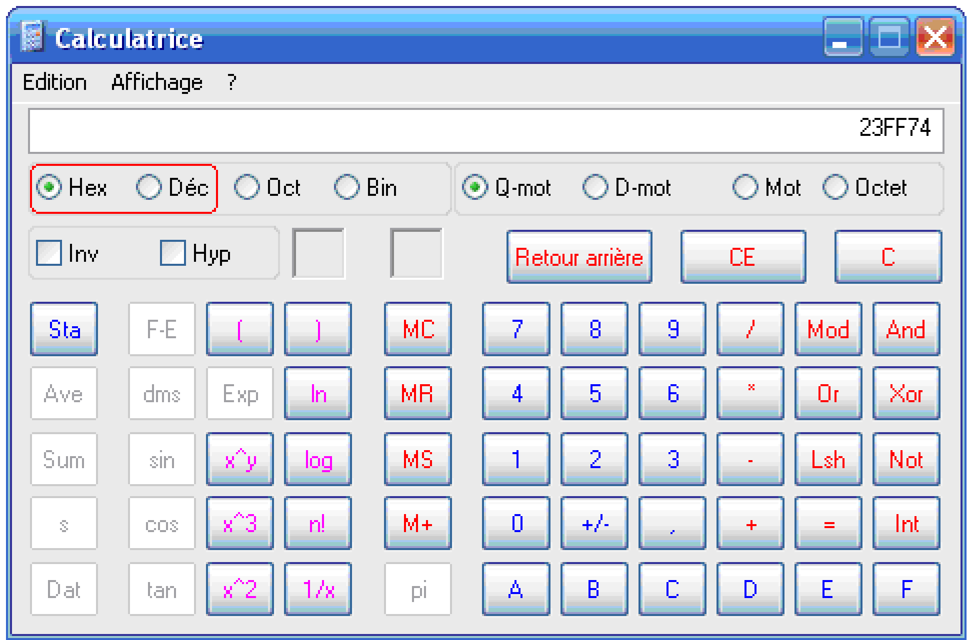
```
L'adresse de la variable age est : 0023FF74
```

Ce que vous voyez là est l'adresse de la variable age au moment où j'ai lancé le programme sur mon ordinateur. Oui oui, c'est un nombre.
0023FF74 est un nombre, il est simplement écrit dans le système hexadécimal, au lieu du système décimal auquel nous avons l'habitude.

Si vous remplacez le %p par un %ld, vous devriez obtenir le nombre en système décimal (plus compréhensible pour nous pauvres humains). Toutefois, le %p a été fait spécialement pour afficher des adresses, donc je préfère en général l'utiliser à la place de %ld.

Sans rentrer dans les détails, juste pour que vous soyez pas trop perturbés, sachez que le fameux système décimal représente tous les nombres avec 10 chiffres : 0 1 2 3 4 5 6 7 8 9
En hexadécimal (un mode dans lequel l'ordinateur travaille souvent), les nombres sont représentés avec 16 chiffres : 0 1 2 3 4 5 6 7 8 9 A B C D E F (les lettres sont en fait des chiffres supplémentaires pour représenter les nombres).

Tout nombre en hexadécimal peut se convertir en décimal et inversement. Ainsi, A vaut 10, B vaut 11, C vaut 12... F vaut 15, 10 vaut 16, 11 vaut 17, 12 vaut 18 et ainsi de suite.
Si vous avez une calculatrice (au hasard la calculatrice de Windows en mode scientifique), vous pouvez convertir les nombres.



La calculatrice de Windows peut convertir les hexadécimaux

Vous devez d'abord vous assurer que vous êtes dans le mode scientifique : Affichage / Scientifique.
Ensuite, cliquez sur Hex (j'ai entouré en rouge sur ma capture d'écran). Tapez le nombre en hexadécimal que vous avez. Puis, cliquez sur Déc juste à côté pour transformer en décimal. Et voilà le travail
Ca marche aussi en sens inverse bien sûr

Ainsi, j'ai pu voir que 0023FF74 correspondait en fait au nombre 2359156. Bon, on s'en fout un peu, ça ne changera pas notre vie, mais ça fait du bien de savoir comment ça marche non ?

Si vous exécutez ce programme sur votre ordinateur, l'adresse sera très certainement différente. Tout dépend de la place que vous avez en mémoire, des programmes que vous avez lancés etc... Il est totalement impossible de prédire à quelle adresse la variable sera stockée chez vous
Si vous lancez votre programme plusieurs fois d'affilée, il se peut que l'adresse soit identique, la mémoire n'ayant pas beaucoup changé entre temps. Si vous redémarrez votre ordinateur par contre, vous aurez sûrement une valeur différente.

Où je veux en venir avec tout ça ?
Eh bien en fait, je veux vous faire retenir la chose suivante toute bête :

- age : affiche **la VALEUR** de la variable.
- &age : affiche **l'ADRESSE** de la variable.

Avec "age", l'ordinateur va lire la valeur de la variable en mémoire et vous renvoie cette valeur. Avec "&age", votre ordinateur vous dit en revanche à quelle adresse se trouve la variable.

UTILISER DES POINTEURS

Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres.
Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses : ce sont justement ce qu'on appelle des pointeurs.

Mais... Les adresses sont des nombres aussi non ? Ca revient à stocker des nombres encore et toujours !

C'est exact. Mais ces nombres auront une signification particulière : ils indiqueront l'adresse d'une autre variable en mémoire.

Créer un pointeur

Pour créer une variable de type pointeur, on doit rajouter le symbole * devant le nom de la variable.

Code : C

```
long *monPointeur;
```

Notez qu'on peut aussi écrire...

Code : C

```
long* pointeurSurAge;
```

Cela revient exactement au même.
Cependant, la première méthode est à préférer. En effet, si vous voulez déclarer plusieurs pointeurs sur la même ligne, vous serez obligés de mettre l'étoile devant le nom (première méthode) :

Code : C

```
long *pointeur1, *pointeur2, *pointeur3;
```

Comme je vous l'ai appris, il est important d'initialiser dès le début ses variables (en leur donnant la valeur 0 par exemple). C'est encore plus important de le faire avec les pointeurs !
Pour initialiser un pointeur (lui donner une valeur par défaut), on n'utilise généralement pas le nombre 0 mais le mot-clé NULL (les majuscules sont importantes attention) :

Code : C

```
long *monPointeur = NULL;
```

Là, vous avez un pointeur initialisé à NULL. Comme ça, vous saurez dans la suite de votre programme que votre pointeur ne contient aucune adresse.

Que se passe-t-il ? Ce code va réserver une case en mémoire comme si vous aviez créé une variable normale. Cependant, et c'est ce qui change, la valeur du pointeur est faite pour contenir une adresse. L'adresse... d'une autre variable.

Pourquoi pas l'adresse de la variable age ? Vous savez maintenant comment indiquer l'adresse d'une variable au lieu de sa valeur (en utilisant le symbole &) alors zou ! Ca nous donne :

Code : C

```
long age = 10;
long *pointeurSurAge = &age;
```

La première ligne signifie : *"Créer une variable de type long dont la valeur vaut 10"*
La deuxième ligne signifie : *"Créer une variable de type pointeur dont la valeur vaut l'adresse de la variable age"*.

Vous avez remarqué qu'il n'y a pas de type "pointeur" comme il y a un type "int", un type "double" ou encore un type "long".
On n'écrit donc pas :

Code : C

```
pointeur pointeurSurAge;
```

Au lieu de ça, on utilise le symbole *, mais on continue à écrire "long". Qu'est-ce que ça signifie ?
En fait, (accrochez-vous), on doit indiquer quel est le type de la variable dont le pointeur va contenir l'adresse. Comme notre pointeur pointeurSurAge va contenir l'adresse de la variable age (qui est de type long) alors mon pointeur doit être de type "long*" ! Si ma variable age avait été de type int, alors j'aurais dû écrire "int *monPointeur".

Vocabulaire : on dit que le pointeur pointeurSurAge pointe sur la variable age.

Un petit schéma de ce qu'il se passe en mémoire :

	Adresse	Valeur
	0	145
	1	3
	2	82
pointeurSurAge	3	177450
	...	...
age	177450	10
	...	...
	3 448 765 900 126 (et des poussières)	940

Dans ce schéma, la variable age a été placée à l'adresse 177450 (vous voyez d'ailleurs que sa valeur est 10), et le pointeur pointeurSurAge a été placé à l'adresse 3 (c'est tout à fait le fruit du hasard hein, j'invente).

Lorsque mon pointeur est créé, le système d'exploitation réserve une case en mémoire comme il l'a fait pour age. La différence ici, c'est que la valeur de pointeurSurAge est un peu particulière. Regardez bien le schéma : c'est l'adresse de la variable age !

Ceci, mesdames et messieurs, est le secret absolu de tout programme écrit en langage C (et donc aussi en langage C++). On y est, nous venons de rentrer dans le monde merveilleux des pointeurs !

Ouah, super. Et ça fait quoi ton truc ?

Ca ne transforme pas encore votre ordinateur en machine à café, certes.

Seulement maintenant, on a un pointeurSurAge qui contient l'adresse de la variable age.
Essayons de voir ce que contient le pointeur en faisant un printf dessus :

Code : C

```c
long age = 10;
long *pointeurSurAge = &age;

printf("%ld", pointeurSurAge);
```

Code : Console

```
177450
```

Hum. En fait, cela n'est pas très étonnant. On demande la valeur de pointeurSurAge, et sa valeur c'est l'adresse de la variable age (177450).
Comment faire pour demander à avoir la valeur de la variable se trouvant à l'adresse indiquée dans pointeurSurAge ?
Il faut mettre le symbole * devant le nom du pointeur :

Code : C

```c
long age = 10;
long *pointeurSurAge = &age;

printf("%ld", *pointeurSurAge);
```

Code : Console

```
10
```

Hourra !
On y est arrivés ! En mettant le symbole * devant le nom du pointeur, on accède à la valeur de la variable age

Si au contraire on avait mis le symbole & devant le nom du pointeur, on aurait eu l'adresse où se trouve le pointeur (ici, c'est 3)

Je ne vois pas ce qu'on y gagne. Après tout, sans pointeur on peut très bien afficher la valeur de la variable age !

Cette question (que vous devez inévitablement vous poser) me fait sourire pour 2 raisons :

- La première, c'est que j'étais sûr que vous me diriez ça. Et après tout, qui pourrait vous en vouloir ? Actuellement l'intérêt n'est pas évident, mais petit à petit, au fur et à mesure des chapitres suivants, vous comprendrez que tout ce tintouin n'a pas été inventé par pur plaisir de compliquer les choses
- La seconde, elle est toute bête : c'est que je me souviens exactement que j'étais comme vous à ce moment-là lorsque j'apprenais le C Bref, je comprends exactement la frustration que vous devez ressentir. Mais c'est une bien maigre consolation pour vous j'en conviens

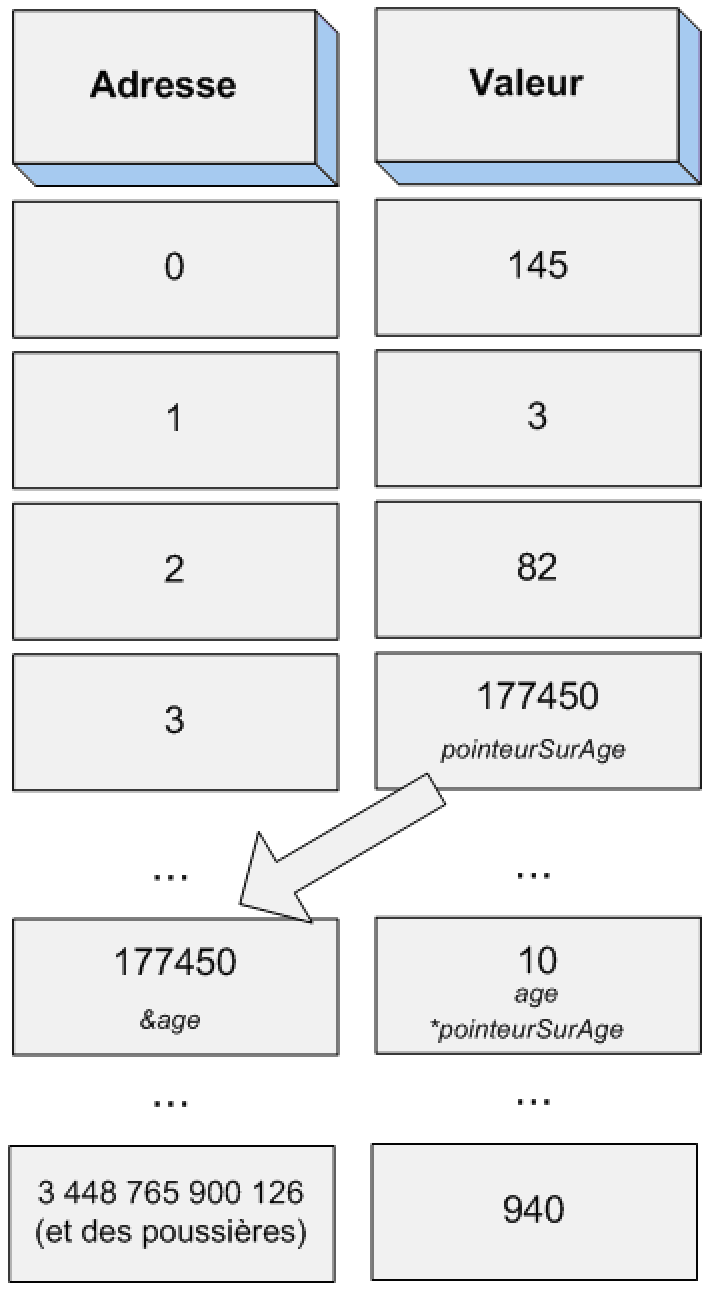
A retenir absolument

Avant d'aller plus loin, s'il y avait une chose à retenir pour le moment ce serait cela en 4 points :

- Sur une variable, comme la variable age :
  - "age" signifie : "Je veux la valeur de la variable age".
  - "&age" signifie : "Je veux l'adresse où se trouve la variable age".

- Sur un pointeur, comme pointeurSurAge :
  - "pointeurSurAge" signifie : "Je veux la valeur de pointeurSurAge" (cette valeur étant une adresse).
  - "*pointeurSurAge" signifie : "Je veux la valeur de la variable qui se trouve à l'adresse contenue dans pointeurSurAge"

Contentez-vous de bien retenir ces 4 points. Faites des tests et vérifiez que ça marche.
Voici un schéma qui va vous permettre de bien situer qui désigne quoi :



Attention à ne pas confondre les différentes significations de l'étoile ! Lorsque vous déclarez un pointeur, l'étoile sert juste à indiquer qu'on veut créer un pointeur :

Code : C

```c
long *pointeurSurAge;
```

En revanche, lorsque vous utilisez votre pointeur ensuite en écrivant :

Code : C

```
printf("%ld", *pointeurSurAge);
```

... cela ne signifie pas "Je veux créer un pointeur" mais : "Je veux la valeur de la variable sur laquelle pointe mon pointeurSurAge".

Tout cela est fon-da-men-tal. Il faut savoir cela par cœur, et surtout le comprendre. Même pas la peine de continuer ce chapitre si vous n'avez pas compris cela, je préfère être franc

N'hésitez pas à lire et relire ce qu'on vient d'apprendre. Je ne peux pas vous en vouloir si vous n'avez pas compris du premier coup, et ce n'est pas une honte non plus d'ailleurs.
Pour info, avant que j'arrive à comprendre cela il a dû se passer une petite semaine (bon pas à temps plein je l'avoue).
Et pour comprendre la plupart des subtilités des pointeurs, je crois qu'il m'a fallu bien 2 ou 3 mois au moins (pas à temps plein là non plus, rassurez-vous)

Bref, si vous vous sentez un peu perdus, pensez à ces gens qui sont aujourd'hui des grands gourous de la programmation : aucun d'entre eux n'a compris tout le fonctionnement des pointeurs du premier coup.
Et si jamais cette personne existe, croyez-moi j'aimerais la rencontrer

Envoyer un pointeur à une fonction

Le gros intérêt des pointeurs (mais ce n'est pas le seul), c'est de les envoyer à des fonctions pour qu'ils modifient directement une variable en mémoire, et non une copie comme on l'a vu.
Comment ça marche ? Il y a en fait plusieurs façons de faire. Voici un premier exemple :

Code : C

```c
void triplePointeur(long *pointeurSurNombre);

int main(int argc, char *argv[])
{
    long nombre = 5;

    triplePointeur(&nombre); // On envoie l'adresse de nombre à la fonction
    printf("%ld", nombre); // On affiche la variable nombre. La fonction a directement
modifié la valeur de la variable car elle connaissait son adresse

    return 0;
}

void triplePointeur(long *pointeurSurNombre)
{
    *pointeurSurNombre *= 3; // On multiplie par 3 la valeur de la variable nombre
}
```

Code : Console

```
15
```

La fonction triplePointeur prend un paramètre de type long* (c'est-à-dire un pointeur sur long). Voici ce qu'il se passe dans l'ordre, en partant du début du main :

1. Une variable nombre est créée dans le main. On lui affecte la valeur 5. Ca, vous connaissez.
2. On appelle la fonction triplePointeur. On lui envoie en paramètre l'adresse de notre variable nombre.
3. La fonction triplePointeur reçoit cette adresse dans pointeurSurNombre. A l'intérieur de la fonction triplePointeur, on a donc un pointeur pointeurSurNombre qui contient l'adresse de la variable nombre.
4. Maintenant qu'on a un pointeur sur nombre, on peut modifier directement la variable nombre en mémoire ! Il suffit d'utiliser *pointeurSurNombre pour désigner la variable nombre ! Pour l'exemple, on fait un simple test : on multiplie la variable nombre par 3.

5. De retour dans la fonction main, notre nombre vaut maintenant 15 car la fonction triplePointeur a modifié directement la valeur de nombre.

Bien sûr, j'aurais pu faire un simple return comme on a appris à le faire dans le chapitre sur les fonctions. Mais l'intérêt là, c'est que de cette manière en utilisant des pointeurs on peut modifier la valeur de plusieurs variables en mémoire (on peut donc "renvoyer plusieurs valeurs"). On n'est plus limités à une seule valeur !

Quel est l'intérêt du coup d'utiliser un return dans une fonction si on peut se servir des pointeurs pour modifier des valeurs ?

Ca dépendra de vous et de votre programme. C'est à vous de décider. Il faut savoir que les return sont bel et bien toujours utilisés en C. Le plus souvent, on s'en sert pour renvoyer ce qu'on appelle *un code d'erreur* : la fonction renvoie 1 (vrai) si tout s'est bien passé, et 0 (faux) s'il y a eu une erreur pendant le déroulement de la fonction. Mais bon, on aura le temps de voir comment gérer les erreurs en C plus tard

Une autre façon d'envoyer un pointeur à une fonction

Dans le code source qu'on vient de voir, il n'y avait pas de pointeur dans la fonction main. Juste une variable nombre. Le seul pointeur qu'il y avait vraiment était dans la fonction tripleNombre (de type long*).

Il faut absolument que vous sachiez qu'il y a une autre façon d'écrire le code précédent, en ajoutant un pointeur dans la fonction main :

Code : C

```c
void triplePointeur(long *pointeurSurNombre);

int main(int argc, char *argv[])
{
    long nombre = 5;
    long *pointeur = &nombre; // pointeur prend l'adresse de nombre

    triplePointeur(pointeur); // On envoie pointeur (l'adresse de nombre) à la fonction
    printf("%ld", *pointeur); // On affiche la valeur de nombre, en tapant *pointeur

    return 0;
}

void triplePointeur(long *pointeurSurNombre)
{
    *pointeurSurNombre *= 3; // On multiplie par 3 la valeur de la variable nombre
}
```

Pour que vous ayez les 2 codes sources côte à côte, je vous mets celui de tout à l'heure ci-dessous. Comparez-les bien, il y a de subtiles différences et vous devez arriver à comprendre pourquoi il y a ces différences entre ces 2 codes sources.

Code : C

```
void triplePointeur(long *pointeurSurNombre);

int main(int argc, char *argv[])
{
    long nombre = 5;

    triplePointeur(&nombre); // On envoie l'adresse de nombre à la fonction
    printf("%ld", nombre); // On affiche la variable nombre. La fonction a directement
modifié la valeur de la variable car elle connaissait son adresse

    return 0;
}

void triplePointeur(long *pointeurSurNombre)
{
    *pointeurSurNombre *= 3; // On multiplie par 3 la valeur de la variable nombre
}
```

Et le résultat dans les deux cas est le même :

Code : Console

```
15
```

Ce qui compte, c'est d'envoyer l'adresse de la variable nombre à la fonction. Or, pointeur vaut l'adresse de la variable nombre, donc c'est bon de ce côté ! On le fait juste d'une manière différente en créant un pointeur dans la fonction main.
Dans le printf (et c'est juste pour l'exercice), j'affiche le contenu de la variable nombre en tapant *pointeur. Notez que j'aurais pu à la place taper "nombre" : ça aurait été pareil car cela désigne la même chose dans la mémoire.

J'ai mis des semaines avant de comprendre que ces 2 codes faisaient effectivement la même chose, mais d'une manière différente. Si vous arrivez à comprendre ça, alors bravo, respect, bien joué, vous avez compris tout ce que je voulais vous enseigner sur les pointeurs

Comme je vous le disais tout à l'heure, dans le programme "Plus ou Moins" nous avons utilisé des pointeurs sans vraiment savoir. C'était en fait en appelant la fonction scanf. En effet, cette fonction a pour rôle de lire ce que l'utilisateur a rentré au clavier et de renvoyer cela.
Pour que la fonction puisse modifier *directement* le contenu de votre variable afin d'y mettre la valeur tapée au clavier, elle a besoin de l'adresse de la variable :

Code : C

```
long nombre = 0;
scanf("%ld", &nombre);
```

La fonction travaille avec un pointeur sur la variable nombre, et peut ainsi modifier directement le contenu de nombre.
Comme on vient de le voir, on pourrait créer un pointeur qu'on enverrait à la fonction scanf :

Code : C

```
long nombre = 0;
long *pointeur = &nombre;
scanf("%ld", pointeur);
```

Attention à ne pas mettre le symbole & devant pointeur dans la fonction scanf ! Ici, pointeur contient lui-même l'adresse de la variable nombre, pas besoin de mettre un & ! Si vous faisiez ça, vous enverriez l'adresse où se trouve le pointeur, et ça, excusez mon langage, mais on s'en fout complètement

Qui a dit : "Un problème bien ennuyeux" ?

Le chapitre est sur le point de s'achever, il est temps de retrouver notre fil rouge
Si vous avez compris ce chapitre, vous devriez être capables de résoudre le problème maintenant.

...

Quoi qu'est-ce que vous attendez ? Allez au boulot tas d'feignasses !

...
...

Vous voulez la solution pour comparer ? La voici !

Code : C

```
void decoupeMinutes(long* pointeurHeures, long* pointeurMinutes);

int main(int argc, char *argv[])
{
    long heures = 0, minutes = 90;

    // On envoie l'adresse de heures et minutes
    decoupeMinutes(&heures, &minutes);

    // Cette fois, les valeurs ont été modifiées !
    printf("%ld heures et %ld minutes", heures, minutes);

    return 0;
}

void decoupeMinutes(long* pointeurHeures, long* pointeurMinutes)
{
    /* Attention à ne pas oublier de mettre une étoile devant le nom
    des pointeurs ! Comme ça, vous pouvez modifier la valeur des variables,
    et non leur adresse ! Vous ne voudriez pas diviser des adresses
    n'est-ce pas ? ;o) */
    *pointeurHeures = *pointeurMinutes / 60;
    *pointeurMinutes = *pointeurMinutes % 60;
}
```

Résultat :

Code : Console

```
1 heures et 30 minutes
```

Est-ce que j'ai besoin de vous expliquer encore une fois comment ça marche ? En théorie, mes explications précédentes devraient suffire, je ne peux rien vous apprendre de nouveau.
Mais bon allez, pour la forme, et parce que c'est un chapitre important, je vais me répéter encore une fois. Il paraît qu'en rabâchant les mêmes choses ça finit par rentrer, si c'est le cas tant mieux pour vous

Explications :

1. Les variables heures et minutes sont créées dans le main.
2. On envoie à la fonction decoupeMinutes l'adresse de heures et minutes.
3. La fonction decoupeMinutes récupère ces adresses dans des pointeurs appelés pointeurHeures et pointeurMinutes. Notez que, là encore, le nom importe peu. J'aurais pu les appeler h et m, ou même encore heures et minutes (mais je ne veux pas que vous risquiez de confondre avec les variables heures et minutes du main, qui ne sont pas les mêmes)
4. La fonction decoupeMinutes modifie directement les valeurs des variables heures et minutes en mémoire car elle possède leurs adresses dans des pointeurs. La seule contrainte, un peu gênante je dois le reconnaître, c'est qu'il faut impérativement mettre une étoile devant le nom des pointeurs si on veut modifier la valeur de

heures et minutes. Si on n'avait pas fait ça, on aurait modifié l'adresse contenue dans les pointeurs, ce qui aurait servi... à rien

De nombreux lecteurs m'ont fait remarquer qu'il était possible de résoudre le "problème" sans utiliser de pointeurs. Oui, bien sûr que je sais que c'est possible, mais il faut contourner les règles que nous nous sommes fixées : on peut utiliser des variables globales (bêrk), ou encore faire un printf dans la fonction (alors que c'est dans le main qu'on veut faire le printf !)
Bref, si vous aussi vous trouvez un moyen de résoudre le problème *autrement*, vous emballez pas. Ce n'était qu'un exemple un peu "théorique" pour vous montrer l'intérêt des pointeurs. Dans les prochains chapitres cet intérêt vous paraîtra de plus en plus évident

Comme le disait mon prof d'info : "*Les pointeurs c'est bon, mangez-en*"
Moi, les premiers temps, ça m'a surtout donné une sacrée migraine

Y'a pas de secret, pour bien comprendre les pointeurs, il faut pratiquer.
Là encore les exemples étaient simples, mais bientôt nous ferons des programmes plus complexes (ne serait-ce que dans les prochains chapitres) et il faudra savoir être patient.
Si vous êtes comme moi, vous allez faire planter misérablement vos programmes. Et pas qu'une fois. Je vous l'ai dit, j'ai mis des mois à acquérir ce que j'appelle "le réflexe des pointeurs". Pendant ce laps de temps, je mélangeais complètement *truc, &machin, truc, machin... J'avançais à petits pas, en essayant de modifier 2-3 caractères par-ci par-là pour essayer de faire marcher mon programme et, surtout, comprendre ce que je faisais.

Aujourd'hui, j'arrive enfin à ne pas me planter trop lamentablement quand je programme. En général, je ne fais plus d'erreurs de base, même si ça arrive à tout le monde hein, même aux meilleurs
Quant à vous, je ne saurais trop vous conseiller de relire ce chapitre autant de fois que nécessaire et de faire des tests. Ne vous affolez pas si les premiers temps vous n'y arrivez pas bien, vous savez désormais que c'est un phénomène complètement normal

Petit résumé avant de se quitter

Les pointeurs ont un gros défaut : ils vous font mélanger plein de choses. Je le sais : dès que j'ai voulu apprendre à me servir des pointeurs je confondais tout.
Je ne peux pas vraiment éviter ça pour vous : il va falloir que vous repassiez ce chapitre encore et encore pour ne plus confondre.

Ceci étant, un énième résumé avant de terminer le chapitre ne fera de mal à personne.
Voici donc comment je résumerais les choses très simplement :

- En C, on peut créer 2 choses dans la mémoire : des variables et des pointeurs.
- **Les variables** : c'est avec elles que nous avons travaillé jusqu'ici. Créer une variable est très simple : il suffit d'indiquer le type de la variable ainsi que son nom.

  Code : C
  ```c
  long maVariable = 0; // Variable créée en mémoire (valeur mise à 0)
  ```
- Si on écrit &maVariable, on obtient l'adresse de la variable en mémoire
- **Les pointeurs** : ce sont des variables un peu particulières car elles prennent pour valeur l'adresse d'autres variables. Pour créer un pointeur vide (qui ne contient l'adresse d'aucune variable), on fait ceci :

  Code : C
  ```c
  long *monPointeur = NULL; // Pointeur créé en mémoire (valeur mise à NULL
  (similaire à 0))
  ```

Un pointeur devient utile lorsqu'on lui donne pour valeur l'adresse d'une variable (par exemple &maVariable) :

Code : C

```
long *monPointeur = &maVariable; // Le pointeur contiendra l'adresse de la variable
```

- On peut alors écrire *monPointeur : si on fait ça, ça sera exactement comme si on écrivait **maVariable** dans le code source (car monPointeur contient l'adresse de maVariable)
  On peut donc écrire :

Code : C

```
printf("%ld", *monPointeur);
```

... cela sera exactement comme si on avait écrit :

Code : C

```
printf("%ld", maVariable);
```

Le résultat est exactement le même, sauf que dans le premier cas on passe par un pointeur pour accéder à la variable.

L'intérêt des pointeurs n'est pas évident. Au final, on en revient à écrire **monPointeur* au lieu de *maVariable* tout court. Quelle perte de temps hein ?
Eh bien non, au contraire les pointeurs sont totalement indispensables en C : on l'a vu dans un petit exemple de ce chapitre (comment modifier la valeur de plusieurs variables depuis une autre fonction), et on n'arrêtera pas de découvrir l'intérêt des pointeurs dans les prochains chapitres.

Soyez donc prêts avant de passer à la suite
N'abandonnez pas ! Les pointeurs seront certainement votre plus gros obstacle dans votre apprentissage du C. Le reste sera plus facile je vous le promets

Les tableaux

Ce chapitre est vraiment la suite directe des pointeurs, et c'est un autre exemple de l'utilité des pointeurs. Vous comptiez y échapper ? C'est raté
Les pointeurs sont partout, je vous avais prévenus

Dans ce chapitre, nous apprendrons à créer des variables de type "tableaux". Les tableaux sont très utilisés en C car ils sont vraiment pratiques

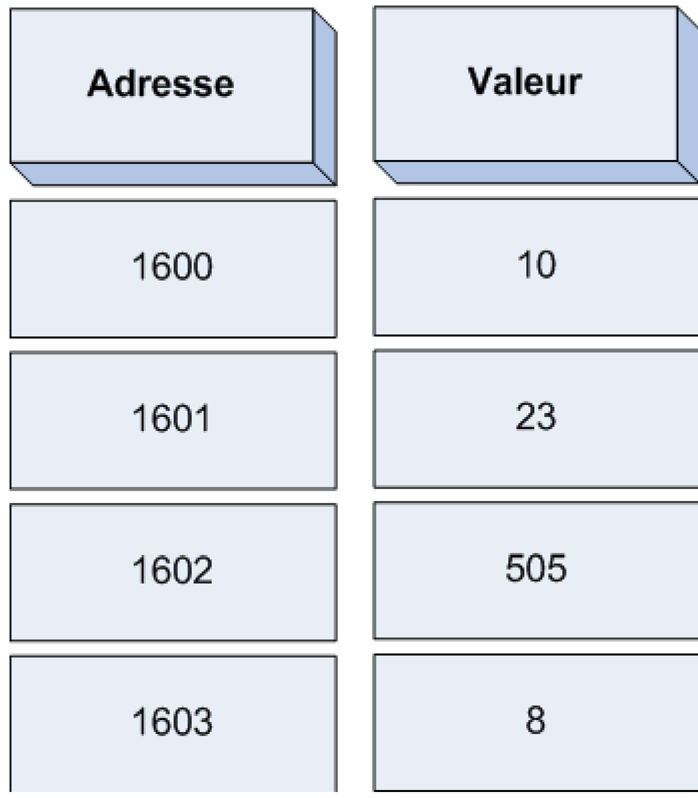
Nous commencerons dans un premier temps par quelques explications sur le fonctionnement des tableaux en mémoire (schémas à l'appui). Croyez-moi, ces petites introductions sur la mémoire sont extrêmement importantes : elles vous permettent de *comprendre* comment cela fonctionne. Un programmeur qui comprend ce qu'il fait, c'est quand même un peu plus rassurant pour la stabilité des programmes non ?

LES TABLEAUX DANS LA MÉMOIRE

"Les tableaux sont une suite de variables de même type, situées dans un espace contigu en mémoire"

Bon, je reconnais que ça fait un peu définition de dictionnaire tout ça
Concrètement, il s'agit de "grosses variables" pouvant contenir plusieurs nombres du même type (long, int, char, double...)

Un tableau a une dimension bien précise. Il peut occuper 2 cases, 3 cases, 10 cases, 150 cases, 2500 cases, c'est vous qui décidez.
Ci-dessous, voici un schéma d'un tableau de 4 cases en mémoire qui commence à l'adresse 1600 :



Lorsque vous demandez à créer un tableau de 4 cases en mémoire, votre programme demande à l'OS la permission d'utiliser 4 cases en mémoire. Ces 4 cases doivent être contiguës, c'est-à-dire les unes à la suite des autres.
Comme vous le voyez, les adresses se suivent : 1600, 1601, 1602, 1603. Il n'y a pas de "trou" au milieu.
Enfin, chaque case du tableau contient un nombre du même type. Si le tableau est de type long, alors chaque case du tableau contiendra un long. On ne peut pas faire de tableau contenant à la fois des long et des double par exemple.

En résumé, voici ce qu'il faut retenir sur les tableaux :

- Lorsqu'un tableau est créé, il prend **un espace contigu en mémoire** : les cases sont les unes à la suite des autres.
- Toutes les cases d'un tableau sont **du même type**. Ainsi, un tableau de int contiendra uniquement des int, et pas autre chose.

DÉFINIR UN TABLEAU

Pour commencer, nous allons voir comment définir un tableau de 4 long :

Code : C

```
long tableau[4];
```

Voilà c'est tout

Il suffit donc de rajouter entre crochets le nombre de cases que vous voulez mettre dans votre tableau. Il n'y a pas de limite (à part peut-être la taille de votre mémoire).

Maintenant, comment accéder à chaque case du tableau ?
Il faut écrire tableau[numeroDeLaCase].

Très important : un tableau commence à l'indice n°0 ! Notre tableau de 4 long a donc les indices 0, 1, 2 et 3. Il n'y a pas d'indice 4 ! C'est une source d'erreurs très courante, souvenez-vous en !

Si je veux mettre dans mon tableau les mêmes valeurs que celles indiquées dans mon schéma, je devrai donc faire :

Code : C

```
long tableau[4];

tableau[0] = 10;
tableau[1] = 23;
tableau[2] = 505;
tableau[3] = 8;
```

Tu as dit qu'il y avait des pointeurs avec les tableaux. Où ça, je n'en vois pas ?

En fait, si vous écrivez juste "tableau", vous avez un pointeur. C'est un pointeur sur la première case du tableau. Faites le test :

Code : C

```
long tableau[4];

printf("%ld", tableau);
```

Résultat, on voit l'adresse où se trouve tableau :

Code : Console

```
1600
```

En revanche, si vous indiquez l'indice de la case du tableau entre crochets, vous obtenez la valeur :

Code : C

```
long tableau[4];

printf("%ld", tableau[0]);
```

Code : Console

```
10
```

De même pour les autres indices.

Notez que, comme tableau est un pointeur, on peut utiliser le symbole * pour connaître la première valeur :

Code : C

```
long tableau[4];

printf("%ld", *tableau);
```

Code : Console

```
10
```

Il est aussi possible d'avoir la valeur de la seconde case en tapant `*(tableau + 1)` (adresse de tableau + 1).
Les 2 lignes suivantes sont donc identiques :

Code : C

```
tableau[1] // Renvoie la valeur contenue dans la seconde case (la première case étant 0)
*(tableau + 1) // Identique : renvoie la valeur contenue dans la seconde case
```

Donc, quand vous écrivez tableau[0], vous demandez la valeur qui se trouve à l'adresse tableau + 0 cases (c'est-à-dire 1600).
Si vous écrivez tableau[1], vous demandez la valeur se trouvant à l'adresse tableau + 1 case (c'est-à-dire 1601).
Et ainsi de suite pour les autres valeurs

Les tableaux à taille dynamique

Le langage C existe en plusieurs versions.
Une version récente, appelée le C99, autorise la création de tableaux à taille dynamique, c'est-à-dire de tableaux dont la taille est définie par une variable :

Code : C

```
long taille = 5;
long tableau[taille];
```

Or, cela n'est pas forcément reconnu par tous les compilateurs, certains planteront sur la ligne n°2.
Le langage C que je vous enseigne depuis le début (appelé le C89) n'autorise pas ce genre de choses. Nous considèrerons donc que faire cela est interdit.

Nous allons nous mettre d'accord sur la chose suivante : vous n'avez pas le droit de mettre une variable entre crochets pour la définition de la taille du tableau, <u>même si cette variable est une constante</u> ! `const long taille = 5;` ne marchera donc pas mieux.
Le tableau doit avoir une dimension fixe, c'est-à-dire que vous devez écrire **noir sur blanc** le nombre correspondant à la taille :

Code : C

```
long tableau[5];
```

Mais... Alors il est interdit de créer un tableau en fonction de la taille d'une variable ?

Non rassurez-vous, c'est possible ! (même en C89)

Mais pour faire cela nous utiliserons une autre technique (plus sûre et qui marche partout) appelée **l'allocation dynamique**. Nous verrons cela bien plus tard dans la partie II de ce cours.

PARCOURIR UN TABLEAU

Supposons que je veuille maintenant afficher les valeurs de chaque case du tableau.
Je pourrais faire autant de *printf* qu'il n'y a de cases. Mais bon, c'est répétitif, un peu lourd, et imaginez le bazar si le tableau contenait 8000 nombres

Le mieux c'est de se servir d'une boucle. Pourquoi pas d'une boucle for ? Les boucles for sont très pratiques pour parcourir un tableau :

Code : C

```
int main(int argc, char *argv[])
{
    long tableau[4], i = 0;

    tableau[0] = 10;
    tableau[1] = 23;
    tableau[2] = 505;
    tableau[3] = 8;

    for (i = 0 ; i < 4 ; i++)
    {
        printf("%ld\n", tableau[i]);
    }

    return 0;
}
```

Code : Console

```
10
23
505
8
```

Notre boucle parcourt le tableau à l'aide d'une variable appelée `i` (c'est le nom super original que les programmeurs donnent en général à la variable qui leur permet de parcourir le tableau)

Ce qui est particulièrement pratique, c'est qu'on peut mettre une variable entre crochets. En effet, la variable était interdite pour la création du tableau (pour définir sa taille), mais elle est heureusement autorisée pour "parcourir" le tableau, c'est-à-dire afficher ses valeurs !
Ici, on a mis la variable i, qui vaut successivement 0, 1, 2, 3. Ainsi, on va donc afficher la valeur de tableau[0], tableau[1], tableau[2] et tableau[3] !

Attention à ne pas tenter d'afficher la valeur de tableau[4] ! Un tableau de 4 cases possède les indices 0, 1, 2 et 3, point barre. Si vous tentez d'afficher tableau[4], vous aurez soit n'importe quoi, soit une belle erreur, l'OS coupant votre programme car il aura tenté d'accéder à une adresse ne lui appartenant pas.

Voilà la technique
Ce n'est pas bien bien compliqué vous voyez.

Initialiser un tableau

Maintenant que l'on sait parcourir un tableau, on est capables d'initialiser toutes ses valeurs à 0 en faisant une boucle

!

Bon, parcourir le tableau pour mettre 0 à chaque case, c'est pas trop dur vous devriez arriver à le faire
Voici le code :

Code : C

```
int main(int argc, char *argv[])
{
    long tableau[4], i = 0;

    // Initialisation du tableau
    for (i = 0 ; i < 4 ; i++)
    {
        tableau[i] = 0;
    }

    // Affichage de ses valeurs pour vérifier
    for (i = 0 ; i < 4 ; i++)
    {
        printf("%ld\n", tableau[i]);
    }

    return 0;
}
```

Code : Console

```
0
0
0
0
```

Une autre façon d'initialiser

Il faut savoir qu'il existe une autre façon d'initialiser un tableau un peu plus automatisée en C.
Elle consiste à écrire tableau[4] = {valeur1, valeur2, valeur3, valeur4}
En clair, vous mettez les valeurs une à une entre accolades, séparées par des virgules :

Code : C

```
int main(int argc, char *argv[])
{
    long tableau[4] = {0, 0, 0, 0}, i = 0;

    for (i = 0 ; i < 4 ; i++)
    {
        printf("%ld\n", tableau[i]);
    }

    return 0;
}
```

Code : Console

```
0
0
0
0
```

Mais en fait, c'est même mieux que ça : vous pouvez définir les valeurs des premières cases du tableau, toutes celles que vous n'aurez pas renseignées seront automatiquement mises à 0.

Ainsi, si je fais :

Code : C

```
long tableau[4] = {10, 23}; // Valeur insérées : 10, 23, 0, 0
```

La case n°0 prendra la valeur 10, la n°1 prendra 23, et toutes les autres prendront la valeur 0 (par défaut).

Comment initialiser tout le tableau à 0 en sachant ça ?
Eh bien, il vous suffit d'initialiser au moins la première valeur à 0, et toutes les autres valeurs non indiquées prendront la valeur 0

Code : C

```
long tableau[4] = {0}; // Toutes les cases du tableau seront initialisées à 0
```

Cette technique a l'avantage de fonctionner avec un tableau de n'importe quelle taille (là ça marche pour 4 cases, mais s'il en avait eu 100 ça aurait été bon aussi)

PASSAGE DE TABLEAUX À UNE FONCTION

Vous aurez sûrement souvent besoin d'afficher tout le contenu de votre tableau.
Pourquoi ne pas écrire une fonction qui fait ça ? Ca va nous permettre de voir comment on envoie un tableau à une fonction en plus, donc ça m'arrange

Il va falloir envoyer 2 informations à la fonction : le tableau (enfin, l'adresse du tableau) et aussi et surtout sa taille !
En effet, notre fonction doit être capable d'initialiser un tableau de n'importe quelle taille. Or, dans votre fonction vous ne connaissez pas la taille de votre tableau. C'est pour cela qu'il faut envoyer en plus une variable que vous appellerez par exemple `tailleTableau`.

Comme je vous l'ai dit, tableau peut être considéré comme un pointeur. On peut donc l'envoyer à la fonction comme on l'aurait fait avec un vulgaire pointeur :

Code : C

```
// Prototype de la fonction d'affichage
void affiche(long *tableau, long tailleTableau);

int main(int argc, char *argv[])
{
    long tableau[4] = {10, 15, 3}, i = 0;

    // On affiche le contenu du tableau
    affiche(tableau, 4);

    return 0;
}

void affiche(long *tableau, long tailleTableau)
{
    long i;

    for (i = 0 ; i < tailleTableau ; i++)
    {
        printf("%ld\n", tableau[i]);
    }
}
```

Code : Console

```
10
15
3
0
```

La fonction n'est pas différente de celles que l'on a étudiées dans le chapitre sur les pointeurs. Elle prend en paramètre un pointeur sur long (notre tableau), ainsi que la taille du tableau (très important pour savoir quand s'arrêter dans la boucle !).
Tout le contenu du tableau est affiché par la fonction via une boucle.

Important : il existe une autre façon d'indiquer que la fonction reçoit un tableau. Plutôt que d'indiquer que la fonction attend un long *tableau, mettez ceci :

Code : C
```c
void affiche(long tableau[], long tailleTableau)
```

Cela revient exactement au même, mais la présence des crochets permet au programmeur de bien voir que c'est un tableau que la fonction prend, et non un simple pointeur. Ca permet d'éviter des confusions
J'utilise personnellement tout le temps les crochets dans mes fonctions pour bien montrer que la fonction attend un tableau. Je vous conseille de faire de même. Il n'est pas nécessaire de mettre la taille du tableau entre les crochets cette fois.

Quelques exercices !

J'ai plein d'idées d'exercices pour vous entraîner !
Je vous propose de réaliser des fonctions travaillant sur des tableaux.

Je donne juste les énoncés des exercices ici pour vous forcer à réfléchir à vos fonctions. Si vous avez du mal à réaliser ces fonctions, rendez-vous sur les forums pour poser vos questions

- **Exercice 1** : créer une fonction *sommeTableau* qui renvoie la somme des valeurs contenues dans le tableau (utilisez un return pour renvoyer la valeur).
  Pour vous aider, voici le prototype de la fonction à créer :

  Code : C
  ```c
  long sommeTableau(long tableau[], long tailleTableau);
  ```

- **Exercice 2** : créer une fonction *moyenneTableau* qui calcule et renvoie la moyenne des valeurs.
  Prototype :

  Code : C
  ```c
  double moyenneTableau(long tableau[], long tailleTableau);
  ```

  La fonction renvoie un double car une moyenne est parfois un nombre décimal (souvent même)

- **Exercice 3** : créer une fonction *copierTableau* qui prend en paramètre 2 tableaux. Le contenu du premier tableau devra être copié dans le second tableau.
  Prototype :

  Code : C
  ```c
  void copie(long tableauOriginal[], long tableauCopie[], long tailleTableau);
  ```

- **Exercice 4** : créer une fonction *maximumTableau* qui aura pour rôle de remettre à 0 toutes les cases du tableau ayant une valeur supérieure à un maximum. Cette fonction prendra en paramètre le tableau ainsi que le nombre maximum autorisé (*valeurMax*). Toutes les cases qui contiennent un nombre supérieur à *valeurMax* doivent être mises à 0.
  Prototype :

  Code : C
  ```c
  void maximumTableau(long tableau[], long tailleTableau, long valeurMax);
  ```

- **Exercice 5** (plus difficile) : créer une fonction *ordonnerTableau* qui classe les valeurs d'un tableau dans l'ordre croissant. Ainsi, un tableau qui vaut {15, 81, 22, 13} doit à la fin de la fonction valoir {13, 15, 22, 81} !
  Cet exercice est un peu plus difficile que les autres, mais est tout à fait réalisable. Ca va vous occuper un petit moment

  Prototype :

  Code : C
  ```c
  void ordonnerTableau(long tableau[], long tailleTableau);
  ```

Faites-vous un petit fichier de fonctions appelé tableaux.c (avec son homologue tableaux.h qui contiendra les prototypes bien sûr !) contenant toutes les fonctions de votre cru réalisant des opérations sur des tableaux
Vous entraîner comme ça, c'est le meilleur moyen de vous former

Au boulot ! Lorsqu'on a appris à se servir des pointeurs, généralement le reste coule de source. Je ne pense pas que ce chapitre vous aura posé trop de problèmes (enfin je peux me tromper hein)
Attention toutefois, cela ne veut pas dire qu'il n'y a pas de pièges. Si je devais vous faire retenir 2 choses auxquelles il faut faire très attention ce serait :

- N'oubliez JAMAIS qu'un tableau commence à l'indice 0, et non pas l'indice 1
- Quand vous envoyez un tableau à une fonction, envoyez toujours à côté la taille du tableau. Sinon, il n'est pas possible de connaître la taille du tableau lorsqu'on doit le parcourir !

Ah au fait, j'ai une bonne nouvelle. **Vous avez maintenant le niveau pour manipuler des chaînes de caractères**, c'est-à-dire du texte. Vous allez pouvoir être capables de retenir du texte dans la mémoire, et donc de demander à l'utilisateur son nom par exemple
Ah il en aura fallu du temps pour faire une chose aussi simple, comme quoi vous voyez même ça c'était pas simple

Nous allons justement étudier les chaînes de caractères dans le prochain chapitre.
(hop hop hop, vous avez vu la transition de dingue que je viens de faire là ?)

Les chaînes de caractères

Une *chaîne de caractères*, c'est un nom programmatiquement correct pour désigner... **du texte**, tout simplement
Une chaîne de caractères est donc du texte que l'on peut retenir sous forme de variable en mémoire. On pourrait ainsi stocker le nom de l'utilisateur.

Comme nous l'avons dit plus tôt, notre ordinateur ne peut retenir que des nombres. Les lettres sont exclues.
Comment diable les programmeurs font-ils pour manipuler du texte alors ?
Y sont malins, z'allez voir

LE TYPE CHAR

Dans ce chapitre, nous allons porter une attention particulière au type char.
Si vous vous souvenez bien, le type char permet de stocker des nombres compris entre -128 et 127.

Si ce type char permet de stocker des nombres, il faut savoir qu'en C on l'utilise rarement pour ça. En général, même si le nombre est petit, on le stocke dans un long ou un int. Certes, ça prend un peu plus de place en mémoire, mais aujourd'hui la mémoire c'est vraiment pas ce qui manque sur un ordinateur
Vous ne tuerez pas votre ordi parce que vous utilisez des int ou des long

Le type char est en fait prévu pour stocker... une lettre !
Attention, j'ai bien dit : UNE lettre.

Comme la mémoire ne peut stocker que des nombres, on a inventé une table qui fait la conversion entre les nombres et les lettres. Cette table indique ainsi par exemple que le nombre 65 équivaut à la lettre A.

Le langage C permet de faire très facilement la traduction : lettre => nombre correspondant. Pour obtenir le nombre associé à une lettre, il suffit de mettre cette lettre entre apostrophes, comme ceci : 'A'. A la compilation, 'A' sera remplacé par la valeur correspondante.

Testons :

Code : C

```c
int main(int argc, char *argv[])
{
    char lettre = 'A';

    printf("%ld\n", lettre);

    return 0;
}
```

Code : Console

```
65
```

On sait donc que la lettre A majuscule est représentée par le nombre 65. B vaut 66, C vaut 67 etc.
Testez avec des minuscules, et vous verrez que les valeurs sont différentes. En effet, la lettre 'a' n'est pas identique à la lettre 'A', l'ordinateur faisant la différence entre les majuscules et les minuscules.

La plupart des caractères "de base" sont codés entre les nombres 0 et 127.

Afficher un caractère

La fonction printf, qui n'a décidemment pas fini de nous étonner, peut aussi afficher un caractère. Pour cela, on doit utiliser le symbole %c (c comme caractère) :

Code : C

```
int main(int argc, char *argv[])
{
    char lettre = 'A';

    printf("%c\n", lettre);

    return 0;
}
```

Code : Console

```
A
```

Hourra !
Nous savons afficher une lettre

On peut aussi demander à l'utilisateur de rentrer une lettre en utilisant le %c dans un scanf :

Code : C

```
int main(int argc, char *argv[])
{
    char lettre = 0;

    scanf("%c", &lettre);
    printf("%c\n", lettre);

    return 0;
}
```

Si je tape la lettre B, je verrai :

Code : Console

```
B
B
```

(le premier des 2 B étant celui que j'ai tapé au clavier, et le second étant affiché par le printf)

Voici à peu près tout ce qu'il faut savoir sur le type char. Il cachait décidemment bien son jeu celui-là

Retenez bien :

- Le type char permet de stocker des nombres allant de -128 à 127, unsigned char des nombres de 0 à 255.
- Il y a une table que votre ordinateur utilise pour convertir les lettres en nombres et inversement.
- On peut donc utiliser le type char pour stocker UNE lettre.
- 'A' est remplacé à la compilation par la valeur correspondante (65 en l'occurrence). On utilise donc les apostrophes pour obtenir la valeur d'une lettre.

LES CHAÎNES SONT DES TABLEAUX DE CHAR !

Arf, j'ai tout dit dans le titre, qu'est-ce que je vais bien pouvoir raconter maintenant
Ben oui, tout y est : une chaîne de caractères n'est rien d'autre qu'un tableau de type char. Un bête tableau de rien du tout

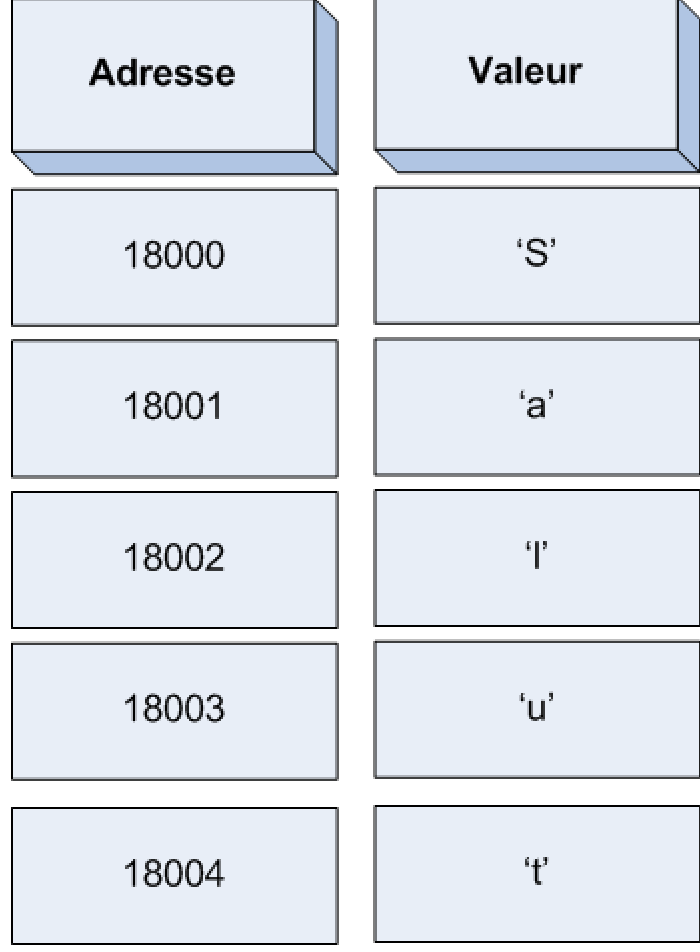
Si on crée un tableau :

Code : C

```
char chaine[5];
```

... et qu'on met dans chaine[0] la lettre 'S', dans chaine[1] la lettre 'a', etc... On peut ainsi former une chaîne de caractères, c'est-à-dire du texte

Voici un schéma de la façon dont ça pourrait être stocké en mémoire (attention je vous préviens de suite, c'est un peu plus compliqué que ça en réalité, je vous explique après pourquoi) :



Comme on peut le voir, c'est un tableau qui prend 5 cases en mémoire pour représenter le mot "Salut". Pour la valeur, j'ai mis exprès les lettres entre apostrophes, pour indiquer que c'est un nombre qui est stocké et non une lettre. En réalité, dans la mémoire ce sont bel et bien les valeurs de ces lettres qui sont stockées

<u>Oui mais attention</u>, une chaîne de caractères ne contient pas que des lettres ! Le schéma que vous voyez ci-dessus est en fait incomplet.
Une chaîne de caractère **doit impérativement contenir un caractère spécial à la fin de la chaîne**, appelé "Caractère de fin de chaîne". Ce caractère s'écrit '\0'.

Pourquoi devoir terminer une chaîne de caractères par un \0 ?

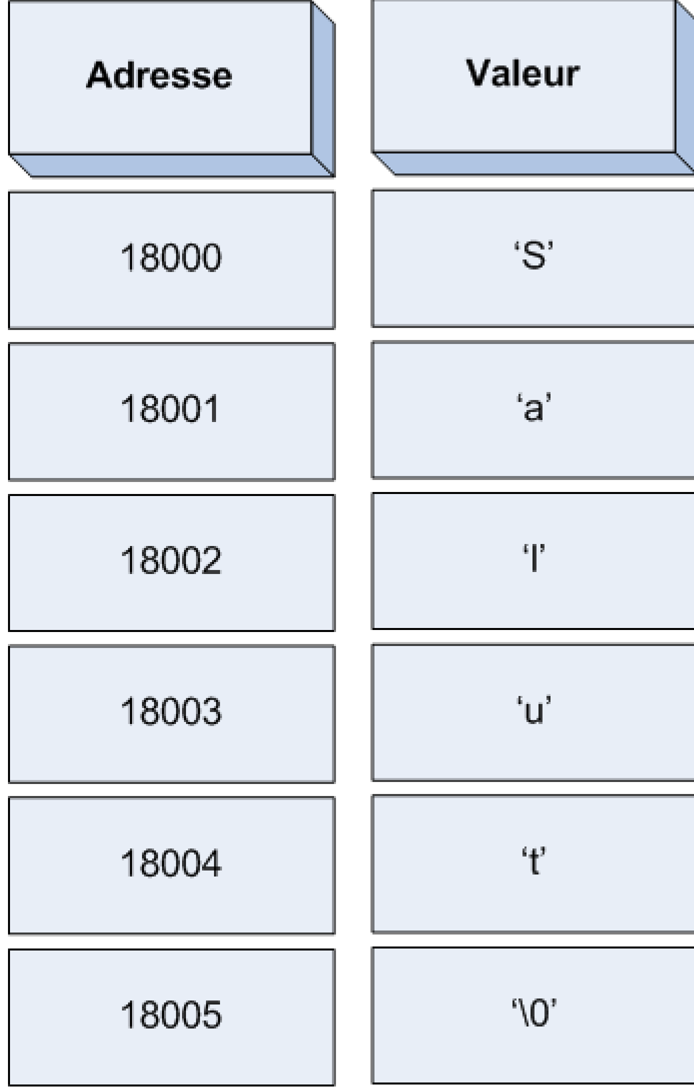
Tout simplement pour que votre ordinateur sache quand s'arrête la chaîne ! Le caractère \0 permet de dire : "Stop, c'est fini, y'a plus rien à lire après circulez !"

Par conséquent, pour stocker le mot "Salut" (qui comprend 5 lettres) en mémoire, **il ne faut pas un tableau de 5 char, il faut un tableau de 6 char** !
A chaque fois que vous créez une chaîne de caractères, <u>vous allez donc devoir penser à prévoir de la place pour le</u>

caractère de fin de chaîne. Il faut toujours toujours toujours rajouter un bloc de plus dans le tableau pour stocker ce caractère \0, c'est impératif !

Oublier le caractère de fin \0, c'est une source d'erreurs impitoyable du langage C. Je le sais, j'en ai fait les frais et pas qu'une fois

En clair, le **bon** schéma de la chaîne de caractères "Salut" en mémoire est le suivant :



Comme vous le voyez, la chaîne prend 6 caractères et non pas 5, il va falloir s'y faire
La chaîne se termine par '\0', le caractère de fin de chaîne qui permet d'indiquer à l'ordinateur que la chaîne se termine là.

Voyez le caractère \0 comme un avantage. Grâce à lui, vous n'aurez pas à retenir la taille de votre tableau car il indique que le tableau s'arrête là Vous pourrez passer votre tableau de char à une fonction sans avoir à ajouter à côté une variable indiquant la taille du tableau.
Cela n'est valable que pour les chaînes de caractères (c'est-à-dire le type char*, qu'on peut aussi écrire char[]). Pour les autres types de tableaux, vous êtes toujours obligés de retenir la taille du tableau quelque part.

Création et initialisation de la chaîne

Si on veut initialiser notre tableau chaine avec le texte "Salut", on peut utiliser la méthode old-school un peu bourrin :

Code : C

```
char chaine[6]; // Tableau de 6 char pour stocker S-a-l-u-t + le \0

chaine[0] = 'S';
chaine[1] = 'a';
chaine[2] = 'l';
chaine[3] = 'u';
chaine[4] = 't';
chaine[5] = '\0';
```

Cette méthode marche.
On peut le vérifier en faisant un printf.

Ah oui, j'allais oublier le printf : y'a encore un nouveau symbole à retenir
C'est le %s (s comme string, qui signifie "chaîne" en anglais).
Voici le code complet qui crée une chaîne "Salut" en mémoire et qui l'affiche :

Code : C

```
#include <stdio.h>
#include <stdlib.h>



int main(int argc, char *argv[])
{
    char chaine[6]; // Tableau de 6 char pour stocker S-a-l-u-t + le \0

    // Initialisation de la chaîne (on écrit les caractères 1 à 1 en mémoire)
    chaine[0] = 'S';
    chaine[1] = 'a';
    chaine[2] = 'l';
    chaine[3] = 'u';
    chaine[4] = 't';
    chaine[5] = '\0';

    // Affichage de la chaîne grâce au %s du printf
    printf("%s", chaine);

    return 0;
}
```

Résultat :

Code : Console

```
Salut
```

Pfiou
Tout ça pour stocker "Salut" en mémoire et l'afficher quand même

C'est un peu fatigant et répétitif de devoir écrire les caractères un à un comme on l'a fait dans le tableau chaine.
Pour initialiser une chaîne, il existe heureusement une méthode plus simple :

Code : C

```
int main(int argc, char *argv[])
{
    char chaine[] = "Salut"; // La taille du tableau chaine est automatiquement calculée


    printf("%s", chaine);

    return 0;
}
```

Code : Console

```
Salut
```

Comme vous le voyez à la première ligne, je crée une variable de type char[]. J'aurais pu écrire aussi char*, le résultat aurait été le même.

En tapant entre guillemets la chaîne que vous voulez mettre dans votre tableau, le compilateur C calcule automatiquement la taille nécessaire. C'est-à-dire qu'il compte les lettres et rajoute 1 pour placer le caractère \0. Il écrit ensuite une à une les lettres du mot "Salut" en mémoire et rajoute l'\0 comme on l'a fait nous-mêmes manuellement quelques instants plus tôt.
Bref, c'est bien pratique

Défaut : ça ne marche que pour l'initialisation ! Vous ne pouvez pas écrire plus loin dans le code :

Code : C

```
chaine = "Salut";
```

Cette technique est donc à réserver à l'initialisation. Après cela, il faudra écrire les caractères manuellement un à un en mémoire

Récupération d'une chaîne via un scanf

Vous pouvez enregistrer une chaîne rentrée par l'utilisateur via un scanf, en utilisant là encore le symbole %s.
Seul problème : vous ne savez pas combien de caractères l'utilisateur va rentrer. Si vous lui demandez son prénom, il s'appelle peut-être Luc (3 caractères), mais qui vous dit qu'il ne s'appelle pas Jean-Edouard (beaucoup plus de caractères) ?

Pour ça, il n'y a pas 36 solutions. Il va falloir créer un tableau de char très grand, suffisamment grand pour pouvoir stocker le prénom. On va donc créer un char[100] pour stocker le prénom. Ca donne l'impression de gâcher de la mémoire, mais souvenez-vous encore une fois que de la place en mémoire c'est pas ce qui manque (et y'a des programmes qui gâchent de la mémoire de manière bien pire que ça, si vous saviez)

Code : C

```
int main(int argc, char *argv[])
{
    char prenom[100];

    printf("Comment t'appelles-tu petit Zer0 ? ");
    scanf("%s", prenom);
    printf("Salut %s, je suis heureux de te rencontrer !", prenom);

    return 0;
}
```

Code : Console

```
Comment t'appelles-tu petit Zer0 ? Mateo21
Salut Mateo21, je suis heureux de te rencontrer !
```

Voilà en gros comment ça se passe pour demander d'entrer du texte à l'utilisateur

FONCTIONS DE MANIPULATION DES CHAÎNES

Les chaînes de caractères sont, vous vous en doutez, fréquemment utilisées. Tous les mots, tous les textes que vous voyez à votre écran sont en fait des tableaux de char en mémoire qui fonctionnent de la manière que je viens de vous expliquer (ah on ne voit plus son ordinateur de la même façon du coup hein ?)
Afin de nous aider un peu à manipuler les chaînes, on nous fournit dans la librairie string.h une pléthore de fonctions dédiées aux calculs sur des chaînes.

Je ne peux pas vraiment toutes vous les présenter ici, ce serait un peu long et elles ne sont pas toutes forcément indispensables.
Je vais me contenter de vous parler des principales dont vous aurez très certainement besoin dans peu de temps, ce qui fait déjà pas mal

Pensez à inclure string.h

Même si cela devrait vous paraître évident, je préfère vous le préciser encore au cas où : comme on va utiliser une nouvelle librairie appelée string.h, vous devez l'inclure en haut des fichiers .c où vous en avez besoin :

Code : C

```
#include <string.h>
```

Si vous ne le faites pas, l'ordinateur ne connaîtra pas les fonctions que je vais vous présenter car il n'aura pas les prototypes, et la compilation plantera.
Bref, n'oubliez pas d'inclure cette librairie à chaque fois que vous utilisez des fonctions de manipulation de chaînes

strlen : calculer la longueur d'une chaîne

strlen est une fonction qui calcule la longueur d'une chaîne de caractères (sans compter le caractère \0).
Vous devez lui envoyer un seul paramètre : votre chaîne de caractères ! Cette fonction vous retourne la longueur de la chaîne.

Maintenant que vous savez ce qu'est un prototype, je vais vous donner le prototype des fonctions dont je vous parle. Les programmeurs s'en servent comme "mode d'emploi" de la fonction (même si quelques explications à côté ne sont jamais superflues) :

Code : C

```
size_t strlen(const char* chaine);
```

size_t est un type spécial qui signifie que la fonction renvoie un nombre correspondant à une taille. Ce n'est pas un type de base comme int, long ou char, c'est un type "inventé". Nous apprendrons nous aussi à créer nos propres types de variables quelques chapitres plus loin.
Pour le moment, on va se contenter de stocker la valeur renvoyée par strlen dans une variable de type *long* (l'ordinateur convertira de *size_t* en *long* automatiquement). En toute rigueur, il faudrait plutôt stocker le résultat dans une variable de type *size_t*, mais en pratique un *long* est suffisant pour cela.

La fonction prend un paramètre de type const char*. Le const (qui signifie constante, rappelez-vous) fait que la

fonction strlen "s'interdit" en quelque sorte de modifier votre chaîne. Quand vous voyez un const, vous savez que la variable n'est pas modifiée par la fonction, elle est juste lue.

Testons la fonction strlen :

Code : C

```
int main(int argc, char *argv[])
{
    char chaine[] = "Salut";
    long longueurChaine = 0;

    // On récupère la longueur de la chaîne dans longueurChaine
    longueurChaine = strlen(chaine);

    // On affiche la longueur de la chaîne
    printf("La chaine %s fait %ld caracteres de long", chaine, longueurChaine);

    return 0;
}
```

Code : Console

```
La chaine Salut fait 5 caracteres de long
```

Cette fonction strlen est d'ailleurs facile à écrire. Il suffit de faire une boucle sur le tableau de char qui s'arrête quand on tombe sur le caractère \0. Un compteur s'incrémente à chaque tour de boucle, et c'est ce compteur que la fonction retourne.

Allez, ça m'a donné envie d'écrire moi-même une fonction similaire à strlen
Ca vous permettra en plus de bien comprendre comment la fonction marche :

Code : C

```
long longueurChaine(const char* chaine);

int main(int argc, char *argv[])
{
    char chaine[] = "Salut";
    long longueur = 0;

    longueur = longueurChaine(chaine);

    printf("La chaine %s fait %ld caracteres de long", chaine, longueur);

    return 0;
}

long longueurChaine(const char* chaine)
{
    long nombreDeCaracteres = 0;
    char caractereActuel = 0;

    do
    {
        caractereActuel = chaine[nombreDeCaracteres];
        nombreDeCaracteres++;
    }
    while(caractereActuel != '\0'); // On boucle tant qu'on n'est pas arrivé à l'\0

    nombreDeCaracteres--; // On retire 1 caractère de long pour ne pas compter l'\0

    return nombreDeCaracteres;
}
```

La fonction longueurChaine fait une boucle sur le tableau chaine. Elle stocke les caractères un par un dans

caractereActuel. Dès que caractèreActuel vaut '\0', la boucle s'arrête.
A chaque passage dans la boucle, on ajoute 1 au nombre de caractères qu'on a analysé.

A la fin de la boucle, on retire 1 caractère au nombre total de caractères qu'on a comptés. Cela permet de ne pas compter le caractère \0 dans le lot.
Enfin, on retourne nombreDeCaracteres, et le tour est joué

strcpy : copier une chaîne dans une autre

La fonction strcpy (comme "string copy") permet de copier une chaîne à l'intérieur d'une autre.
Son prototype est :

Code : C

```
char* strcpy(char* copieDeLaChaine, const char* chaineACopier);
```

Cette fonction prend 2 paramètres :

- copieDeLaChaine : c'est un pointeur vers un char* (tableau de char). C'est dans ce tableau que la chaîne sera copiée.
- chaineACopier : c'est un pointeur vers un autre tableau de char. Cette chaîne sera copiée dans copieDeLaChaine.

La fonction renvoie un pointeur sur copieDeLaChaine, ce qui n'est pas très utile. En général, on ne récupère pas ce que cette fonction renvoie.

Allons tester ça :

Code : C

```
int main(int argc, char *argv[])
{
    /* On crée une chaine "chaine" qui contient un peu de texte
    et une copie (vide) de taille 100 pour être sûr d'avoir la place
    pour la copie */
    char chaine[] = "Texte", copie[100] = {0};

    strcpy(copie, chaine); // On copie "chaine" dans "copie"

    // Si tout s'est bien passé, la copie devrait être identique à chaine
    printf("chaine vaut : %s\n", chaine);
    printf("copie vaut : %s\n", copie);


    return 0;
}
```

Code : Console

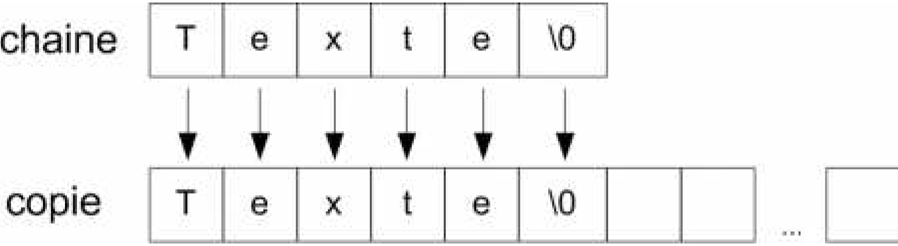
```
chaine vaut : Texte
copie vaut : Texte
```

On voit que chaine vaut "Texte". Bon ça c'est normal
Par contre, on voit aussi que la variable copie, qui était vide au départ, a été remplie par le contenu de chaine. La chaine a donc bien été copiée dans "copie"

Vérifiez que la chaîne "copie" est assez grande pour accueillir le contenu de "chaine". Si, dans mon

exemple, j'avais défini copie[5] (ce qui n'est pas suffisant car il n'y aurait pas eu de place pour l'\0), la fonction strcpy aurait "débordé en mémoire" et probablement fait planter votre programme. A éviter à tout prix, sauf si vous aimez faire crasher votre ordinateur bien sûr

Schématiquement, il s'est passé ça en mémoire :



Chaque caractère de chaine a été placé dans copie.
La chaîne copie contient de nombreux caractères inutilisés, vous l'aurez remarqué. Je lui ai donné la taille 100 par sécurité, mais en toute rigueur la taille 6 aurait suffit. L'avantage de créer un tableau un peu plus grand, c'est que de cette façon la chaîne copie sera capable de recevoir d'autres chaînes peut-être plus grandes dans la suite du programme.

strcat : concaténer 2 chaînes

Cette fonction ajoute une chaîne à la suite d'une autre. On appelle cela la concaténation.
Si j'ai :

- chaine1 = "Salut "
- chaine2 = "Mateo21"

Si je concatène chaine2 dans chaine1, alors chaine1 vaudra "Salut Mateo21".
chaine2, elle, n'aura pas changé et vaudra donc toujours "Mateo21". Seule chaine1 est modifiée.

C'est exactement ce que fait strcat, dont voici le prototype :

Code : C

```
char* strcat(char* chaine1, const char* chaine2);
```

Comme vous pouvez le voir, chaine2 ne peut pas être modifiée car elle est définie comme constante dans le prototype de la fonction.
La fonction retourne un pointeur vers chaine1 ce qui, comme pour strcpy, ne sert pas à grand-chose dans le cas présent, donc on peut ignorer ce que la fonction nous renvoie.

La fonction ajoute à chaine1 le contenu de chaine2. Regardons-y de plus près :

Code : C

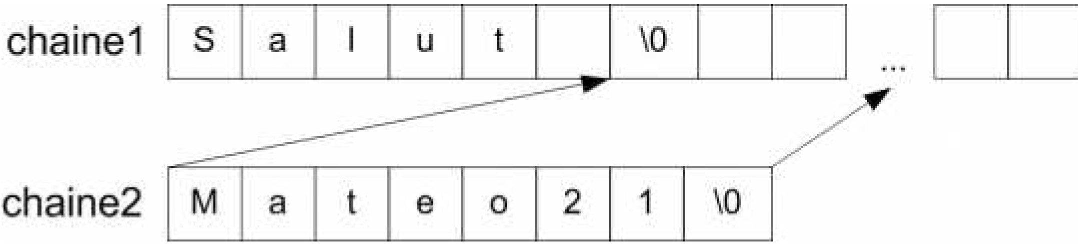
```
int main(int argc, char *argv[])
{
    /* On crée 2 chaînes. chaine1 doit être assez grande pour accueillir
    le contenu de chaine2 en plus, sinon risque de plantage */
    char chaine1[100] = "Salut ", chaine2[] = "Mateo21";

    strcat(chaine1, chaine2); // On concatène chaine2 dans chaine1

    // Si tout s'est bien passé, chaine1 vaut "Salut Mateo21"
    printf("chaine1 vaut : %s\n", chaine1);
    // chaine2 n'a pas changé :
    printf("chaine2 vaut toujours : %s\n", chaine2);

    return 0;
}
```

Code : Console

```
chaine1 vaut : Salut Mateo21
chaine2 vaut toujours : Mateo21
```

Vérifiez absolument que chaine1 est assez grande pour qu'on puisse lui rajouter le contenu de chaine2, sinon vous ferez un débordement en mémoire qui peut conduire à un plantage.
C'est pour cela que j'ai défini chaine1 de taille 100. Quant à chaine2, j'ai laissé l'ordinateur calculer sa taille (je n'ai donc pas précisé la taille) car cette chaîne n'est pas modifiée, il n'y a donc pas besoin de la rendre plus grande que nécessaire

Schématiquement il s'est passé ça :



Le tableau chaine2 a été ajouté à la suite de chaine1 (qui comprenait une centaine de cases)
L'\0 de chaine1 a été supprimé (en fait il a été remplacé par le M de Mateo21). En effet, il ne faut pas laisser un \0 au milieu de la chaîne sinon celle-ci aurait été "coupée" au milieu ! On ne met qu'un \0 à la fin de la chaîne, une fois qu'elle est finie.

strcmp : comparer 2 chaînes

strcmp compare 2 chaînes entre elles. Voici son prototype :

Code : C

```
int strcmp(const char* chaine1, const char* chaine2);
```

Les variables chaine1 et chaine2 sont comparées. Comme vous le voyez, aucune d'elles n'est modifiée car elles sont indiquées comme constantes.

Il est important de récupérer ce que la fonction renvoie. En effet, strcmp renvoie :

- 0 si les chaînes sont identiques
- Une autre valeur (positive ou négative) si les chaînes sont différentes

Il aurait été plus logique, je le reconnais, que la fonction renvoie 1 si les chaînes sont identiques pour dire "vrai" (rappelez-vous des booléens). Toutefois, comme ce n'est pas moi qui ai codé la fonction...

Plus sérieusement, la fonction compare les valeurs de chacun des caractères un à un. Si tous les caractères sont identiques, elle renvoie 0. Si les caractères de la chaine1 sont supérieurs à ceux de la chaine2, la fonction renvoie un nombre positif. Si c'est l'inverse, la fonction renvoie un nombre négatif.
Dans la pratique, on se sert surtout de strcmp pour vérifier si 2 chaînes sont identiques, point barre

Voici un code de test :

Code : C

```c
int main(int argc, char *argv[])
{
    char chaine1[] = "Texte de test", chaine2[] = "Texte de test";

    if (strcmp(chaine1, chaine2) == 0) // Si strcmp renvoie 0 (chaînes identiques)
    {
        printf("Les chaines sont identiques\n");
    }
    else
    {
        printf("Les chaines sont differentes\n");
    }

    return 0;
}
```

Code : Console

```
Les chaines sont identiques
```

Les chaînes étant identiques, la fonction strcmp a renvoyé le nombre 0.
Notez que j'aurais pu stocker ce que renvoie strcmp dans une variable de type int. Toutefois, ce n'est pas obligatoire, on peut directement mettre la fonction dans le if comme je l'ai fait.

Je n'ai pas grand-chose à rajouter à propos de cette fonction. Elle est assez simple à utiliser en fait, mais il ne faut pas oublier que 0 signifie "identique" et une autre valeur signifie "différent". C'est la seule source d'erreurs possible ici.

strchr : rechercher un caractère

La fonction strchr recherche un caractère dans une chaîne.
Prototype :

Code : C

```c
char* strchr(const char* chaine, int caractereARechercher);
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite.
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

Vous remarquerez que caractereARechercher est de type int et non de type char. Ce n'est pas réellement un problème car, au fond, un caractère est et restera toujours un nombre
Néanmoins, on utilise quand même plus souvent un char qu'un int pour stocker un caractère en mémoire.

La fonction renvoie un pointeur vers **le premier caractère** qu'elle a trouvé, c'est-à-dire qu'elle renvoie l'adresse de ce caractère dans la mémoire. Elle renvoie NULL si elle n'a rien trouvé.
Dans l'exemple suivant, je récupère ce pointeur dans suiteChaine :

Code : C

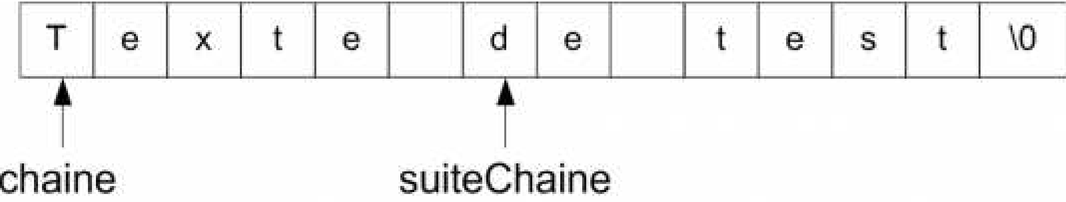
```
int main(int argc, char *argv[])
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;

    suiteChaine = strchr(chaine, 'd');
    if (suiteChaine != NULL) // Si on a trouvé quelque chose
    {
        printf("Voici la fin de la chaine a partir du premier d : %s", suiteChaine);
    }

    return 0;
}
```

Code : Console

```
Voici la fin de la chaine a partir du premier d : de test
```

Avez-vous bien compris ce qu'il se passe ici ? C'est un peu particulier.
En fait, suiteChaine est un pointeur comme chaine. Sauf que chaine pointe sur le premier caractère (le 'T' majuscule), tandis que suiteChaine pointe sur le premier caractère 'd' qui a été trouvé dans chaine.

Le schéma suivant vous montre où pointe chaque pointeur :



chaine commence au début de la chaine ('T' majuscule), tandis que suiteChaine pointe sur le 'd' minuscule.

Lorsque je fais un printf de suiteChaine, il est donc normal que l'on m'affiche juste "de test". La fonction printf affiche tous les caractères qu'elle rencontre ('d', 'e', ' ', 't', 'e', 's', 't') jusqu'à ce qu'elle tombe sur l'\0 qui lui dit que la chaîne s'arrête là.

Variante

Il existe une fonction strrchr strictement identique à strchr, sauf que celle-là renvoie un pointeur vers **le dernier caractère** qu'elle a trouvé dans la chaîne au lieu du premier

strpbrk : premier caractère de la liste

Cette fonction ressemble beaucoup à la précédente. Celle-ci recherche un des caractères dans la liste que vous lui

donnez sous forme de chaîne, contrairement à strchr qui ne peut rechercher qu'un seul caractère à la fois.

Par exemple, si on forme la chaîne "xds" et qu'on en fait une recherche dans "Texte de test", la fonction renvoie un pointeur vers le premier de ces caractères qu'elle a trouvé dedans. En l'occurrence, le premier caractère de "xds" qu'elle trouve dans "Texte de test" est le x, donc strpbrk renverra un pointeur sur 'x'.

Prototype :

Code : C

```
char* strpbrk(const char* chaine, const char* lettresARechercher);
```

Test :

Code : C

```
int main(int argc, char *argv[])
{
    char *suiteChaine;

// On cherche la première occurrence de x, d ou s dans "Texte de test"
    suiteChaine = strpbrk("Texte de test", "xds");

    if (suiteChaine != NULL)
    {
        printf("Voici la fin de la chaine a partir du premier des caracteres trouves :
%s", suiteChaine);
    }

    return 0;
}
```

Code : Console

```
Voici la fin de la chaine a partir du premier des caracteres trouves : xte de test
```

Pour cet exemple, j'ai directement écrit les valeurs à envoyer à la fonction (entre guillemets). On n'est en effet pas obligés d'employer une variable à tous les coups, on peut très bien écrire la chaîne directement.
Il faut simplement retenir la règle suivante :

- Si vous utilisez les guillemets "", cela signifie **chaîne**.
- Si vous utilisez les apostrophes '', cela signifie **caractère**.

strstr : rechercher une chaîne dans une autre

Cette fonction recherche la première occurrence d'une chaîne dans une autre chaîne.
Son prototype est :

Code : C

```
char* strstr(const char* chaine, const char* chaineARechercher);
```

Le prototype est similaire à strpbrk, mais attention à ne pas confondre : strpbrk recherche UN des caractères, tandis que strstr recherche toute la chaîne.

Exemple :

Code : C

```
int main(int argc, char *argv[])
{
    char *suiteChaine;

    // On cherche la première occurrence de "test" dans "Texte de test" :
    suiteChaine = strstr("Texte de test", "test");
    if (suiteChaine != NULL)
    {
        printf("Premiere occurence de test dans Texte de test : %s\n", suiteChaine);
    }

    return 0;
}
```

Code : Console

```
Premiere occurence de test dans Texte de test : test
```

La fonction strstr recherche la chaîne "test" dans "Texte de test".
Elle renvoie, comme les autres, un pointeur quand elle a trouvé ce qu'elle cherchait. Elle renvoie NULL si elle n'a rien trouvé.

Pensez à vérifier si vos fonctions de recherche n'ont pas renvoyé NULL. Si vous ne le faites pas et que vous essayez d'afficher une chaîne qui pointe sur NULL, votre programme plantera. Votre OS fermera brutalement votre programme car il aura essayé d'accéder à l'adresse NULL à laquelle il n'a pas le droit d'accéder.

Jusqu'ici, je me suis contenté d'afficher la chaîne à partir du pointeur retourné par les fonctions. Dans la pratique, ça n'est pas très utile Vous ferez juste un `if (resultat != NULL)` pour savoir si la recherche a trouvé quelque chose ou si elle n'a rien trouvé, et vous afficherez "*Le texte que vous recherchiez a été trouvé*".
Enfin, cela dépend de votre programme, mais en tout cas ces fonctions sont la base si vous voulez faire un traitement de texte

sprintf : écrire dans une chaîne

Cette fonction se trouve dans stdio.h contrairement aux autres fonctions que nous avons étudiées jusqu'ici, qui étaient dans string.h

Ce nom doit vaguement vous rappeler quelque chose
Cette fonction ressemble énormément au printf que vous connaissez mais, au lieu d'écrire à l'écran, sprintf écrit dans... une chaîne ! D'où son nom d'ailleurs, qui commence par le "s" de "string" (chaîne en anglais).

C'est une fonction très pratique pour mettre en forme une chaîne. Petit exemple :

Code : C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        char chaine[100];
        long age = 15;

        // On écrit "Tu as 15 ans" dans chaine
        sprintf(chaine, "Tu as %ld ans !", age);

        // On affiche chaine pour vérifier qu'elle contient bien cela :
        printf("%s", chaine);

        return 0;
}
```

Code : Console

```
Tu as 15 ans !
```

Elle s'utilise de la même manière que printf, mis à part le fait que vous devez lui donner en premier paramètre un pointeur vers la chaîne qui doit recevoir le texte.
Dans mon exemple, j'écris dans chaine "Tu as %ld ans", où %ld est remplacé par le contenu de la variable age. Toutes les règles du printf s'appliquent, vous pouvez donc si vous le voulez mettre des %s pour insérer d'autres chaînes à l'intérieur de votre chaîne

Comme d'hab, vérifiez que votre chaîne est suffisamment grande pour accueillir tout le texte que le sprintf va lui envoyer. Sinon, ben... boum
Les chaînes de caractères sont, il faut dire ce qui est, assez délicates à manipuler en langage C.

Sachez que je ne connais pas moi-même toutes les fonctions de string.h : je ne vous demande donc pas de les retenir par coeur. En revanche, vous devez savoir comment une chaîne de caractères fonctionne avec l'\0 et tout ça

Souvenez-vous que le langage C est globalement "assez bas niveau" c'est-à-dire que vous êtes près du fonctionnement de votre ordinateur.
L'avantage est que vous comprenez aujourd'hui comment votre ordinateur gère le texte. Ce que vous apprenez là sera payant dans le futur, je peux vous l'assurer. Contrairement à quelqu'un qui programme en Java ou en Basic qui n'a pas besoin de savoir comment marche un ordinateur, vous vous commencez à vraiment comprendre le fonctionnement de votre ordinateur, et ça c'est je trouve très important.

Bien entendu, il y a un défaut : ce chapitre est un petit peu compliqué. Il faut bien prévoir la taille de son tableau, penser à enregistrer l'\0 etc... Les chaînes de caractères ne sont pas évidentes à manipuler pour un débutant donc, mais avec un peu de pratique ça vient tout seul.

La pratique, parlons-en justement ! J'ai du boulot pour vous
Je vous conseille ultra fortement de vous entraîner. Quoi de mieux que de travailler sur les chaînes de caractères pour ça ? Ca vous fait travailler les chaînes, les tableaux et les pointeurs à la fois... si c'est pas merveilleux

Voici ce que je vous propose de faire : nous avons étudié sur la fin de ce chapitre un petit nombre de fonctions issues de la librairie string.h. Vous êtes parfaitement capables de les écrire vous-mêmes. Faites-le.

Mais, est-ce bien utile ? Si les fonctions ont été écrites avant nous, on va pas s'embêter à les refaire !

Effectivement ce n'est pas très utile, et vous utiliserez dans le futur sûrement les fonctions de la librairie string.h et non les vôtres. Toutefois, comme je vous l'ai dit, ça vous permet de vous entraîner et c'est, je trouve, un très bon exercice. Je vous ai déjà montré le fonctionnement de strlen d'ailleurs, ça devrait vous aider à réaliser les autres fonctions.
En revanche, n'essayez pas de recoder sprintf qui est une fonction assez complexe. Contentez-vous des fonctions issues de string.h

Si vous coincez sur une fonction, n'hésitez pas à demander de l'aide sur les forums !

Allez au boulot !

Le préprocesseur

Après ces derniers chapitres harassants sur les pointeurs, tableaux et chaînes de caractères, nous allons faire **une pause**.
Je veux dire par là que vous avez dû encaisser pas mal de chocs dans les chapitres précédents, et que je ne peux donc pas vous refuser de souffler un peu

Ceci étant, pas question de se reposer sans rien apprendre (ça va pas la tête ?). On va donc voir ensemble un chapitre simple, contenant d'ailleurs quelques rappels. Ce chapitre va traiter du préprocesseur, ce programme qui s'exécute juste avant la compilation.
Ne vous y trompez pas : les informations contenues dans ce chapitre vous seront utiles. Elles sont juste faciles à comprendre... et ça nous arrange

Les includes

Comme je vous l'ai expliqué dans les tous premiers chapitres du cours, on trouve dans les codes sources des lignes un peu particulières appelées **directives de préprocesseur**.
Ces directives de préprocesseur ont la caractéristique suivante : elles commencent toujours par le symbole #. Elles sont donc faciles à reconnaître.

La première (et seule) directive que nous ayons vue pour l'instant est #include.
Cette directive permet d'inclure le contenu d'un fichier dans un autre, je vous l'ai dit plus tôt.
On s'en sert en particulier pour inclure des fichiers .h comme les fichiers .h des librairies (stdlib.h, stdio.h, string.h, math.h...) et vos propres fichiers .h.

Pour inclure un fichier .h se trouvant dans **le dossier où est installé votre IDE**, vous devez utiliser les chevrons < > :

Code : C

```
#include <stdlib.h>
```

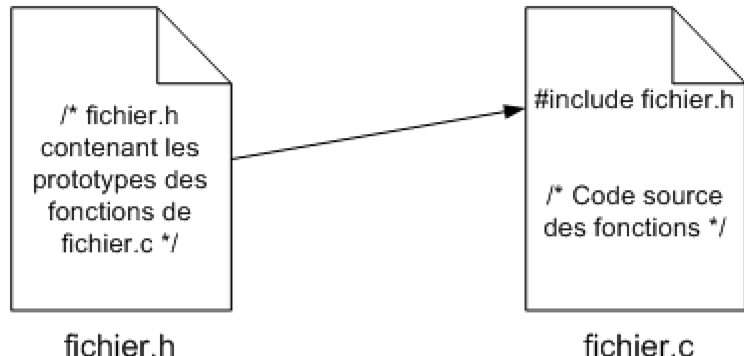
Pour inclure un fichier .h se trouvant dans **le dossier de votre projet**, vous devez utiliser les guillemets :

Code : C

```
#include "monfichier.h"
```

Concrètement, le préprocesseur est démarré avant la compilation. Il parcourt tous vos fichiers à la recherche de directives de préprocesseur, ces fameuses lignes qui commencent par un #
Lorsqu'il rencontre la directive #include, il met littéralement le contenu du fichier indiqué à l'endroit du #include.

Supposons que j'aie un "fichier.c" contenant le code de mes fonctions et un "fichier.h" contenant les prototypes des fonctions de fichier.c. On pourrait résumer la situation dans ce schéma tout simple :



Tout le contenu de fichier.h est mis à l'intérieur de fichier.c, à l'endroit où il y a la directive #include fichier.h

Imaginons qu'on ait dans le fichier.c :

Code : C

```
#include "fichier.h"

long maFonction(int truc, double bidule)
{
    /* Code de la fonction */
}

void autreFonction(long valeur)
{
    /* Code de la fonction */
}
```

Et dans le fichier.h :

Code : C

```
long maFonction(int truc, double bidule);
void autreFonction(long valeur);
```

Lorsque le préprocesseur passe par là, juste avant la compilation de fichier.c, il met fichier.h dans fichier.c. Au final, le code source de fichier.c *juste avant* la compilation ressemble à ça :

Code : C

```
long maFonction(int truc, double bidule);
void autreFonction(long valeur);

long maFonction(int truc, double bidule)
{
    /* Code de la fonction */
}

void autreFonction(long valeur)
{
    /* Code de la fonction */
}
```

Le contenu du .h est venu se mettre à l'emplacement de la ligne #include

Ce n'est pas bien compliqué à comprendre, je pense d'ailleurs que bon nombre d'entre vous devaient se douter que ça fonctionnait comme ça.

Au moins là avec ces explications supplémentaires, j'espère avoir mis tout le monde d'accord Le #include ne fait rien d'autre qu'insérer un fichier dans un autre, c'est important de bien le comprendre.

Si on a décidé de mettre les prototypes dans les .h, au lieu de tout mettre dans les .c, c'est principalement par principe. On pourrait à priori mettre les prototypes en haut des .c (d'ailleurs dans certains très petits programmes on le fait parfois), mais pour des questions d'organisation il est *vivement* conseillé de placer ses prototypes dans des .h.

LES DEFINES

Nous allons découvrir maintenant une nouvelle directive de préprocesseur : le #define.

Cette directive permet de définir une **constante de préprocesseur**. Cela permet d'associer une valeur à un mot. Voici un exemple :

Code : C

```
#define NOMBRE_VIES_INITIALES 3
```

Vous mettez dans l'ordre :

- le #define
- le mot auquel la valeur va être associée
- la valeur du mot

Attention, malgré les apparences (notamment le nom que l'on a l'habitude de mettre en majuscules) c'est très différent des constantes que nous avons étudiées jusqu'ici, telles que :

Code : C

```
const long NOMBRE_VIES_INITIALES = 3;
```

Les constantes occupaient de la place en mémoire. Même si la valeur ne changeait pas, votre nombre "3" était stocké quelque part dans la mémoire. Ce n'est pas le cas des constantes de préprocesseur

Comment ça fonctionne ? En fait, le #define remplace dans votre code source tous les mots par leur valeur correspondante. C'est comme la fonction "rechercher / remplacer" de Word si vous voulez
Ainsi, la ligne :

Code : C

```
#define NOMBRE_VIES_INITIALES 3
```

... remplace dans le fichier chaque NOMBRE_VIES_INITIALES par 3.

Voici un exemple de fichier .c avant passage du préprocesseur :

Code : C

```c
#define NOMBRE_VIES_INITIALES 3

int main(int argc, char *argv[])
{
    long vies = NOMBRES_VIES_INITIALES;

    /* Code ...*/
```

Après passage du préprocesseur :

Code : C

```c
int main(int argc, char *argv[])
{
    long vies = 3;

    /* Code ...*/
```

Avant la compilation, tous les #define auront donc été remplacés par les valeurs correspondantes. Le compilateur "voit" le fichier après passage du préprocesseur, dans lequel tous les remplacements auront été effectués.

Quel intérêt par rapport à l'utilisation de constantes comme on l'a vu jusqu'ici ?

Eh bien, comme je vous l'ai dit ça **ne prend pas de place en mémoire**. C'est logique, vu que lors de la compilation il ne reste plus que des nombres dans le code source.

Un autre intérêt est que le remplacement se fait dans **tout le fichier** dans lequel se trouve le #define. Si vous aviez défini une constante en mémoire dans une fonction, celle-ci n'aurait été valable que dans la fonction puis aurait été supprimée. Le #define en revanche s'appliquera à toutes les fonctions du fichier, ce qui peut s'avérer vraiment pratique pour le programmeur.

Un exemple concret d'utilisation des #define ?
En voici un que vous ne tarderez pas à utiliser. Lorsque vous ouvrirez une fenêtre en C, vous aurez probablement envie de définir des constantes de préprocesseur pour indiquer les dimensions de la fenêtre :

Code : C

```c
#define LARGEUR_FENETRE  800
#define HAUTEUR_FENETRE  600
```

L'avantage est que si plus tard vous décidez de changer la taille de la fenêtre (parce que ça vous semble trop petit), il vous suffira de modifier les #define puis de recompiler.

A noter : les #define sont généralement placés dans des .h, à côté des prototypes (vous pouvez d'ailleurs aller voir les .h des librairies comme stdlib.h, vous verrez qu'il y a des #define !).
Les #define sont donc "faciles d'accès", vous pouvez changer les dimensions de la fenêtre en modifiant les #define plutôt que d'aller chercher au fond de vos fonctions l'endroit où vous ouvrez la fenêtre pour modifier les dimensions.
C'est donc du temps de gagné pour le programmeur

En résumé, les constantes de préprocesseur permettent de "configurer" votre programme avant sa compilation. C'est une sorte de mini-configuration en fait

Un define pour la taille des tableaux

On utilise souvent les defines pour définir la taille des tableaux. On fait par exemple :

Code : C

```
#define TAILLE_MAX      1000

int main(int argc, char *argv[])
{
    char chaine1[TAILLE_MAX], chaine2[TAILLE_MAX];
    // ...
```

Mais... Je croyais qu'on ne pouvait pas mettre de variable entre les crochets lors d'une définition de tableau ?

Oui, mais TAILLE_MAX n'est PAS une variable
En effet je vous l'ai dit, le préprocesseur transforme le fichier avant compilation en :

Code : C

```
int main(int argc, char *argv[])
{
    char chaine1[1000], chaine2[1000];
    // ...
```

... et cela est valide

En définissant TAILLE_MAX ainsi, vous pouvez vous en servir pour créer des tableaux d'une certaine taille. Si cela s'avère insuffisant par le futur, vous n'aurez qu'à modifier la ligne du #define, recompiler, et vos tableaux de char prendront tous la nouvelle taille que vous aurez indiquée

Calculs dans les defines

Il est possible de faire quelques petits calculs dans les defines.
Par exemple, ce code crée une constante LARGEUR_FENETRE, une autre HAUTEUR_FENETRE, puis une troisième NOMBRE_PIXELS qui contiendra le nombre de pixels affichés à l'intérieur de la fenêtre (le calcul est simple : largeur * hauteur) :

Code : C

```
#define LARGEUR_FENETRE  800
#define HAUTEUR_FENETRE  600
#define NOMBRE_PIXELS    (LARGEUR_FENETRE * HAUTEUR_FENETRE)
```

La valeur de NOMBRE_PIXELS est remplacée avant la compilation par (LARGEUR_FENETRE * HAUTEUR_FENETRE), c'est-à-dire par (800 * 600), ce qui fait 480000
Mettez toujours votre calcul entre parenthèses comme je l'ai fait.

Vous pouvez faire toutes les opérations de base que vous connaissez : addition (+), soustraction (-), multiplication (*), division (/) et modulo (%)

Les constantes prédéfinies

En plus des constantes que vous pouvez définir vous-mêmes, il existe quelques constantes prédéfinies par le préprocesseur. A l'heure où j'écris ces lignes, je dois vous dire que je ne les ai encore jamais utilisées, mais il n'est pas impossible que vous leur en trouviez une utilité donc je vais vous les présenter

Chacune de ces constantes commence et se termine par 2 symboles underscore _ (que vous trouverez sous le chiffre 8, tout du moins si vous avez un clavier AZERTY).

- __LINE__ : donne le numéro de la ligne actuelle
- __FILE__ : donne le nom du fichier actuel
- __DATE__ : donne la date de la compilation
- __TIME__ : donne l'heure de la compilation

Je pense que ces constantes peuvent être utiles pour gérer des erreurs, en faisant par exemple ceci :

Code : C

```
printf("Erreur a la ligne %ld du fichier %s\n", __LINE__, __FILE__);
printf("Ce fichier a ete compile le %s a %s\n", __DATE__, __TIME__);
```

Code : Console

```
Erreur a la ligne 9 du fichier main.c
Ce fichier a ete compile le Jan 13 2006 a 19:21:10
```

Les définitions simples

Il est aussi possible de faire tout simplement :

Code : C

```
#define CONSTANTE
```

... sans préciser de valeur.
Cela veut dire pour le préprocesseur que le mot CONSTANTE est défini, tout simplement. Il n'a pas de valeur, mais il "existe".

L'intérêt est moins évident que tout à l'heure, mais il y en a un et nous allons le découvrir tout à l'heure

Les macros

Nous avons vu qu'avec le #define on pouvait demander au préprocesseur de remplacer un mot par une valeur. Par exemple :

Code : C

```
#define NOMBRE 9
```

... signifie que tous les "NOMBRE" de votre code seront remplacés par 9. Nous avons vu qu'il s'agissait en fait d'un simple rechercher / remplacer fait par le préprocesseur avant la compilation.

J'ai du nouveau !
En fait, le #define est encore plus puissant que ça. Il permet de remplacer aussi par... du code source tout entier !

Quand on utilise #define pour rechercher / remplacer un mot par un code source, on dit qu'on crée **une macro**.

Macro sans paramètres

Voici un exemple de macro très simple :

Code : C

```
#define COUCOU() printf("Coucou");
```

Ce qui change ici, c'est les parenthèses qu'on a ajoutées après le mot-clé (ici **COUCOU()**). Nous verrons à quoi elles peuvent servir tout à l'heure.

Testons la macro dans un code source :

Code : C

```
#define COUCOU() printf("Coucou");

int main(int argc, char *argv[])
{
    COUCOU()

    return 0;
}
```

Code : Console

```
Coucou
```

Je vous l'accorde, ce n'est pas original pour le moment

Ce qu'il faut bien comprendre déjà, c'est que les macros ne sont en fait que des bouts de code qui sont directement remplacés dans votre code source juste avant la compilation.
Le code qu'on vient de voir ressemblera en fait à ça lors de la compilation :

Code : C

```
int main(int argc, char *argv[])
{
    printf("Coucou");

    return 0;
}
```

Si vous avez compris ça, vous avez compris le principe de base des macros déjà

Mais... On ne peut mettre qu'une seule ligne de code par macro ?

Non, heureusement il est possible de mettre plusieurs lignes de code à la fois. Il suffit de mettre un \ avant chaque nouvelle ligne, comme ceci :

Code : C

```
#define RACONTER_SA_VIE()   printf("Coucou, je m'appelle Brice\n"); \
                            printf("J'habite a Nice\n"); \
                            printf("J'aime la glisse\n");

int main(int argc, char *argv[])
{
    RACONTER_SA_VIE()

    return 0;
}
```

Code : Console

```
Coucou, je m'appelle Brice
J'habite a Nice
J'aime la glisse
```

Remarquez dans le main que l'appel de la macro ne prend pas de point-virgule à la fin. En effet, c'est une ligne pour le préprocesseur, elle ne nécessite donc pas d'être terminée par un point-virgule

Macro avec paramètres

Pour le moment, on a vu comment faire une macro sans paramètre, c'est-à-dire avec rien entre les parenthèses. Le principal intérêt de ce type de macros, c'est de pouvoir "raccourcir" un code un peu long, surtout s'il est amené à être répété de nombreuses fois dans votre code source.

Cependant, les macros deviennent réellement intéressantes lorsqu'on leur met des paramètres. Cela marche quasiment comme avec les fonctions.

Code : C

```
#define MAJEUR(age) if (age >= 18) \
                    printf("Vous etes majeur\n");

int main(int argc, char *argv[])
{
    MAJEUR(22)

    return 0;
}
```

Code : Console

```
Vous etes majeur
```

Notez qu'on aurait aussi pu rajouter un else pour afficher "Vous êtes mineur". Essayez de le faire pour vous entraîner, ce n'est pas bien difficile. N'oubliez pas de mettre un antislash \ avant chaque nouvelle ligne

Le principe de notre macro est assez intuitif je trouve :

Code : C

```
#define MAJEUR(age) if (age >= 18) \
                    printf("Vous etes majeur\n");
```

On met entre parenthèses le nom d'une "variable" qu'on nomme age. Dans tout notre code de remplacement, age sera remplacé par le nombre qui est indiqué lors de l'appel à la macro (ici c'est 22).

Ainsi, notre code source de tout à l'heure ressemblera à ceci juste après le passage du préprocesseur :

Code : C

```
int main(int argc, char *argv[])
{
    if (22 >= 18)
    printf("Vous etes majeur\n");

    return 0;
}
```

Le code source a été mis à la place de l'appel de la macro, et la valeur de la "variable" age a été mise directement dans le code source de remplacement.

Il est possible aussi de créer une macro qui prend plusieurs paramètres :

Code : C

```
#define MAJEUR(age, nom) if (age >= 18) \
                    printf("Vous etes majeur %s\n", nom);

int main(int argc, char *argv[])
{
    MAJEUR(22, "Maxime")

    return 0;
}
```

Voilà en gros tout ce qu'on peut dire sur les macros. Il faut donc retenir que c'est un simple remplacement de code source qui a l'avantage de pouvoir prendre des paramètres.

Normalement vous ne devriez pas avoir besoin d'utiliser les macros très souvent. Toutefois, certaines librairies assez complexes comme wxWidgets ou QT (librairies de création de fenêtres que nous étudierons bien plus tard) utilisent beaucoup de macros. Il est donc préférable de savoir comment cela fonctionne dès maintenant pour ne pas être bêtement perdu plus tard

LES CONDITIONS

Tenez-vous bien : il est possible de réaliser des conditions en langage préprocesseur
Voici comment cela fonctionne :

Code : C

```
#if condition
    /* Code source à compiler si la condition est vraie */
#elif condition2
    /* Sinon` si la condition 2 est vraie` compiler ce code source */
#endif
```

Le mot-clé #if permet d'insérer une condition de préprocesseur. #elif signifie "else if" (sinon si).
La condition s'arrête lorsque vous insérez un #endif. Vous noterez qu'il n'y a pas d'accolades en préprocesseur.

L'intérêt, c'est qu'on peut ainsi faire des **compilations conditionnelles**.
En effet, si la condition est vraie, le code qui suit sera compilé. Sinon, il sera tout simplement supprimé du fichier pour le temps de la compilation. Il n'apparaîtra donc pas dans le programme final.

#ifdef, #ifndef

Nous allons voir maintenant l'intérêt de faire un #define d'une constante sans préciser de valeur, comme je vous l'ai montré tout à l'heure :

Code : C

```
#define CONSTANTE
```

En effet, il est possible d'utiliser #ifdef pour dire "Si la constante est définie".
#ifndef, lui, sert à dire "Si la constante n'est pas définie".

On peut alors imaginer ceci :

Code : C

```
#define WINDOWS

#ifdef WINDOWS
    /* Code source pour Windows */
#endif

#ifdef LINUX
    /* Code source pour Linux */
#endif

#ifdef MAC
    /* Code source pour Mac */
#endif
```

C'est comme ça que font les programmes multi plateformes pour s'adapter à l'OS par exemple
Alors, bien entendu, il faut recompiler le programme pour chaque OS (ce n'est pas magique :D). Si vous êtes sous Windows, vous mettez un `#define WINDOWS` en haut, puis vous compilez.
Si vous voulez compiler votre programme pour Linux (avec la partie du code source spécifique à Linux), alors vous devrez modifier le define et mettre à la place : `#define LINUX`. Recompilez, et cette fois c'est la portion de code source pour Linux qui sera compilée, les autres parties étant ignorées.

#ifndef pour éviter les inclusions infinies

#ifndef est très utilisé dans les .h pour éviter les "inclusions infinies".

Imaginez, c'est très simple.
J'ai un fichier A.h et un fichier B.h. Le fichier A.h contient un include du fichier B.h. Le fichier B est donc inclus dans le fichier A.
Mais, et c'est là le hic, supposez que le fichier B.h contienne à son tour un include du fichier A.h ? Ca arrive quelques fois en programmation ! Le premier fichier a besoin du second pour fonctionner, et le second a besoin du premier.

Si on y réfléchit 10 petites secondes, on imagine vite ce qu'il va se passer :

1. L'ordinateur lit A.h et voit qu'il faut inclure B.h
2. Il lit B.h pour l'inclure, et là il voit qu'il faut inclure A.h
3. Donc il inclut A.h dans B.h, mais dans A.h on lui indique qu'il doit inclure B.h !
4. Rebelote, il va voir B.h et voit à nouveau qu'il faut inclure A.h
5. etc etc.

Pas besoin d'être un pro pour comprendre que ça ne s'arrêtera jamais !
En fait, à force de faire trop d'inclusions, le préprocesseur s'arrêtera en disant "*y'en a marre des inclusions !*" et du coup bah votre compilation plantera

Comment diable faire pour éviter cet affreux cauchemar ?
Voici l'astuce. Désormais, je vous demande de faire comme ça **dans TOUS vos fichiers .h** sans exception :

Code : C

```
#ifndef DEF_NOMDUFICHIER // Si la constante n'a pas été définie` le fichier n'a jamais été inclus
#define DEF_NOMDUFICHIER // On définit la constante pour que la prochaine fois le fichier ne soit plus inclus

/* Contenu de votre fichier .h (autres includes` prototypes de vos fonctions` defines...) */

#endif
```

Vous mettrez en fait tout le contenu de votre fichier .h (à savoir vos autres includes, vos prototypes, vos defines...) *entre* le #ifndef et le #endif.
Comprenez-vous bien comment ce code fonctionne ? Je ne l'ai pas compris du premier coup quand on me l'a expliqué moi pour être tout à fait franc

Imaginez que le fichier .h est inclus pour la première fois. Il lit la condition "Si la constante DEF_NOMDUFICHIER n'a pas été définie". Comme c'est la première fois que le fichier est lu, la constante n'est pas définie, donc le préprocesseur rentre à l'intérieur du if.

La première instruction qu'il rencontre est justement :

Code : C

```
#define DEF_NOMDUFICHIER
```

Maintenant, la constante est définie. La prochaine fois que le fichier sera inclus, la condition ne sera plus vraie, et donc le fichier ne risque plus d'être réinclus.

Bien entendu, vous appelez votre constante comme vous voulez. Moi je l'appelle DEF_NOMDUFICHIER par habitude, maintenant c'est chacun ses petites manies
Ce qui compte en revanche, et j'espère que vous l'aviez bien compris, c'est de changer de nom de constante à chaque fichier .h différent. Il ne faut pas que ça soit la même constante pour tous les fichiers .h, sinon seul le premier fichier .h serait lu et pas les autres
Vous remplacerez donc NOMDUFICHIER par le nom de votre fichier .h.

Si vous voulez vérifier que je ne suis pas en train de vous raconter des bêtises, je vous invite à aller consulter les .h des librairies standard sur votre disque dur. Vous verrez qu'ils sont TOUS construits sur le même principe (un ifndef au début et un endif à la fin). Ils s'assurent ainsi qu'il ne pourra pas y avoir d'inclusions infinies.

C'est marrant, j'ai presque l'impression de vous avoir enseigné un nouveau langage de programmation là
Et c'est un peu vrai d'ailleurs, car le préprocesseur, ce fameux programme qui lit vos codes sources juste avant de les envoyer au compilateur, a son propre langage à lui.

On peut faire 2-3 autres petites choses dont je ne vous ai pas parlé ici, mais globalement on en a fait le tour. On utilise beaucoup les directives de préprocesseur dans les .h comme vous l'avez vu.

Ah, aussi une petite remarque qui ne mange pas de pain avant de terminer : je vous conseille vivement de mettre quelques retours à la ligne après le #endif à la fin de vos fichiers .h.
Evitez que #endif soit la dernière ligne du fichier, j'ai déjà eu des erreurs de compilation à cause de ça et j'ai eu du mal à comprendre au début d'où ça venait. J'avais l'erreur "No new ligne at the end of file bidule"

Mettez donc 2-3 retours à la ligne après le #endif comme ceci :

Code : C

```
#endif


```

Cette remarque vaut d'ailleurs aussi pour la fin des fichiers .c. Mettez toujours quelques retours à la ligne vides à la fin, ça ne coûte rien et ça évite des prises de tête inutiles.

Je n'ai jamais dit que la programmation était une science exacte hein
Parfois, on tombe sur des erreurs tellement bizarres qu'on en vient à se demander s'il ne faudrait pas faire d'incantations vaudoues

Vous inquiétez pas si ça vous arrive, c'est l'métier qui rentre

Le langage C nous permet de faire quelque chose de très puissant : il nous permet de créer nos propres types de variables.
Des "types de variables personnalisés", nous allons en voir 2 sortes :

- **Les structures** : elles permettent de créer des variables composées de plusieurs sous-variables.
- **Les énumérations** : elles permettent de définir une liste de valeurs possibles pour une variable.

Créer de nouveaux types de variables devient indispensable quand l'on cherche à faire des programmes plus complexes. Par "plus complexe", je veux dire "autre chose que faire un Plus ou Moins"

Ce n'est (heureusement) pas bien compliqué à comprendre et à manipuler. Restez attentifs tout de même parce que nous réutiliserons les structures tout le temps à partir du prochain chapitre.
Il faut savoir que les librairies définissent généralement leurs propres types. Vous ne tarderez donc pas à manipuler un type de variable "Fichier" ou encore, un peu plus tard, un type de variable "Fenetre", "Audio", "Clavier" etc

DÉFINIR UNE STRUCTURE

Une structure est **un assemblage de variables** qui peuvent avoir différents types.
Contrairement aux tableaux qui vous obligent à utiliser le même type dans tout le tableau, vous pouvez créer une structure comportant des variables de type long, char, int, double à la fois

Les structures sont généralement **définies dans les fichiers .h**, au même titre donc que les prototypes et les defines.

Voici un exemple de structure :

Code : C

```c
struct NomDeVotreStructure
{
    long variable1;
    long variable2;
    int autreVariable;
    double nombreDecimal;
};
```

Une définition de structure commence par le mot-clé struct, suivi du nom de votre structure (par exemple "Fichier", ou encore "Ecran").

J'ai personnellement l'habitude de nommer mes structures en suivant les mêmes règles que pour les noms de variables, excepté que je mets la première lettre en majuscule pour pouvoir faire la différence. Ainsi, quand je vois le mot "ageDuCapitaine" dans mon code, je sais que c'est une variable car cela commence par une lettre minuscule. Quand je vois "MorceauAudio" je sais qu'il s'agit d'une structure (un type personnalisé) car cela commence par une majuscule.
Vous n'êtes pas obligés de faire comme moi. Le principal est que vous soyez organisés

Après le nom de votre structure, vous ouvrez les accolades et les fermez plus loin, comme pour une fonction.

Attention, ici c'est particulier : vous DEVEZ mettre un point-virgule après l'accolade fermante. C'est obligatoire. Si vous ne le faites pas, la compilation plantera.

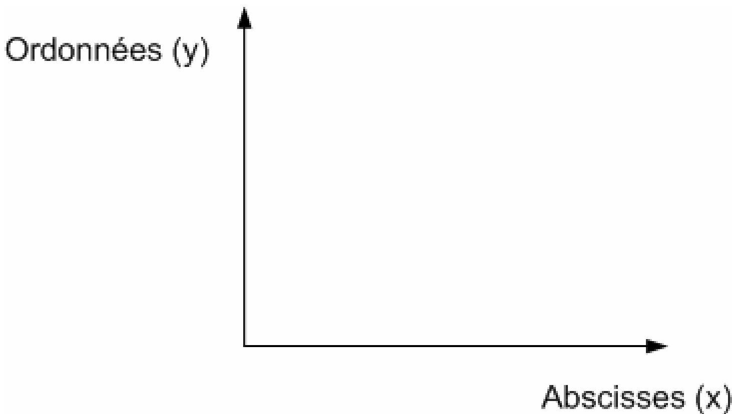
Et maintenant, que mettre entre les accolades ?
C'est simple, vous mettez les variables dont est composée votre structure. Une structure est généralement composée d'au moins 2 "sous-variables", sinon elle n'a pas trop d'intérêt

Comme vous le voyez, la création d'un type de variable personnalisé n'est pas bien complexe. Toutes les structures que vous verrez sont en fait des "assemblages" de variables de type de base, comme long, int, double etc... Il n'y a pas de miracle, un type "Fichier" n'est donc composé que de nombres de base

Exemple de structure

Des idées de structures, j'en ai des centaines en tête
Imaginons par exemple que vous vouliez créer une variable qui stocke les coordonnées d'un point à l'écran. Vous aurez très certainement besoin d'une structure comme cela lorsque vous ferez des jeux 2D dans la partie III, c'est donc l'occasion de s'avancer un peu là

Pour ceux chez qui le mot "géométrie" provoque des apparitions de boutons inexplicables sur tout le visage, voici un petit rappel fondamental sur la 2D :



Lorsqu'on travaille en 2D (2 dimensions), on a 2 axes : l'axe des abscisses (de gauche à droite) et l'axe des ordonnées (de haut en bas).
On a l'habitude d'exprimer les abscisses par une variable appelée x, et les ordonnées par y.

Etes-vous capables d'écrire une structure "Coordonnees" capable de stocker à la fois la valeur de l'abscisse (x) et celle de l'ordonnée (y) d'un point ?
Allons allons, ce n'est pas bien difficile

Code : C

```
struct Coordonnees
{
    long x; // Abscisses
    long y; // Ordonnées
};
```

Notre structure s'appelle Coordonnee et est composée de 2 variables : x et y, c'est-à-dire l'abscisse et l'ordonnée.

Si on le voulait, on pourrait facilement faire une structure Coordonnees pour de la 3D : il suffirait d'ajouter une troisième variable (par exemple "z") qui indiquerait la hauteur. Et hop, on aurait une structure faite pour gérer des points en 3D dans l'espace

Tableaux dans une structure

Les structures peuvent contenir des tableaux. Ca tombe bien, on va pouvoir ainsi placer des tableaux de char (chaînes de caractères) sans problème !
Allez, imaginons une structure "Personne" qui stockerait diverses informations sur une personne :

Code : C

```
struct Personne
{
    char nom[100];
    char prenom[100];
    char adresse[1000];

    long age;
    int garcon; // Booléen : 1 = garçon, 0 = fille
};
```

Cette structure est composée de 5 sous-variables. Les 3 premières sont des chaînes, qui stockeront le nom, le prénom et l'adresse de la personne.
Les 2 dernières stockent l'âge et le sexe de la personne. Le sexe est un booléen, 1 = vrai = garçon, 0 = faux = fille.

Cette structure pourrait servir à créer un programme de carnet d'adresses. Bien entendu, vous pouvez rajouter des variables dans la structure pour la compléter si vous le voulez. Il n'y a pas de limite au nombre de variables dans une

structure (enfin évitez d'en mettre une centaine ça va faire beaucoup quand même).

UTILISATION D'UNE STRUCTURE

Maintenant que notre structure est définie dans le .h, on va pouvoir l'utiliser dans une fonction de notre fichier .c.
Voici comment créer une variable de type Coordonnees (la structure qu'on a définie plus haut) :

Code : C

```c
#include "main.h" // Inclusion du .h qui contient les prototypes et structures

int main(int argc, char *argv[])
{
    struct Coordonnees point; // Création d'une variable appelée "point" de type
Coordonnees

    return 0;
}
```

Nous avons ainsi créé une variable "point" de type Coordonnees. Cette variable est automatiquement composée de 2 sous-variables : x et y (son abscisse et son ordonnée).

On est obligés de mettre le mot "struct" lors de la définition de la variable ?

Oui, cela permet à l'ordinateur de différencier un type de base (comme "int") d'un type personnalisé, comme "Coordonnees".
Toutefois, les programmeurs trouvent un peu lourd de mettre le mot "struct" à chaque définition de variable personnalisée. Pour régler ce problème, ils ont inventé une instruction spéciale : **le typedef**.

Le typedef

Retournons dans le fichier .h qui contient la définition de notre structure Coordonnees.
Nous allons ajouter une instruction appelée typedef qui sert à créer un alias de structure, c'est-à-dire à dire que telle chose est équivalente à écrire telle autre chose.

Nous allons ajouter une ligne commençant par typedef juste avant la définition de la structure :

Code : C

```c
typedef struct Coordonnees Coordonnees;
struct Coordonnees
{
    long x;
    long y;
};
```

Cette ligne doit être découpée en 3 morceaux (non je n'ai pas bégayé le mot "Coordonnees") :

- **typedef** : indique que nous allons créer un alias de structure.
- **struct Coordonnees** : c'est le nom de la structure dont vous allez créez un alias (c'est-à-dire un "équivalent").
- **Coordonnees** : c'est le nom de l'équivalent.

En clair, cette ligne dit "Ecrire le mot *Coordonnees* est désormais équivalent à écrire *struct Coordonnees*". En faisant cela, vous n'aurez plus besoin de mettre le mot struct à chaque définition de variable de type Coordonnees.
On peut donc retourner dans notre main et écrire tout simplement :

Code : C

```
int main(int argc, char *argv[])
{
    Coordonnees point; // L'ordinateur comprend qu'il s'agit de "struct Coordonnees"
grâce au typedef
    return 0;
}
```

Je vous recommande de faire un typedef comme je l'ai fait ici pour le type Coordonnees. La plupart des programmeurs font comme cela. Ca leur évite d'avoir à écrire le mot "struct" partout, et vu que ce sont des feignasses ça les arrange (je rigole hein les gars, tapez pas)

Modifier les composantes de la structure

Maintenant que notre variable point est créée, nous voulons modifier ses coordonnées.
Comment accéder au x et au y de point ? Comme cela :

Code : C

```
int main(int argc, char *argv[])
{
    Coordonnees point;

    point.x = 10;
    point.y = 20;

    return 0;
}
```

On a ainsi modifié la valeur de point, en lui donnant une abscisse de 10 et une ordonnée de 20. Notre point se situe désormais à la position (10 ; 20) (c'est comme cela qu'on note une coordonnée en mathématiques)

Pour accéder donc à chaque composante de la structure, vous devez écrire :

Code : C

```
variable.nomDeLaComposante
```

Le point fait la séparation entre la variable et la composante.

Si on prend la structure "Personne" de tout à l'heure et qu'on demande le nom et le prénom, , on devra faire comme ça :

Code : C

```
int main(int argc, char *argv[])
{
    Personne utilisateur;

    printf("Quel est votre nom ? ");
    scanf("%s", utilisateur.nom);
    printf("Votre prenom ? ");
    scanf("%s", utilisateur.prenom);

    printf("Vous vous appelez %s %s", utilisateur.prenom, utilisateur.nom);

    return 0;
}
```

Code : Console

```
Quel est votre nom ? Dupont
Votre prenom ? Jean
Vous vous appelez Jean Dupont
```

On envoie la variable "utilisateur.nom" à scanf qui écrira directement dans notre variable "utilisateur".
On fait de même pour "prenom", et on pourrait aussi le faire pour l'adresse, l'âge et le sexe, mais j'ai la flême de le faire ici (je dois être programmeur, c'est pour ça).

Vous auriez pu faire la même chose sans connaître les structures, en créant juste une variable nom et une autre "prenom".
Mais l'intérêt ici est que vous pouvez créer une autre variable de type "Personne" qui aura aussi son propre nom, son propre prénom etc etc
On peut donc faire :

Code : C

```c
Personne joueur1, joueur2;
```

... et stocker ainsi les informations sur chaque joueur.

Mais le C, c'est plus fort encore ! On peut créer un tableau de Personne !
C'est facile à faire :

Code : C

```c
Personne joueurs[2];
```

Et ensuite, vous accédez par exemple au nom du joueur n°0 en tapant :

Code : C

```c
joueurs[0].nom
```

L'avantage d'utiliser un tableau ici, c'est que vous pouvez faire une boucle pour demander les infos du joueur 1 et du joueur 2, sans avoir à répéter 2 fois le même code. Il suffit de parcourir le tableau joueur et de demander à chaque fois nom, prénom, adresse...

Exercice : créez ce tableau de type Personne et demandez les infos de chacun grâce à une boucle (qui se répète tant qu'il y a des joueurs). Faites un petit tableau de 2 joueurs pour commencer, mais si ça vous amuse vous pourrez agrandir la taille du tableau plus tard.
Affichez à la fin du programme les infos que vous avez recueillies sur chacun des joueurs

Ce code est facile à écrire, vous n'avez pas besoin de correction, vous êtes des grands maintenant

Initialiser une structure

Pour les structures comme pour les variables, tableaux et pointeurs, il est vivement conseillé de les initialiser dès leur création pour éviter qu'elles ne contiennent "n'importe quoi". En effet, je vous le rappelle, une variable qui est créée prend la valeur de ce qui se trouve en mémoire là où elle a été placée. Parfois cette valeur est 0, parfois c'est un résidu d'un autre programme qui est passé par là avant vous et la variable vaut une valeur qui n'a aucun sens, comme -84570.

Pour rappel, voici comment on initialise :

- **Une variable** : on met sa valeur à 0 (ça c'est simple).
- **Un pointeur** : on met sa valeur à NULL. NULL est en fait un #define situé dans stdlib.h qui vaut généralement 0, mais on continue à utiliser NULL par convention sur les pointeurs pour bien voir qu'il s'agit de pointeurs et non de variables ordinaires.
- **Un tableau** : on met chacune de ses valeurs à 0.

Pour les structures, ça va un peu ressembler à l'initialisation d'un tableau qu'on avait vue. En effet, on peut faire à la déclaration de la variable :

Code : C

```
Coordonnees point = {0, 0};
```

Cela mettra, dans l'ordre, point.x = 0 et point.y = 0.

Revenons à la structure Personne (qui contient des chaînes). Vous avez aussi le droit d'initialiser une chaîne en mettant juste "" (rien entre les guillemets). Je ne vous ai pas parlé de cette possibilité dans le chapitre sur les chaînes je crois, mais il n'est jamais trop tard pour l'apprendre
On peut donc initialiser dans l'ordre nom, prenom, adresse, age et garcon comme ceci :

Code : C

```
Personne utilisateur = {"", "", "", 0, 0};
```

Toutefois, j'utilise assez peu cette technique personnellement. Je préfère envoyer par exemple ma variable point à une fonction initialiserCoordonnees qui se charge de faire les initialisations pour moi sur ma variable.
Toutefois, pour faire cela il faut envoyer un pointeur de ma variable. En effet si j'envoie juste ma variable, une copie en sera réalisée dans la fonction (comme pour une variable de base) et la fonction modifiera les valeurs de la copie et non celle de ma vraie variable. Revoyez le "fil rouge" du chapitre sur les pointeurs si vous avez un trou de mémoire à ce sujet

Il va donc falloir apprendre à utiliser des pointeurs sur des structures (hummm ça donne faim ça).
Nous allons justement voir comment faire ça maintenant

Pointeur de structure

Un pointeur de structure se crée de la même manière qu'un pointeur de long, de double ou de n'importe quelle autre type de base :

Code : C

```
Coordonnees* point = NULL;
```

On a ainsi un pointeur de Coordonnees appelé "point".
Comme un rappel ne fera de mal à personne, je tiens à vous répéter que l'on aurait aussi pu mettre l'étoile devant le nom du pointeur, cela revient exactement au même :

Code : C

```
Coordonnees *point = NULL;
```

Je fais d'ailleurs assez souvent comme cela, car si l'on veut définir plusieurs pointeurs sur la même ligne, on est obligés de mettre l'étoile devant chaque nom de pointeur :

Code : C

```c
Coordonnees *point1 = NULL, *point2 = NULL;
```

Bon, mais ça pour le moment c'est du domaine des pointeurs, ça n'a pas de rapport direct avec les structures

Envoi de la structure à une fonction

Ce qui nous intéresse ici, c'est de savoir comment envoyer un pointeur de structure à une fonction, pour que celle-ci puisse modifier le contenu de la variable.

On va faire ceci pour cet exemple : on va juste créer une variable de type Coordonnees dans le main et envoyer son adresse à la fonction initialiserCoordonnees. Cette fonction aura pour rôle de mettre tous les éléments de la structure à 0.

Nous faisons cela juste à titre d'exemple. En pratique, il serait plus simple d'initialiser la structure comme on a appris à le faire un peu plus haut :

Code : C

```c
Coordonnees point = {0};
```

Cela mettra tous les éléments de la structure à 0.

Notre fonction initialiserCoordonnees va prendre un paramètre : un pointeur sur une structure de type Coordonnees (un Coordonnees* donc).

Code : C

```c
int main(int argc, char *argv[])
{
    Coordonnees monPoint;

    initialiserCoordonnees(&monPoint);

    return 0;
}


void initialiserCoordonnees(Coordonnees* point)
{
    // Initialisation de chacun des membres de la structure ici
}
```

Ma variable monPoint est donc créée dans le main. On envoie son adresse à initialiserCoordonnees qui récupère cette variable sous la forme d'un pointeur appelé "point" (on aurait d'ailleurs pu l'appeler n'importe comment dans la fonction, même "monPoint" ou "trucBidule").

Bien, et maintenant que nous sommes dans initialiserCoordonnees, nous allons initialiser chacune des valeurs une à une.
Il ne faut pas oublier de mettre une étoile devant le nom du pointeur pour accéder à la variable. Si vous ne le faites pas, vous risquez de modifier l'adresse, et ce n'est pas ce que nous voulons faire

Oui mais voilà, problème... On ne peut pas vraiment faire :

Code : C

```
void initialiserCoordonnees(Coordonnees* point)
{
    *point.x = 0;
    *point.y = 0;
}
```

C'aurait été trop facile bien entendu

Pourquoi on ne peut pas faire ça ? Parce que le point de séparation s'applique sur le mot "point" et non *point en entier. Or, nous ce qu'on veut c'est accéder à *point pour en modifier la valeur.
Pour régler le problème, il faut mettre des parenthèses autour de *point. Comme cela, le point de séparation s'appliquera à *point et non juste à point :

Code : C

```
void initialiserCoordonnees(Coordonnees* point)
{
    (*point).x = 0;
    (*point).y = 0;
}
```

Voilà

Ce code fonctionne, vous pouvez tester. La variable de type Coordonnees a été transmise à la fonction qui a initialisé x et y à 0.

En C, on initialise généralement nos structures avec la méthode simple qu'on a vue plus haut.
En C++ en revanche, les initialisations sont plus souvent faites dans des "fonctions".
Nous n'en sommes pas encore à étudier le C++, mais vous verrez que, lorsque nous y arriverons, le C++ n'est en fait rien d'autre qu'une sorte de "super-amélioration" des structures. Bien entendu, beaucoup de choses découleront de cela, mais nous les étudierons en temps voulu. En attendant, je tiens à vous rappeler de vous concentrer sur le C, car tout ce que vous apprenez ici vous le réutiliserez avec le C++

Un raccourci pratique et très utilisé

Vous allez voir qu'on manipulera très souvent des pointeurs de structures. Pour être franc, je dois même vous dire qu'en C on utilise plus souvent des pointeurs de structures que des structures tout court
Quand je vous disais que les pointeurs vous poursuivraient jusque dans votre tombe, je ne le disais pas en rigolant
Vous ne pouvez y échapper, c'est votre destinée.

Hum...
En fait je voulais vous parler d'un truc sérieux là
Comme les pointeurs de structures sont très utilisés, on sera souvent amenés à écrire ceci :

Code : C

```
(*point).x = 0;
```

Oui mais voilà, encore une fois les programmeurs trouvent ça trop long. Les parenthèses autour de *point, quelle plaie ! Alors, comme les programmeurs sont de grosses feignasses (comment ça je l'ai déjà dit ?!), il ont inventé le raccourci suivant :

Code : C

```
point->x = 0;
```

Ce raccourci consiste à former une flèche avec un tiret suivi d'un chevron ">".
Ecrire `point->x` est donc STRICTEMENT équivalent à écrire `(*point).x`
Ca éclaircira votre code vous verrez

N'oubliez pas qu'on ne peut utiliser la flèche que sur un pointeur !
Si vous travaillez directement sur la variable, vous devez utiliser le symbole "point" comme on l'a vu au début.

Reprenons notre fonction initialiserCoordonnes. Nous pouvons donc l'écrire comme ceci :

Code : C

```
void initialiserCoordonnees(Coordonnees* point)
{
    point->x = 0;
    point->y = 0;
}
```

Retenez bien ce raccourci de la flèche, nous allons le réutiliser et pas qu'une seule fois

Et surtout, surtout, ne confondez pas la flèche avec le symbole "point".
La flèche est réservée aux pointeurs, le "point" est réservé aux variables. Utilisez ce petit exemple pour vous en souvenir :

Code : C

```
int main(int argc, char *argv[])
{
    Coordonnees monPoint;
    Coordonnees *pointeur = &monPoint;

    monPoint.x = 10; // On travaille sur une variable, on utilise le "point"
    pointeur->x = 10; // On travaille sur un pointeur, on utilise la flèche

    return 0;
}
```

On met le x à 10 de deux manières différentes ici, la première fois en travaillant directement sur la variable, la seconde fois en passant par le pointeur.

LES ÉNUMÉRATIONS

Les énumérations sont une façon un peu différente de créer ses propres types de variables.

Une énumération ne contient pas de "sous-variables" comme c'était le cas pour les structures. C'est une liste de "valeurs possibles" pour une variable. Une énumération ne prend donc qu'une case en mémoire, et cette case peut prendre une des valeurs que vous définissez (et une seule à la fois).

Voici un exemple d'énumération :

Code : C

```c
typedef enum Volume Volume;
enum Volume
{
    FAIBLE, MOYEN, FORT
};
```

Vous noterez qu'on utilise un typedef là aussi, comme on l'a fait jusqu'ici.

Pour créer une énumération, on utilise le mot-clé enum. Notre énumération s'appelle ici "Volume". C'est un type de variable personnalisé qui peut prendre **une des 3 valeurs** qu'on a indiquées : soit FAIBLE, soit MOYEN, soit FORT.

On va pouvoir créer une variable de type Volume, par exemple *musique*, qui stockera le volume actuel de la musique.
On peut par exemple initialiser la musique au volume MOYEN :

Code : C

```c
Volume musique = MOYEN;
```

Voilà qui est fait
Plus tard dans le programme, on pourra modifier la valeur du volume et la mettre soit à FAIBLE, soit à FORT.

Association de nombres aux valeurs

Vous avez remarqué que j'ai mis les valeurs possibles de l'énumération en majuscules. Ca devrait vous rappeler les constantes et les defines non ?

En effet, c'est assez similaire mais ce n'est pourtant pas exactement la même chose.
Le compilateur associe automatiquement un nombre à chacune des valeurs possibles de l'énumération.

Dans le cas de notre énumération Volume, FAIBLE vaut 0, MOYEN vaut 1 et FORT vaut 2. L'association est automatique et commence à 0.

Contrairement au #define, c'est le compilateur qui associe MOYEN à 1 par exemple, et non le préprocesseur. Au bout du compte, ça revient un peu au même
Quand on a initialisé la variable *musique* à MOYEN, on a donc en fait mis la case en mémoire à la valeur 1.

Non. En général vous vous en moquez
C'est le compilateur qui associe automatiquement un nombre à chaque valeur. Grâce à ça, vous n'avez plus qu'à faire :

Code : C

```c
if (musique == MOYEN)
{
    // Jouer la musique au volume moyen
}
```

Peu importe la valeur de MOYEN, vous laissez le compilateur se charger de gérer les nombres.

L'intérêt de tout ça ? C'est que du coup votre code est très lisible. En effet, tout le monde peut facilement lire le if

précédent (on comprend bien que la condition signifie "Si la musique est au volume moyen").

Notez qu'on aurait très bien pu faire ça sans utiliser d'énumération. On aurait par exemple pu créer une variable musique de type long, et "retenir" que 1 signifie "volume moyen" par exemple.

Code : C

```c
if (musique == 1)
{

}
```

Mais comme vous le voyez dans l'exemple ci-dessus, c'est moins facile à lire pour le programmeur

Associer une valeur précise

Pour le moment, c'est le compilateur qui décide d'associer le nombre 0 à la première valeur, puis 1, 2, 3 dans l'ordre.

Il est possible de demander d'associer une valeur précise à chaque élément de l'énumération. Mais quel intérêt est-ce que ça peut bien avoir ?

Eh bien, supposons que sur votre ordinateur le volume soit géré entre 0 et 100 (0 = pas de son, 100 = 100% du son). Il est alors pratique d'associer une valeur précise à chaque élément :

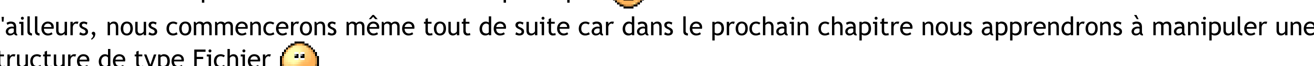
Code : C

```c
typedef enum Volume Volume;
enum Volume
{
    FAIBLE = 10, MOYEN = 50, FORT = 100
};
```

Ici, le volume FAIBLE correspondra à 10% de volume, le volume MOYEN à 50% etc.
On pourrait facilement ajouter de nouvelles valeurs possibles comme MUET. On mettrait dans ce cas MUET à la valeur... 0 ! Bravo vous avez compris le truc La possibilité d'utiliser des types de variables personnalisés est vraiment un atout majeur du langage C. Grâce à cela, les informations peuvent être regroupées entre elles, centralisées, traitées etc...

En résumé, nous avons vu :

- **Les structures** : elles permettent de créer des types de variables composés de plusieurs sous-variables.
- **Les énumérations** : elles permettent de créer des types de variables qui peuvent avoir une des valeurs définies dans l'énumération.

Nous ne tarderons pas à utiliser tout cela en pratique

D'ailleurs, nous commencerons même tout de suite car dans le prochain chapitre nous apprendrons à manipuler une structure de type Fichier

Quelques ajouts sur les structures

Avant de terminer ce chapitre, je tiens à faire quelques petits ajouts sur les structures pour vous montrer tout ce

qu'on peut faire avec.

On peut vraiment tout mettre à l'intérieur d'une structure. Je vous ai dit qu'on pouvait mettre des types de base (int, long...) ainsi que des tableaux, mais sachez qu'il est aussi possible de mettre des pointeurs:

Code : C

```
struct MaStructure
{
    long* monPointeur; // Pointeur sur long
    int monBooleen;
    char maChaine[10];
};
```

Plus fort encore, le C vous autorise à mettre une structure *dans* une structure :

Code : C

```
struct MaStructure
{
    Coordonnees element; // MaStructure contient une variable de type Coordonnees !
    int monBooleen;
    char maChaine[10];
};
```

Ca a l'air de compliquer un peu à priori, et pourtant c'est justement tout ce qui rend le langage *puissant*.

Attention si vous faites ça : il faudra définir la structure Coordonnees avant MaStructure (plus haut dans le fichier), car sinon le compilateur ne la connaîtra pas au moment de lire MaStructure et il plantera en disant que le type "Coordonnees" n'existe pas.

Si vous avez une variable appelée "test" de type MaStructure, vous pouvez du coup accéder aux coordonnées de "element" comme ceci :

Code : C

```
test.element.x = 0;
test.element.y = 0;
```

Je vous laisse méditer sur ce dernier code source et imaginer les structures que vous allez pouvoir créer et imbriquer entre elles

Lire et écrire dans des fichiers

Le défaut avec les variables, c'est qu'elles n'existent que dans la mémoire vive. Une fois votre programme arrêté, toutes vos variables sont supprimées de la mémoire et il n'est pas possible de retrouver ensuite leur valeur.

Comment, dans ce cas-là, peut-on enregistrer les meilleurs scores obtenus à son jeu ?
Comment peut-on faire un éditeur de texte si tout le texte écrit disparaît une fois lorsqu'on arrête le programme ?

Heureusement, on peut lire et écrire dans des fichiers en langage C. Et ça tombe bien, parce si on n'avait pas pu le

faire, nos programmes auraient été vraiment pauvres
Les fichiers seront écrits sur le disque dur de votre ordinateur : l'avantage est donc qu'ils restent là même si vous arrêtez le programme ou l'ordinateur

Pour lire et écrire dans des fichiers, nous allons avoir besoin de réutiliser tout ce que nous avons appris jusqu'ici : pointeurs, structures, pointeurs de structures, chaînes de caractères etc. Pour une bonne compréhension de ce chapitre, il faut donc que tout cela soit clair dans votre tête. Si ce n'est pas le cas, pas de panique : allez relire les chapitres précédents.
Rien ne presse, apprendre le langage C n'est pas une course. Les meilleurs seront ceux qui y seront allés le plus doucement, donc le plus sûrement

Ouvrir et fermer un fichier

Pour lire et écrire dans des fichiers, nous allons nous servir de fonctions situées dans la librairie **stdio** que nous avons déjà utilisée.
Oui, cette librairie-là contient aussi les fonctions printf et scanf que nous avons longuement utilisées jusqu'ici ! Mais elle ne contient pas que ça : il y a aussi d'autres fonctions, notamment des fonctions faites pour travailler sur des fichiers.

> Toutes les librairies que je vous ai fait utiliser jusqu'ici (stdlib.h, stdio.h, math.h, string.h...) sont ce qu'on appelle **des librairies standard**. Ce sont des librairies automatiquement livrées avec votre IDE qui ont la particularité de fonctionner sur tous les OS. Vous pouvez donc les utiliser partout, que vous soyez sous Windows, Linux, Mac ou autre
> Les librairies standard ne sont pas très nombreuses et ne permettent de faire que des choses très basiques, comme ce que nous avons fait jusqu'ici. Pour faire des choses plus avancées, comme ouvrir des fenêtres, il faudra télécharger et installer de nouvelles librairies. Nous verrons cela bientôt

Assurez-vous donc, pour commencer, que vous incluez bien au moins les librairies stdio.h et stdlib.h en haut de votre fichier .c :

Code : C

```c
#include <stdlib.h>
#include <stdio.h>
```

Ces librairies sont tellement fondamentales, tellement basiques, que je vous recommande d'ailleurs de les inclure dans tous vos futurs programmes, quels qu'ils soient.

Bien. Maintenant que les bonnes librairies sont incluses, nous allons pouvoir attaquer les choses sérieuses
Voici ce qu'il faut faire à chaque fois dans l'ordre quand on veut ouvrir un fichier (que ce soit pour lire ou pour écrire dedans) :

- On appelle la fonction d'**ouverture de fichier** *fopen* qui nous renvoie un pointeur sur le fichier.
- **On vérifie si l'ouverture a réussi** (c'est-à-dire si le fichier existait) en testant la valeur du pointeur qu'on a reçu. Si le pointeur vaut NULL, c'est que l'ouverture du fichier n'a pas marché, dans ce cas on ne peut pas continuer (il faut afficher un message d'erreur).
- Si l'ouverture a marché (si le pointeur est différent de NULL donc), alors on peut s'amuser à **lire et écrire dans le fichier** à travers des fonctions que nous verrons un peu plus loin.
- Une fois qu'on a **terminé de travailler sur le fichier**, il faut penser à le "fermer" avec la fonction *fclose*.

Nous allons dans un premier temps apprendre à nous servir de *fopen* et *fclose*. Une fois que vous saurez faire cela, nous apprendrons à lire le contenu du fichier et à écrire dedans.

fopen : ouverture du fichier

Dans le chapitre sur les chaînes, nous nous sommes servis des prototypes des fonctions comme de "mode d'emploi". C'est comme ça que les programmeurs font en général : ils lisent le prototype et comprennent alors le fonctionnement de la fonction (bon, je reconnais que des fois même eux ont besoin de quelques petites explications à côté)

Voyons voir justement le prototype de la fonction *fopen* :

Code : C

```
FILE* fopen(const char* nomDuFichier, const char* modeOuverture);
```

Cette fonction attend 2 paramètres :

- Le nom du fichier à ouvrir.
- Le mode d'ouverture du fichier, c'est-à-dire une indication qui dit si vous voulez juste écrire dans le fichier, juste lire dans le fichier, ou les deux à la fois

Cette fonction renvoie... un pointeur sur FILE

Eh bien ça les amis, vous devriez comprendre ce que c'est maintenant
C'est un pointeur sur une structure de type FILE. Cette structure est définie dans stdio.h. Vous pouvez ouvrir ce fichier pour voir de quoi est constitué le type FILE, mais ça n'a aucun intérêt je vous le dis de suite

Pourquoi le nom de la structure est-il tout en majuscules (FILE) ? Je croyais que les noms tout en majuscules étaient réservés aux constantes et aux defines ?

Cette "règle", c'est moi qui me la suis fixée (et pas mal d'autres programmeurs la suivent d'ailleurs). Ca n'a jamais été une obligation. Force est de croire que ceux qui ont programmé stdio ne suivaient pas exactement les mêmes règles
Cela ne doit pas vous perturber pour autant. Vous verrez d'ailleurs que les librairies que nous étudierons ensuite respectent les mêmes règles que moi, à savoir juste la première lettre en majuscule pour une structure.

Revenons à notre fonction fopen. Elle renvoie un FILE*. Il est extrêmement important de récupérer ce pointeur, pour pouvoir ensuite lire et écrire dans le fichier.
Nous allons donc créer un pointeur de FILE au début de notre fonction (par exemple la fonction main) :

Code : C

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;

    return 0;
}
```

Le pointeur est initialisé à NULL dès le début. Je vous rappelle que c'est une règle **fondamentale** que d'initialiser ses pointeurs à NULL dès le début si on n'a pas d'autre valeur à leur donner. Si vous ne le faites pas, vous risquez moultes plantages par la suite

Vous noterez qu'il n'est pas nécessaire d'écrire `struct FILE* fichier = NULL`. Les créateurs de

stdio ont donc fait un typedef comme je vous ai appris à le faire il n'y a pas longtemps
Notez que la forme de la structure peut changer d'un OS à l'autre (elle ne contient pas forcément les mêmes sous-variables partout). Pour cette raison, on ne modifiera jamais le contenu d'un FILE directement (on ne fera pas *fichier.truc* par exemple). On passera par des fonctions qui manipulent le FILE à notre place.

Maintenant, nous allons appeler la fonction *fopen* et récupérer la valeur qu'elle renvoie dans le pointeur "fichier".
Mais avant ça, il faut que je vous explique comment se servir du second paramètre, le paramètre "modeOuverture".
En effet, il y a un code à envoyer qui indiquera à l'ordinateur si vous ouvrez le fichier en mode de lecture seule, d'écriture seule, ou des deux à la fois.
Voici les modes d'ouvertures possibles :

- **"r" : lecture seule**. Vous pourrez lire le contenu du fichier, mais pas écrire dedans. *Le fichier doit avoir été créé au préalable.*
- **"w" : écriture seule**. Vous pourrez écrire dans le fichier, mais pas lire son contenu. *Si le fichier n'existe pas, il sera créé.*
- **"a" : mode d'ajout**. Vous écrirez dans le fichier, en partant de la fin du fichier. Vous rajouterez donc du texte à la fin du fichier. *Si le fichier n'existe pas, il sera créé.*
- **"r+" : lecture et écriture**. Vous pourrez lire et écrire dans le fichier. *Le fichier doit avoir été créé au préalable.*
- **"w+" : lecture et écriture, avec suppression du contenu au préalable**. Le fichier est donc d'abord vidé de son contenu, et vous écrivez et lisez ensuite dedans. *Si le fichier n'existe pas, il sera créé.*
- **"a+" : ajout en lecture / écriture à la fin**. Vous écrivez et lisez du texte à partir de la fin du fichier. *Si le fichier n'existe pas, il sera créé.*

Et encore, je n'ai pas tout mis là ! Il y a le double de ça en réalité ! Pour chaque mode qu'on a vu là, si vous rajoutez un "b" après le premier caractère ("rb", "wb", "ab", "rb+", "wb+", "ab+"), alors le fichier est ouvert en mode binaire. C'est un mode un peu particulier que nous ne verrons pas ici. En fait, le mode texte est fait pour stocker... du texte comme le nom l'indique (uniquement des caractères affichables), tandis que le mode binaire permet de stocker... des informations octet par octet (des nombres principalement). C'est sensiblement différent. Vous utiliseriez par exemple le mode binaire pour lire et écrire des fichiers Word octet par octet.
Le fonctionnement est quasiment le même de toute façon que ce que nous allons voir ici.

On a déjà fort à faire avec ces **6 modes d'ouverture** à retenir
Personnellement, j'utilise souvent "r" (lecture), "w" (écriture) et "r+" (lecture et écriture). Le mode "w+" est un peu dangereux parce qu'il vide de suite le contenu du fichier, sans demande de confirmation. Il ne doit être utilisé que si vous voulez d'abord réinitialiser le fichier.
Le mode d'ajout ("a") peut être utile dans certains cas, si vous voulez juste rajouter des informations à la fin du fichier.

Si vous avez juste l'intention de lire un fichier, il est conseillé de mettre "r". Certes, le mode "r+" aurait fonctionné lui aussi, mais en mettant "r" vous vous assurez que le fichier ne pourra pas être modifié, ce qui est en quelque sorte une sécurité.

Si vous écrivez une fonction "chargerNiveau" (pour charger le niveau d'un jeu par exemple), le mode "r" suffit. Si vous écrivez une fonction "enregistrerNiveau", le mode "w" sera alors adapté.

Le code suivant ouvre le fichier test.txt en mode "r+" (lecture / écriture) :

Code : C

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "r+");

    return 0;
}
```

Le pointeur "fichier" devient alors un pointeur sur "test.txt".

Où doit être situé test.txt ?

Il doit être situé dans le même dossier que votre exécutable (.exe).
Pour les besoins de ce chapitre, créez un fichier "test.txt" comme moi dans le même dossier que le .exe :

Comme vous le voyez, je travaille actuellement avec l'IDE Code::Blocks, ce qui explique la présence d'un fichier de projet .cbp (au lieu de .dev si vous avez Dev-C++, ou .sln si vous avez Visual C++).
Bref, ce qui compte c'est de bien voir que mon programme (tests.exe) est situé dans le même dossier que le fichier dans lequel on va lire et écrire (test.txt).

Le fichier doit-il être de type .txt ?

Non. C'est vous qui choisissez l'extension lorsque vous ouvrez le fichier. Vous pouvez très bien inventer votre propre format de fichier ".niveau" pour enregistrer les niveaux de vos jeux par exemple 😉

Le fichier doit-il être obligatoirement dans le même répertoire que l'exécutable ?

Non plus. Il peut être dans un sous-dossier :

Code : C

```
fichier = fopen("dossier/test.txt", "r+");
```

Ici, le fichier test.txt est dans un sous-dossier appelé "dossier". Cette méthode, que l'on appelle *chemin relatif* est plus pratique. Comme ça, cela fonctionnera peu importe l'endroit où est installé votre programme. C'est donc plus pratique.

Il est aussi possible d'ouvrir un autre fichier n'importe où ailleurs sur le disque dur. Dans ce cas, il faut écrire le chemin complet (ce qu'on appelle le *chemin absolu*) :

Code : C

```
fichier = fopen("C:\\Program Files\\Notepad++\\readme.txt", "r+");
```

Ce code ouvre le fichier readme.txt situé dans "C:\Program Files\Notepad++"

J'ai dû mettre 2 antislash \ comme vous l'avez remarqué. En effet, si j'en avais mis un seul, votre ordinateur aurait cru que vous essayiez d'insérer un symbole spécial comme \n ou \t. Pour mettre un antislash dans une chaîne, il faut donc l'écrire 2 fois. Votre ordinateur comprend alors que c'est bien le symbole \ que vous vouliez utiliser.

Le défaut des chemins absolus, c'est qu'ils ne marchent que sur un OS précis. Ce n'est pas une solution *portable* donc. Si vous aviez été sous Linux, vous auriez dû écrire un chemin à-la-linux, tel que :

Code : C

```
fichier = fopen("/home/mateo/dossier/readme.txt", "r+");
```

Je vous recommande donc d'utiliser des chemins relatifs plutôt que des chemins absolus. N'utilisez les chemins absolus que si votre programme est fait pour un OS précis et doit modifier un fichier précis quelque part sur votre disque dur.

Tester l'ouverture du fichier

Le pointeur "fichier" devrait contenir l'adresse de la structure de type FILE qui sert de descripteur de fichier. Celui-ci a été chargé en mémoire pour vous par la fonction fopen().
A partir de là, 2 possibilités :

- Soit l'ouverture a réussi, et vous pouvez continuer (c'est-à-dire commencer à lire et écrire dans le fichier).
- Soit l'ouverture a échoué parce que le fichier n'existait pas ou était utilisé par un autre programme. Dans ce cas, vous devez arrêter de travailler sur le fichier.

Juste après l'ouverture du fichier, **il FAUT absolument vérifier si l'ouverture a réussi ou pas**. Pour faire ça, c'est très simple : si le pointeur vaut NULL, l'ouverture a échoué. S'il vaut autre chose que NULL, l'ouverture a réussi.
On va donc suivre systématiquement le schéma suivant :

Code : C

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "r+");

    if (fichier != NULL)
    {
        // On peut lire et écrire dans le fichier
    }
    else
    {
        // On affiche un message d'erreur si on veut
        printf("Impossible d'ouvrir le fichier test.txt");
    }

    return 0;
}
```

Faites toujours ça lorsque vous ouvrez un fichier. Si vous ne le faites pas et que le fichier n'existe pas, vous risquez un plantage du programme ensuite.

fclose : fermer le fichier

Si l'ouverture du fichier a réussi, vous pouvez lire et écrire dedans (nous allons voir de suite après comment faire).
Une fois que vous aurez fini de travailler avec le fichier, il faudra le "fermer". On utilise pour cela la fonction fclose qui a pour rôle de libérer la mémoire, c'est-à-dire supprimer votre fichier chargé dans la mémoire vive.

Son prototype est :

Code : C

```
int fclose(FILE* pointeurSurFichier);
```

Cette fonction prend un paramètre : votre pointeur sur le fichier.
Elle renvoie un int qui indique si elle a réussi à fermer le fichier. Cet int vaut :

- 0 : si la fermeture a marché
- EOF : si la fermeture a échoué. EOF est un define situé dans stdio.h qui correspond à un nombre spécial, utilisé pour dire soit qu'il y a eu une erreur, soit qu'on est arrivés à la fin du fichier. Dans le cas présent cela signifie qu'il y a eu une erreur.

A priori, la fermeture se passe toujours bien donc je n'ai pas l'habitude de tester si le fclose a marché. Vous pouvez le faire néanmoins si vous le voulez.

Pour fermer le fichier, on va donc écrire :

Code : C

```
fclose(fichier);
```

Tout simplement

Au final, le schéma que nous allons suivre pour ouvrir et fermer un fichier sera le suivant :

Code : C

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "r+");

    if (fichier != NULL)
    {
        // On lit et on écrit dans le fichier

        // ...

        fclose(fichier); // On ferme le fichier qui a été ouvert
    }

    return 0;
}
```

Je n'ai pas mis le else ici pour afficher un message d'erreur si l'ouverture a échoué, mais vous pouvez le faire si vous le désirez.

Il faut toujours penser à fermer son fichier une fois que l'on a fini de travailler avec. Cela permet de libérer de la mémoire.
Si vous oubliez de libérer la mémoire, votre programme risque à la fin de prendre énormément de mémoire qu'il n'utilise plus. Sur un petit exemple comme ça ce n'est pas flagrant, mais sur un gros programme, bonjour les dégâts

Oublier de libérer la mémoire, ça arrive. Ca vous arrivera d'ailleurs très certainement. Dans ce cas, vous serez témoins de ce que l'on appelle des **fuites mémoire**. Votre programme se mettra alors à utiliser plus de mémoire que nécessaire sans que vous arriviez à comprendre pourquoi. Bien souvent, il s'agit simplement d'un ou deux trucs comme des petits *fclose* oubliés. Comme quoi la solution à un problème mystique est parfois toute bête (je dis bien "parfois")

DIFFÉRENTES MÉTHODES DE LECTURE / ÉCRITURE

Maintenant que nous avons écrit le code qui ouvre et ferme le fichier, nous n'avons plus qu'à insérer le code qui lit et écrit dedans

Nous allons commencer par voir comment écrire dans un fichier (ce qui est un peu plus simple), puis nous verrons ensuite comment lire dans un fichier.

Ecrire dans le fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier. Ce sera à vous de choisir celle qui est la plus adaptée à votre cas.
Voici les 3 fonctions que nous allons étudier :

- fputc : écrit un caractère dans le fichier (UN SEUL caractère à la fois).
- fputs : écrit une chaîne dans le fichier
- fprintf : écrit une chaîne "formatée" dans le fichier, fonctionnement quasi-identique à printf

fputc

Cette fonction écrit un caractère à la fois dans le fichier. Son prototype est :

Code : C

```
int fputc(int caractere, FILE* pointeurSurFichier);
```

Elle prend 2 paramètres :

- Le caractère à écrire (de type int, ce qui comme je vous l'ai dit revient plus ou moins à utiliser un char, sauf que le nombre de caractères utilisables est ici plus grand). Vous pouvez donc écrire directement 'A' par exemple.
- Le pointeur sur le fichier dans lequel écrire. Dans notre exemple, notre pointeur s'appelle "fichier". L'avantage de demander le pointeur de fichier à chaque fois, c'est que vous pouvez ouvrir plusieurs fichiers en même temps et donc lire et écrire dans chacun de ces fichiers. Vous n'êtes pas limités à un seul fichier ouvert à la fois.

La fonction retourne un int, c'est un code d'erreur. Cet int vaut EOF si l'écriture a échoué, sinon il vaut autre chose. Comme le fichier a été ouvert avec succès normalement, je n'ai pas l'habitude de tester si chacun de mes fputc a réussi, mais vous pouvez le faire encore une fois si vous le voulez.

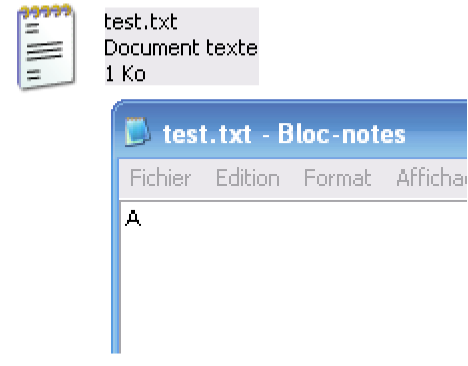
Le code suivant écrit la lettre 'A' dans test.txt (si le fichier existe, il est remplacé ; si il n'existe pas, il est créé). Il y a tout dans ce code : ouverture, test de l'ouverture, écriture et fermeture.

Code : C

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "w");

    if (fichier != NULL)
    {
        fputc('A', fichier); // Ecriture du caractère A
        fclose(fichier);
    }

    return 0;
}
```

Ouvrez votre fichier "test.txt". Que voyez-vous ?
C'est magique (enfin pas tellement), le fichier contient maintenant la lettre 'A' !

La preuve en image :



fputs

Cette fonction est très similaire à fputc, à la différence près qu'elle écrit tout une chaîne, ce qui est en général plus pratique que d'écrire caractère par caractère
Ceci dit, fputc reste utile lorsque vous devez écrire caractère par caractère, ce qui arrive fréquemment

Prototype de la fonction :

Code : C

```
char* fputs(const char* chaine, FILE* pointeurSurFichier);
```

Les 2 paramètres sont faciles à comprendre :

- chaine : la chaîne à écrire. Notez que le type ici est **const char*** : en rajoutant le mot const dans le prototype, la fonction indique que pour elle la chaîne sera considérée comme une constante. En 1 mot comme en 100 : elle s'interdit de modifier le contenu de votre chaîne. C'est logique quand on y pense : fputs doit juste lire votre chaîne, pas la modifier. C'est donc pour vous une information (et une sécurité) comme quoi votre chaîne ne subira pas de modification
- pointeurSurFichier : comme pour fputc, il s'agit de votre pointeur de type FILE* sur le fichier que vous avez ouvert.

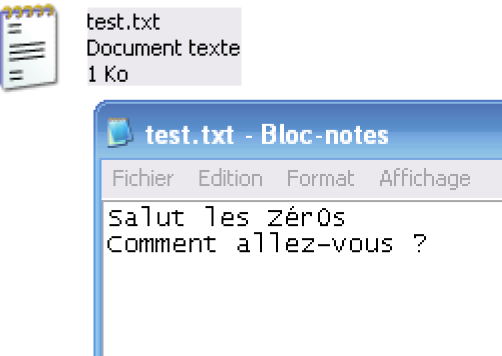
La fonction renvoie EOF s'il y a eu une erreur, sinon c'est que cela a fonctionné. Là non plus, je ne teste en général pas la valeur de retour.

Testons l'écriture d'une chaîne dans le fichier :

Code : C

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "w");

    if (fichier != NULL)
    {
        fputs("Salut les Zér0s\nComment allez-vous ?", fichier);
        fclose(fichier);
    }

    return 0;
}
```

Preuve que ce code fonctionne :



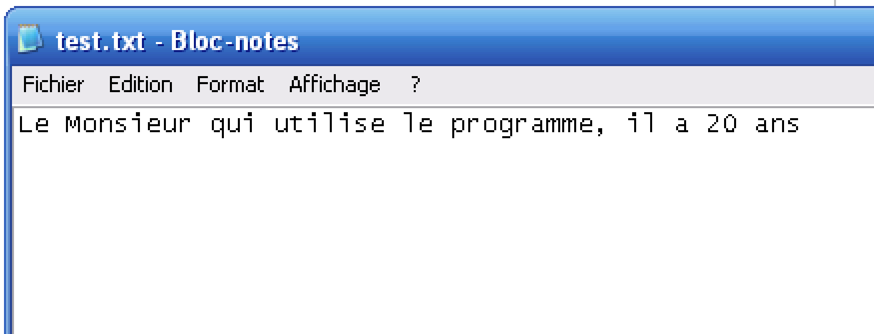
fprintf

Voici un autre exemplaire de la fonction printf. Celle-ci peut être utilisée pour écrire dans un fichier. Elle s'utilise de la même manière que printf d'ailleurs, excepté le fait que vous devez indiquer un pointeur de FILE en premier paramètre.

Ce code demande l'âge de l'utilisateur et l'écrit dans le fichier :

Code : C

```c
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    long age = 0;

    fichier = fopen("test.txt", "w");

    if (fichier != NULL)
    {
        // On demande l'âge
        printf("Quel age avez-vous ? ");
        scanf("%ld", &age);

        // On l'écrit dans le fichier
        fprintf(fichier, "Le Monsieur qui utilise le programme, il a %ld ans", age);
        fclose(fichier);
    }

    return 0;
}
```



Vous pouvez ainsi facilement réutiliser ce que vous savez de printf pour écrire dans un fichier, c'est pas génial ça ?
C'est pour cette raison d'ailleurs que j'utilise le plus souvent fprintf pour écrire dans des fichiers, car c'est pratique et pas trop compliqué

Lire dans un fichier

Nous pouvons utiliser quasiment les mêmes fonctions que pour l'écriture, le nom change juste un petit peu :

- fgetc : lit un caractère
- fgets : lit une chaîne
- fscanf : lit une chaîne formatée

Je vais aller un peu plus vite cette fois dans l'explication de ces fonctions, si vous avez compris ce que j'ai écrit plus haut ça ne devrait pas poser de problème

fgetc

Tout d'abord le prototype :

Code : C

```
int fgetc(FILE* pointeurDeFichier);
```

Cette fonction retourne un int : c'est le caractère qui a été lu.
Si la fonction n'a pas pu lire de caractère, elle retourne EOF.

? Mais comment savoir quel caractère on lit ? Si on veut lire le 3ème caractère, ainsi que le 10ème caractère, comment doit-on faire ?

En fait, au fur et à mesure que vous lisez un fichier, vous avez un "curseur" qui avance. C'est un curseur virtuel hein, vous ne le voyez pas à l'écran
Mais vous pouvez imaginer que ce curseur est comme la barre clignotante lorsque vous éditez un fichier sous Bloc-Notes. Il indique où vous en êtes dans la lecture du fichier.
Nous verrons peu après comment savoir à quelle position le curseur est situé dans le fichier, et aussi comment modifier la position du curseur (pour le remettre au début du fichier par exemple, ou le placer à un caractère précis, comme le 10ème caractère).

fgetc avance le curseur d'un caractère à chaque fois que vous en lisez un. Si vous appelez fgetc une seconde fois, la fonction lira donc le second caractère, puis le troisième et ainsi de suite
Vous pouvez faire une boucle pour lire les caractères un par un dans le fichier donc

On va écrire un code qui lit tous les caractères d'un fichier un à un, et qui les écrit à chaque fois à l'écran.
La boucle s'arrête quand fgetc renvoie EOF (qui signifie End Of File, c'est-à-dire "fin du fichier").

Code : C

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    int caractereActuel = 0;

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        // Boucle de lecture des caractères un à un
        do
        {
            caractereActuel = fgetc(fichier); // On lit le caractère
            printf("%c", caractereActuel); // On l'affiche
        } while (caractereActuel != EOF); // On continue tant que fgetc n'a pas retourné EOF (fin de fichier)

        fclose(fichier);
    }

    return 0;
}
```

La console affichera tout le contenu du fichier, par exemple :

Code : Console

```
Coucou, je suis le contenu du fichier test.txt !
```

fgets

Cette fonction lit une chaîne dans le fichier. Ca vous évite d'avoir à lire tous les caractères un par un. La fonction **lit au maximum une ligne** (elle s'arrête au premier \n qu'elle rencontre). Si vous voulez lire plusieurs lignes, il faudra faire une boucle.

Voici le prototype de fgets :

Code : C

```
char* fgets(char* chaine, int nombreDeCaracteresALire, FILE* pointeurSurFichier);
```

Cette fonction demande un paramètre un peu particulier, qui va en fait s'avérer très pratique : le nombre de caractères à lire. Cela demande à la fonction fgets de s'arrêter de lire la ligne si elle contient plus de X caractères. Avantage : ça nous permet de nous assurer que l'on ne fera pas de dépassement de mémoire ! En effet, si la ligne est trop grosse pour rentrer dans chaine, la fonction aurait lu plus de caractères qu'il n'y a de place, ce qui aurait probablement provoqué un plantage du programme.

Lire une ligne avec fgets

Nous allons d'abord voir comment lire une ligne avec fgets (nous verrons ensuite comment lire tout le fichier).

On crée une chaîne suffisamment grande pour stocker le contenu de la ligne qu'on va lire (du moins on espère). Vous allez voir là tout l'intérêt d'utiliser un define pour définir la taille du tableau :

Code : C

```
#define TAILLE_MAX 1000 // Tableau de taille 1000

int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    char chaine[TAILLE_MAX] = ""; // Chaîne vide de taille TAILLE_MAX

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        fgets(chaine, TAILLE_MAX, fichier); // On lit maximum TAILLE_MAX caractères du
fichier, on stocke le tout dans "chaine"
        printf("%s", chaine); // On affiche la chaîne

        fclose(fichier);
    }

    return 0;
}
```

Le résultat est le même que pour le code de tout à l'heure, à savoir que le contenu s'écrit dans la console :

Code : Console

```
Coucou, je suis le contenu du fichier test.txt !
```

La différence, c'est qu'ici on ne fait pas de boucle. On affiche toute la chaîne d'un coup.

Vous aurez sûrement remarqué maintenant l'intérêt que peut avoir un #define dans son code pour

définir la taille maximale d'un tableau par exemple. En effet, TAILLE_MAX est ici utilisé à 2 endroits du code :

- Une première fois pour définir la taille du tableau à créer
- Une autre fois dans le fgets pour limiter le nombre de caractères à lire.

L'avantage ici, c'est que si vous vous rendez compte que la chaîne n'est pas assez grande pour lire le fichier, vous n'avez qu'à changer la ligne du define et recompiler Cela vous évite d'avoir à chercher tous les endroits du code qui indiquent la taille du tableau. Le préprocesseur remplacera tous les TAILLE_MAX dans le code par la nouvelle valeur

Lire tout le fichier avec fgets

Comme je vous l'ai dit, fgets lit au maximum toute une ligne à la fois. Elle s'arrête de lire la ligne si elle dépasse le nombre de caractères maximum que vous autorisez.

Oui mais voilà, pour le moment on ne sait lire qu'une seule ligne à la fois avec fgets. Comment diable lire tout le fichier ?
La réponse est simple : avec une boucle

La fonction fgets renvoie NULL si elle n'est pas parvenue à lire ce que vous avez demandé.
La boucle doit donc s'arrêter dès que fgets se met à renvoyer NULL.

On n'a plus qu'à faire un while pour boucler tant que fgets ne renvoit pas NULL, et zou !

Code : C

```c
#define TAILLE_MAX 1000

int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    char chaine[TAILLE_MAX] = "";

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        while (fgets(chaine, TAILLE_MAX, fichier) != NULL) // On lit le fichier tant
qu'on ne reçoit pas d'erreur (NULL)
        {
            printf("%s", chaine); // On affiche la chaîne qu'on vient de lire
        }

        fclose(fichier);
    }

    return 0;
}
```

Ce code source lit et affiche tout le contenu de mon fichier, ligne par ligne.

La ligne de code la plus intéressante c'est celle du while (en fait c'est la seule nouvelle chose par rapport à tout à l'heure) :

Code : C

```c
while (fgets(chaine, TAILLE_MAX, fichier) != NULL)
```

La ligne du while fait 2 choses : elle lit une ligne dans le fichier et vérifie si fgets ne renvoie pas NULL. Elle peut donc se traduire donc comme ceci : "*Lire une ligne du fichier tant qu'on **n'est pas** arrivés à la fin du fichier*".

fscanf

C'est le même principe que la fonction scanf là encore.
Cette fonction lit dans un fichier qui doit avoir été écrit d'une manière précise.

Supposons que votre fichier contienne 3 nombres séparés par un espace, qui sont par exemple les 3 plus hauts scores obtenus à votre jeu :

```
15 20 30
```

Vous voudriez récupérer chacun de ces nombres dans une variable de type long.
La fonction fscanf va vous permettre de faire ça rapidement.

Code : C

```c
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    long score[3] = {0}; // Tableau des 3 meilleurs scores

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        fscanf(fichier, "%ld %ld %ld", &score[0], &score[1], &score[2]);
        printf("Les meilleurs scores sont : %ld, %ld et %ld", score[0], score[1],
score[2]);

        fclose(fichier);
    }

    return 0;
}
```

Code : Console

```
Les meilleurs scores sont : 15, 20 et 30
```

Comme vous le voyez, la fonction fscanf attend 3 nombres séparés par un espace (`"%ld %ld %ld"`). Elle les stocke ici dans notre tableau de 3 blocs.

On affiche ensuite chacun des nombres récupérés.

Jusqu'ici, je ne vous avais fait mettre qu'un seul "%ld" entre guillemets pour la fonction scanf. Vous découvrez aujourd'hui qu'on peut en mettre plusieurs, les combiner. Si votre fichier est écrit d'une façon bien précise, cela permet d'aller plus vite pour récupérer chacune des valeurs

SE DÉPLACER DANS UN FICHIER

Je vous ai parlé d'une espèce de "curseur" virtuel tout à l'heure. Nous allons l'étudier maintenant plus en détail.
A chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier. Vous pouvez imaginer que c'est exactement comme le curseur de votre éditeur de texte (tel bloc-notes). Il indique où vous êtes dans le fichier, et donc où vous allez écrire.

En résumé : le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier.

Il existe 3 fonctions à connaître :

- ftell : indique à quelle position vous êtes actuellement dans le fichier
- fseek : positionne le curseur à un endroit précis
- rewind : remet le curseur au début du fichier (c'est équivalent à demander à la fonction fseek de positionner le curseur au début).

ftell : position dans le fichier

Cette fonction est très simple à utiliser. Elle renvoie la position actuelle du curseur sous la forme d'un *long* :

Code : C

```
long ftell(FILE* pointeurSurFichier);
```

Le nombre renvoyé indique donc la position du curseur dans le fichier.

fseek : se positionner dans le fichier

Le prototype de fseek est le suivant :

Code : C

```
int fseek(FILE* pointeurSurFichier, long deplacement, int origine);
```

La fonction fseek permet de déplacer le "curseur" d'un certain nombre de caractères (indiqué par *deplacement*) à partir de la position indiquée par *origine*.

- Le nombre *deplacement* peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant au nombre *origine*, vous pouvez mettre comme valeur l'une des 3 constantes (généralement des defines) listées ci-dessous :
  - **SEEK_SET** : indique le début du fichier.
  - **SEEK_CUR** : indique la position actuelle du curseur.
  - **SEEK_END** : indique la fin du fichier.

Voici quelques exemples pour bien comprendre comment on jongle avec *deplacement* et *origine*

- Le code suivant place le curseur 2 caractères *après* le début :

  Code : C

  ```
  fseek(fichier, 2, SEEK_SET);
  ```

- Le code suivant place le curseur 4 caractères *avant* la position courante :

  Code : C

  ```
  fseek(fichier, -4, SEEK_CUR);
  ```

(remarquez que deplacment est négatif car on se déplace en arrière)

- Le code suivant place le curseur à la fin du fichier :

Code : C

```
fseek(fichier, 0, SEEK_END);
```

Si vous écrivez après avoir fait un fseek qui mène à la fin du fichier, cela rajoutera vos informations à la suite dans le fichier (ça complètera votre fichier).
En revanche, si vous placez le curseur au début et que vous écrivez, cela écrasera le texte qui se trouvait là. Il n'y a pas de moyen d'"insérer" de texte dans le fichier (à moins de coder soi-même une fonction qui lit les caractères d'après pour s'en souvenir avant de les écraser !).

Mais comment je sais à quelle position je dois aller lire et écrire dans le fichier ?

Alors ça, c'est vous qui gérez
Si c'est un fichier que vous avez vous-même écrit, vous savez comment il est construit. Vous savez donc où aller chercher vos informations (par exemple les meilleurs scores sont en position 0, les noms des derniers joueurs sont en position 50 etc...)

On fera un TP un peu plus tard dans lequel vous comprendrez (si ce n'est pas déjà le cas) comment on fait pour aller chercher l'information qui nous intéresse.
N'oubliez pas que c'est **vous** qui définissez comment votre fichier est construit. C'est donc à vous de dire : "je place le score du meilleur joueur sur la première ligne, celui du second meilleur joueur sur la seconde ligne" etc...

La fonction fseek peut se comporter bizarrement sur des fichiers ouverts en mode texte. En général, on l'utilise plutôt pour se déplacer dans des fichiers ouverts en mode binaire.
Quand on lit / écrit dans un fichier en mode texte, on le fait généralement caractère par caractère. La seule chose qu'on se permet en mode texte avec fseek c'est de revenir au début ou de se placer à la fin.

En résumé : fseek c'est bien, mais à utiliser plutôt avec des fichiers binaires. On ne se déplace en général pas avec fseek sur des fichiers texte.

rewind : retour au début

Cette fonction est équivalente à utiliser fseek pour nous renvoyer à la position 0 dans le fichier. Si vous avez eu un magnétoscope un jour dans votre vie, eh bien c'est le même nom que la touche qui permet de revenir en arrière.

Le prototype est tout bête :

Code : C

```
void rewind(FILE* pointeurSurFichier);
```

L'utilisation est aussi bête que le prototype. Pour la peine, je ne vous donne pas d'exemple cette fois (je commence à avoir les doigts ankylosés à ce moment de l'écriture du chapitre)

Renommer et supprimer un fichier

Nous terminerons ce chapitre en douceur par l'étude de 2 fonctions très simples :

- rename : renomme un fichier
- remove : supprime un fichier

La particularité de ces fonctions est **qu'elles ne nécessitent pas de pointeur de fichier pour fonctionner**. Il suffira juste d'indiquer le nom du fichier à renommer / supprimer
Bref, c'est encore plus simple que simple

rename : renommer un fichier

Ca va être vite étudié vous allez voir

Code : C

```
int rename(const char* ancienNom, const char* nouveauNom);
```

La fonction renvoie 0 si elle a réussi à renommer, sinon elle renvoie... autre chose que 0

Avez-vous vraiment besoin d'un exemple ?
Bon allez

Code : C

```
int main(int argc, char *argv[])
{
    rename("test.txt", "test_renomme.txt");

    return 0;
}
```

Ouahouh c'était duuuur
Et voilà mon fichier renommé

remove : supprimer un fichier

Cette fonction supprime un fichier sans demander son reste :

Code : C

```
int remove(const char* fichierASupprimer);
```

Faites très attention en utilisant cette fonction ! Elle supprime le fichier indiqué sans demander de confirmation ! Le fichier n'est pas mis dans la corbeille ni rien, il est littéralement supprimé du disque dur. Il n'est pas possible de récupérer un fichier supprimé.

Cette fonction tombe à pic pour la fin du chapitre, je n'ai justement plus besoin du fichier test.txt, je vais donc le supprimer

Code : C

```
int main(int argc, char *argv[])
{
    remove("test.txt");

    return 0;
}
```

Aussi étonnant que cela puisse paraître, ce chapitre ne vous a enseigné aucune nouvelle connaissance en langage C.
Les pointeurs, les tableaux, les structures : ça c'était de l'étude du langage C.
Ici, nous n'avons fait qu'utiliser des fonctions de stdio, une librairie standard. **Nous avons donc fait l'étude d'une librairie** (pas en entier ceci dit, il y a quelques autres fonctions dans stdio, mais on en a vu un bon gros morceau).

Je ne dis pas que nous avons déjà fait le tour du langage C mais... presque
En fait, le langage C est "vite" appris pour ce qui est de la base. Le plus long est certainement d'arriver à comprendre les pointeurs, à ne pas confondre valeur et adresse etc.

Mais, une fois que c'est fait, vous êtes en théorie capables de tout programmer. **Le tout est de savoir utiliser des librairies**, c'est-à-dire faire ce qu'on vient de faire ici : apprendre à utiliser des fonctions de ces librairies. Sans librairie, un programme ne peut donc rien faire. Même le printf vous n'auriez pas pu le faire, vu qu'il est situé dans stdio

Lorsque nous aurons terminé la partie II (ce qui ne va plus tarder maintenant), vous aurez fait le tour du langage C. Il restera, certes, 2-3 choses dont je n'aurai pas parlé (je ne peux pas parler de tout). Mais, franchement, si vous arrivez à comprendre et à retenir tout ce que je vous aurai appris dans la partie II bah... chapeau Vous serez désormais aptes à programmer avec n'importe quelle librairie.

La prochaine partie (partie III) sera entièrement dédiée à l'utilisation d'une librairie appelée la SDL. Comble du bonheur, cette librairie contient des fonctions permettant d'ouvrir des fenêtres, dessiner à l'écran, jouer du son, lire des CD Audio, manipuler le clavier, la souris et même le joystick
Alors allez, un peu de courage, vous allez bientôt arriver dans la partie "amusante" de la programmation. Il faut bien que vos efforts soient récompensés, non ?

L'allocation dynamique

Inspirez un grand coup : ce chapitre est le dernier chapitre "théorique" que vous lirez avant un bon moment

De quoi va-t-on parler aujourd'hui ? On va voir comment créer une variable *manuellement* (= dynamiquement).
Quand on déclare une variable, on dit qu'on **demande à allouer de la mémoire** :

Code : C

```
long monNombre = 0;
```

Lorsque le programme arrive à une ligne comme celle-là, il se passe en fait les choses suivantes :

1. Votre programme demande au système d'exploitation (Windows, Linux, Mac OS...) la permission d'utiliser un peu de mémoire.
2. Le système d'exploitation répond à votre programme en lui indiquant *où il peut stocker cette variable* (il lui

donne l'adresse qu'il lui a réservée).
3. Lorsque la fonction est terminée, la variable est *automatiquement* supprimée de la mémoire. Votre programme dit au système d'exploitation : "*Je n'ai plus besoin de l'espace en mémoire que tu m'avais réservé à telle adresse, merci*" (Nota : l'histoire ne précise pas si le programme dit "merci" à l'OS, mais c'est tout dans son intérêt parce que c'est l'OS qui contrôle la mémoire)

Jusqu'ici, les choses étaient automatiques. Lorsqu'on déclarait une variable, le système d'exploitation était automatiquement appelé par le programme.
Que diriez-vous de faire cela manuellement ? Non pas par pur plaisir de faire quelque chose de compliqué (même si c'est tentant), mais plutôt parce que parfois on est obligés de faire comme ça.

Dans ce chapitre, nous allons :

- Etudier le fonctionnement de la mémoire (oui, encore !) pour **voir la taille que prend une variable en fonction de son type**.
- Puis, nous attaquerons le gros du sujet : nous verrons comment demander manuellement de la mémoire au système d'exploitation. On fera ce qu'on appelle de **l'allocation dynamique de mémoire**.
- Enfin, nous verrons l'intérêt de faire une allocation dynamique de mémoire en apprenant à **créer un tableau dont la taille n'est connue qu'à l'exécution du programme**.

Il est impératif de bien savoir manipuler les pointeurs pour pouvoir lire ce chapitre ! Si vous avez encore des doutes sur les pointeurs, je vous recommande d'aller relire le chapitre sur les pointeurs avant de commencer quoi que ce soit !

La taille des variables

Selon le type de variable que vous demandez à créer (char, int, double, float...), vous avez besoin de plus ou moins de mémoire.

En effet, pour stocker un nombre compris entre -128 et 127 (un char), on n'a besoin que d'un octet en mémoire (c'est tout petit).
En revanche, un int occupe généralement 4 octets en mémoire. Quant au double, il occupe 8 octets.

Le problème est... que ce n'est pas toujours le cas. Cela dépend de votre ordinateur : peut-être que chez vous un int occupe 8 octets, qui sait ?
Notre objectif ici est de vérifier quelle taille occupe chacun des types sur votre ordinateur.

Il y a un moyen très facile pour savoir cela : utiliser l'opérateur sizeof().
Contrairement aux apparences, ce n'est pas une fonction mais une fonctionnalité *de base* du langage C. Vous devez juste indiquer entre parenthèses le type que vous voulez analyser.
Pour connaître la taille d'un int, on devra donc écrire :

Code : C

```c
sizeof(int)
```

A la compilation, cela sera remplacé par un nombre : le nombre d'octets que prend int en mémoire. Chez moi, sizeof(int) vaut 4, ce qui signifie que int occupe 4 octets. Chez vous, c'est probablement la même valeur, mais ce n'est pas une règle. Testez, vous verrez
Vous pouvez faire des printf pour afficher cela :

Code : C

```c
printf("char : %ld octets\n", sizeof(char));
printf("int : %ld octets\n", sizeof(int));
printf("long : %ld octets\n", sizeof(long));
printf("double : %ld octets\n", sizeof(double));
```

Chez moi, cela affiche :

Code : Console

```
char : 1 octets
int : 4 octets
long : 4 octets
double : 8 octets
```

Je n'ai pas mis tous les types que nous connaissons, je vous laisse le soin de tester vous-même la taille des autres types
Vous remarquerez que long et int occupent la même place en mémoire. Créer un long revient donc ici exactement à créer un int, cela prend 4 octets dans la mémoire.

> En fait, le type "long" est équivalent à un type appelé "long int", qui est ici équivalent au type... "int". Bref, ça fait beaucoup de noms différents pour pas grand-chose au final Avoir de nombreux types différents était utile à une époque où on n'avait pas beaucoup de mémoire. On cherchait à utiliser le minimum de mémoire possible en utilisant le type le plus adapté.
> Aujourd'hui, cela ne sert plus vraiment car la mémoire d'un ordinateur est très grande.

Oui ! sizeof marche aussi sur les structures !

Code : C

```c
typedef struct Coordonnees Coordonnees;
struct Coordonnees
{
    long x;
    long y;
};

int main(int argc, char *argv[])
{
    printf("Coordonnees : %ld octets\n", sizeof(Coordonnees));

    return 0;
}
```

Code : Console

```
Coordonnees : 8 octets
```

Plus une structure contient de sous-variables, plus elle prend de mémoire. Terriblement logique n'est-ce pas ?

Une nouvelle façon de voir la mémoire

Jusqu'ici, mes schémas de mémoire étaient encore assez imprécis. On va enfin pouvoir les rendre précis et corrects maintenant qu'on connaît la taille de chacun des types de variables (c'est pas trop tôt)

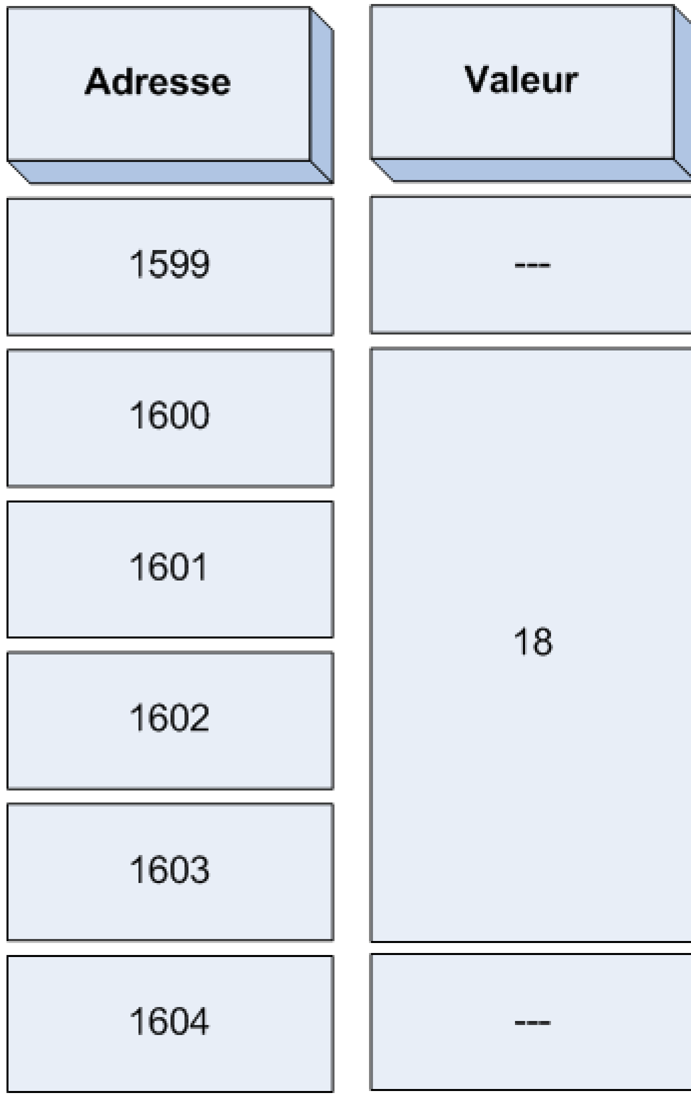
Si on déclare une variable de type long :

Code : C

```
long nombre = 18;
```

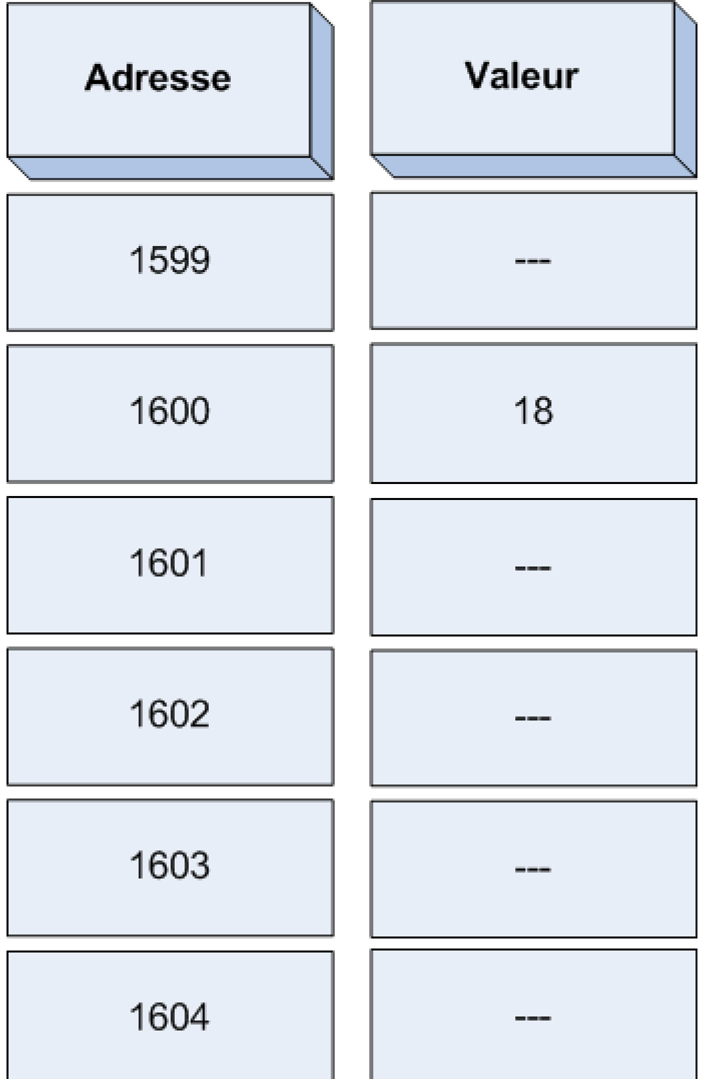
... et que sizeof(long) indique 4 octets sur notre ordinateur, alors la variable occupera 4 octets en mémoire !

Supposons que la variable nombre soit allouée à l'adresse 1600 en mémoire. On aurait alors le schéma suivant :



Ici, on voit bien que notre variable "nombre" de type long qui vaut 18 occupe 4 octets dans la mémoire.
Elle commence à l'adresse 1600 (c'est son adresse) et termine à l'adresse 1603. La prochaine variable ne pourra donc être stockée qu'à partir de l'adresse 1604 !

Si on avait fait la même chose avec un char, alors on n'aurait occupé qu'un seul octet en mémoire :



Imaginez maintenant un tableau de long !
Chaque "case" du tableau occupera 4 octets. Si notre tableau fait 100 cases :

Code : C
```
long tableau[100];
```

Alors on occupera en réalité 4 * 100 = 400 octets en mémoire

Bien sûr ! La place en mémoire est réservée, aucun autre programme n'a le droit d'y toucher (à part le vôtre). Une fois qu'une variable est déclarée, elle prend immédiatement de la place en mémoire.

Notez que si on crée un tableau de type "Coordonnees" :

Code : C
```
Coordonnees tableau[100];
```

... on utilisera (allez c'est facile) : 8 * 100 = 800 octets en mémoire.

Il est important de bien comprendre ces petits calculs pour la suite du chapitre.
C'est de la multiplication de niveau Primaire ça

Allocation de mémoire dynamique

Rentrons maintenant dans le vif du sujet.
Le but du chapitre, c'était quoi justement ?

Ah oui : apprendre à demander de la mémoire manuellement.
On va avoir besoin d'inclure la librairie <stdlib.h> (si vous avez suivi mes conseils, vous devriez avoir inclus cette librairie dans tous vos programmes de toute façon).

Cette librairie contient 2 fonctions dont nous allons avoir besoin :

- **malloc** (*"Memory ALLOCation", c'est-à-dire "Allocation de mémoire"*) : demande au système d'exploitation la permission d'utiliser de la mémoire.
- **free** (*"Libérer"*) : permet d'indiquer à l'OS que l'on n'a plus besoin de la mémoire qu'on avait demandée. La place en mémoire est libérée, un autre programme peut maintenant s'en servir au besoin.

Quand vous faites une allocation manuelle de mémoire (ce qu'on va apprendre à faire maintenant), vous devez toujours suivre ces 3 étapes :

1. Appeler **malloc** pour demander de la mémoire
2. **Vérifier la valeur retournée par malloc** pour savoir si l'OS a bien réussi à allouer la mémoire.
3. Une fois qu'on a fini d'utiliser la mémoire, on doit la libérer avec **free**. Si on ne le fait pas, on s'expose à des fuites de mémoire, c'est-à-dire que votre programme risque au final de prendre beaucoup de mémoire alors qu'il n'a en réalité plus besoin de tout cet espace.

Tiens, ces 3 étapes ça vous rappelle pas le chapitre sur les fichiers ça ?
Ben moi si

Le principe est exactement le même qu'avec les fichiers : on alloue, on vérifie si l'allocation a marché, on utilise la mémoire, puis on libère quand on a fini d'utiliser.

Nous allons maintenant étudier la fonction malloc.

malloc : demande d'allocation de mémoire

Le prototype de la fonction malloc est assez comique vous allez voir :

Code : C

```
void* malloc(size_t nombreOctetsNecessaires);
```

La fonction prend un paramètre : le nombre d'octets à réserver. Ainsi, il suffira d'écrire sizeof(long) dans ce paramètre pour réserver suffisamment d'espace pour stocker un long.

Mais c'est surtout ce que la fonction renvoie qui est curieux : elle renvoie un... void*
Si vous vous souvenez du chapitre sur les fonctions, je vous avais dit que "void" signifiait "vide" et qu'on utilisait ce type pour indiquer que la fonction ne retournait aucune valeur.

Alors ici, on aurait une fonction qui retourne un "pointeur sur vide" ?

En voilà une bien bonne !
Ces programmeurs ont décidemment un sens de l'humour très développé

Ca te dérangerait pas trop de nous donner quelques explications ?

Oui oui j'y viens Je me rappelle juste la première fois que j'ai vu le prototype de malloc, je suis resté la bouche ouverte un petit moment devant mon écran avant de comprendre

En fait, cette fonction renvoie un pointeur indiquant l'adresse que l'OS a réservé pour votre variable. Si l'OS a trouvé de la place pour vous à l'adresse 1600, la fonction renvoie donc un pointeur contenant l'adresse 1600.
Le problème, c'est que la fonction malloc ne sait pas quel type de variable vous cherchez à créer. En effet, vous ne lui donnez qu'un paramètre : le nombre d'octets en mémoire dont vous avez besoin. Si vous demandez 4 octets, ça pourrait aussi bien être un int qu'un long par exemple !

Comme malloc ne sait pas quel type elle doit retourner, elle renvoie le type void*. **Ce sera un pointeur sur n'importe quel type**. On peut dire que c'est un pointeur universel.

Passons à la pratique. Si je veux m'amuser (*hahem*) à créer manuellement une variable de type long en mémoire, je devrai indiquer à malloc que j'ai besoin de sizeof(long) octets en mémoire.
Je récupère le résultat du malloc dans un pointeur sur long.

Code : C

```c
long* memoireAllouee = NULL; // On crée un pointeur sur long

memoireAllouee = malloc(sizeof(long)); // La fonction malloc inscrit dans notre pointeur
l'adresse qui a été reservee.
```

A la fin de ce code, memoireAllouee est un pointeur contenant une adresse qui vous a été réservée par l'OS, par exemple l'adresse 1600 (pour reprendre mes schémas de tout à l'heure).

Tester le pointeur

La fonction malloc a donc renvoyé dans notre pointeur memoireAllouee l'adresse qui a été réservée pour vous en mémoire.
2 possibilités :

- Si l'allocation a marché, notre pointeur contient une adresse.
- Si l'allocation a échoué, notre pointeur contient l'adresse NULL.

Il est peu probable qu'une allocation échoue, mais cela peut arriver. Imaginez que vous demandiez à utiliser 34 Go de mémoire vive, il y a très peu de chances que l'OS vous réponde favorablement

Il est néanmoins recommandé de **tester si l'allocation a marché**. On va faire ceci : si l'allocation a échoué, c'est qu'il n'y avait plus de mémoire de libre (c'est un cas critique). Dans un tel cas, le mieux est d'arrêter immédiatement le programme parce qu'il ne pourra pas continuer convenablement de toute manière.

On va utiliser une fonction standard qu'on n'avait pas encore vue jusqu'ici : exit(). Elle arrête immédiatement le programme. Elle prend un paramètre : la valeur que le programme doit retourner (ça correspond au return du main()).

Code : C

```
int main(int argc, char *argv[])
{
    long* memoireAllouee = NULL;

    memoireAllouee = malloc(sizeof(long));
    if (memoireAllouee == NULL) // Si l'allocation a échoué
    {
        exit(0); // On arrête immédiatement le programme
    }

    // On peut continuer le programme normalement sinon.

    return 0;
}
```

Si le pointeur est différent de NULL, le programme peut continuer, sinon il faut afficher un message d'erreur ou même mettre fin au programme, parce qu'il ne pourra pas continuer correctement s'il n'y a plus de place en mémoire.

free : libérer de la mémoire

Tout comme on utilisait la fonction fclose pour fermer un fichier dont on n'avait plus besoin, on va utiliser la fonction free pour libérer la mémoire quand on n'en a plus besoin.

Code : C

```
void free(void* pointeur);
```

La fonction free a juste besoin de l'adresse mémoire à libérer. On va donc lui envoyer notre pointeur, c'est-à-dire memoireAllouee dans notre exemple.
Voici le schéma complet et final, ressemblant à s'y méprendre à ce qu'on a vu dans le chapitre sur les fichiers :

Code : C

```
int main(int argc, char *argv[])
{
    long* memoireAllouee = NULL;

    memoireAllouee = malloc(sizeof(long));
    if (memoireAllouee == NULL) // On vérifie si la mémoire a été allouée
    {
        exit(0); // Erreur : on arrête tout !
    }

    // On peut utiliser ici la mémoire

    free(memoireAllouee); // On n'a plus besoin de la mémoire, on la libère

    return 0;
}
```

Exemple concret d'utilisation

On va faire quelque chose qu'on a appris à faire il y a longtemps : on va demander l'âge de l'utilisateur et on va le lui afficher.
La seule différence avec ce qu'on faisait avant, c'est qu'ici la variable va être allouée *manuellement* (on dit aussi *dynamiquement*) au lieu d'automatiquement comme auparavant. Alors oui, du coup, le code est un peu plus compliqué. Mais faites l'effort de bien essayer de le comprendre, c'est important :

Code : C

```
int main(int argc, char *argv[])
{
    long* memoireAllouee = NULL;

    memoireAllouee = malloc(sizeof(long)); // Allocation de la mémoire
    if (memoireAllouee == NULL)
    {
        exit(0);
    }

    // Utilisation de la mémoire
    printf("Quel age avez-vous ? ");
    scanf("%ld", memoireAllouee);
    printf("Vous avez %ld ans\n", *memoireAllouee);

    free(memoireAllouee); // Libération de mémoire

    return 0;
}
```

Code : Console

```
Quel age avez-vous ? 31
Vous avez 31 ans
```

Attention : comme memoireAllouee est un pointeur, on ne l'utilise pas de la même manière qu'une vraie variable. Pour obtenir la valeur de la variable, il faut mettre une étoile devant : "**memoireAllouee*" (regardez le printf). Tandis que pour indiquer l'adresse, on a juste besoin d'écrire le nom du pointeur "*memoireAllouee*" (regardez le scanf)
Tout cela a été expliqué dans le chapitre sur les pointeurs. Toutefois, on met généralement du temps à s'y faire, et il est probable que vous confondiez encore. Si c'est votre cas, vous DEVEZ relire le chapitre sur les pointeurs, qui est fondamental.

Revenons à notre code. On y a alloué dynamiquement une variable de type long.
Au final, ce qu'on a écrit revient exactement au même que d'utiliser la méthode "automatique" qu'on connaît bien maintenant :

Code : C

```
int main(int argc, char *argv[])
{
    long maVariable = 0; // Allocation de la mémoire (automatique)

    // Utilisation de la mémoire
    printf("Quel age avez-vous ? ");
    scanf("%ld", &maVariable);
    printf("Vous avez %ld ans\n", maVariable);

    return 0;
} // Libération de la mémoire (automatique à la fin de la fonction)
```

Code : Console

```
Quel age avez-vous ? 31
Vous avez 31 ans
```

En résumé : il y a 2 façons de créer une variable, c'est-à-dire d'allouer de la mémoire. Soit on le fait :

- *Automatiquement* : c'est la méthode que vous connaissez et qu'on a utilisée jusqu'ici.
- *Manuellement* (= *dynamiquement*) : c'est la méthode que je vous enseigne dans ce chapitre.

Un peu plus compliquée... certes.
Mais inutile, non ! On est parfois obligé d'allouer manuellement de la mémoire, comme on va le voir maintenant

ALLOCATION DYNAMIQUE D'UN TABLEAU

Pour le moment, on s'est servis de l'allocation dynamique uniquement pour créer une petite variable. Or en général, on ne se sert pas de l'allocation dynamique pour ça On utilise la méthode automatique qui est plus simple.

Quand a-t-on besoin de l'allocation dynamique me direz-vous ?
Le plus souvent, on se sert de l'allocation dynamique pour créer un tableau dont on ne connaît pas la taille avant l'exécution du programme.

Imaginons par exemple un programme qui stocke l'âge de tous les amis de l'utilisateur dans un tableau. Vous pourriez créer un tableau de long pour stocker les âges, comme ceci :

Code : C

```
long ageAmis[15];
```

Mais qui vous dit que l'utilisateur a 15 amis ? Peut-être qu'il en a plus que ça !
Lorsque vous rédigez le code source, vous ne connaissez pas la taille que vous devez donner à votre tableau. Vous ne le saurez qu'à l'exécution, lorsque vous demanderez à l'utilisateur combien il a d'amis.
L'intérêt de l'allocation dynamique est là : on va demander le nombre d'amis à l'utilisateur, puis on fera une allocation dynamique pour créer un tableau ayant exactement la taille nécessaire (ni trop petit, ni trop grand).
Si l'utilisateur a 15 amis, on créera un tableau de 15 long, s'il en a 28 on créera un tableau de 28 long etc.

Comme je vous l'ai appris, il est interdit en C de créer un tableau en indiquant sa taille à l'aide d'une variable :

Code : C

```
long amis[nombreDAmis];
```

(Notez : ce code marche peut-être sur certains compilateurs mais uniquement dans des cas précis, il est recommandé de ne pas l'utiliser !)
L'avantage de l'allocation dynamique, c'est qu'elle nous permet de créer un tableau qui a exactement la taille de la variable nombreDAmis, et cela grâce à un code qui marchera partout !

On va demander au malloc de nous réserver nombreDAmis * sizeof(long) octets en mémoire :

Code : C

```
amis = malloc(nombreDAmis * sizeof(long));
```

Ce code permet de créer un tableau de type long qui a une taille correspondant exactement au nombre de ses amis !

Voici ce que va faire le programme dans l'ordre :

1. On demande à l'utilisateur combien il a d'amis
2. On crée un tableau de long faisant une taille égale à son nombre d'amis (via malloc)
3. On demande l'âge de chacun de ses amis un à un, qu'on stocke dans le tableau
4. On affiche l'âge des amis pour montrer qu'on a bien mémorisé tout cela
5. A la fin, on n'a plus besoin du tableau contenant l'âge des amis : on le libère avec la fonction free.

Code : C

```
int main(int argc, char *argv[])
{
    long nombreDAmis = 0, i = 0;
    long* ageAmis = NULL; // Ce pointeur va servir de tableau après l'appel du malloc

    // On demande le nombre d'amis à l'utilisateur
    printf("Combien d'amis avez-vous ? ");
    scanf("%ld", &nombreDAmis);

    if (nombreDAmis > 0) // Il faut qu'il ait au moins un ami (je le plains un peu sinon
:p)
    {
        ageAmis = malloc(nombreDAmis * sizeof(long)); // On alloue de la mémoire pour le
tableau
        if (ageAmis == NULL) // On vérifie si l'allocation a marché ou pas
        {
            exit(0); // On arrête tout
        }

        // On demande l'âge des amis un à un
        for (i = 0 ; i < nombreDAmis ; i++)
        {
            printf("Quel age a l'ami numero %ld ? ", i + 1);
            scanf("%ld", &ageAmis[i]);
        }

        // On affiche les âges stockés un à un
        printf("\n\nVos amis ont les ages suivants :\n");
        for (i = 0 ; i < nombreDAmis ; i++)
        {
            printf("%ld ans\n", ageAmis[i]);
        }

        // On libère la mémoire allouée avec malloc, on n'en a plus besoin
        free(ageAmis);
    }

    return 0;
}
```

Code : Console

```
Combien d'amis avez-vous ? 5
Quel age a l'ami numero 1 ? 16
Quel age a l'ami numero 2 ? 18
Quel age a l'ami numero 3 ? 20
Quel age a l'ami numero 4 ? 26
Quel age a l'ami numero 5 ? 27


Vos amis ont les ages suivants :
16 ans
18 ans
20 ans
26 ans
27 ans
```

Ce programme est tout à fait inutile : il demande les âges et les affiche ensuite. J'ai choisi de faire cela car c'est un exemple "simple" (enfin si vous avez compris le malloc).

Que je vous rassure, dans la suite du cours nous aurons l'occasion d'utiliser le malloc pour des choses plus intéressantes que le stockage de l'âge de ses amis Ce chapitre n'était pas très évident je le reconnais, et il a dû être encore moins rigolo pour ceux qui n'avaient pas encore bien assimilé les pointeurs ! D'ailleurs je vous avais prévenu au début du chapitre à ce sujet
Tout cela est encore une preuve qu'il n'y a rien à faire pour combattre les pointeurs, on ne peut pas les éviter quand on programme en C. Il faut donc apprendre à les connaître et les comprendre si on veut vraiment se prétendre programmeur en C. Ca met plus ou moins de temps selon les gens, mais si on est motivé on finit toujours par y arriver

Ce chapitre marque la fin d'une ère (enfin presque). L'allocation dynamique était une des choses les plus difficiles que j'avais à vous expliquer. On a vu la plupart de la théorie du langage C qu'il faut connaître.
Maintenant ce qu'il vous manque c'est de la pratique. Je vais donc tout mettre en oeuvre pour vous faire pratiquer à partir de cet instant.

Justement, le chapitre suivant sera un TP. Prenez-le au sérieux et **prenez le temps qu'il faut pour l'assimiler**. Il va vous demander de faire des efforts car c'est de la pratique pure de tout ce qu'on a appris jusqu'ici (et la pratique, vous le savez bien maintenant, ça n'a rien à voir avec la théorie)

TP : Réalisation d'un pendu

Ah le pendu... Voilà un grand classique des jeux de lettres
Dans ce chapitre, vous allez ~~essayer de~~ réaliser un jeu de pendu en console en langage C.

L'objectif est de vous faire manipuler tout ce que vous avez appris dans la partie II jusqu'ici (et vous en avez appris des choses !). Au menu : pointeurs, chaînes de caractères, fichiers, tableaux... que du bon quoi !

A taaaaable !

LES CONSIGNES

Je veux tout d'abord qu'on se mette bien d'accord sur les règles du pendu à réaliser. Je vais donc vous donner ici les consignes, c'est-à-dire vous expliquer comment doit fonctionner le jeu que vous allez créer.

Tout le monde connaît le pendu n'est-ce pas ? Allez, un petit rappel ne peut pas faire de mal
Le but du pendu est de retrouver un mot caché en moins de 10 coups (mais vous pouvez changer ce nombre maximal pour corser la difficulté bien sûr !).

Déroulement d'une partie

Supposons que le mot caché soit ROUGE.
Vous proposez une lettre à l'ordinateur, par exemple la lettre A. L'ordinateur vérifie si cette lettre se trouve dans le mot caché

Rappelez-vous : il y a une fonction toute prête dans string.h pour rechercher une lettre dans un mot ! Notez que vous n'êtes pas obligés de l'utiliser toutefois (personnellement je ne m'en suis pas servi).

A partir de là, 2 possibilités :

- La lettre se trouve effectivement dans le mot : dans ce cas on dévoile le mot avec les lettres qu'on a déjà trouvées.
- La lettre ne se trouve pas dans le mot (c'est le cas ici, car A n'est pas dans ROUGE) : on indique au joueur que la lettre ne s'y trouve pas, et on diminue le nombre de coups restants. Quand il ne nous reste plus de coups (0 coups) le jeu est terminé et on a perdu.

Dans un "vrai" pendu, il y aurait normalement un dessin d'un bonhomme qui se fait pendre au fur et à mesure que l'on fait des erreurs. Bon, comme là on travaille en console, ce serait un peu hardcore de dessiner un bonhomme qui se fait pendre rien qu'avec du texte, donc on va se contenter d'afficher une simple phrase comme "Il vous reste X coups avant une mort certaine".
Les plus courageux d'entre vous essaieront peut-être de dessiner quand même le bonhomme en console à grands coups de printf, mais je vous préviens : ce n'est pas le but du TP

Supposons maintenant que le joueur tape la lettre G. Celle-ci se trouve dans le mot caché, donc on ne diminue pas le nombre de coups restants pour le joueur. On affiche le mot secret avec les lettres qu'on a déjà découvertes, c'est-à-dire qu'on affiche quelque chose comme :

Code : Console

```
Mot secret : ***G*
```

Si ensuite on tape un R, comme la lettre s'y trouve, on la rajoute à la liste des lettres trouvées et on affiche le mot avec les lettres déjà découvertes :

Code : Console

```
Mot secret : R**G*
```

Le cas des lettres multiples

Dans certains mots, une même lettre peut apparaître 2, 3 fois, voire même plus !
Par exemple, il y a 2 Z dans PUZZLE
De même, il y a 3 E dans ELEMENT

Que fait dans un cas comme ça ? Les règles du pendu sont claires : si le joueur tape la lettre E, toutes les lettres E du mot ELEMENT doivent être découvertes d'un seul coup :

Code : Console

```
Mot secret : E*E*E**
```

Il ne faut donc pas taper 3 fois la lettre E pour que tous les E soient découverts. Ca peut paraître évident à certains d'entre vous, mais je préfère le dire avant on sait jamais

Exemple d'une partie complète

Voici à quoi devrait ressembler une partie complète en console de votre programme lorsqu'il sera terminé :

Code : Console

```
Bienvenue dans le Pendu !



Il vous reste 10 coups a jouer
Quel est le mot secret ? ******
Proposez une lettre : E


Il vous reste 9 coups a jouer
Quel est le mot secret ? ******
Proposez une lettre : A


Il vous reste 9 coups a jouer
Quel est le mot secret ? *A****
Proposez une lettre : O


Il vous reste 9 coups a jouer
Quel est le mot secret ? *A**O*
Proposez une lettre : P


Il vous reste 8 coups a jouer
Quel est le mot secret ? *A**O*
Proposez une lettre : M


Il vous reste 8 coups a jouer
Quel est le mot secret ? MA**O*
Proposez une lettre : N


Il vous reste 8 coups a jouer
Quel est le mot secret ? MA**ON
Proposez une lettre : R


Gagne ! Le mot secret etait bien : MARRON
```

Saisie d'une lettre en console

La lecture d'une lettre dans la console est plus compliquée qu'il n'y paraît.
Intuitivement, pour récupérer un caractère, vous devriez avoir pensé à :

Code : C

```c
scanf("%c", &maLettre);
```

Et effectivement, c'est bien. %c indique que l'on attend un caractère, qu'on stockera dans maLettre (une variable de type char).

Tout se passe très bien... tant qu'on ne refait pas un scanf. En effet, vous pouvez tester le code suivant :

Code : C

```
int main(int argc, char* argv[])
{
    char maLettre = 0;

    scanf("%c", &maLettre);
    printf("%c", maLettre);

    scanf("%c", &maLettre);
    printf("%c", maLettre);

    return 0;
}
```

Normalement, ce code est censé vous demander une lettre et vous l'afficher, et cela 2 fois.
Testez. Que se passe-t-il ? Vous rentrez une lettre d'accord mais... le programme s'arrête de suite après, il ne vous demande pas la seconde lettre ! On dirait qu'il ignore le second scanf.

Que s'est-il passé ?

En fait, quand vous rentrez du texte en console, tout ce que vous tapez est stocké quelque part en mémoire, y compris l'appui sur la touche Entrée (\n).

Ainsi, la première fois que vous rentrez une lettre (par exemple A) puis que vous tapez Entrée, c'est la lettre A qui est renvoyée par le scanf. Mais la seconde fois, scanf renvoie le \n correspondant à la touche "Entrée" que vous aviez tapée auparavant !

Oulah, comment éviter cela ?

Le mieux, c'est de créer notre propre petite fonction lireCaractere() :

Code : C

```
char lireCaractere()
{
    char caractere = 0;

    caractere = getchar(); // On lit le premier caractère
    caractere = toupper(caractere); // On met la lettre en majuscule si elle ne l'est pas
déjà

    // On lit les autres caractères mémorisés un à un jusqu'à l'\n (pour les effacer)
    while (getchar() != '\n') ;

    return caractere; // On retourne le premier caractère qu'on a lu
}
```

Cette fonction utilise getchar() qui est une fonction de stdio identique à `scanf("%c", &lettre);`. La fonction getchar renvoie le caractère que le joueur a tapé.
Après, j'utilise une fonction standard qu'on n'a pas eu l'occasion d'étudier dans le cours : *toupper()*. Cette fonction transforme la lettre indiquée en majuscule. Comme ça, le jeu fonctionnera même si le joueur tape des lettres

minuscules. **Il faut inclure ctype.h pour pouvoir utiliser cette fonction** (ne l'oubliez pas !)

Ensuite vient la partie la plus intéressante : celle où je vide les autres caractères qui auraient pu avoir été tapés. En effet, en refaisant un getchar on prend le caractère suivant que l'utilisateur a tapé (par exemple l'Entrée \n).
Ce que je fais est simple et tient en une ligne : j'appelle la fonction getchar en boucle jusqu'à tomber sur le caractère \n. La boucle s'arrête dès qu'on tombe sur l'\n, ce qui signifie qu'on a "lu" tous les autres caractères de la mémoire, ils ont donc été vidés de la mémoire. **On dit qu'on vide le buffer**.

Euh pourquoi il y a un point-virgule à la fin du while et pourquoi il n'y a pas d'accolades ?

En fait, je fait une boucle qui ne contient pas d'instructions (la seule instruction c'est le getchar entre les parenthèses). Les accolades ne sont pas nécessaires vu que je n'ai rien d'autre à faire qu'un getchar. Je mets donc un point-virgule pour remplacer les accolades. Ce point-virgule signifie "*ne rien faire à chaque passage dans la boucle*". C'est un peu particulier je le reconnais, mais c'est une technique à connaître qu'utilisent les programmeurs pour faire des boucles très courtes et très simples.

Pour mieux comprendre, dites vous que le while aurait aussi pu être écrit comme ceci :

Code : C
```c
while (getchar() != '\n')
{
}
```

Il n'y a rien entre accolades c'est volontaire, vu qu'on n'a rien d'autre à faire. Ma technique de placer juste un point-virgule est plus courte que celle avec des accolades, c'est tout

Enfin, la fonction lireCaractere retourne le premier caractère qu'elle a lu (la variable *caractere*).

En résumé : pour récupérer une lettre dans votre code, vous n'utiliserez pas :

Code : C
```c
scanf("%c", &maLettre);
```

Mais vous utiliserez à la place notre super fonction :

Code : C
```c
maLettre = lireCaractere();
```

Dictionnaire de mots

Dans un premier temps pour vos tests, je vais vous demander de fixer le mot secret directement dans votre code. Vous écrirez donc par exemple :

Code : C
```c
char motSecret[] = "MARRON";
```

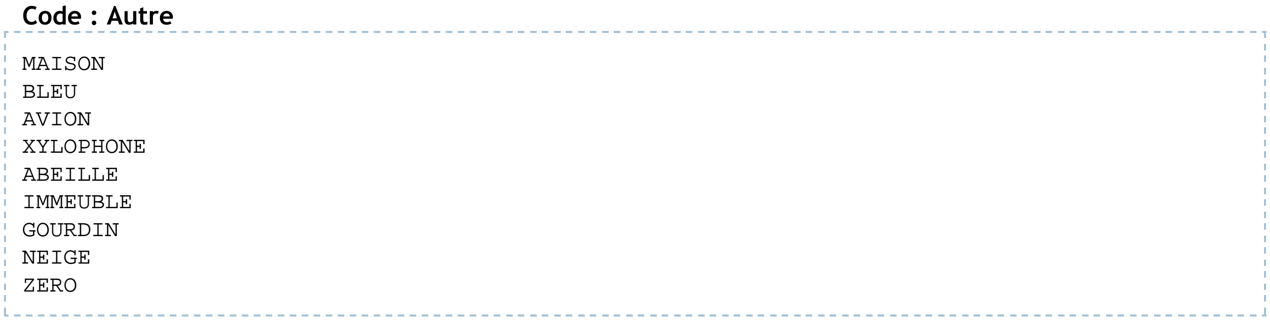
Alors oui, bien sûr le mot secret sera toujours le même si on laisse ça comme ça, ce qui n'est pas très rigolo
Je vous demande de faire comme ça dans un premier temps pour ne pas mélanger les problèmes. En effet, une fois que votre jeu de pendu fonctionnera correctement (et seulement à partir de ce moment-là) vous attaquerez la 2ème

phase : la création du dictionnaire de mots.

Qu'est-ce que c'est le "dictionnaire de mots" ?

C'est un fichier qui contiendra plein de mots pour votre jeu de pendu. Il doit y avoir un mot par ligne. Exemple :

Code : Autre



```
MAISON
BLEU
AVION
XYLOPHONE
ABEILLE
IMMEUBLE
GOURDIN
NEIGE
ZERO
```

A chaque nouvelle partie, votre programme devra ouvrir ce fichier et prendre un des mots au hasard dans la liste.
Grâce à cette technique, vous aurez un fichier à part que vous pourrez éditer tant que vous voudrez pour ajouter des mots secrets possibles pour le pendu

Vous aurez remarqué que depuis le début je fais exprès de mettre tous mes mots en majuscule. En effet, dans le pendu on ne fait pas la distinction entre les majuscules et les minuscules, donc le mieux est de se dire dès le début : "tous mes mots seront en majuscules". A vous de prévenir le joueur (dans le mode d'emploi du jeu par exemple) qu'il est censé rentrer des lettres majuscules et non des minuscules
Par ailleurs, on fait exprès d'ignorer les accents pour simplifier (si on doit commencer à tester le é, le è, le ê, le ë... on n'a pas fini). Vous devrez donc écrire vos mots dans le dictionnaire **entièrement en majuscules et sans accents.**

Le problème qui se posera rapidement pour vous sera de savoir combien il y a de mots dans le dictionnaire. En effet, si vous voulez choisir un mot au hasard, il faudra tirer au sort un nombre entre 0 et X, et vous ne savez pas combien de mots contient votre fichier à priori.
Pour résoudre le problème, 2 solutions :

- Soit vous indiquez sur la première ligne du fichier le nombre de mots qu'il contient :
  Code : Autre

```
3
MAISON
BLEU
AVION
```

- Mais cette technique est ennuyeuse car il faudra recompter manuellement le nombre de mots à chaque fois que vous rajoutez un mot. Aussi je vous propose plutôt de compter automatiquement le nombre de mots en lisant une première fois le fichier avec votre programme. Pour savoir combien il y a de mots, c'est simple : vous comptez le nombre d'\n (retours à la ligne) dans le fichier
  Une fois que vous aurez lu le fichier une première fois pour compter les \n, vous ferez un rewind pour revenir au début. Vous n'aurez alors plus qu'à tirer un nombre au sort parmi le nombre de mots que vous avez compté, puis à vous rendre au mot que vous avez choisi et à le stocker dans une chaîne en mémoire.

Je vous laisse réfléchir un peu là-dessus, je vais pas trop vous aider quand même sinon ça sera plus un TP
Sachez que vous avez acquis toutes les connaissances qu'il faut dans les chapitres précédents, donc vous êtes parfaitement capables de réaliser ce jeu. Ca va prendre plus ou moins de temps et c'est moins facile qu'il n'y paraît,

mais en vous organisant correctement (et en créant suffisamment de fonctions) vous y arriverez

Bon courage, et surtout : per-sé-vé-rez !

LA SOLUTION (1 : LE CODE DU JEU)

Si vous lisez ces lignes, c'est soit que vous avez terminé le programme, soit... que vous n'arrivez pas à le terminer

J'ai personnellement mis plus de temps que je ne le pensais pour réaliser ce petit jeu apparemment tout bête. C'est souvent comme ça : on se dit "boah c'est facile" alors qu'en fait il y a plein de cas à gérer
Je persiste toutefois à dire que **vous êtes tous capables de le faire**. Il vous faudra plus ou moins de temps (quelques minutes, quelques heures, quelques jours ?), mais ça n'a jamais été une course. Je préfère que vous y passiez beaucoup de temps et que vous y arriviez, plutôt que vous n'essayiez que 5 minutes et que vous regardiez la solution.

> Rappelez-vous que c'est précisément dans les TP que le gros du travail se fait : la pratique, c'est vraiment pas la même chose que la théorie
> C'est donc principalement pendant les TP que vous progressez, raison de plus pour y mettre tous vos efforts

N'allez pas croire que j'ai écrit le programme d'une traite. Moi aussi, comme vous, j'y suis allé pas à pas. J'ai commencé par faire quelque chose de très simple, puis petit à petit j'ai amélioré le code pour finalement arriver au résultat final
J'ai fait plusieurs erreurs en codant : j'ai oublié à un moment d'initialiser une variable correctement, j'ai oublié de mettre un prototype de fonction ou encore de supprimer une variable qui ne servait plus dans mon code. J'ai même, je l'avoue, oublié un bête point-virgule à un moment à la fin d'une ligne.

Tout ça pour dire quoi ? Que je ne suis pas infaillible et que je vis à peu près les mêmes frustrations que vous : "ESPECE DE PROGRAMME DE ***** TU VAS TE METTRE A MARCHER OUI OU NON !?".

Je vais vous présenter la solution en 2 temps :

1. D'abord je vais vous montrer comment j'ai fait le code du jeu lui-même, en fixant le mot caché directement dans le code (j'ai pris le mot MARRON car il me permet de tester si je gère bien les lettres en double comme R ici).
2. Ensuite, je vous montrerai comment dans un second temps j'ai ajouté la gestion du dictionnaire de mots pour tirer au sort un mot secret pour le joueur.

Bien sûr, je pourrais vous montrer tout le code d'un coup mais... ça ferait beaucoup à la fois, et nombre d'entre vous n'auraient pas le courage de se pencher dans le code (je suis comme vous, quand y'a un gros code à comprendre je mets plus de temps que pour un petit code)

Je vais essayer de vous expliquer pas à pas mon raisonnement. Ce qui compte, ce n'est pas le résultat, mais la façon dont on réfléchit.

Analyse de la fonction main

Comme tout le monde le sait, tout commence par un main. On n'oublie pas d'inclure les librairies stdio, stdlib et ctype (pour la fonction toupper()) dont on aura besoin :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char* argv[])
{

    return 0;
}
```

Ok, jusque là tout le monde devrait suivre
Notre main va gérer la plupart du jeu et faire appel à 2-3 de nos fonctions quand il en aura besoin.

Commençons par déclarer les variables dont on va avoir besoin. Rassurez-vous, je n'ai pas pensé de suite à toutes ces variables, il y en avait un peu moins la première fois que j'ai écrit le code

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char* argv[])
{
    char lettre = 0; // Stocke la lettre proposée par l'utilisateur (retour du scanf)
    char motSecret[] = "MARRON"; // C'est le mot à trouver
    int lettreTrouvee[6] = {0}; // Un tableau de booléens. Chaque case correspond à une
lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée
    long coupsRestants = 10; // Compteur de coups restants (0 = mort)
    long i = 0; // Une petite variable pour parcourir les tableaux

    return 0;
}
```

J'ai fait exprès de mettre une déclaration de variable par ligne ainsi que pas mal de commentaires pour que vous compreniez. En pratique, vous n'aurez pas forcément besoin de mettre tous ces commentaires, et vous pourrez grouper plusieurs déclarations de variables sur la même ligne.

Je pense que la plupart de ces variables semblent logiques : la variable lettre stocke la lettre que l'utilisateur tape à chaque fois, le motSecret c'est le mot à trouver, coupsRestants le nombre de coups etc.
La variable i est une petite variable que j'utilise pour parcourir mes tableaux avec des for. Elle n'est donc pas extrêmement importante mais elle est nécessaire si on veut faire nos boucles.

Enfin, la variable à laquelle il fallait penser, celle qui fait la différence, c'est mon tableau de booléens *lettreTrouvee*. Vous remarquerez que je lui ai donné pour taille le nombre de lettres du mot secret (6). Ce n'est pas un hasard : chaque case de ce tableau de booléens représente une lettre du mot secret. Ainsi, la première case représente la première lettre, la seconde case représente la seconde lettre etc.
Les cases du tableau sont au départ initialisées à 0, ce qui signifie "Lettre non trouvée". Au fur et à mesure de l'avancement du jeu, ce tableau sera modifié. Pour chaque lettre du mot secret trouvée, la case correspondante du tableau *lettreTrouvee* sera mise à 1.

Par exemple, si à un moment du jeu j'ai l'affichage suivant : M*RR*N, c'est que mon tableau d'int a les valeurs : 101101 (1 pour chaque lettre qui a été trouvée).
Il est ainsi facile de savoir quand on a gagné : il suffit de vérifier si le tableau de booléens ne contient que des 1.
En revanche, on a perdu si le compteur coupsRestants tombe à 0.

Passons à la suite :

Code : C

```c
printf("Bienvenue dans le Pendu !\n\n");
```

Bon, ça c'est un message de bienvenue, j'ai rien à ajouter

Ensuite, on commence la boucle principale du jeu :

Code : C

```
while (coupsRestants > 0 && !gagne(lettreTrouvee))
{
```

Le jeu continue tant qu'il reste des coups (coupsRestants > 0) et tant qu'on n'a pas gagné.
Si on n'a plus de coups à jouer, c'est qu'on a perdu. Si on a gagné, c'est... qu'on a gagné

"gagne" est une fonction qui analyse le tableau lettreTrouvee. Elle renvoie vrai (1) si le joueur a gagné (le tableau lettreTrouvee ne contient que des 1), faux (0) si le joueur n'a pas encore gagné.
Je ne vous explique pas le fonctionnement de cette fonction en détail pour le moment. On verra cela plus tard. Pour le moment, vous avez juste besoin de savoir ce que fait la fonction.

La suite :

Code : C

```
printf("\n\nIl vous reste %ld coups a jouer", coupsRestants);
printf("\nQuel est le mot secret ? ");

/* On affiche le mot secret en masquant les lettres non trouvees
        Exemple : *A**ON */
        for (i = 0 ; i < 6 ; i++)
        {
            if (lettreTrouvee[i]) // Si on a trouvé la lettre n°i
                printf("%c", motSecret[i]); // On l'affiche
            else
                printf("*"); // Sinon, on affiche une étoile pour les lettres non
trouvées
        }
```

On affiche à chaque coup le nombre de coups restants ainsi que le mot secret (masqué par des * pour les lettres non trouvées).
L'affichage du mot secret masqué par des * se fait grâce à une boucle for. On analyse pour chaque lettre si elle a été trouvée (if lettreTrouvee[i]). Si c'est le cas, on affiche la lettre. Sinon, on affiche une * de remplacement pour masquer la lettre.

Maintenant qu'on a affiché ce qu'il fallait, on va demander au joueur de saisir une lettre :

Code : C

```
printf("\nProposez une lettre : ");
lettre = lireCaractere();
```

Je fais appel à notre fonction lireCaractere(). Celle-ci lit le premier caractère tapé, le met en majuscule et vide le buffer (c'est-à-dire qu'elle vide les autres caractères qui auraient pu être restés dans la mémoire).

Code : C

```
        // Si ce n'était PAS la bonne lettre
        if (!rechercheLettre(lettre, motSecret, lettreTrouvee))
        {
            coupsRestants--; // On enlève un coup au joueur
        }
    }
```

On teste si la lettre entrée se trouve dans motSecret. On fait appel pour cela à une fonction maison appelée rechercheLettre. Nous verrons peu après le code de cette fonction.
Pour le moment, tout ce que vous avez besoin de savoir, c'est que cette fonction renvoie "vrai" si la lettre se trouve dans le mot, "faux" si elle ne s'y trouve pas.

Mon if, vous l'aurez remarqué, commence par un point d'exclamation "!" qui signifie "non". La condition se lit donc "Si la lettre n'a pas été trouvée".
Que fait-on si la lettre n'a pas été trouvée ? On diminue le nombre de coups restants.

Notez que la fonction rechercheLettre met aussi à jour le tableau de booléens lettreTrouvee. Elle met des 1 dans les cases des lettres qui ont été trouvées.

La boucle principale du jeu s'arrête là. On recommence donc au début de la boucle et on vérifie s'il reste des coups à jouer et si on n'a pas déjà gagné.

Lorsqu'on sort de la boucle principale du jeu, il reste à afficher si on a gagné ou pas avant que le programme ne s'arrête :

Code : C

```
    if (gagne(lettreTrouvee))
        printf("\n\nGagne ! Le mot secret etait bien : %s", motSecret);
    else
        printf("\n\nPerdu ! Le mot secret etait : %s", motSecret);

        return 0;
}
```

On fait appel à la fonction "gagne" pour vérifier si on a gagné. Si c'est le cas, alors on affiche le message "Gagné !", sinon c'est qu'on n'avait plus de coups à jouer, on a été pendu.

Analyse de la fonction gagne

Voyons voir maintenant le code de la fonction gagne :

Code : C

```
int gagne(int lettreTrouvee[])
{
    long i = 0;
    int joueurGagne = 1;

    for (i = 0 ; i < 6 ; i++)
    {
        if (lettreTrouvee[i] == 0)
            joueurGagne = 0;
    }

    return joueurGagne;
}
```

Cette fonction prend le tableau de booléens lettreTrouvee pour paramètre. Elle renvoie un booléen : vrai si on a gagné, faux si on a perdu.

Le code de cette fonction est plutôt simple, vous devriez tous le comprendre. On parcourt lettreTrouvee et on vérifie si UNE des cases vaut faux (0). Si une des lettres n'a pas encore été trouvée, c'est qu'on a perdu : on met alors le booléen joueurGagne à faux (0). Sinon, si toutes les lettres ont été trouvée, le booléen vaut vrai (1) et la fonction

renverra donc vrai.

Analyse de la fonction rechercheLettre

La fonction rechercheLettre a 2 missions :

- Renvoyer un booléen indiquant si la lettre se trouvait bien dans le mot secret
- Mettre à jour les cases du tableau lettreTrouvee correspondant aux positions de la lettre qui a été trouvée à 1.

Code : C

```
int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[])
{
    long i = 0;
    int bonneLettre = 0;

    // On parcourt motSecret pour vérifier si la lettre proposée y est
    for (i = 0 ; motSecret[i] != '\0' ; i++)
    {
        if (lettre == motSecret[i]) // Si la lettre y est
        {
            bonneLettre = 1; // On mémorise que c'était une bonne lettre
            lettreTrouvee[i] = 1; // On met à 1 le case du tableau de booléens
correspondant à la lettre actuelle
        }
    }

    return bonneLettre;
}
```

On parcourt donc la chaîne motSecret caractère par caractère. A chaque fois, on vérifie si la lettre que le joueur a proposée est une lettre du mot. Si la lettre correspond, alors on fait 2 choses :

- On change la valeur du booléen bonneLettre à 1, pour que la fonction retourne 1 car la lettre se trouvait effectivement dans motSecret.
- On met à jour le tableau lettreTrouvee à la position actuelle pour indiquer que cette lettre a été trouvée.

L'avantage de cette technique c'est qu'ainsi on parcourt tout le tableau (on ne s'arrête pas à la première lettre trouvée). Cela nous permet de bien mettre à jour le tableau lettreTrouvee, au cas où une lettre serait présente en plusieurs exemplaires dans le mot secret (comme c'est le cas pour les 2 R dans MARRON).

Pfiou !

Et voilà, on a fait le tour
Je résume le code source que j'utilise :

Code : C

```
/*
Jeu du pendu
Par M@teo21, pour le Site du Zér0
www.siteduzero.com

main.c
------

Fonctions principales de gestion du jeu
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int gagne(int lettreTrouvee[]);
int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[]);
char lireCaractere();

int main(int argc, char* argv[])
{
    char lettre = 0; // Stocke la lettre proposée par l'utilisateur (retour du scanf)
    char motSecret[] = "MARRON"; // C'est le mot à trouver
    int lettreTrouvee[6] = {0}; // Un tableau de booléens. Chaque case correspond à une
lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée
    long coupsRestants = 10; // Compteur de coups restants (0 = mort)
    long i = 0; // Une petite variable pour parcourir les tableaux

    printf("Bienvenue dans le Pendu !\n\n");

    // On continue à jouer tant qu'il reste au moins un coup à jouer ou qu'on
    // n'a pas gagné
    while (coupsRestants > 0 && !gagne(lettreTrouvee))
    {
        printf("\n\nIl vous reste %ld coups a jouer", coupsRestants);
        printf("\nQuel est le mot secret ? ");

        /* On affiche le mot secret en masquant les lettres non trouvées
        Exemple : *A**ON */
        for (i = 0 ; i < 6 ; i++)
        {
            if (lettreTrouvee[i]) // Si on a trouvé la lettre n°i
                printf("%c", motSecret[i]); // On l'affiche
            else
                printf("*"); // Sinon, on affiche une étoile pour les lettres non
trouvées
        }

        printf("\nProposez une lettre : ");
        lettre = lireCaractere();

        // Si ce n'était PAS la bonne lettre
        if (!rechercheLettre(lettre, motSecret, lettreTrouvee))
        {
            coupsRestants--; // On enlève un coup au joueur
        }
    }


    if (gagne(lettreTrouvee))
        printf("\n\nGagne ! Le mot secret etait bien : %s", motSecret);
    else
        printf("\n\nPerdu ! Le mot secret etait : %s", motSecret);

        return 0;
}

char lireCaractere()
{
    char caractere = 0;

    caractere = getchar(); // On lit le premier caractère
    caractere = toupper(caractere); // On met la lettre en majuscule si elle ne l'est pas
déjà

    // On lit les autres caractères mémorisés un à un jusqu'à l'\n
    while (getchar() != '\n') ;
```

Vous noterez que j'ai mis pour le moment les prototypes en haut du fichier, au lieu de les mettre dans un .h. C'est simplement parce que pour le moment le programme n'est pas encore bien compliqué et ne nécessite pas de créer plus que le simple fichier main.c de base.
Cependant, les choses vont changer dès qu'on attaquera la phase 2 avec la gestion du dictionnaire

Il y a plein de façons différentes d'imaginer le code du jeu du pendu, aussi si vous y étiez arrivés mais avec un code complètement différent, sachez que c'est normal. Je ne dis pas que mon code est le meilleur, je dis juste que c'était une bonne façon de faire.

Comment savoir si vous avez bien fait ?
C'est simple :

- Si votre code ne fait pas 2000 lignes, déjà, c'est bien. Généralement, un bon code est court et concis.
- Si votre programme met 5 minutes à vérifier si la lettre tapée se trouve dans le mot secret, c'est généralement qu'il y a un problème et que votre code n'est pas assez optimisé. Rassurez-vous, sur un jeu du pendu cela ne devrait pas arriver, mais à l'avenir il faudra y penser.
- Si votre programme prend 450 Mo de mémoire vive, il faut commencer à vous inquiéter et vérifier si vous ne prenez pas un peu trop de mémoire. Là encore, sur un petit programme comme ça c'est difficile à faire (à moins de faire une boucle infinie de malloc)

La meilleure façon d'obtenir des commentaires c'est bien souvent de proposer votre code sur les forums du Site du Zér0 pour que des gens plus expérimentés puissent vous donner leur avis. N'hésitez donc pas à le faire, on apprend généralement des tas de choses comme ça

La solution (2 : la gestion du dictionnaire)

Pour qu'on puisse faire des tests, la première chose à faire c'est de créer ce fameux dictionnaire de mots. Même s'il est court, c'est pas grave c'est pour les tests.

Je vais donc créer un fichier dico.txt dans le même répertoire que mon projet. Pour le moment, je mets les mots suivants :

Code : Autre

```
MAISON
BLEU
AVION
XYLOPHONE
ABEILLE
IMMEUBLE
GOURDIN
NEIGE
ZERO
```

Une fois que j'aurai terminé de coder le programme, je reviendrai bien sûr sur ce dictionnaire et j'y ajouterai des tooonnes de mots tordus à rallonge comme ANTICONSTITUTIONNELLEMENT

Mais pour le moment, retournons à nos instructions.

Préparation des nouveaux fichiers

La lecture du dico va demander pas mal de lignes de codes (du moins je le pressens). Je prends donc les devants en ajoutant **un nouveau fichier à mon projet** : dico.c (qui sera chargé de la lecture du dico). Dans la foulée, je crée le dico.h qui contiendra les prototypes des fonctions de dico.h.

Dans dico.c, je commence par inclure les librairies dont j'aurai besoin ainsi que mon dico.h
A priori, comme souvent, j'aurai besoin de stdio et stdlib ici. En plus de cela, je vais être amené à piocher un nombre

au hasard dans le dico, donc je vais inclure time.h comme on l'avait fait pour notre premier projet "Plus ou Moins"

Je vais aussi avoir besoin de string.h pour faire un strlen vers la fin de la fonction :

Code : C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "dico.h"
```

La fonction piocherMot

Maintenant, attaquons la fonction qui va être chargée de piocher un mot au hasard dans le dictionnaire.

Cette fonction va prendre un paramètre : un pointeur sur la zone en mémoire où elle pourra écrire le mot. Ce pointeur sera fourni par le main().
La fonction renverra un int qui sera un booléen : 1 = tout s'est bien passé, 0 = il y a eu une erreur.

Voici le début de la fonction :

Code : C

```
int piocherMot(char *motPioche)
{
    FILE* dico = NULL; // Le pointeur de fichier qui va contenir notre fichier
    long nombreMots = 0, numMotChoisi = 0, i = 0;
    int caractereLu = 0;
```

Je définis quelques variables dont je vais avoir besoin. Comme pour le main(), je n'ai pas pensé à mettre toutes ces variables dès le début, il y en a certaines que j'ai rajoutées par la suite lorsque je me suis rendu compte que j'en avais besoin

Grosso modo, les noms des variables parlent d'eux-mêmes. On a notre pointeur sur fichier dico dont on va se servir pour lire le fichier dico.txt, des variables temporaires qui vont stocker les caractères etc.
Notez que j'utilise ici un int pour stocker un caractère (caractereLu) car la fonction fgetc que je vais utiliser renvoie un int. Il est donc préférable de stocker le résultat dans un int.

Passons à la suite :

Code : C

```
dico = fopen("dico.txt", "r"); // On ouvre le dictionnaire en lecture seule

// On vérifie si on a réussi à ouvrir le dictionnaire
if (dico == NULL) // Si on n'a PAS réussi à ouvrir le fichier
{
    printf("\nImpossible de charger le dictionnaire de mots");
    return 0; // On retourne 0 pour indiquer que la fonction a échoué
    // A la lecture du return, la fonction s'arrête immédiatement.
}
```

Je n'ai pas grand chose à rajouter là. J'ouvre le fichier dico.txt en lecture seule ("r") et je vérifie si j'ai réussi en testant si dico vaut NULL ou pas. Si dico vaut NULL, le fichier n'a pas pu être ouvert (fichier introuvable ou occupé par un autre programme). Dans ce cas j'affiche une erreur et je fais un return 0.

Pourquoi un return là ? En fait, l'instruction return commande l'arrêt de la fonction. Si le dico n'a pas pu être ouvert,

la fonction s'arrête là et l'ordinateur n'ira pas lire plus loin. On retourne 0 pour indiquer au main que la fonction a échoué.

Dans la suite de la fonction, on suppose donc que le fichier a été bien ouvert.

Code : C

```
// On compte le nombre de mots dans le fichier (il suffit de compter les
// entrées \n
do
{
    caractereLu = fgetc(dico);
    if (caractereLu == '\n')
        nombreMots++;
} while(caractereLu != EOF);
```

Là, on parcourt tout le fichier à coups de fgetc (caractère par caractère). On compte le nombre d'\n (entrées) qu'on détecte. A chaque fois qu'on tombe sur un \n, on incrémente la variable nombreMots.
Grâce à ce bout de code, on obtient dans nombreMots le nombre de mots dans le fichier (rappelez-vous que le fichier contient un mot par ligne).

Code : C

```
numMotChoisi = nombreAleatoire(nombreMots); // On pioche un mot au hasard
```

Là, je fais appel à une fonction de mon crû qui va génèrer un nombre aléatoire entre 1 et nombreMots (le paramètre qu'on envoie à la fonction).
C'est une fonction toute simple que j'ai placée aussi dans dico.c (je vous la détaillerai tout à l'heure).

Bref, elle renvoie un numéro au hasard qu'on stocke dans numMotChoisi.

Code : C

```
// On recommence à lire le fichier depuis le début. On s'arrête lorsqu'on est arrivés au
bon mot
rewind(dico);
while (numMotChoisi > 0)
{
    caractereLu = fgetc(dico);
    if (caractereLu == '\n')
        numMotChoisi--;
}
```

Maintenant qu'on a le numéro du mot qu'on veut piocher, on repart au début grâce à un appel à rewind()
On parcourt là encore le fichier caractère par caractère en comptant les \n. Cette fois, on décrémente la variable numMotChoisi. Si par exemple on a choisi le mot numéro 5, à chaque entrée le variable va être décrémentée de 1.
Elle va donc valoir 5, puis 4, 3, 2, 1... et 0.
Lorsque la variable vaut 0, on sort du while (la condition du while `numMotChoisi > 0` n'est plus remplie).

Ce bout de code, que vous devez impérativement comprendre, vous montre donc comment on parcourt un fichier pour se placer à la position voulue. Ce n'est pas bien compliqué, mais ce n'est pas non plus "super évident".
Assurez-vous donc de bien comprendre ce que je fais là.

Maintenant, on devrait avoir un curseur positionné juste devant le mot secret qu'on a choisi de piocher.
On va le stocker dans motPioche (le paramètre que la fonction reçoit) grâce à un simple fgets qui va lire le mot :

Code : C

```
/* Le curseur du fichier est positionné au bon endroit.
On n'a plus qu'à faire un fgets qui lira la ligne */
fgets(motPioche, 100, dico);

// On vire l'\n à la fin
motPioche[strlen(motPioche) - 1] = '\0';
```

On demande au fgets de ne pas lire plus de 100 caractères (c'est la taille du tableau motPioche, qu'on a défini dans le main).
N'oubliez pas que fgets lit toute une ligne, y compris l'\n.
Comme on ne veut pas garder cet \n dans le mot final, on le supprime en le remplaçant par un \0. Cela aura pour effet de couper la chaîne juste avant l'\n.

Et... voilà qui est fait
On a écrit le mot secret dans la mémoire à l'adresse de motPioche.

On n'a plus qu'à fermer le fichier et à retourner 1 pour que la fonction s'arrête et pour dire que tout s'est bien passé :

Code : C

```

    fclose(dico);

    return 1; // Tout s'est bien passé, on retourne 1
}
```

Yahou
C'est tout pour la fonction piocherMot

La fonction nombreAleatoire

C'est la fonction dont j'avais promis de vous parler tout à l'heure. On tire un nombre au hasard et on le renvoie :

Code : C

```
long nombreAleatoire(long nombreMax)
{
    srand(time(NULL));
    return (rand() % nombreMax);
}
```

La première ligne initialise le générateur de nombres aléatoires, comme on a appris à le faire dans le premier TP "Plus ou Moins".
La seconde ligne prend un nombre au hasard entre 0 et nombreMax et renvoie ça (j'ai tout mis dans le return comme un gros bourrin que je suis)

Le fichier dico.h

Il s'agit juste des prototypes des fonctions.
Vous remarquerez qu'il y a la "protection" #ifndef que je vous avais demandé d'inclure dans tous vos fichiers .h (revoyez le chapitre sur le préprocesseur au besoin)

Code : C

```
#ifndef DEF_DICO
#define DEF_DICO


int piocherMot(char *motPioche);
long nombreAleatoire(long nombreMax);

#endif
```

Le fichier dico.c

Voici le fichier dico.c en entier :

Code : C

```
/*
Jeu du pendu
Par M@teo21, pour le Site du Zér0
www.siteduzero.com

dico.c
------

Ces fonctions piochent au hasard un mot dans un fichier dictionnaire
pour le jeu du pendu
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "dico.h"


int piocherMot(char *motPioche)
{
    FILE* dico = NULL; // Le pointeur de fichier qui va contenir notre fichier
    long nombreMots = 0, numMotChoisi = 0;
    int caractereLu = 0;

    dico = fopen("dico.txt", "r"); // On ouvre le dictionnaire en lecture seule

    // On vérifie si on a réussi à ouvrir le dictionnaire
    if (dico == NULL) // Si on n'a PAS réussi à ouvrir le fichier
    {
        printf("\nImpossible de charger le dictionnaire de mots");
        return 0; // On retourne 0 pour indiquer que la fonction a échoué
        // A la lecture du return, la fonction s'arrête immédiatement.
    }

    // On compte le nombre de mots dans le fichier (il suffit de compter les
    // entrées \n). Pensez à laisser une Entrée après le dernier mot du dico !
    do
    {
        caractereLu = fgetc(dico);
        if (caractereLu == '\n')
            nombreMots++;
    } while(caractereLu != EOF);

    numMotChoisi = nombreAleatoire(nombreMots); // On pioche un mot au hasard

    // On recommence à lire le fichier depuis le début. On s'arrête lorsqu'on est arrivés
au bon mot
    rewind(dico);
    while (numMotChoisi > 0)
    {
        caractereLu = fgetc(dico);
        if (caractereLu == '\n')
            numMotChoisi--;
    }

    /* Le curseur du fichier est positionné au bon endroit.
    On n'a plus qu'à faire un fgets qui lira la ligne */
    fgets(motPioche, 100, dico);

    // On vire l'\n à la fin
    motPioche[strlen(motPioche) - 1] = '\0';

    fclose(dico);

    return 1; // Tout s'est bien passé, on retourne 1
}


long nombreAleatoire(long nombreMax)
{
    srand(time(NULL));
    return (rand() % nombreMax);
}
```

Il va falloir modifier le main !

Maintenant que le fichier dico.c est prêt, on retourne dans le main() pour l'adapter un petit peu aux quelques changements qu'on vient de faire.

Déjà, on commence par inclure dico.h si on veut pouvoir faire appel aux fonctions de dico.c
Ensuite, on va inclure string.h en plus là aussi car on va devoir faire un strlen :

Code : C

```
#include <string.h>
#include "dico.h"
```

Pour commencer, les définitions de variables vont un peu changer. Déjà, on n'initialise plus la chaine motSecret, on crée juste un grand tableau de char (100 cases)...

Quant au tableau lettreTrouvee... sa taille dépendra de la longueur du mot secret qu'on aura pioché. Comme on ne connaît pas encore cette taille, on crée un simple pointeur. Tout à l'heure, on fera un malloc et on fera pointer ce pointeur vers la zone mémoire qu'on aura allouée.
Ceci est un exemple parfait où l'allocation dynamique est indispensable : on ne connaît pas la taille du tableau avant la compilation, on est donc obligés de créer un pointeur et de faire un malloc.
Je ne dois pas oublier de libérer la mémoire ensuite quand je n'en ai plus besoin, d'où la présence d'un free() à la fin du main().

On va aussi avoir besoin d'une variable tailleMot qui va stocker... la taille du mot En effet, si vous regardez le main() tel qu'il était dans la première partie, on supposait que le mot faisait 6 caractères partout (et c'était vrai car MARRON fait 6 lettres). Mais maintenant que le mot peut changer de taille, il va falloir être capable de s'adapter à tous les mots

Voici donc les définitions de variables du main en version finale :

Code : C

```
int main(int argc, char* argv[])
{
    char lettre = 0; // Stocke la lettre proposée par l'utilisateur (retour du scanf)
    char motSecret[100] = {0}; // Ce sera le mot à trouver
    int *lettreTrouvee = NULL; // Un tableau de booléens. Chaque case correspond à une
lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée
    long coupsRestants = 10; // Compteur de coups restants (0 = mort)
    long i = 0; // Une petite variable pour parcourir les tableaux
    long tailleMot = 0;
```

C'est principalement le début du main() qui va changer, donc analysons-le de plus près :

Code : C

```
if (!piocherMot(motSecret))
    exit(0);
```

On fait d'abord appel à piocherMot directement dans le if.
piocherMot va placer dans motSecret le mot qu'elle aura pioché.

De plus, piocherMot va renvoyer un booléen pour nous dire si la fonction a réussi ou échoué. Le rôle du if est d'analyser ce booléen. Si ça n'a PAS marché (le "!" permet d'exprimer la négation), alors on arrête tout (*exit(0)*).

Code : C

```
tailleMot = strlen(motSecret);
```

On stocke la taille du motSecret dans tailleMot comme je vous l'ai dit tout à l'heure.

Code : C

```
lettreTrouvee = malloc(tailleMot * sizeof(int)); // On alloue dynamiquement le tableau
lettreTrouvee (dont on ne connaissait pas la taille au départ)
if (lettreTrouvee == NULL)
    exit(0);
```

Maintenant on doit allouer la mémoire pour le tableau lettreTrouvee comme je vous l'ai dit. On lui donne la taille du mot (tailleMot).
On vérifie ensuite si le pointeur n'est pas NULL. Si c'est le cas, c'est que l'allocation a échoué. Dans ce cas, on arrête immédiatement le programme (on fait appel à exit()).

Si les lignes suivantes sont lues, c'est que tout s'est bien passé.

Voilà, grosso modo ce sont tous les préparatifs qu'il vous fallait faire ici.
J'ai dû ensuite modifier le reste du fichier main.c pour remplacer tous les nombres **6** (l'ancienne longueur de MARRON qu'on avait fixée) par la variable tailleMot
Par exemple :

Code : C

```
for (i = 0 ; i < tailleMot ; i++)
    lettreTrouvee[i] = 0;
```

Ce code met toutes les cases du tableau lettreTrouvee à 0 (en s'arrêtant lorsqu'on a parcouru *tailleMot* cases).

J'ai dû aussi modifier le prototype de la fonction gagne pour ajouter la variable tailleMot. Sans cela, la fonction n'aurait pas su quand arrêter sa boucle.

Voici le fichier main.c final en entier :

Code : C

```
/*
Jeu du pendu
Par M@teo21, pour le Site du Zér0
www.siteduzero.com

main.c
------

Fonctions principales de gestion du jeu
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#include "dico.h"

int gagne(int lettreTrouvee[], long tailleMot);
int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[]);
char lireCaractere();

int main(int argc, char* argv[])
{
    char lettre = 0; // Stocke la lettre proposée par l'utilisateur (retour du scanf)
    char motSecret[100] = {0}; // Ce sera le mot à trouver
    int *lettreTrouvee = NULL; // Un tableau de booléens. Chaque case correspond à une
lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée
    long coupsRestants = 10; // Compteur de coups restants (0 = mort)
    long i = 0; // Une petite variable pour parcourir les tableaux
    long tailleMot = 0;

    printf("Bienvenue dans le Pendu !\n\n");

    if (!piocherMot(motSecret))
        exit(0);

    tailleMot = strlen(motSecret);

    lettreTrouvee = malloc(tailleMot * sizeof(int)); // On alloue dynamiquement le
tableau lettreTrouvee (dont on ne connaissait pas la taille au départ)
    if (lettreTrouvee == NULL)
        exit(0);

    for (i = 0 ; i < tailleMot ; i++)
        lettreTrouvee[i] = 0;

    // On continue à jouer tant qu'il reste au moins un coup à jouer ou qu'on
    // n'a pas gagné
    while (coupsRestants > 0 && !gagne(lettreTrouvee, tailleMot))
    {
        printf("\n\nIl vous reste %ld coups a jouer", coupsRestants);
        printf("\nQuel est le mot secret ? ");

        /* On affiche le mot secret en masquant les lettres non trouvées
        Exemple : *A**ON */
        for (i = 0 ; i < tailleMot ; i++)
        {
            if (lettreTrouvee[i]) // Si on a trouvé la lettre n°i
                printf("%c", motSecret[i]); // On l'affiche
            else
                printf("*"); // Sinon, on affiche une étoile pour les lettres non
trouvées
        }

        printf("\nProposez une lettre : ");
        lettre = lireCaractere();

        // Si ce n'était PAS la bonne lettre
        if (!rechercheLettre(lettre, motSecret, lettreTrouvee))
        {
            coupsRestants--; // On enlève un coup au joueur
        }
    }


    if (gagne(lettreTrouvee, tailleMot))
```

IDÉES D'AMÉLIORATION

Télécharger le projet

Pour commencer, je vous invite à télécharger le projet de Pendu que j'ai fait :

Télécharger le projet Pendu (10 Ko)

Si vous êtes sous Linux ou Mac, supprimez le fichier dico.txt et recréez-vous en un. Les fichiers sont enregistrés de manière différente sous Windows, donc si vous utilisez le mien vous risquez d'avoir des bugs.

N'oubliez pas qu'il faut qu'il y ait une Entrée après chaque mot du dictionnaire. Pensez en particulier à mettre une Entrée après le dernier mot de la liste.

Ca va vous permettre de tester par vous-mêmes le fonctionnement du projet, de faire éventuellement des améliorations par-dessus etc etc. Bien entendu, le mieux serait que vous ayez déjà réussi le pendu par vous-mêmes et que vous n'ayez même pas besoin de voir mon projet pour voir comment j'ai fait mais... je suis réaliste, je sais que ce TP a dû être assez délicat pour bon nombre d'entre vous

Vous trouverez dans ce zip les fichiers .c et .h ainsi que le fichier .cbp du projet. C'est un projet fait sous Code::Blocks oui
Si vous utilisez un autre IDE, pas de panique. Vous créez un nouveau projet console et vous y ajoutez manuellement les .c et .h que vous trouverez dans le zip.

Vous trouverez aussi l'exécutable (.exe Windows) ainsi qu'un petit dictionnaire (dico.txt) pour commencer

Améliorez le pendu !

Mine de rien, le pendu est déjà assez évolué comme ça
On a un jeu qui lit un fichier de dictionnaire et qui prend à chaque fois un mot au hasard.

Voici quand même quelques idées d'amélioration qui me passent par la tête :

- Actuellement, on ne vous propose de jouer qu'une fois. Il serait bien de pouvoir reboucler à la fin du main pour **faire une nouvelle partie** si le joueur le désire
- Vous pourriez créer **un mode 2 joueurs** dans lequel le premier joueur rentre un mot que le deuxième joueur doit deviner
- Ce n'est pas utile (donc c'est indispensable) : pourquoi ne pas **dessiner un bonhomme qui se pend** à chaque fois que l'on fait une erreur ?
  A coups de printf bien sûr, on est en console rappelez-vous

Prenez bien le temps de comprendre ce jeu de Pendu et améliorez-le jusqu'au maximum. Il faut que vous soyez capables de refaire ce petit jeu de Pendu les yeux fermés !

Allez courage Pfiou !

Ca n'a pas été facile, mais on y arrive toujours n'est-ce pas ? Si vous voulez *vraiment* progresser en C, il vous faut pratiquer. Inventez des jeux si c'est ce qui vous amuse, inventez des petits utilitaires (éditeur de texte en console) si c'est votre domaine à vous... Faites ce que vous voulez, mais programmez ! Entraînez-vous !

Une fois que vous serez bien à l'aise avec les programmes en console...
...

... Oh mais... qu'entends-je ? Qu'ouïe-je ?
Mais oui ! C'est bien l'appel de la librairie SDL !

J'entrevois... une foule de possibilités qui s'offre à vous maintenant !
Des programmes en fenêtre, en plein écran, de la 2D, du son, le contrôle du clavier, de la souris, du joystick...

Attends-moi SDL, j'arriiiiiiiiiiiiiive !!!!!

Si vous arrivez jusque-là, vous pouvez vous dire que le plus dur est fait !
Certes la partie II comporte son lot de difficultés, mais avec un peu de bonne volonté on arrive à tout !

Une récompense attend tous ceux qui seront parvenus à comprendre toute la partie II. Cette récompense... c'est la partie III !

PARTIE 3 : [LIBRAIRIE C] CRÉATION DE JEUX 2D EN SDL

Arrivés à ce stade, vous connaissez la plupart des bases du C. Vous avez donc la théorie nécessaire pour réaliser à peu près n'importe quel programme. Mais... pour le moment nous n'avons fait que des *printf* en console, ce qui fait que nos programmes sont encore bien monotones.

Dans la partie III, ça va changer ! Nous allons étudier une librairie qui a pour nom **SDL** (*Simple Directmedia Layer*).
Cette librairie, une fois installée, rajoute de nombreuses possibilités. Vous allez pouvoir en effet ouvrir des fenêtres, faire du plein écran, dessiner, gérer le contrôle du clavier, de la souris, du joystick...

Bref, à partir de maintenant nous allons vraiment pouvoir nous amuser !

Installation de la SDL

Dans la partie III, fini la théorie : nous allons enfin passer au concret
Nous allons étudier une librairie tierce.

Une librairie quoi ?

Il faut savoir qu'il existe 2 sortes de librairies :

- **La librairie standard** : c'est la librairie "de base" qui fonctionne sur tous les OS (d'où le mot "standard") et qui permet de faire des choses très basiques comme des *printf*. Elle a été automatiquement installée lorsque vous avez téléchargé votre IDE.
  Durant la partie I et la partie II, nous avons uniquement utilisé la librairie standard (stdlib.h, stdio.h, string.h, time.h...).

  Nous n'avons pas étudié la librairie standard en entier mais nous en avons vu un assez gros morceau. Si vous voulez tout savoir sur la librairie standard, tapez "C standard library" dans google et vous aurez la liste des prototypes ainsi qu'une brève explication de chacune des fonctions.

J'envisage moi aussi de faire une liste (mais en français cette fois). Si un jour je trouve la motivation pour m'en occuper, vous trouverez cette liste en annexe du cours.

- **Les librairies tierces** : ce sont des librairies qui ne sont pas installées avec votre IDE. Vous devez les télécharger sur Internet et les installer sur votre ordinateur.
  Contrairement à la librairie standard, qui est relativement simple et qui contient assez peu de fonctions, il existe des milliers de librairies tierces écrites par d'autres programmeurs. Certaines sont bonnes, d'autres moins, certaines sont payantes, d'autres gratuites etc. Le tout est de trouver de préférence des librairies bonnes et gratuites à la fois

Je ne peux pas faire un cours pour toutes les librairies tierces qui existent. Même en y passant toute ma vie 24h/24 je ne pourrais pas
J'ai donc fait le choix de vous présenter une librairie écrite en C, et donc utilisable par des programmeurs en langage C tels que vous.

Cette librairie a pour nom **la SDL**. Pourquoi ai-je choisi cette librairie plutôt qu'une autre ? Que permet-elle de faire ?

Autant de questions auxquelles je vais commencer par répondre

Pourquoi avoir choisi la SDL ?

Choisir une librairie : pas facile !

Comme je vous l'ai dit en introduction, il existe des milliers et des milliers de librairies à télécharger.
Certaines librairies sont simples, d'autres plus complexes. Certaines sont tellement grosses que même tout un cours entier comme celui que vous êtes en train de lire ne suffirait pas !

Faire un choix est donc dur. En plus c'est la première librairie que vous allez apprendre à utiliser (si on ne compte pas la librairie standard), donc il vaut mieux commencer par une librairie simple.

Vous, vous voudriez je pense commencer à voir comment on ouvre des fenêtres, comment on peut créer des jeux etc etc. *(enfin si vous aimez la console on peut continuer longtemps si vous voulez... Non ? Ah bon tiens c'est curieux)*
Quant à moi, non seulement j'ai bien envie de vous montrer comment on peut faire tout ça, mais en plus je veux vous faire pratiquer. En effet, nous avons bien fait quelques TP dans les parties I et II, mais ce n'est pas assez ! C'est en forgeant que l'on devient forgeron, et c'est en programmant que euh... Bref vous m'avez compris

Je suis donc parti pour vous à la recherche d'une librairie à la fois simple et puissante pour que vous puissiez rapidement réaliser vos rêves les plus fous sans avoir envie de vous suicider une fois toutes les 20 minutes

La SDL est un bon choix !

J'ai mis un moment avant de me décider et c'est finalement la SDL que nous allons étudier.
Pourquoi ?

- C'est **une librairie écrite en C**, elle peut donc être utilisée par des programmeurs en C tels que vous (notez qu'elle pourra aussi être utilisée dans un programme écrit en C++). En revanche, l'inverse n'est pas vrai : les librairies écrites en C++ ne sont pas utilisables dans des programmes en C, donc toutes celles-là on peut déjà les exclure

- C'est **une librairie libre et gratuite** : je vous ai dit dès le début que vous n'auriez pas à débourser un sou, je tiendrai mes promesses. Contrairement à ce que l'on pourrait penser, trouver des librairies libres et gratuites n'est pas très difficile, il en existe beaucoup aujourd'hui.

Qu'est-ce qu'une librairie libre ?

C'est tout simplement une librairie dont vous pouvez voir le code source.
En ce qui nous concerne, voir le code source de la SDL ne nous intéressera pas. Toutefois, le fait que la librairie soit libre vous garantit plusieurs choses, notamment sa gratuité et sa pérennité (si le développeur principal arrête de s'en occuper, d'autres personnes pourront la continuer à sa place). La librairie ne risque donc pas de mourir du jour au lendemain

- **Vous pouvez réaliser des programmes commerciaux et propriétaires avec**. Bon, ok, c'est peut-être un peu trop vouloir anticiper, mais tant qu'à faire autant choisir une librairie gratuite qui vous laisse un maximum de libertés. En effet, il existe 2 types de librairies libres :
  - Les librairies sous **license GPL** : elles sont gratuites et vous pouvez avoir le code source, mais vous êtes obligés en contrepartie de fournir le code source des programmes que vous réalisez avec.
  - Les librairies sous **license LGPL** : c'est la même chose grosso modo, sauf que cette fois vous n'êtes pas obligés de fournir le code source de vos programmes. Vous pouvez donc réaliser des programmes propriétaires avec.

  Les premiers temps toutefois, je vous conseille de montrer la source de vos programmes pour avoir des conseils de programmeurs plus expérimentés que vous. Cela vous permettra de vous améliorer.
  Après, c'est vous qui choisirez de faire des programmes libres ou propriétaires, c'est surtout une question de mentalité. Je ne rentrerai pas dans le débat ici pas plus que je ne prendrai position, on peut tirer du bon comme du mauvais dans chacun de ces 2 types de programmes.

- C'est **une librairie multiplateforme**. Que vous soyez sous Windows, Mac ou Linux, la SDL fonctionnera chez vous. C'est même d'ailleurs ce qui fait que cette librairie est impressionnante aux yeux des programmeurs : elle fonctionne sur un très grand nombre de systèmes d'exploitation. Il y a Windows, Mac et Linux certes, mais cela peut aussi fonctionner sur Atari, Amiga, Symbian, Dreamcast etc.

- Enfin, la librairie permet de **faire des choses amusantes**. Je ne dis pas qu'une librairie mathématique capable de résoudre des équations du quatrième degré n'est pas intéressante, mais je pense quand même que la plupart d'entre vous aimeraient plutôt voir comment on peut créer des jeux vidéo

La SDL n'est pas une librairie spécialement faite pour créer des jeux vidéo. Bon ok, la plupart des programmes utilisant la SDL sont des jeux vidéo, mais cela ne veut pas dire que vous êtes forcément obligés d'en faire. A priori, tout est possible avec plus ou moins de travail (je connais des gens qui ont fait un éditeur de texte en SDL par exemple)

Les possibilités offertes par la SDL

La SDL est une librairie **bas niveau**. Vous vous souvenez de ce que je vous avais dit au tout début du cours à propos des langages haut niveau et bas niveau ? Eh bien ça s'applique aussi aux librairies.

- **Une librairie bas niveau** : c'est une librairie disposant de fonctions très basiques. Il y a en général peu de fonctions car on peut tout faire à l'aide de ces fonctions basiques. Comme les fonctions sont basiques, elles sont très rapides. Les programmes réalisés à l'aide d'une telle librairie sont donc en général ce qui se fait de plus rapide (sauf si vous codez avec les pieds bien sûr)
- **Une librairie haut niveau** : elle possède en général beaucoup de fonctions capables de faire de nombreuses choses différentes. Cela la rend plus simple d'utilisation.
  Toutefois, une librairie de ce genre est généralement "grosse", donc plus difficile à étudier et à connaître

entièrement. En outre, elle est souvent plus lente qu'une librairie bas niveau (bien que parfois ça ne soit pas vraiment visible).

Bien entendu, il faut nuancer. On ne peut pas dire "une librairie bas niveau c'est mal" ou "une librairie haut niveau c'est mal". Il y a des avantages et des défauts à chacun des 2 types, et la SDL fait partie des librairies bas niveau c'est tout.

Il faut donc retenir que la SDL ne propose que des fonctions basiques. Vous avez par exemple la possibilité de dessiner pixel par pixel, de dessiner des rectangles ou encore d'afficher des images. C'est tout, et c'est suffisant.

- En faisant bouger une image, vous pouvez faire déplacer un personnage.
- En affichant plusieurs images d'affilée, vous pouvez créer une animation.
- En combinant plusieurs images côte à côte, vous pouvez créer un véritable jeu.

Pour vous donner une idée de jeu codable en SDL, sachez que le jeu "**Civilization : Call to power**" a été adapté pour Linux à l'aide de la librairie SDL.
Voici quelques captures d'écran :

Ce qu'il faut bien comprendre, c'est qu'en fait tout dépend de vous. Vous pouvez faire des jeux encore plus beaux en créant de plus beaux graphismes, ou en faire des plus moches en utilisant des graphismes plus moches
Cela devrait quand même je pense vous donner une petite idée.

La seule limite de la SDL, c'est la 2D. Elle n'est pas conçue pour la 3D (qui est de toute manière plus complexe, nous l'étudierons plus tard).
Voici une liste de jeux parfaitement codables en SDL (ce n'est qu'une petite liste, tout est possible à priori tant que ça reste de la 2D) :

- Casse-briques
- Bomberman
- Tetris
- Jeu de plateforme : Super Mario Bros, Sonic, Rayman...
- RPG 2D : Zelda, les premiers Final Fantasy etc...

Il m'est impossible de faire une liste complète, la seule limite ici étant l'imagination
Bien entendu, y arriver demandera parfois beaucoup de travail, parfois énormément de travail. Mais ça reste

possible.

Allez, on arrête de rêver et on redescend sur Terre : je ne fais que vous parler de la SDL depuis tout à l'heure, mais on ne l'a toujours pas installée !

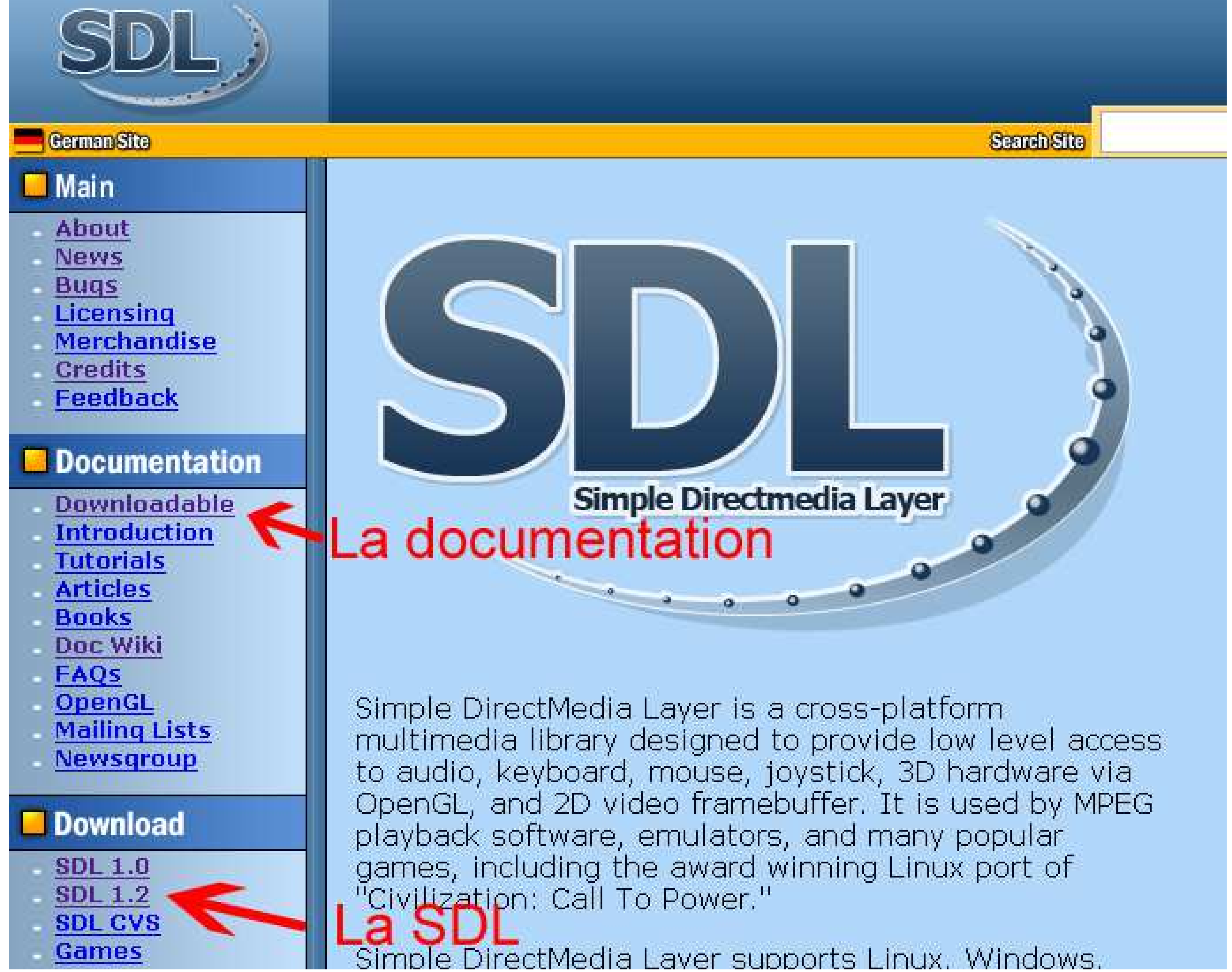
TÉLÉCHARGEMENT DE LA SDL

Voici un nouveau site à mettre en favori :

http://www.libsdl.org

Là-bas, vous trouverez tout ce dont vous avez besoin, en particulier la librairie elle-même ainsi que sa documentation

Voici à quoi ressemble le site de la SDL. Repérez bien les menus à gauche, c'est là que tout se passe :



Rendez-vous dans le menu à gauche, section "Download".
Téléchargez la version de la SDL la plus récente que vous voyez (SDL 1.2 au moment où j'écris ces lignes).

La page de téléchargement est séparée en plusieurs parties :

- **Source code** : vous pouvez télécharger le code source de la SDL. Comme je vous l'ai dit, le code source ne nous intéresse pas. Je sais que vous êtes curieux et que vous voudriez savoir *comment c'est fait* mais actuellement ça ne vous apportera rien. Pire, ça vous embrouillera et c'est pas le but
- **Runtime libraries** : ce sont les fichiers que vous aurez besoin de distribuer en même temps que votre exécutable lorsque vous donnerez votre programme à d'autres personnes. Sous Windows, il s'agit tout simplement d'un fichier SDL.dll.
Celui-ci devra se trouver :
  - Soit dans le même dossier que l'exécutable (ce que je recommande)
  - Soit dans le dossier c:\Windows

  L'idéal est de toujours donner la DLL avec votre exécutable et de la laisser dans le même dossier. Si vous mettez la DLL dans le dossier de Windows, vous n'aurez plus besoin de mettre une DLL dans chaque dossier contenant un programme SDL. Toutefois, cela peut poser des problèmes de conflits de version si vous écrasez une DLL plus récente.
- **Development libraries** : ce sont les fichiers .a (ou .lib sous visual) et .h vous permettant de créer des programmes SDL. Ces fichiers ne sont nécessaires que pour vous, le programmeur. Vous n'aurez donc pas à les distribuer avec votre programme une fois qu'il sera fini.
Si vous êtes sous Windows, on vous propose 2 versions dépendant chacune de votre compilateur :
  - **VC6** : pour ceux qui utilisent Visual Studio (il y aura donc des fichiers .lib)
  - **mingw32** : pour ceux qui utilisent Code::Blocks ou Dev-C++ (il y aura donc des fichiers .a)

Source Code:
SDL-1.2.9.tar.gz - GPG signed
SDL-1.2.9-1.src.rpm - GPG signed
SDL-1.2.9.zip - GPG signed
SDL-1.2.9.sea.bin - GPG signed

Runtime Libraries:
Linux:
SDL-1.2.9-1.i386.rpm
http://packages.debian.org/stable/libs/
Win32:
SDL-1.2.9-win32.zip
BeOS:
LibPak sdl for users package (BeOS 5.0)
MacOS (Classic):
SDL-1.2.9-PPC.sea.bin
MacOS X:
SDL-1.2.9.dmg

Development Libraries:
Linux:
SDL-devel-1.2.9-1.i386.rpm
http://packages.debian.org/stable/libdevel/
Win32:
SDL-devel-1.2.9-VC6.zip (Visual C++ 5,6,7)
SDL-devel-1.2.9-mingw32.tar.gz (Mingw32)

La particularité, c'est que les "Development libraries" contiennent tout ce qu'il faut : les .h et .a (ou .lib) bien sûr, mais aussi la SDL.dll à distribuer avec votre application ainsi que la documentation de la SDL !
Bref, tout ce que vous avez à faire est de télécharger les "**Development libraries**". Tout ce dont vous avez besoin se trouve à l'intérieur

Ne vous trompez pas de lien ! Prenez bien la SDL dans la section "Development libraries" (plus bas sur la page) et non le code source de la section "Source code" !

Qu'est-ce que la documentation ?

Une **documentation**, c'est la liste complète des fonctions d'une librairie. Toutes les documentations sont écrites en anglais (oui même les librairies écrites par des français ont leur documentation en anglais). Voilà une raison de plus de bien vous entraîner en anglais !

La documentation n'est pas un tutorial, elle est en général assez austère. L'avantage par rapport à un tutorial, c'est qu'elle est complète. Elle contient la liste de TOUTES les fonctions, c'est donc LA référence du programmeur.
Bien souvent, vous rencontrerez des librairies pour lesquelles il n'y a pas de tutorial. Vous aurez uniquement la *doc'* comme on l'appelle, et vous serez capables de vous débrouiller avec seulement ça (même si parfois c'est un peu dur de démarrer sans aide). Un vrai bon programmeur peut donc découvrir le fonctionnement d'une librairie uniquement en lisant sa doc.

A priori, vous n'aurez pas besoin de la doc de la SDL de suite car je vais moi-même vous expliquer comment elle fonctionne. Toutefois, c'est comme pour la librairie standard : je ne pourrai pas vous parler de toutes les fonctions. Vous aurez donc certainement besoin de lire la doc plus tard.

La documentation se trouve déjà dans le package "Development libraries", mais si vous le voulez vous pouvez la télécharger à part en vous rendant dans le menu "Documentation" / "Downloadable".
Je vous recommande de mettre les fichiers HTML de la documentation dans un dossier spécial (par exemple "Doc SDL") et de faire un raccourci dans le menu Démarrer vers le fichier index.html. Le but est que vous puissiez accéder rapidement à la documentation lorsque vous en avez besoin

Créer un projet SDL

Comment on installe la SDL ?

L'installation d'une librairie est en général un petit peu plus compliquée que les installations dont vous avez l'habitude. Ici, il n'y a pas d'installeur automatique qui vous demande de bêtement cliquer sur *Suivant - Suivant - Suivant - Terminer*

En général, installer une librairie est assez difficile pour un débutant. Pourtant, si ça peut vous remonter le moral, l'installation de la SDL est beaucoup plus simple que bien d'autres librairies que j'ai eu l'occasion d'utiliser (en général on ne vous donne que le code source de la librairie, et c'est à vous de la recompiler !).

En fait, le mot "installer" n'est peut-être pas celui qui convient le mieux. Nous n'allons rien installer du tout : nous voulons simplement arriver à créer un nouveau projet de type SDL avec notre IDE.
Or, selon l'IDE que vous utilisez la manipulation sera un peu différente. Je vais présenter la manip' pour chacun des IDE que je vous ai montrés au début du cours pour que personne ne se sente désavantagé.

Personnellement j'ai l'habitude de coder le plus souvent sous Visual C++, surtout parce qu'il possède un débugger qui m'aide à retrouver mes erreurs. Si vous n'utilisez pas Visual, je vous recommande Code::Blocks qui est, à mes yeux, un excellent second choix. De préférence, évitez d'utiliser Dev-C++ car cet IDE se fait vieux et n'est plus trop mis à jour.

Je vais maintenant vous montrer comment créer un projet SDL sous chacun de ces 3 IDE.

Création d'un projet SDL sous Code::Blocks

1/ Extraction des fichiers de la SDL

Ouvrez le fichier compressé de "Development Libraries" que vous avez téléchargé.
Ce fichier est un .zip pour Visual et un .tar.gz pour mingw32 (il vous faudra un logiciel comme Winrar pour décompresser le .tar.gz si vous ne l'avez pas encore).

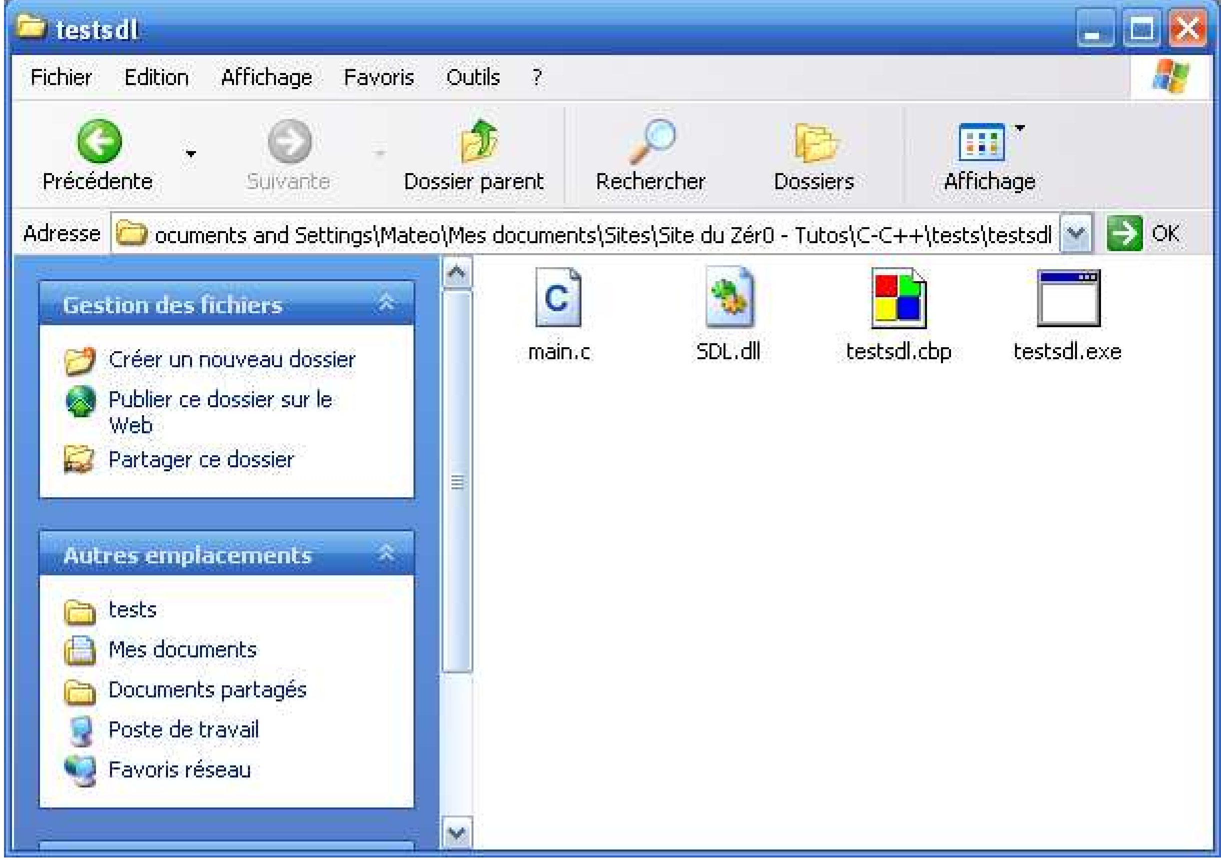
Le fichier compressé contient plusieurs sous-dossiers. Ceux qui nous intéressent sont les suivants :

- **docs** : contient la documentation de la SDL
- **include** : contient les .h
- **lib** : contient les .a (ou .lib pour visual)

Il y a aussi un dossier "bin" dans le fichier pour mingw32 qui contient le fichier SDL.dll (c'est le même que celui que vous auriez téléchargé en allant dans "Runtime libraries" tout à l'heure).
Dans le cas de Visual, la DLL se trouve dans le dossier "lib".

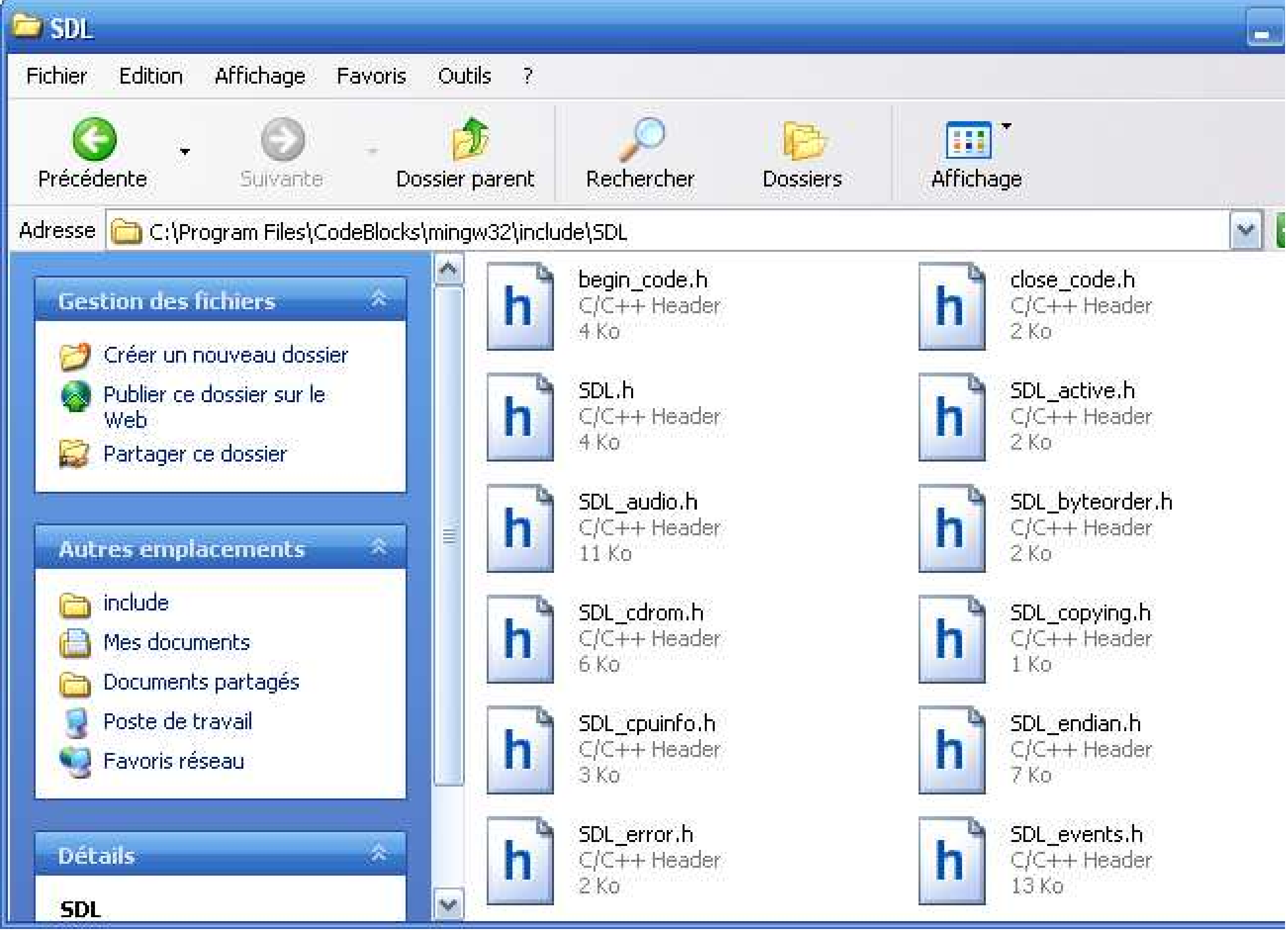
Vous allez extraire les fichiers comme ceci (attention à ne pas vous tromper !) :

- **SDL.dll** : vous la mettrez dans le même dossier que le projet (et donc que l'exécutable). Si elle ne s'y trouve pas au moment où vous lancez votre exécutable, Windows vous affichera une erreur vous disant que SDL.dll est introuvable.
  Votre dossier du projet devra donc contenir ces fichiers :



  L'exécutable n'apparaîtra qu'une fois que vous aurez compilé bien entendu

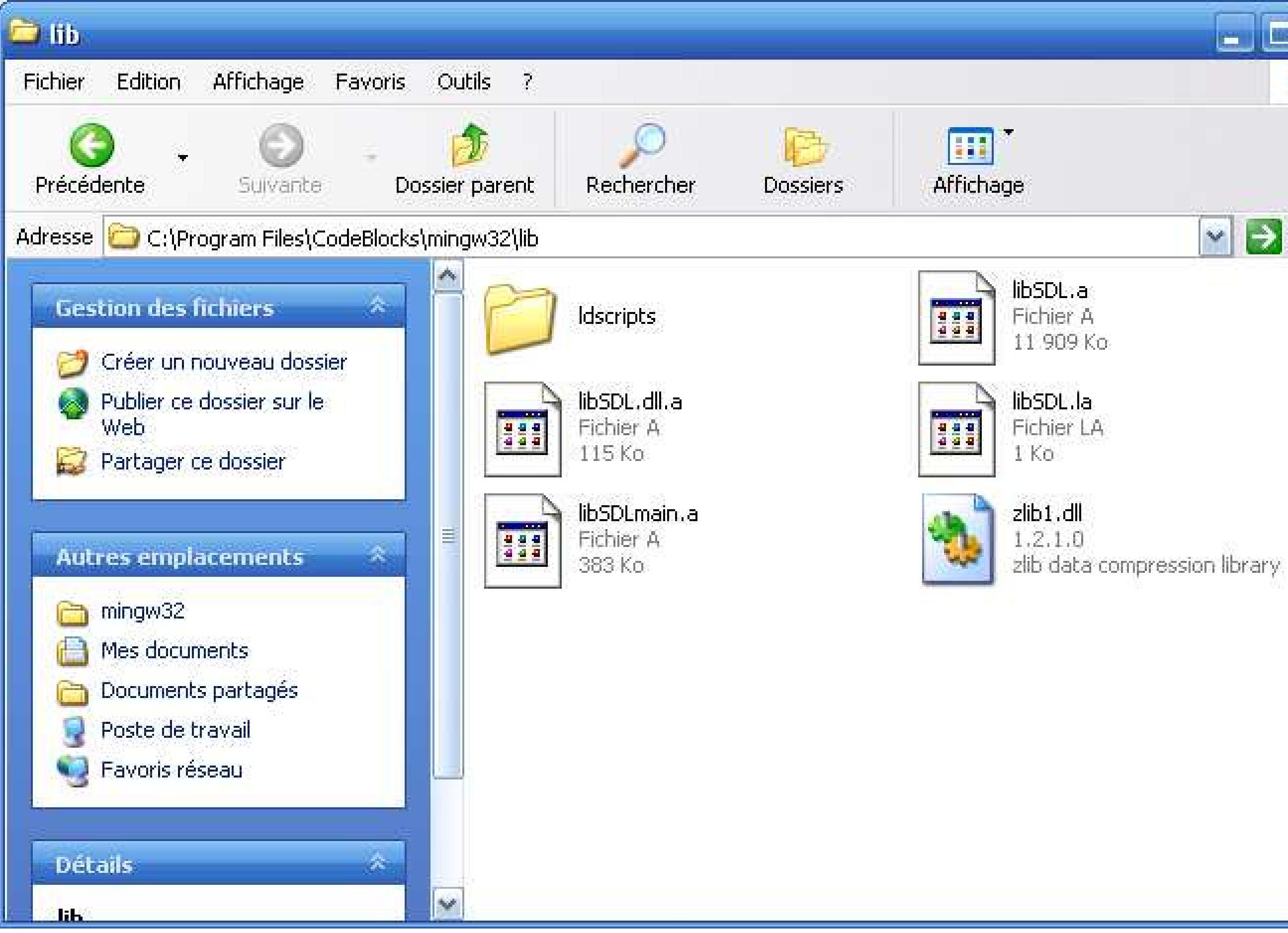
- **Les fichiers .h** : rendez-vous dans le dossier de votre compilateur. Sous Code::Blocks, il faut vous rendre dans le dossier où Code::Blocks est installé, puis dans le sous-dossier mingw32. Là, vous trouverez un dossier "include" (s'il n'existe pas, créez-le). Créez un dossier "SDL" à l'intérieur et placez-y tous vos fichiers .h.



Le dossier dans lequel se trouvent mes .h est donc :
`C:\Program Files\CodeBlocks\mingw32\include\SDL`

Il sera peut-être différent chez vous si vous avez installé CodeBlocks dans un autre dossier.

- **Les fichiers .a (ou .lib)** : vous les mettrez non pas dans le dossier include du compilateur mais dans le dossier lib. Exemple :



Dans mon cas, les fichiers se trouvent donc dans :
`C:\Program Files\CodeBlocks\mingw32\lib`

Il y a des dossiers "include" et "lib" dans le dossier "CodeBlocks" mais vous ne devez pas utiliser ceux-là.
Vous devez utiliser les dossiers "include" et "lib" situés dans "CodeBlocks\mingw32".

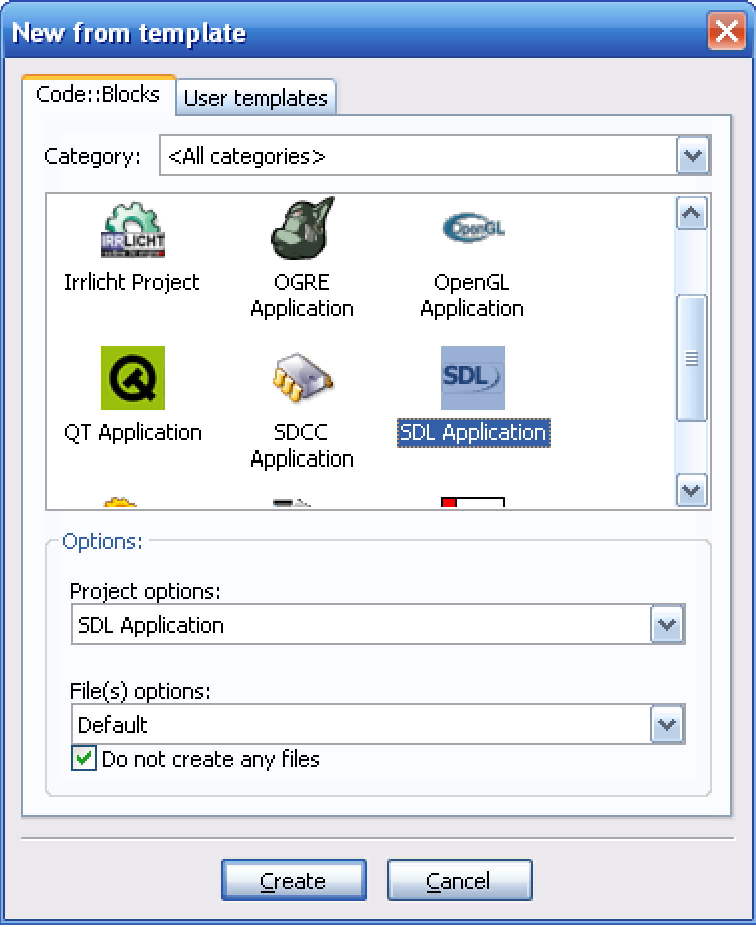
Bon eh bien c'est tout :)
Normalement la SDL est installée.
Vous n'aurez plus à copier les fichiers dans lib et include pour chaque nouveau projet, en revanche pensez à mettre le fichier SDL.dll dans le dossier de chacun de vos projets SDL !

2/ Création du projet SDL

Ouvrez maintenant Code::Blocks et demandez à créer un nouveau projet.

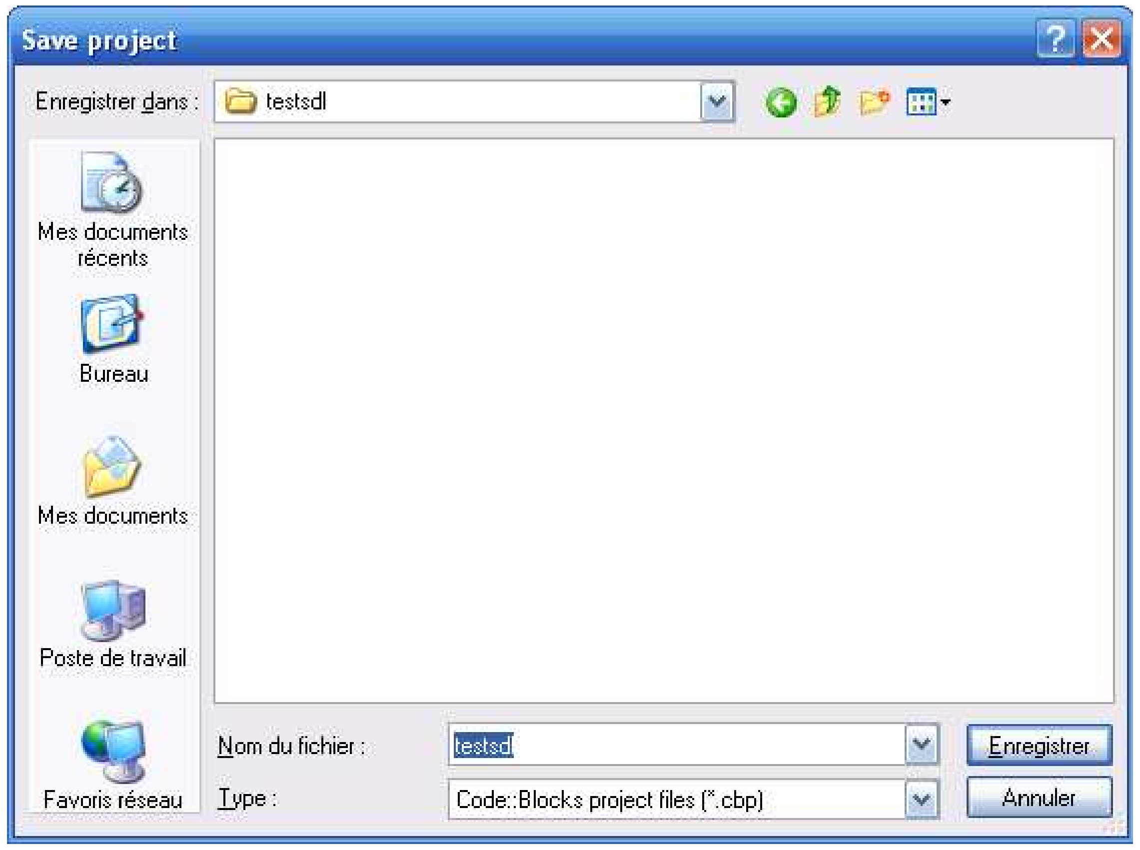
Là, au lieu de créer un projet "*Console Application*" comme vous aviez l'habitude de faire, vous allez demander à créer un projet "*SDL Application*".
Plus bas, **veillez à cocher la case "Do not create any files"**. Si vous ne le faites pas, Code::Blocks va vous préremplir un fichier main.c avec pas mal de code d'exemple, et comme nous débutons cela ferait trop d'un coup. Bref, restons simples.

Votre fenêtre de création de projet devrait donc ressembler à cela :



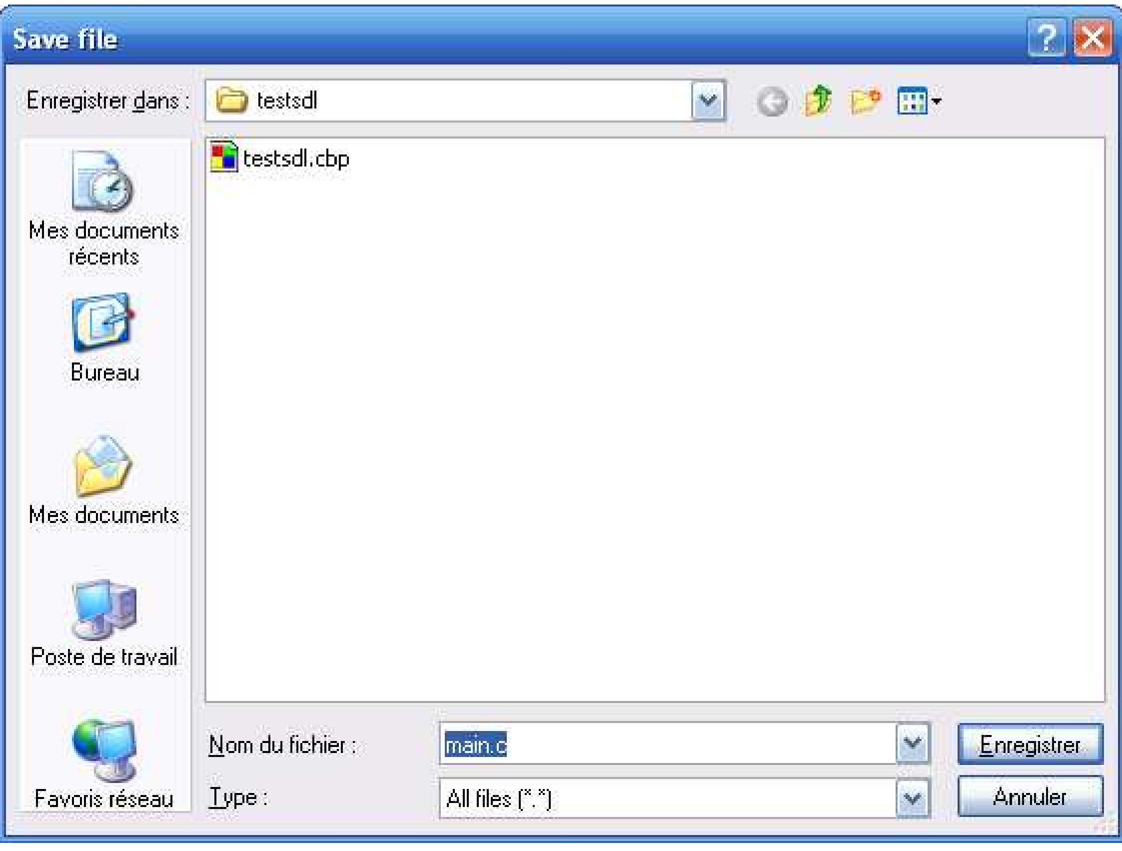
Cliquez sur "Create".
Une boîte de dialogue s'ouvre, vous demandant où vous voulez enregistrer votre projet. C'est donc pour le moment exactement la même chose que la création d'un projet de type Console.

Comme d'habitude, je vous recommande de créer un nouveau dossier spécialement pour ce projet. Vous n'avez qu'à appeler ce dossier (ainsi que votre projet) "testsdl" par exemple :

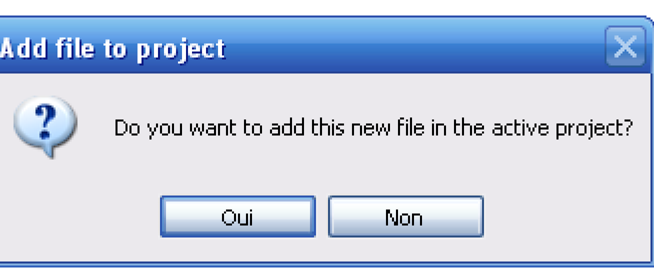


Cliquez sur "Enregistrer". Un nouveau projet vide est créé. Il ne contient pour le moment aucun nouveau fichier, pas même un main.c !

Vous allez donc aller dans le menu "File" / "New File", et demander à créer un fichier "main.c" :



Cliquez sur "Enregistrer".
On vous demande alors si vous voulez ajouter ce fichier à votre projet. La réponse est... bien entendu oui



Cliquez donc sur "Oui".

Voilà, votre premier projet SDL est prêt (quoique un peu vide)
Mettez le code suivant dans votre main.c :

Code : C

```c
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

int main(int argc, char *argv[])
{

    return 0;
}
```

C'est en fait un code de base très similaire à ceux que l'on connaît (un include de stdlib, un autre de stdio, un main...).
La seule chose qui change, c'est l'include d'un fichier SDL.h. C'est le fichier .h de base de la SDL qui se chargera d'inclure tous les autres fichiers .h de la SDL. Bref, en incluant SDL.h, cela inclue automatiquement tous les fichiers .h nécessaires au bon fonctionnement de la SDL (et il y en a plein !).

Comme vous pouvez le voir, l'include est le suivant :

Code : C

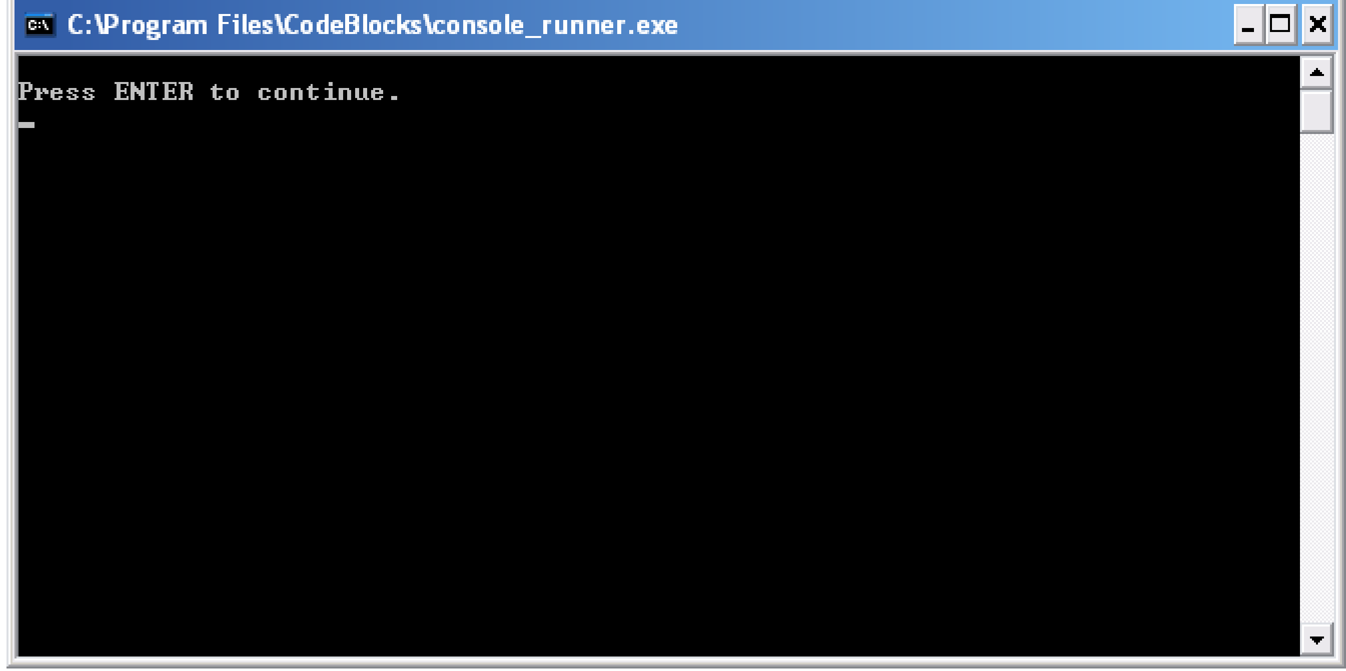
```c
#include <SDL/SDL.h>
```

Les chevrons < et > signifient que le fichier se trouve positionné dans le dossier du compilateur. Toutefois, le fichier inclus est SDL/SDL.h, ce qui veut dire que le fichier SDL.h doit se trouver dans un sous-dossier nommé "SDL".

Créer un sous-dossier SDL permet de regrouper tous les .h de la SDL et donc d'éviter qu'ils se mélangent avec d'autres .h, ce qui serait vite le bordel

3/ Démarrer le programme

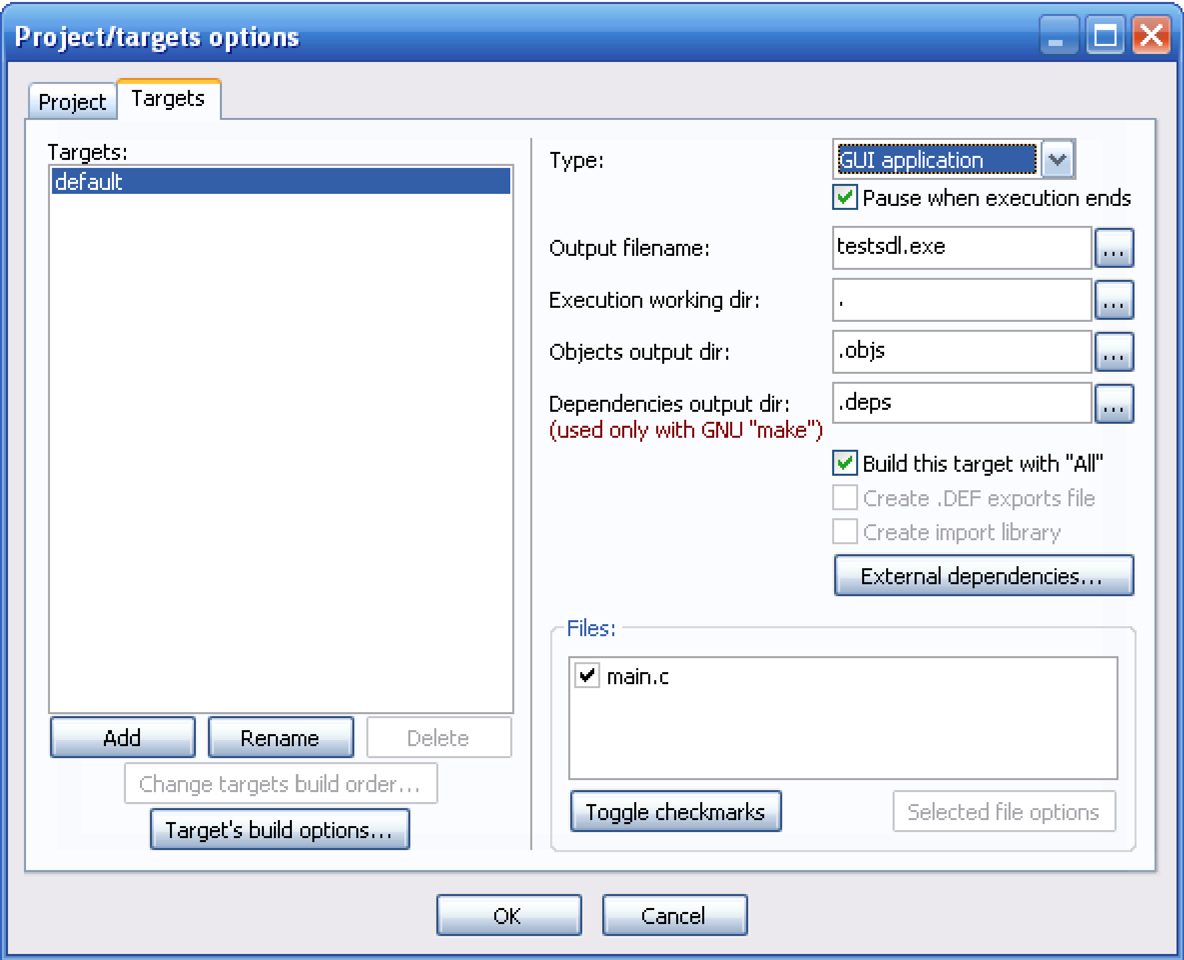
Essayez de compiler votre programme. Si tout se passe bien, une console devrait s'afficher comme ceci :



Si la compilation échoue, c'est que vous avez mal installé quelque chose. Recommencez, vous avez dû vous planter quelque part

? QUOIIII ? Mais je croyais que la console c'était fini, pourquoi est-ce que ça m'ouvre une console encore ? Ouiiiiin

Vous inquiétez pas, c'est juste une option du projet à changer;
Rendez-vous dans le menu Project / Properties. Allez dans l'onglet "Targets" :



Repérez en haut à droite la liste déroulante "Type". Elle devrait être actuellement sur "Console application". Changez ça pour "GUI application" et la console n'apparaîtra plus
En outre, vous pouvez changer le nom de l'exécutable généré juste en-dessous (vous pouvez mettre *testsdl.exe* au lieu du *sdlapp.exe* par défaut).

Cliquez sur OK pour valider et c'est bon

Si vous recompilez cette fois et que vous exécutez, rien ne s'affichera à l'écran (pas même une fenêtre !). C'est normal, vu qu'on n'a mis aucun code source qui demande à afficher une fenêtre. Nous verrons cela dès le chapitre suivant

Ce qui compte, c'est que la compilation se passe sans afficher d'erreur

Création d'un projet SDL sous Dev-C++

1/ Extraction des fichiers de la SDL

L'extraction des fichiers est quasiment identique à Code::Blocks car le compilateur est le même (mingw32).
Rendez-vous dans le dossier de l'IDE Code::Blocks, puis dans le sous-dossier mingw32. Chez moi, le répertoire est :
`C:\Program Files\Dev-Cpp\mingw32`

La manipulation est la même que pour Code::Blocks :

- Placez les fichiers .a dans le dossier Dev-Cpp\mingw32\lib
- Placez les headers (.h) dans le dossier Dev-Cpp\mingw32\include\SDL. Pensez à créer un sous-dossier "SDL"

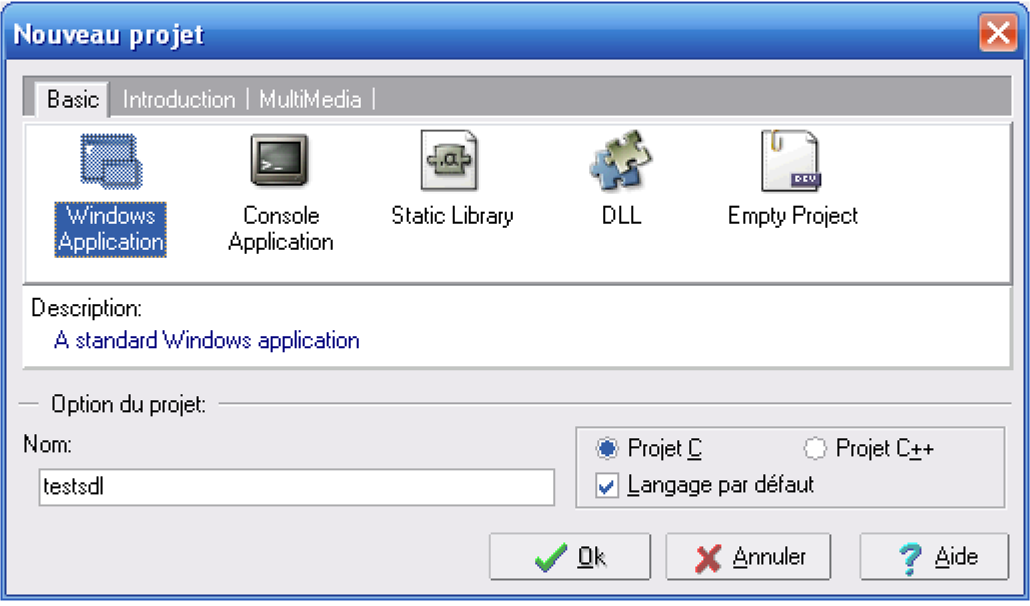
pour y mettre vos .h, c'est important.
- Placez la SDL.dll dans le dossier de votre projet (à côté de l'exécutable).

Si les dossiers dont je vous parle n'existent pas sur votre ordinateur, créez-les.
Attention : il y a peut-être des dossiers lib et include dans le dossier Dev-Cpp, mais ce ne sont pas les bons.
Il faut vous rendre dans le sous-dossier Dev-Cpp\mingw32 !

Si vous voulez plus de détails et des captures d'écrans, remontez un peu plus haut sur cette page et regardez les captures d'écran pour Code::Blocks : c'est la même chose à part le répertoire de l'IDE qui est un peu différent.

2/ Création d'un nouveau projet SDL

Créez un nouveau projet Dev-C++ de type **Windows Application** en langage C. Cochez la case "Langage par défaut".



Cliquez sur OK. On vous demande où vous voulez enregistrer votre projet. Là encore, créez un dossier spécialement pour votre projet et nommez votre projet "testsdl" par exemple.

Dev-C++, contrairement à Code::Blocks qui vous laisse le choix, vous crée automatiquement le main.c et le remplit de... pas mal de code qui n'est pas un code SDL mais un code pour l'API Win32 (librairie de création de fenêtres pour Windows). Or, ce n'est pas ce qu'on veut faire !
Vous allez donc me faire le plaisir de supprimer tout le code de main.c et d'y mettre à la place le même code que celui qu'on a utilisé pour Code::Blocks, à savoir :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

int main(int argc, char *argv[])
{

    return 0;
}
```

3/ Configuration du projet SDL sous Dev-C++

Code::Blocks nous avait automatiquement configuré notre projet quand on lui avait demandé de créer une "SDL Application". Pour Dev, il va falloir configurer cela manuellement. Fort heureusement, c'est simple :)

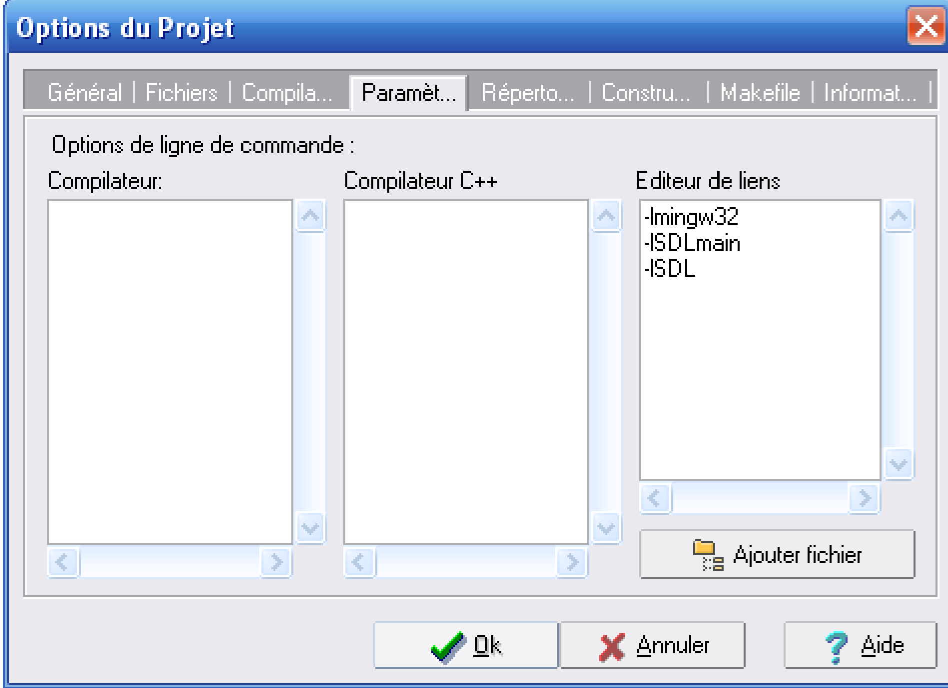
Rendez-vous dans le menu Projet / Options du projet. Allez dans l'onglet "Paramètres".
Là, repérez la zone de texte tout à droite intitulée "Editeur de liens".

Mettez-y le code suivant :

Code : Autre

```
-lmingw32
-lSDLmain
-lSDL
```

Vous devriez voir ceci :



Ce sont les commandes qui vont commander au linker (l'éditeur de liens) d'ajouter les fichiers .a de la SDL lors de l'étape d'édition des liens (celle qui intervient juste après la compilation, si vous vous souvenez bien du schéma que je vous avais fait !).

Cliquez sur OK. Enregistrez tout.
Vous pouvez compiler.

Si tout se passe bien, la compilation s'effectuera sans erreurs. Vous pourrez alors toujours tenter d'exécuter votre programme. Comme pour Code::Blocks, rien ne devrait s'afficher (pas même une fenêtre) car on n'a pas écrit le code pour demander cela. On le fera dans le prochain chapitre.
Du temps qu'aucune erreur n'apparaît, c'est que vous avez tout bon :)

Création d'un projet SDL sous Visual C++

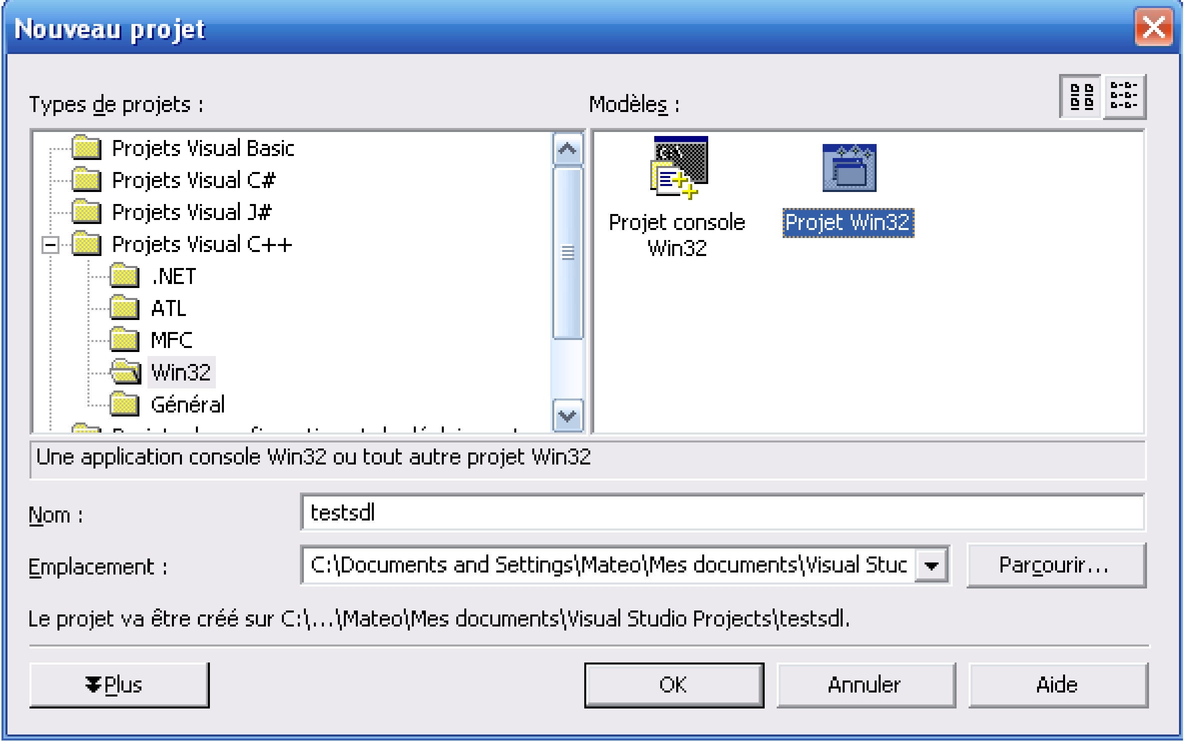
1/ Extraction des fichiers de la SDL

Ouvrez le zip que vous avez téléchargé sur le site de la SDL. Attention : c'est un .zip et non un .tar.gz cette fois.
Il contient la doc (dossier docs), les .h (dossier includes), et les .lib (dossier lib) qui sont l'équivalent des .a pour le compilateur de Visual. Vous trouverez aussi le fichier SDL.dll dans ce dossier lib.

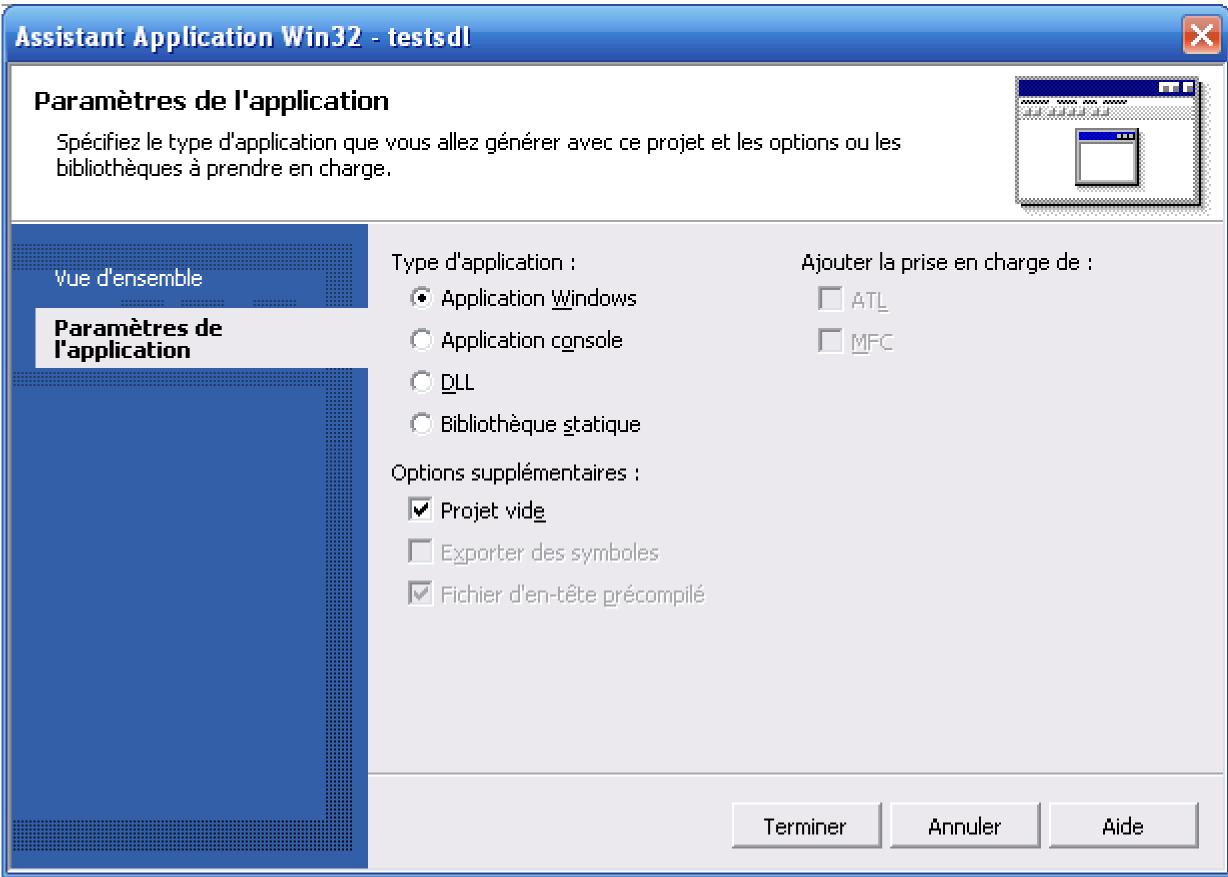
- Copiez SDL.dll dans le dossier de votre projet
- Copiez les .lib dans le dossier lib de Visual C++. Par exemple chez moi il s'agit du dossier :
  `C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\lib`
- Copiez les .h dans le dossier "includes" de Visual C++. Créez un dossier "SDL" dans ce dossier "includes" pour regrouper les .h de la SDL entre eux.
  Chez moi, je mets donc les .h dans le dossier :
  `C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\include\SDL`

2/ Création d'un nouveau projet SDL

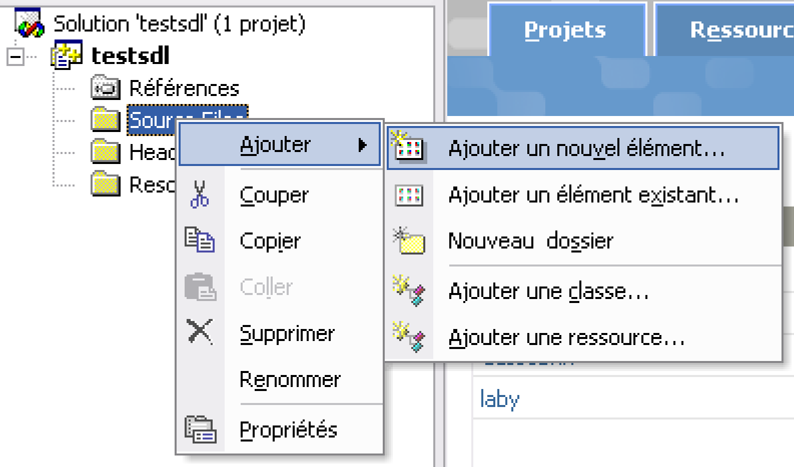
Sous Visual C++ (ici Visual C++ 2003), créez un nouveau projet de type "Projet Win32" :



Un assistant va s'ouvrir. Allez dans "Paramètres de l'application" et vérifiez que "Projet vide" est coché :



Le projet est alors créé. Il est vide.
Ajoutez-y un nouveau fichier en faisant un clic droit sur "Source Files", "Ajouter" / "Ajouter un nouvel élément".



Dans la fenêtre qui s'ouvre, demandez à créer un nouveau fichier de type "Fichier C++ (.cpp)" que vous appellerez "main.c". En mettant l'extension .c dans le nom du fichier, Visual créera un fichier .c et non un fichier .cpp.

Mettez le code "de base" dans votre nouveau fichier vide :

Code : C

```c
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

int main(int argc, char *argv[])
{

    return 0;
}
```

3/ Configuration du projet SDL sous Visual C++

La configuration du projet est un peu plus délicate que pour Dev-C++ et Code::Blocks, mais y'a pas mort d'homme je vous rassure
Allez dans les propriétés de votre projet : "Projet" / "Propriétés de testsdl".

- Dans la section "C / C++ => Génération de code", mettez "Bibliothèque runtime" à "DLL multithread (/MD)
- Dans la section "C/C++ => Avancé", sélectionnez "Compilation sous : Compiler comme code C (/TC)" (sinon visual vous compilera votre projet comme étant du C++).
- Dans la section "Editeur de liens => Entrée", modifiez la valeur de "Dépendances supplémentaires" pour y ajouter "**SDL.lib SDLmain.lib**"

Validez ensuite vos modifications en cliquant sur OK et enregistrez le tout.
Vous pouvez maintenant compiler en allant dans le menu "Générer / Générer la solution".

Rendez-vous dans le dossier de votre projet pour y trouver votre exécutable (il sera peut-être dans un sous-dossier Debug). N'oubliez pas que le fichier SDL.dll doit se trouver dans le même dossier que l'exécutable.
Double-cliquez sur votre .exe : si tout va bien, il ne devrait rien se passer. Sinon, s'il y a une erreur c'est probablement que le fichier SDL.dll ne se trouve pas dans le même dossier

Création d'un projet SDL avec Xcode (Mac OS)

Commencez par télécharger la version de la SDL sur le site officiel. Le fichier doit avoir l'extension *.dmg. Montez (chargez) ce fichier *.dmg, prenez le dossier *.framework (par ex: SDL.framework) et placez-le dans le dossier :
`<Racine Disque>/Library/Frameworks`
Si votre système est en français, il est probable que vous voyiez : `<Racine Disque>/Bibliothèque/Frameworks`

Un dossier *.framework est un dossier contenant tout les fichiers nécessaires, comme les binaires de la SDL et les headers.

Vous obtenez donc ceci :

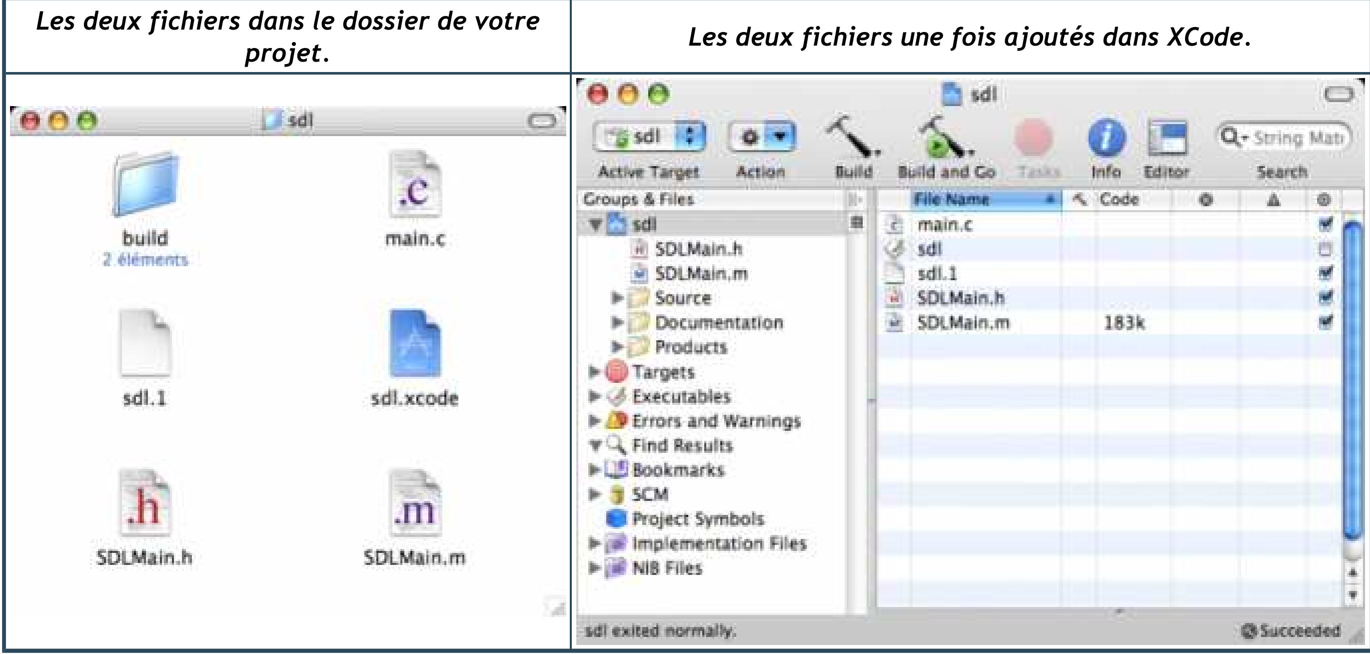
Les frameworks dans leur dossier : vous pouvez voir notamment SDL.framework.

Allez dans l'archive de la SDL (SDL-1.2.11.dmg), allez dans le dossier *devel-lite* et prenez les fichiers *SDLMain.m* et *SDLMain.h* : placez-les en lieu sûr.

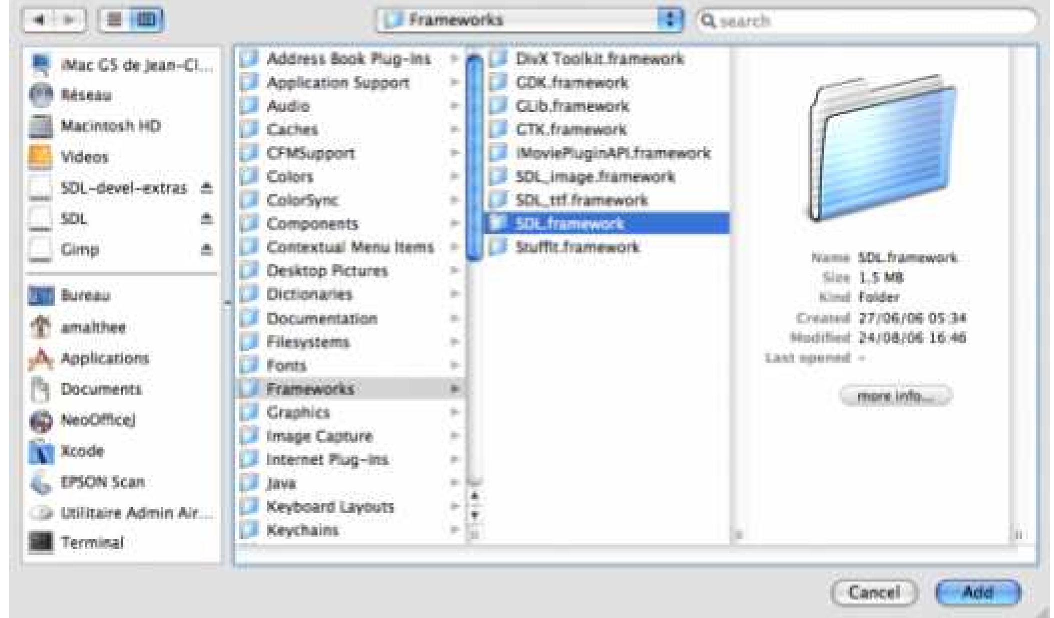
Création du projet

Ouvrez le logiciel XCode, créez un nouveau projet (*Standard Tool*, catégorie *Command Line Utility*). Placez une copie des deux fichiers *SDLMain.m* et *SDLMain.h* dans le dossier de votre projet; ajoutez ensuite ces fichiers au projet dans XCode.
Attention, ces deux fichiers sont essentiels à la SDL. Si vous les oubliez, votre projet ne marchera pas.

Les deux fichiers dans le dossier de votre projet.	*Les deux fichiers une fois ajoutés dans XCode.*

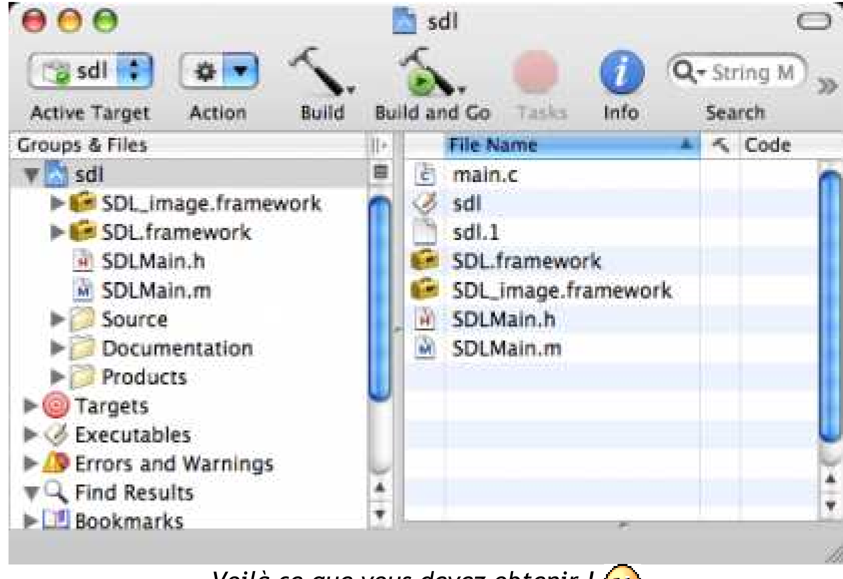


Maintenant, faites un clic droit (<ctrl> + clic) sur votre projet (l'élément surligné du menu, sur la photo de droite), allez dans *Add*, puis cliquez sur *Existing Frameworks ...*
Là, vous tombez (normalement) sur le dossier dans lequel vous avez mis les *.framework. Sélectionnez les librairies dont vous avez besoin (donc au moins SDL.framework) et validez (*Add*). Une boite de dialogue apparaît, cliquez *bêtement* sur *Add*.

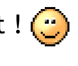


Sélectionnez les librairies dont vous avez besoin.

Cliquez sur Add.



Voilà ce que vous devez obtenir !
Ici, j'ai inclus les librairies SDL et SDL_image (une autre librairie dont on parlera plus tard).

Voilà, votre projet est prêt, tapez votre code dans le fichier, puis appuyez sur *Build & Run* : admirez le résultat ! 
Dans votre code, vous ferez ainsi pour inclure les librairies :

Code : C

```c
#include <SDL/SDL.h>
/* Comme vous pouvez le constater, le nom du dossier d'inclusion correspond au nom du
*.framework !!*/
```

Et sous Linux ?

Si vous compilez sous Linux avec un IDE, il faudra modifier les propriétés du projet (la manipulation sera quasiment la même). Si vous utilisez Code::Blocks, qui existe aussi en version Linux, vous pouvez suivre la même procédure que celle que j'ai décrite plus haut !

Et pour ceux qui compilent à la main ?

Il y en a peut-être parmi vous qui ont pris l'habitude de compiler à la main sous Linux à l'aide d'un Makefile (fichier commandant la compilation).
Si c'est votre cas, voici un **Makefile** que vous pouvez utiliser pour compiler des projets SDL

La seule chose particulière c'est l'ajout de la librairie SDL pour le linker (LDFLAGS).
Il faudra que vous ayez téléchargé la SDL version Linux et que vous l'ayez installée dans le dossier de votre compilateur, de la même manière qu'on le fait sous Windows (dossiers *include\SDL* et *lib*)

Ensuite, vous pourrez taper dans la console :

Code : Console

```
make #Pour compiler le projet
make clean #Pour effacer les fichiers de compilation (les .o inutiles)
make mrproper #Pour tout supprimer sauf les fichiers source
```

A la fin de ce chapitre, vous devriez donc avoir téléchargé et installé la SDL dans le répertoire de votre compilateur.
Vous devez être capables de créer un projet SDL configuré comme il faut avec votre IDE favori les yeux fermés
L'idéal serait même que vous soyez capables de le faire pour les 3 IDE présentés ici.

Dans le prochain chapitre nous ferons nos premiers pas en SDL, avec en particulier l'ouverture d'une fenêtre ! Ca fait longtemps que vous attendez ça hein ?

D'autres librairies ?

Nous allons donc dès le chapitre suivant commencer à étudier ensemble la SDL et découvrir comment elle fonctionne.

Toutefois, peut-être qu'il y en a parmi vous qui ne veulent pas apprendre la SDL (bon ça ça serait dommage) ou qui aimeraient aussi connaître d'autres librairies. Ce que je vous conseillerais ce serait dans tous les cas de travailler la SDL avec moi, ça vous fera de la pratique et vous aurez au moins acquis de l'expérience sur une librairie. Après, si vous le voulez, vous pourrez essayer d'apprendre par vous-mêmes une autre librairie écrite en langage C.

En effet, sachez qu'à votre niveau vous pouvez théoriquement utiliser n'importe quelle librairie écrite en C, pour peu que vous trouviez des tutoriaux pour vous aider à démarrer

Vous savez que je ne pourrai pas vous expliquer ces librairies (j'ai fait le choix de la SDL), mais je peux au moins vous indiquer le chemin si vous voulez commencer à apprendre tous seuls

Voici quelques librairies assez connues qui vous intéresseront probablement :

- **GTK+** : c'est une librairie de fenêtres multiplateforme. Elle a été créée au départ uniquement pour le logiciel de dessin The Gimp, puis elle a été étendue et améliorée pour être utilisable par d'autres programmes. Contrairement à la SDL qui ne permet pas de créer des boutons et de menus (enfin c'est possible mais il faut les simuler c'est un peu délicat) et qui est plutôt adaptée pour les jeux, GTK+ vous propose tout ça. C'est une librairie sous license LGPL aussi, donc vous êtes libres de distribuer vos programmes comme vous le voulez.

  La particularité de GTK+ est que les fenêtres ont une apparence bien particulière. Voici un screenshot d'une fenêtre GTK+ :

Une fenêtre GTK+ (merci à KaZu pour le screenshot)

Vos fenêtres auront donc une apparence différente des fenêtres habituelles de votre OS (enfin sauf pour ceux qui sont sous Gnome sous Linux). Pour que vos applications faites en GTK+ fonctionnent sur l'ordinateur de vos utilisateurs, il faudra qu'ils installent au préalable un programme appelé *le runtime GTK+* (ce qui n'est pas très pratique à mon goût).
Ceci mis à part, c'est une bonne librairie graphique de création de fenêtres. Si cela vous tente, vous trouverez un bon tutorial sur http://www.gtk-fr.org pour démarrer

- **API Win32** : c'est la librairie de création de fenêtres de Windows. Le défaut est que votre programme ne fonctionnera que sous Windows, l'avantage est que... ben en utilisant cette librairie vous pouvez créer de véritables programmes Windows en fenêtres comme ceux que vous avez l'habitude d'utiliser tous les jours (si vous êtes sous Windows). En outre, vous n'aurez aucune DLL à livrer (ou aucun runtime à installer comme c'est le cas pour GTK+) : logique, car ceux-ci sont automatiquement installés avec Windows.

  Vous trouverez un bon tutorial à cette adresse : http://bob.developpez.com/tutapiwin/

> L'API Win32 est écrite en C, mais il faut savoir aussi qu'il existe une librairie appelée la **MFC** (Microsoft Foundation Classes, donc aussi créée par Microsoft) qui vous permet de programmer en C++.
> Tout dépend du langage que vous souhaitez utiliser. A votre niveau de toute façon c'est sur l'API Win32 qu'il faut vous pencher vu que vous ne connaissez pas le C++. La MFC n'est pas plus puissante

que l'API Win32 (je vous avais déjà dit que c'est pas parce que c'est du C++ que c'est mieux), elle permet juste de créer votre programme d'une manière différente.

Voilà, ce n'est qu'une petite liste (vraiment très petite). Ce sont des suggestions de librairies que vous pouvez apprendre si vous le désirez.
Encore une fois j'insiste : je vous recommande dans un premier temps d'étudier la SDL avec moi, pour vous faire les crocs. Une fois que vous serez bien affamés et que vous aurez dévoré la SDL toute crue, vous serez plus aptes à vous lancer tous seuls dans l'étude d'une de ces librairies

Création d'une fenêtre et de surfaces

Dans le premier chapitre de la partie III, nous avons fait un petit tour d'horizon de la SDL pour voir les possibilités que cette librairie nous offre. Vous l'avez normalement téléchargée et vous êtes capables de créer un nouveau projet SDL valide sans aucun problème

Oui mais voilà la tronche de votre programme pour le moment
Ce n'est qu'un main quasiment vide (bon allez je vous l'accorde, il fait un return quand même !). Le programme n'affiche rien, ne fait rien, s'arrête de suite. Toutefois, cela est signe que votre projet SDL est correctement configuré et que vous êtes donc parés à écrire du code SDL.

Bon, qu'est-ce qu'on attend pour commencer ?

Charger et arrêter la SDL

Un grand nombre de librairies écrites en C **nécessitent d'être initialisées et fermées** par des appels à des fonctions. La SDL n'échappe pas à la règle.

En effet, la librairie doit charger un certain nombre d'informations dans la mémoire pour pouvoir fonctionner correctement. Ces informations sont chargées en mémoire dynamiquement par des *malloc* (ils sont très utiles ici !). Or, comme vous le savez, qui dit *malloc* dit... *free* !
Vous **devez** libérer la mémoire que vous avez allouée manuellement et dont vous n'avez plus besoin. Si vous ne le faites pas, votre programme va prendre plus de place en mémoire que nécessaire, et dans certains cas ça peut être complètement catastrophique (imaginez que vous fassiez une boucle infinie de malloc sans le faire exprès, en quelques secondes vous saturerez toute votre mémoire !).

Voici donc les 2 premières fonctions de la SDL à connaître :

- **SDL_Init** : charge la SDL en mémoire (des *malloc* y sont faits).
- **SDL_Quit** : libère la SDL de la mémoire (des *free* y sont faits).

La toute première chose que vous devrez faire dans votre programme sera donc un appel à SDL_Init, et la dernière sera un appel à SDL_Quit.

SDL_Init : chargement de la SDL

La fonction SDL_Init prend un paramètre. Vous devez indiquer quelles parties de la SDL vous chargez.

Ah bon, la SDL est composée de plusieurs parties ?

Eh oui
Il y a une partie de la SDL qui gère l'affichage à l'écran, une autre qui gère le son etc etc...

La SDL met à votre disposition plusieurs constantes pour que vous puissiez indiquer quelle partie vous avez besoin d'utiliser dans votre programme :

Constante	Description
SDL_INIT_VIDEO	Charge le système d'affichage (vidéo). C'est la partie que nous chargerons le plus souvent.
SDL_INIT_AUDIO	Charge le système de son. Vous permettra donc par exemple de jouer de la musique.
SDL_INIT_CDROM	Charge le système de CD-Rom. Vous permettra de manipuler votre lecteur de CD-Rom
SDL_INIT_JOYSTICK	Charge le système de gestion du joystick.
SDL_INIT_TIMER	Charge le système de timer. Cela vous permet de gérer le temps dans votre programme (très pratique).
SDL_INIT_EVERYTHING	Charge tous les systèmes listés ci-dessus à la fois.

Si vous appelez la fonction comme ceci :

Code : C

```c
SDL_Init(SDL_INIT_VIDEO);
```

... alors le système vidéo sera chargé et vous pourrez ouvrir une fenêtre, dessiner dedans etc.
En fait, tout ce que vous faites c'est envoyer un nombre à SDL_Init à l'aide d'une constante. Vous ne savez pas de quel nombre il s'agit, et justement c'est ça qui est bien. Vous avez juste besoin d'écrire la constante, c'est plus facile à lire et à retenir

La fonction SDL_Init regardera le nombre qu'elle reçoit, et en fonction de cela elle saura quels systèmes elle doit charger.

Maintenant si vous faites :

Code : C

```c
SDL_Init(SDL_INIT_EVERYTHING);
```

... vous chargez tous les sytèmes de la SDL. Ne faites cela que si vous avez vraiment besoin de tout, il est inutile de surcharger votre ordinateur en chargeant des choses dont vous ne vous servirez pas.

Supposons que je veuille charger l'audio et la vidéo seulement. Dois-je utiliser SDL_INIT_EVERYTHING ?

Vous n'allez pas utiliser SDL_INIT_EVERYTHING juste parce que vous avez besoin de 2 systèmes, pauvres fous
On peut combiner les options à l'aide du symbole | (la barre verticale).

Code : C

```c
// Chargement de la vidéo et de l'audio
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO);
```

Vous pouvez aussi en combiner 3 sans problèmes :

Code : C

```
// Chargement de la vidéo, de l'audio et du timer
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER);
```

Ces "options" que l'on envoie à SDL_Init sont aussi appelées *flags*. C'est quelque chose que vous rencontrerez assez souvent.
Retenez bien qu'on utilise la barre verticale "|" pour combiner les options. Ca agit un peu comme un opérateur d'addition (d'ailleurs le + peut généralement être utilisé à la place, mais il est préférable d'utiliser le symbole | qui marche dans tous les cas).

SDL_Quit : arrêt de la SDL

La fonction SDL_Quit est super simple à utiliser vu qu'elle ne prend pas de paramètre :

Code : C

```
SDL_Quit();
```

Tous les systèmes initialisés seront arrêtés et libérés de la mémoire.
Bref, c'est un moyen de quitter la SDL proprement. A faire à la fin de votre programme.

En résumé...

En résumé, voici à quoi va ressembler votre programme :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO); // Démarrage de la SDL (ici : chargement du système vidéo)

    /*
    La SDL est chargée.
    Vous pouvez mettre ici le contenu de votre programme
    */

    SDL_Quit(); // Arrêt de la SDL (libération de la mémoire).

    return 0;
}
```

Bien entendu, aucun programme "sérieux" ne tiendra dans le main. Ce que je fais là est schématique. Dans la réalité, votre main contiendra certainement plein d'appels à des fonctions qui feront elles aussi plein d'appels à d'autres fonctions.
Ce qui compte au final, c'est que la SDL soit chargée au début et qu'elle soit fermée à la fin quand vous n'en avez plus besoin.

Gérer les erreurs

La fonction SDL_Init renvoie une valeur :

- -1 en cas d'erreur
- 0 si tout s'est bien passé

Vous n'y êtes pas obligés, mais vous pouvez vérifier la valeur retournée par SDL_Init. Ca peut être un bon moyen de traiter les erreurs de votre programme et donc de vous aider à résoudre vos erreurs.

Excellente question ! On n'a plus de console maintenant, donc comment faire pour stocker / afficher des messages d'erreurs ?

2 possibilités :

- Soit on modifie les options du projet pour qu'il réaffiche la console. On pourra alors faire des *printf*
- Soit on écrit dans un fichier les erreurs. On utilisera *fprintf*.

J'ai choisi d'écrire dans un fichier. Cependant, écrire dans un fichier implique de faire un fopen, un fclose... bref c'est un peu moins facile qu'un printf
Heureusement, il y a une solution plus simple : **utiliser la sortie d'erreur standard.**

Il y a une variable *stderr* qui est définie par stdio.h. Cette variable est automatiquement créée au début du programme et fermée à la fin. Vous n'avez donc pas besoin de faire de fopen ou de fclose.
Du coup, vous pouvez faire un fprintf sur stderr sans utiliser fopen ou fclose :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

int main(int argc, char *argv[])
{
    if (SDL_Init(SDL_INIT_VIDEO) == -1) // Démarrage de la SDL. Si erreur alors...
    {
        fprintf(stderr, "Erreur d'initialisation de la SDL : %s\n", SDL_GetError()); //
Ecriture de l'erreur
        exit(EXIT_FAILURE); // On quitte le programme
    }


    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Quoi de neuf dans ce code ? 2 choses :

- On écrit dans stderr notre erreur. Le %s permet de laisser la SDL indiquer les détails de l'erreur : la fonction **SDL_GetError()** renvoie en effet la dernière erreur de la SDL.

  Sous Windows, si vous avez écrit quelque chose dans stderr, alors un fichier `stderr.txt` sera créé dans le

même dossier que votre exécutable. Vous pourrez donc l'ouvrir pour lire l'erreur.
Si vous êtes sous un autre OS, cela dépend de l'endroit où sont stockées les erreurs habituellement.

- On quitte en utilisant exit() (bon ça rien de nouveau). Toutefois, vous aurez remarqué que j'utilise une constante (EXIT_FAILURE) pour indiquer la valeur que renvoie le programme. De plus, à la fin j'utilise EXIT_SUCCESS au lieu de 0.
  Qu'est-ce que j'ai changé ? En fait j'améliore petit à petit nos codes sources pour les rendre plus "pros". En effet, le nombre qui signifie "erreur" par exemple peut être différent selon les ordinateurs ! Cela dépend là encore de l'OS.
  Pour pallier ce problème, stdlib.h nous fournit 2 constantes (des defines) :

  - **EXIT_FAILURE** : valeur à renvoyer en cas d'échec du programme
  - **EXIT_SUCCESS** : valeur à renvoyer en cas de réussite du programme.

  En utilisant ces constantes au lieu de nombres, vous êtes sûrs de renvoyer une valeur correcte quel que soit l'OS.
  Pourquoi ? Parce que le fichier stdlib.h change selon l'OS sur lequel vous êtes, donc les valeurs des constantes sont adaptées. Votre code source, lui, n'a pas besoin de changer ! C'est ce qui rend le langage C compatible avec tous les OS pour peu que vous programmiez correctement

  Cela n'a pas de grandes conséquences pour nous pour le moment, mais c'est plus "sérieux" d'utiliser ces constantes. C'est donc ce que nous ferons à partir de maintenant

Ouverture d'une fenêtre

Bon, la SDL est initialisée et fermée correctement maintenant
La prochaine étape, si vous le voulez bien (et je suis sûr que vous le voulez bien), c'est l'ouverture d'une fenêtre !

Pour commencer déjà, assurez-vous d'avoir un main qui ressemble à ceci :

Code : C

```c
int main(int argc, char *argv[])
{
    if (SDL_Init(SDL_INIT_VIDEO) == -1)
    {
        fprintf(stderr, "Erreur d'initialisation de la SDL");
        exit(EXIT_FAILURE);
    }



    SDL_Quit();

    return EXIT_SUCCESS;
}
```

(cela devrait être le cas si vous avez bien suivi le début du chapitre)
Pour le moment donc, on initialise juste la vidéo (SDL_INIT_VIDEO), c'est tout ce qui nous intéresse.

Choix du mode vidéo

La première chose à faire après SDL_Init() c'est indiquer le mode vidéo que vous voulez utiliser, c'est-à-dire la résolution, le nombre de couleurs et quelques autres options.

On va utiliser pour cela la fonction **SDL_SetVideoMode()** qui prend 4 paramètres :

- La largeur de la fenêtre désirée (en pixels)
- La hauteur de la fenêtre désirée (en pixels)
- Le nombre de couleurs affichables (en bits / pixel)
- Des options (des *flags*)

Pour la largeur et la hauteur de la fenêtre, je crois que je ne vais pas vous faire l'affront de vous expliquer ce que c'est
Par contre, les 2 paramètres suivants sont plus intéressants.

- **Le nombre de couleurs** : c'est le nombre maximal de couleurs affichables dans votre fenêtre. Si vous jouez aux jeux vidéo, vous devriez avoir l'habitude de cela. Une valeur de 32 bits/pixel permet d'afficher des milliards de couleurs (c'est le maximum). C'est cette valeur que nous utiliserons le plus souvent car désormais tous les ordinateurs gèrent les couleurs en 32 bits.

  Sachez aussi que vous pouvez mettre des valeurs plus faibles comme 16 bits/pixel (65536 couleurs), ou 8 bits/pixel (256 couleurs). Cela peut être utile surtout si vous faites un programme pour un petit appareil genre PDA ou téléphone portable.

- **Les options** : comme pour SDL_Init on doit utiliser des flags pour définir des options. Voici les principaux flags que vous pouvez utiliser (et combiner avec le symbole "|") :
  - **SDL_HWSURFACE** : les données seront chargées dans la mémoire vidéo, c'est-à-dire dans la mémoire de votre carte 3D. Avantage : cette mémoire est plus rapide. Défaut : il y a en général moins d'espace dans cette mémoire que dans l'autre (SDL_SWSURFACE).
  - **SDL_SWSURFACE** : les données seront chargées dans la mémoire système (c'est-à-dire la RAM à priori). Avantage : il y a plein de place dans cette mémoire. Défaut : c'est moins rapide et moins optimisé.
  - **SDL_RESIZABLE** : la fenêtre sera redimensionnable. Par défaut elle ne l'est pas.
  - **SDL_NOFRAME** : la fenêtre n'aura pas de barre de titre ni de bordure.
  - **SDL_FULLSCREEN** : mode plein écran. Dans ce mode, aucune fenêtre n'est ouverte. Votre programme prendra toute la place à l'écran, en changeant automatiquement la résolution de votre écran au besoin.
  - **SDL_DOUBLEBUF** : mode double buffering. C'est une technique très utilisée dans les jeux 2D qui permet de faire en sorte que les déplacements des objets à l'écran soient fluides (sinon ça scintille et c'est moche). Je vous expliquerai les détails de cette technique très intéressante plus loin.

Donc, si je fais :

Code : C

```
SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
```

Cela ouvre une fenêtre de taille 640*480 en 32 bits/pixel (milliards de couleurs) qui sera chargée en mémoire vidéo (c'est la plus rapide, on préfèrera utiliser celle-là).

Autre exemple, si je fais :

Code : C

```
SDL_SetVideoMode(400, 300, 32, SDL_HWSURFACE | SDL_RESIZABLE | SDL_DOUBLEBUF);
```

Cela ouvre une fenêtre redimensionnable de taille initiale 400x300 (32 bits/pixel) en mémoire vidéo, avec le double buffering activé.

Voici un premier code source très simple (j'ai volontairement enlevé la gestion d'erreur) que vous pouvez essayer :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO);

    SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);

    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Testez.
Que se passe-t-il ? La fenêtre apparaît et disparaît à la vitesse de la lumière. En effet, la fonction SDL_SetVideoMode est immédiatement suivie de SDL_Quit, donc tout s'arrête immédiatement.

Mettre en pause le programme

Comment faire pour faire en sorte que la fenêtre se maintienne ?

Il faut faire comme le font tous les programmes, que ce soit des jeux vidéo ou autre : une boucle infinie. En effet, en faisant une bête boucle infinie on empêche notre programme de s'arrêter. Le problème est que cette solution est trop efficace car du coup il n'y a pas de moyen d'arrêter le programme (à part un CTRL+ALT+SUPPR à la rigueur mais c'est bourrin).
Voici un code à ne pas tester, je vous le donne juste à titre explicatif :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO);

    SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);

    while(1);

    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Vous reconnaissez le `while(1);` : c'est la boucle infinie. Comme 1 signifie vrai (rappelez-vous les booléens), la boucle est toujours vraie et tourne en rond indéfiniment sans qu'il y ait moyen de l'arrêter. Ce n'est donc pas une très bonne solution

Pour mettre en pause notre programme afin de pouvoir admirer notre beeelle fenêtre sans faire de boucle interminable, on va utiliser une petite fonction à moi :

Code : C

```
void pause()
{
    int continuer = 1;
    SDL_Event event;

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
        }
    }
}
```

Je ne vous explique pas le détail de cette fonction pour le moment. C'est volontaire, car cela fait appel à la gestion des évènements. Je vous l'expliquerai dans les prochains chapitres. Si je vous explique tout à la fois maintenant vous risquez de tout mélanger
Faites donc pour l'instant confiance à ma fonction de pause, nous ne tarderons pas à l'expliquer.

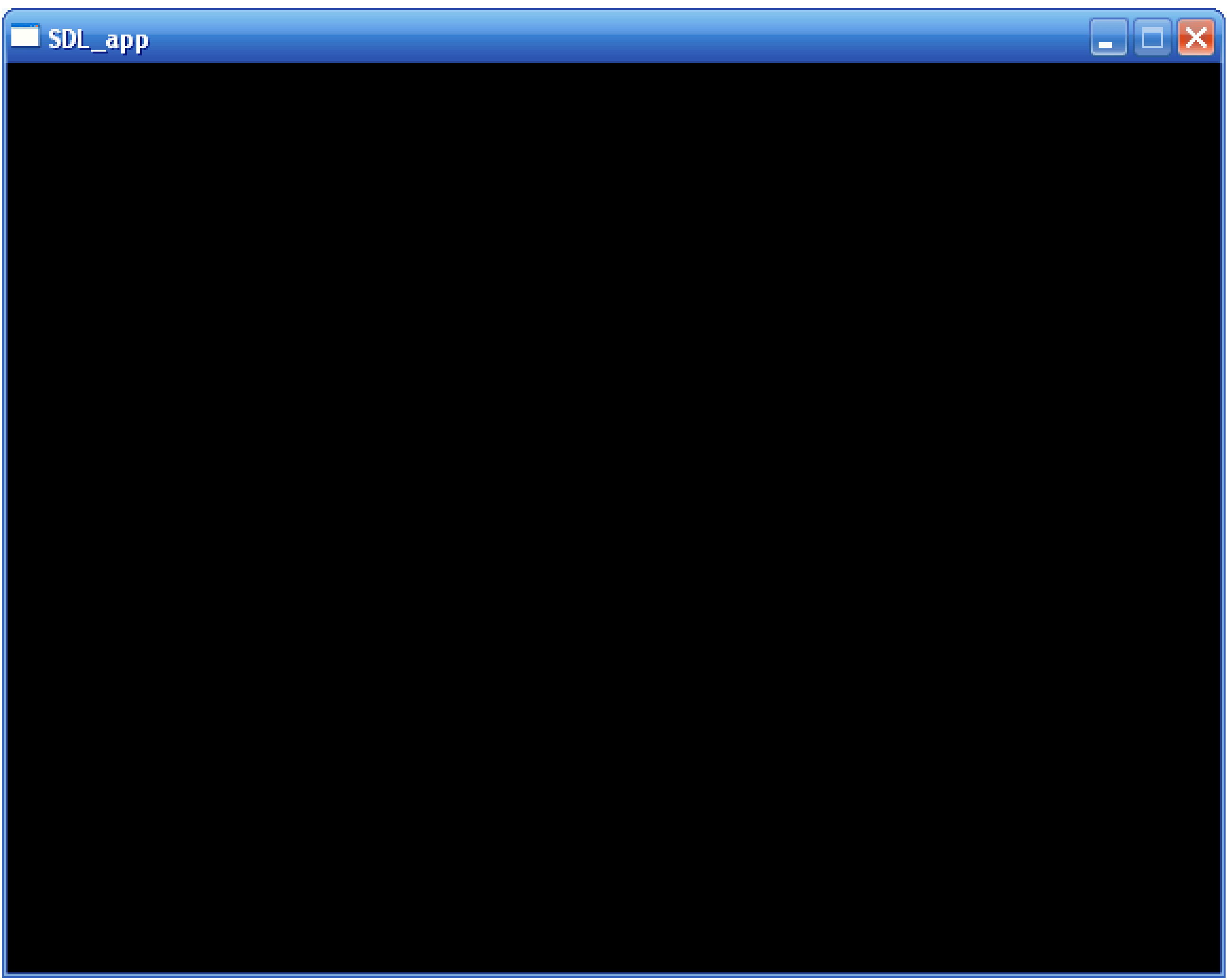
Voici un code source complet et correct que vous pouvez (enfin !) tester :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

void pause();

int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO); // Initialisation de la SDL

    SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE); // Ouverture de la fenêtre

    pause(); // Mise en pause du programme

    SDL_Quit(); // Arrêt de la SDL

    return EXIT_SUCCESS; // Fermeture du programme
}

void pause()
{
    int continuer = 1;
    SDL_Event event;

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
        }
    }
}
```

Vous remarquerez que j'ai mis le prototype de ma fonction pause() en haut.
Je fais appel à la fonction pause() qui fait une boucle infinie un peu plus intelligente que tout à l'heure. Cette boucle s'arrêtera en effet si vous cliquez sur la croix pour fermer la fenêtre

Voici à quoi devrait ressembler la fenêtre que vous avez sous les yeux (ici une fenêtre 640x480) :



Pfiou ! Nous y sommes enfin arrivés !

Si vous voulez vous pouvez mettre le flag "redimensionnable" pour autoriser le redimensionnement de votre fenêtre. Toutefois, dans la plupart des jeux on préfère avoir une fenêtre de taille fixe (c'est plus simple à gérer !), donc nous garderons notre fenêtre fixe pour le moment

Attention au mode plein écran (SDL_FULLSCREEN) et au mode sans bordure (SDL_NOFRAME). Comme il n'y a pas de barre de titre dans ces 2 modes, vous ne pourrez pas fermer le programme et vous serez alors obligés d'utiliser la commande CTRL+ALT+SUPPR.
Attendez d'apprendre à manipuler les évènements SDL (dans quelques chapitres) et vous pourrez alors coder un moyen de sortir de votre programme avec une technique un peu moins hardcore que le CTRL+ALT+SUPPR

Changer le titre de la fenêtre

Pour le moment, notre fenêtre a un titre par défaut (SDL_app sur ma capture d'écran).
Que diriez-vous de changer cela ?

C'est extrêmement simple, il suffit d'utiliser la fonction **SDL_WM_SetCaption**.
Cette fonction prend 2 paramètres. Le premier est le titre que vous voulez donner à la fenêtre, le second est le titre que vous voulez donner à l'icône.

Contrairement à ce qu'on pourrait croire, donner un titre à l'icône ne correspond pas à charger une icône qui s'afficherait dans la barre de titre en haut à gauche. En fait, le titre de l'icône ne s'affiche nulle part et je n'en ai

jamais vu l'utilité. Par conséquent j'envoie la valeur NULL à la fonction pour ne rien mettre.
Sachez qu'il est possible de changer l'icône de la fenêtre, mais nous verrons comment le faire dans le chapitre suivant seulement.

Voici donc le même main que tout à l'heure avec la fonction SDL_WM_SetCaption en plus :

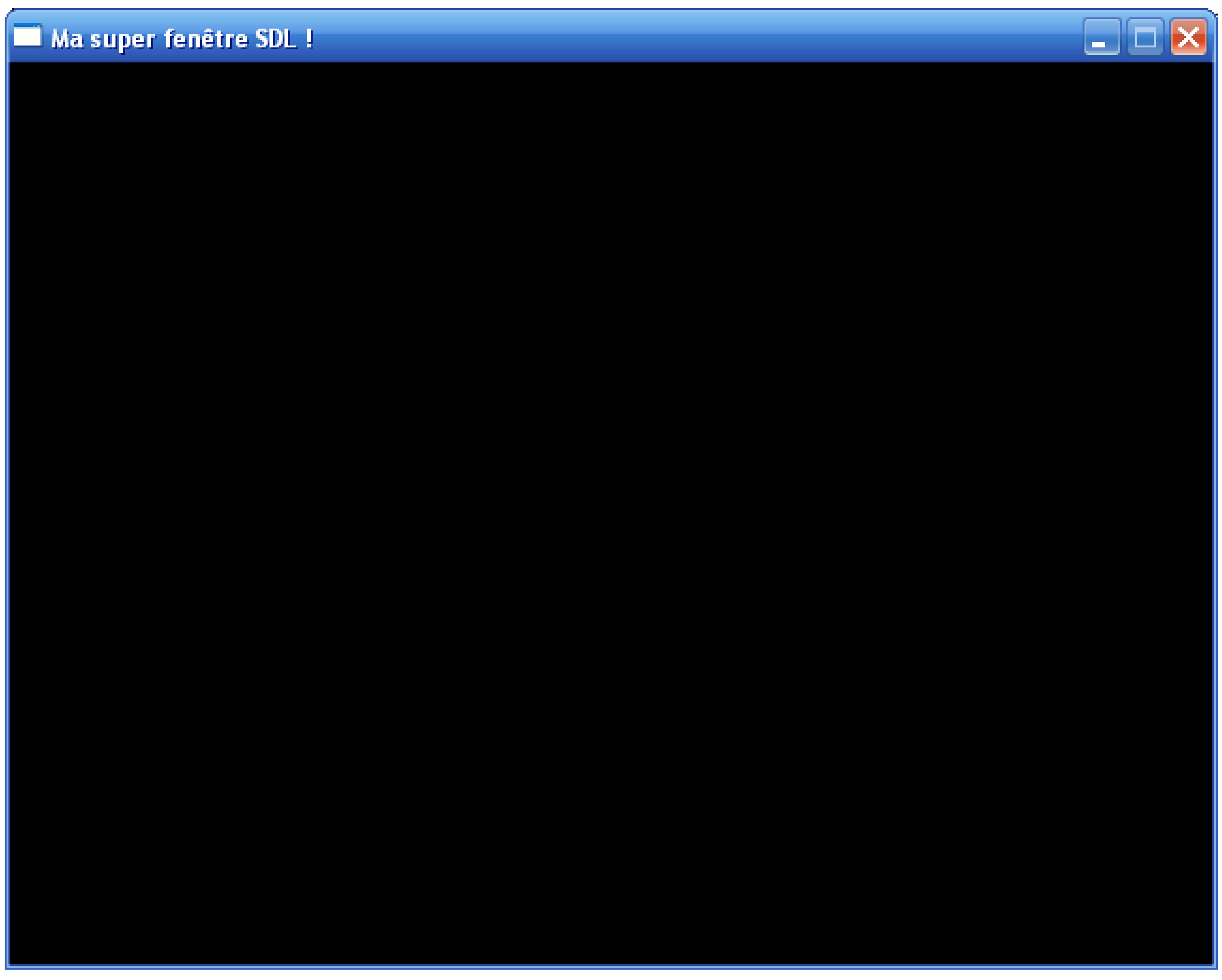
Code : C

```c
int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_VIDEO);

    SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);

    pause();

    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Vous aurez remarqué que j'ai mis NULL pour second paramètre pas très utile (nom de l'icône).

En C, vous êtes obligés d'indiquer tous les paramètres même si certains ne vous intéressent pas, quitte à envoyer NULL comme je l'ai fait ici.
Plus tard, si vous utilisez une librairie codée en C++, celle-ci pourra rendre certains paramètres de fonction facultatifs. C'est un petit + du C++

La fenêtre a maintenant un titre



MANIPULATION DES SURFACES

Pour le moment nous avons une fenêtre avec un fond noir. C'est la fenêtre de base.
Ce qu'on veut faire, c'est la remplir, c'est-à-dire "dessiner" dedans.

La SDL, je vous l'ai dit dans le chapitre précédent, est une librairie bas niveau. Cela veut dire qu'elle ne nous propose que des fonctions de base, très simples.
Et en effet, la seule forme que la SDL nous permet de dessiner, c'est le rectangle ! Tout ce que vous allez faire, c'est assembler des rectangles dans la fenêtre. On appelle ces rectangles **des surfaces**. La surface, c'est la brique de base de la SDL.

Il est possible bien sûr de dessiner d'autres formes, comme des cercles, des triangles etc... Mais il faudra écrire nous-mêmes des fonctions pour le faire, en dessinant pixel par pixel. C'est un peu compliqué, et de toute manière vous verrez que nous n'en aurons pas vraiment besoin dans la pratique

Votre première surface : l'écran

Dans tout programme SDL, il y a au moins une surface que l'on appelle généralement *ecran* (ou *screen* en anglais). C'est une surface qui correspond à toute la fenêtre, c'est-à-dire toute la zone noire de la fenêtre que vous voyez.

Dans notre code source, chaque surface sera mémorisée dans une variable de type **SDL_Surface**. Oui, c'est un type de variable créé par la SDL (une structure en l'occurence).

Comme la première surface que nous devons créer est l'écran, allons-y :

Code : C

```
SDL_Surface *ecran = NULL;
```

Vous remarquerez que je crée un pointeur. Pourquoi je fais ça ? Parce que c'est la SDL qui va allouer de l'espace en mémoire pour notre surface. Une surface n'a en effet pas toujours la même taille, et la SDL est obligée de faire une allocation dynamique pour nous (ici ça dépendra de la taille de la fenêtre que vous avez ouverte).

Je ne vous l'ai pas dit tout à l'heure, mais la fonction SDL_SetVideoMode renvoie une valeur ! Elle renvoie un pointeur sur la surface de l'écran qu'elle a créée en mémoire pour nous.
Cool, on va donc pouvoir récupérer ce pointeur dans *ecran* :

Code : C

```
ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
```

Notre pointeur peut valoir :

- **NULL** : ecran vaut NULL si la SDL_SetVideoMode n'a pas réussi à charger le mode vidéo demandé. Cela arrive si vous demandez une trop grande résolution ou un trop grand nombre de couleurs que ne supporte pas votre ordinateur.
- **Une autre valeur** : si la valeur est différente de NULL, c'est que la SDL a pu allouer la surface en mémoire, donc que tout est bon !

Il serait bien ici de gérer les erreurs, comme on a appris à le faire tout à l'heure pour l'initialisation de la SDL. Voici donc notre main avec la gestion de l'erreur pour SDL_SetVideoMode :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL; // Le pointeur qui va stocker la surface de l'écran


    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE); // On tente d'ouvrir une
fenêtre
    if (ecran == NULL) // Si l'ouverture a échoué, on écrit l'erreur et on arrête
    {
        fprintf(stderr, "Impossible de charger le mode vidéo : %s\nq", SDL_GetError());
        exit(EXIT_FAILURE);
    }

    SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);

    pause();

    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Le message que nous laissera SDL_GetError() nous sera très utile pour savoir ce qui a planté.
Petite anecdote : une fois je me suis planté en voulant faire du plein écran. Au lieu de demander une résolution de 1024*768, j'ai écrit 10244*768. Je ne comprenais pas au départ pourquoi ça ne voulait pas charger, car je ne voyais pas le double 4 dans mon code (oui je devais être un peu fatigué).
Un petit coup d'oeil au fichier stderr.txt, et j'ai tout de suite compris que c'était ma résolution qui avait été rejetée (tiens comme c'est curieux)

Colorer une surface

Il n'y a pas 36 façons de remplir une surface... En fait, il y en a 2 :

- Soit vous remplissez la surface avec une couleur unie
- Soit vous remplissez la surface en chargeant une image

Il est aussi possible de dessiner pixel par pixel dans la surface mais c'est assez compliqué, nous ne le verrons pas ici.

Nous allons dans un premier temps voir comment remplir une surface avec une couleur unie. Dans le chapitre suivant, nous apprendrons à charger une image.

La fonction qui permet de colorer une surface avec une couleur unie s'appelle **SDL_FillRect** (FillRect = "remplir rectangle" en anglais). Elle prend 3 paramètres. Dans l'ordre :

- Un pointeur sur la surface dans laquelle on doit dessiner (par exemple *ecran*).
- La partie de la surface qui doit être remplie. Si vous voulez remplir toute la surface (et c'est ce qu'on veut faire), envoyez NULL.
- La couleur à utiliser pour remplir la surface.

En résumé :

Code : C

```
SDL_FillRect(surface, NULL, couleur);
```

La gestion des couleurs en SDL

En SDL, une couleur est stockée dans un nombre de type Uint32.

Si c'est un nombre, pourquoi ne pas avoir utilisé le type de variable int ou long tout simplement ?

La SDL est une librairie multiplateforme. Or, comme vous le savez maintenant, la taille occupée par un int ou un long peut varier selon votre OS. Pour s'assurer que le nombre occupera toujours la même taille en mémoire, la SDL a donc "inventé" des types pour stocker des entiers qui ont la même taille sur tous les systèmes.
Il y a par exemple :

- Uint32 : un entier de longueur 32 bits (soit 4 octets, quant on sait que 1 octet = 8 bits)
- Uint16 : un entier codé sur 16 bits (2 octets)
- Uint8 : un entier codé sur 8 bits (1 octet)

La SDL ne fait qu'un simple typedef qui changera selon l'OS que vous utilisez. Regardez de plus près le fichier SDL_types.h si vous êtes curieux.

On ne va pas s'attarder là-dessus plus longtemps car les détails de tout cela ne sont pas importants. Vous avez juste besoin de retenir que Uint32 est un type qui stocke un entier, comme un int.

Bon ok, mais comment je sais quel nombre je dois mettre pour la couleur verte, azur, gris foncé, jaune pâle etc. ?

Il existe une fonction qui sert à ça : **SDL_MapRGB**. Elle prend 4 paramètres :

- Le format des couleurs. Ce format dépend du nombre de bits / pixel que vous avez demandé avec SDL_SetVideoMode. Vous pouvez le récupérer, il est stocké dans la sous-variable ecran->format
- La quantité de rouge de la couleur
- La quantité de vert de la couleur
- La quantité de bleu de la couleur

Certains d'entre vous ne le savent peut-être pas, alors expliquons ce bazar. Toute couleur sur un ordinateur est construite à partir de 3 couleurs de base : le rouge, le vert et le bleu.
Chaque quantité peut varier de 0 (pas de couleur) à 255 (toute la couleur).
Donc, si on écrit :

Code : C

```
SDL_MapRGB(ecran->format, 255, 0, 0)
```

... on crée une couleur rouge. Il n'y a pas de vert ni de bleu.
Autre exemple, si on écrit :

Code : C

```
SDL_MapRGB(ecran->format, 0, 0, 255)
```

... cette fois c'est une couleur bleue.

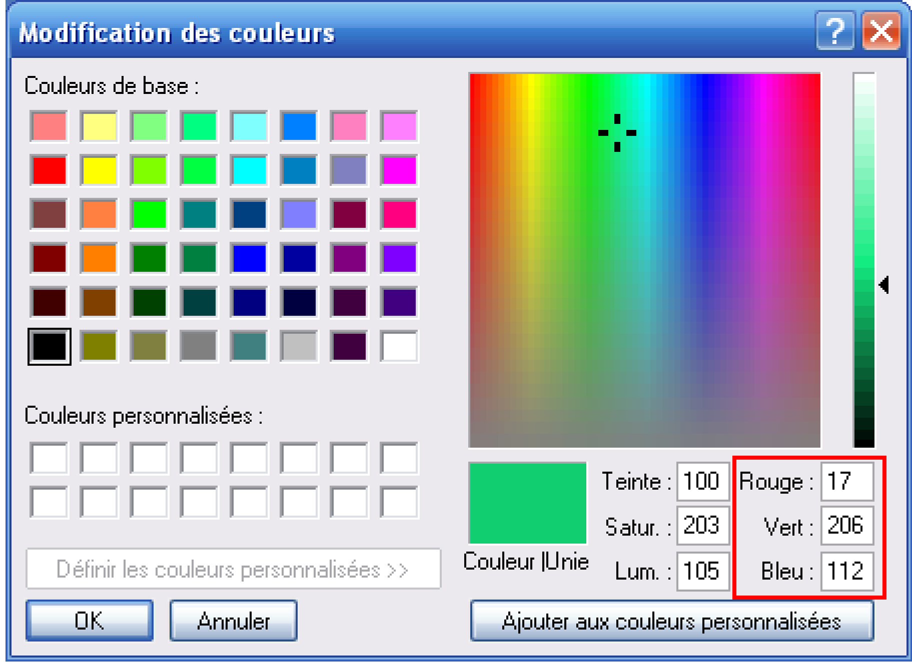
Code : C

```
SDL_MapRGB(ecran->format, 255, 255, 255)
```

... là il s'agit d'une couleur blanche (toutes les couleurs s'additionnent). Si vous voulez du noir, il faut mettre 0, 0, 0.

On ne peut que mettre du rouge, du vert, du bleu, du noir et du blanc ?

Non, c'est à vous de combiner intelligemment les quantités de couleurs
Pour vous aider, ouvrez par exemple le logiciel Paint. Allez dans le menu "Couleurs / Modifier les couleurs". Cliquez sur le bouton "Définir les couleurs personnalisées".
Là, choisissez la couleur qui vous intéresse :



Les composantes de la couleur sont affichées en bas à droite. Ici, ce bleu-vert que j'ai sélectionné se crée à l'aide de 17 de rouge, 206 de vert et 112 de bleu

Coloration de l'écran

La fonction SDL_MapRGB renvoie un Uint32 qui correspond à la couleur demandée.
On peut donc créer une variable bleuVert qui contiendra le code de la couleur bleu-vert :

Code : C

```
Uint32 bleuVert = SDL_MapRGB(ecran->format, 17, 206, 112);
```

Toutefois, ce n'est pas toujours la peine de passer par une variable pour stocker la couleur (à moins que vous en ayez besoin très souvent dans votre programme).
Vous pouvez tout simplement envoyer directement la couleur donnée par SDL_MapRGB à SDL_FillRect.

Si on veut remplir notre écran de bleu-vert, on peut donc écrire :

Code : C

```
SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 17, 206, 112));
```

On combine 2 fonctions, mais ça ne pose aucun problème au langage C

Mise à jour de l'écran

Nous y sommes presque.
Toutefois il manque encore une petite chose : demander la mise à jour de l'écran. En effet, l'instruction SDL_FillRect colorie bien l'écran mais cela ne se fait que dans la mémoire. Il faut ensuite demander à l'ordinateur de mettre à jour l'écran avec les nouvelles données.

Pour cela, on va utiliser la fonction **SDL_Flip**, donc nous reparlerons plus longuement plus tard dans le cours.
Cette fonction prend un paramètre : l'écran

Code : C

```
SDL_Flip(ecran); /* Mise à jour de l'écran */
```

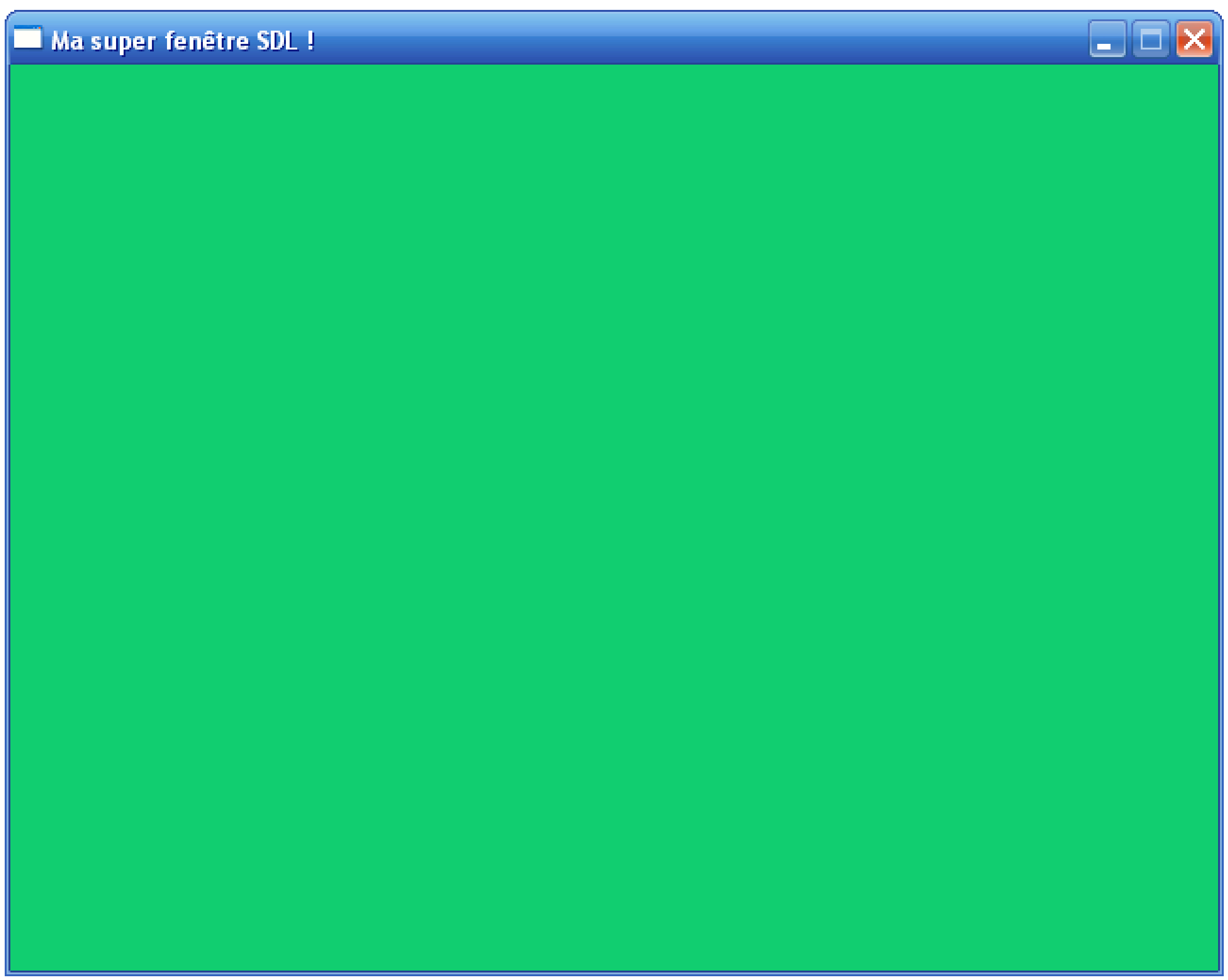
On résume !

Voici une fonction main() qui crée une fenêtre avec un fond bleu-vert :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);

    // Coloration de la surface ecran en bleu-vert
    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 17, 206, 112));

    SDL_Flip(ecran); /* Mise à jour de l'écran avec sa nouvelle couleur */

    pause();

    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Résultat :



Dessiner une nouvelle surface à l'écran

Bon, c'est bien, mais ne nous arrêtons pas en si bon chemin
Pour le moment on n'a qu'une seule surface, c'est l'écran. On aimerait pouvoir dessiner dessus, c'est-à-dire "coller" des surfaces avec une autre couleur par-dessus.

Pour commencer, nous allons avoir besoin de créer une autre variable de type SDL_Surface pour cette nouvelle surface :

Code : C

```c
SDL_Surface *rectangle = NULL;
```

Nous devons ensuite demander à la SDL de nous allouer de l'espace en mémoire pour cette nouvelle surface.
Pour l'écran, nous avons utilisé SDL_SetVideoMode. Toutefois, cette fonction ne marche que pour l'écran (la surface globale), on ne va pas recréer une fenêtre pour chaque rectangle que l'on veut dessiner

Il existe donc une autre fonction pour créer une surface : SDL_CreateRGBSurface. C'est celle que nous utiliserons à chaque fois que nous voulons créer une surface unie comme ici.

Cette fonction prend... beaucoup de paramètres (8 !) D'ailleurs, peu d'entre eux nous intéressent pour l'instant, donc je vais éviter de vous détailler ceux qui ne nous serviront pas de suite.
Comme en C nous sommes obligés d'indiquer tous les paramètres (les paramètres facultatifs n'existent qu'en C++), nous enverrons la valeur 0 quand le paramètre ne nous intéresse pas.

Regardons de plus près les 4 premiers paramètres, les plus intéressants :

- Une liste de flags (des options). Vous avez le choix entre :
  - **SDL_HWSURFACE** : la surface sera chargée en mémoire vidéo. Il y a moins d'espace dans cette mémoire que dans la mémoire système (quoique, avec les cartes 3D qu'on sort de nos jours...), mais cette mémoire est plus optimisée et accélérée.
  - **SDL_SWSURFACE** : la surface sera chargée en mémoire système où il y a beaucoup de place, mais cela obligera votre processeur à faire plus de calculs. Si vous aviez chargé la surface en mémoire vidéo, c'est la carte 3D qui aurait fait la plupart des calculs.
- La largeur de la surface (en pixels)
- La hauteur de la surface (en pixels)
- Le nombre de couleurs (en bits / pixel)

Voici donc comment on alloue notre nouvelle surface en mémoire :

Code : C

```c
rectangle = SDL_CreateRGBSurface(SDL_HWSURFACE, 220, 180, 32, 0, 0, 0, 0);
```

Les 4 derniers paramètres sont mis à 0 comme je vous l'ai dit car ils ne nous intéressent pas.

Comme notre surface a été allouée manuellement, il faudra penser à la libérer de la mémoire avec la fonction SDL_FreeSurface(), à utiliser juste avant SDL_Quit() :

Code : C

```c
SDL_FreeSurface(rectangle);
SDL_Quit();
```

La surface *ecran* n'a pas besoin d'être libérée avec SDL_FreeSurface(), cela est fait lors de SDL_Quit().

On peut maintenant colorer notre nouvelle surface en la remplissant par exemple de blanc :

Code : C

```c
SDL_FillRect(rectangle, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
```

Coller la surface à l'écran

Allez, c'est presque fini courage
Notre surface est prête, mais si vous testez le programme vous verrez qu'elle ne s'affichera pas ! En effet, la SDL n'affiche à l'écran que la surface *ecran*. Pour que l'on puisse voir notre notre nouvelle surface, il va falloir **blitter la surface**, c'est-à-dire la coller sur l'écran. On utilisera pour cela la fonction **SDL_BlitSurface**.

Cette fonction attend :

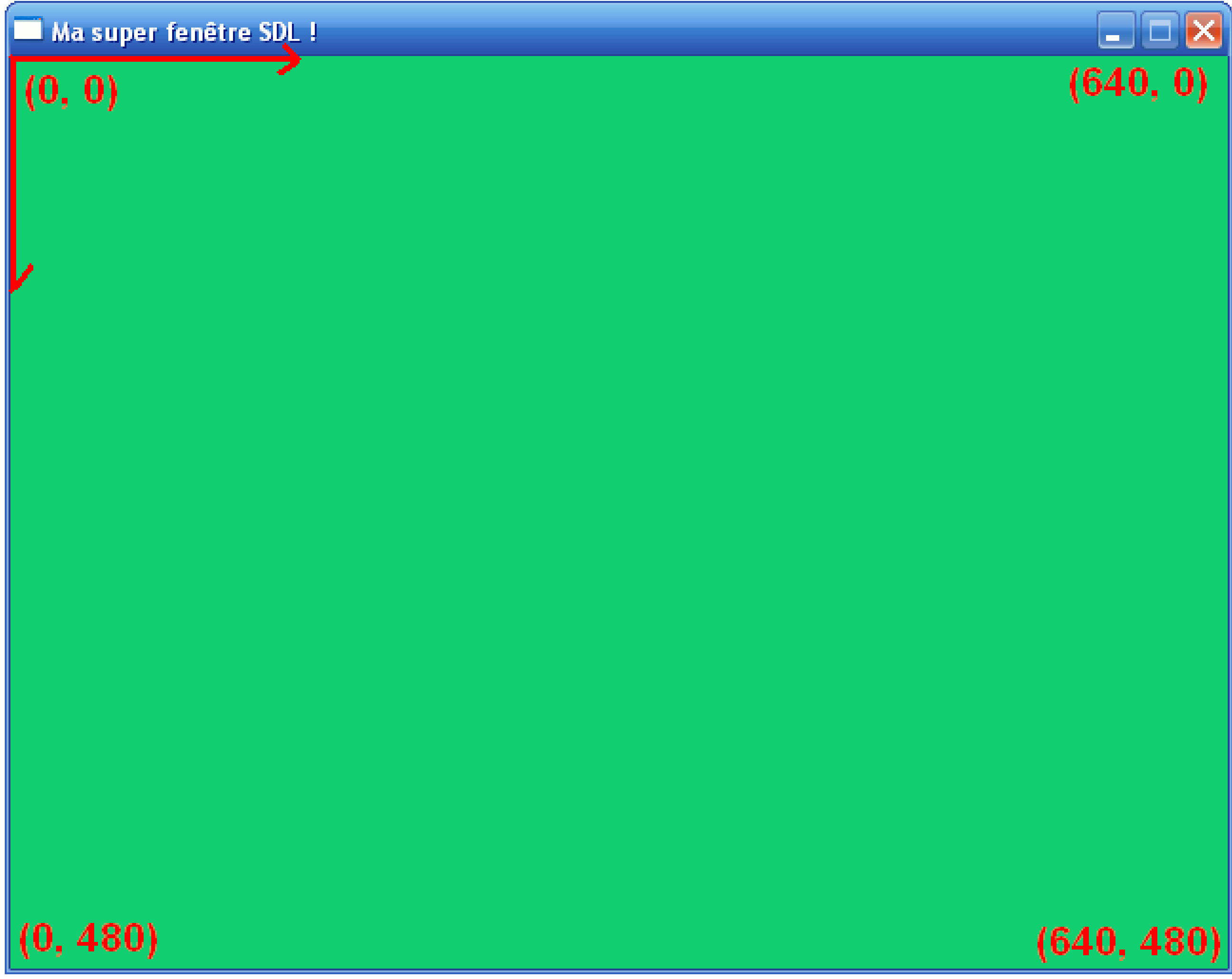
- La surface à coller (ici, ce sera *rectangle*)
- Une information sur la partie de la surface à coller (facultative). Ca ne nous intéresse pas car on veut coller toute la surface, donc on enverra NULL
- La surface sur laquelle on doit coller, c'est-à-dire *ecran* dans notre cas.
- Un pointeur sur une variable contenant des coordonnées. Ces coordonnées indiquent où devra être collée notre surface sur l'écran (la position quoi)

Pour indiquer les coordonnées, on doit utiliser une variable de type SDL_Rect.
C'est une structure qui contient plusieurs sous-variables, dont 2 qui nous intéressent ici :

- **x** : l'abscisse
- **y** : l'ordonnée

Il faut savoir que le point de coordonnées (0, 0) est situé tout en haut à gauche.
En bas à droite, le point a les coordonnées (640, 480) si vous avez ouvert une fenêtre de taille 640x480 comme moi.

Aidez-vous de ce schéma pour vous situer :



Si vous avez déjà fait des maths une fois dans votre vie vous ne devriez pas être trop perturbé

Créons donc une variable position. On va mettre x et y à 0 pour coller la surface en haut à gauche de l'écran :

Code : C

```
SDL_Rect position;

position.x = 0;
position.y = 0;
```

Maintenant qu'on a notre variable *position*, on peut blitter notre rectangle sur l'écran :

Code : C

```
SDL_BlitSurface(rectangle, NULL, ecran, &position);
```

(remarquez le symbole &, car il faut envoyer l'adresse de notre variable position).

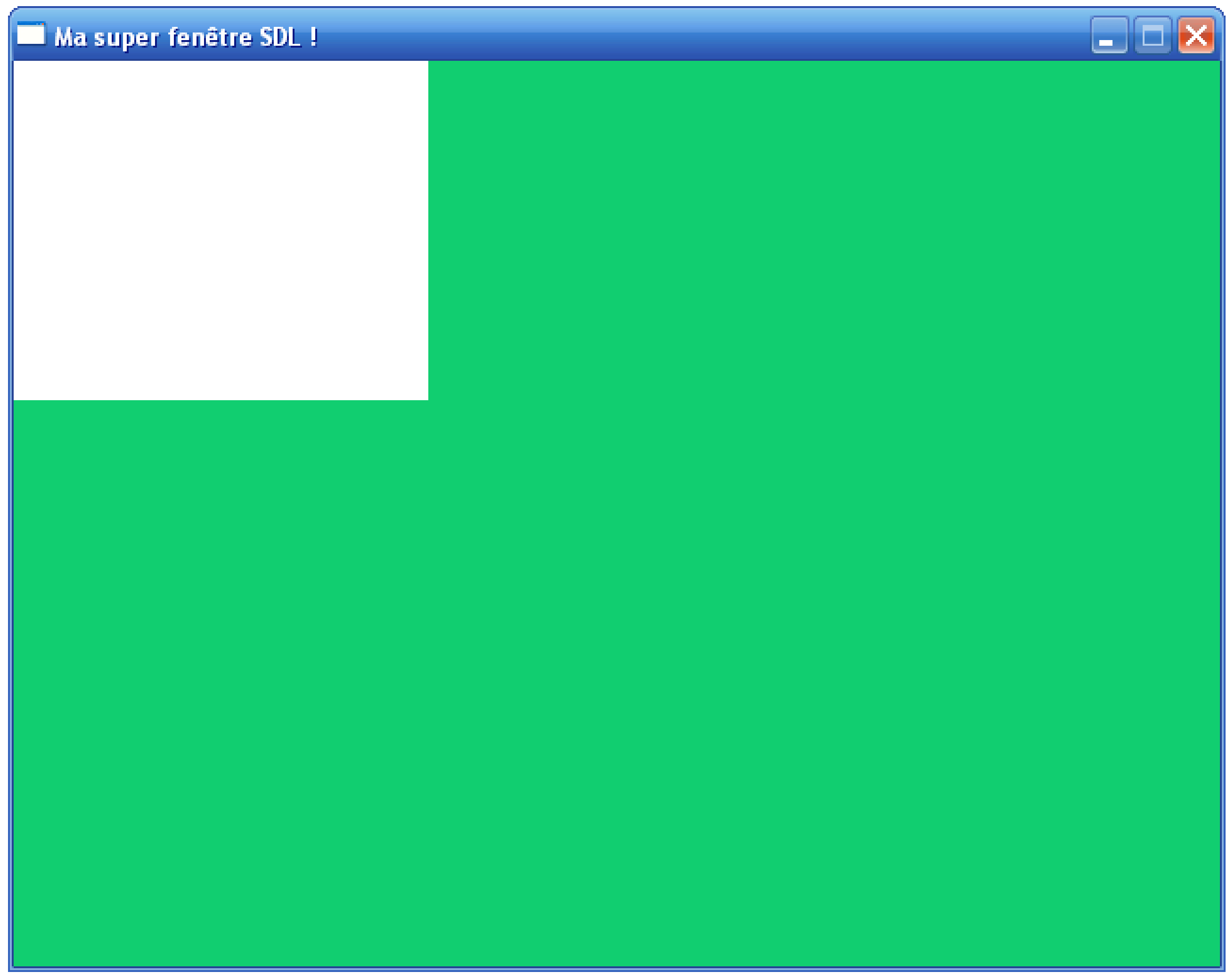
En résumé...

Je crois qu'un petit code pour résumer tout ça ne sera pas superflu

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *rectangle = NULL;
    SDL_Rect position;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    rectangle = SDL_CreateRGBSurface(SDL_HWSURFACE, 220, 180, 32, 0, 0, 0, 0); //
Allocation de la surface
    SDL_WM_SetCaption("Ma super fenêtre SDL !", NULL);

    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 17, 206, 112));

    position.x = 0; // Les coordonnées de la surface seront (0, 0)
    position.y = 0;
    SDL_FillRect(rectangle, NULL, SDL_MapRGB(ecran->format, 255, 255, 255)); //
Remplissage de la surface avec du blanc
    SDL_BlitSurface(rectangle, NULL, ecran, &position); // Collage de la surface sur
l'écran

    SDL_Flip(ecran); // Mise à jour de l'écran

    pause();

    SDL_FreeSurface(rectangle); // Libération de la surface
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Et voilà le travail



Sympa non ?

Centrer la surface à l'écran

On sait afficher la surface en haut à gauche.
Il serait aussi facile de la mettre en bas à droite. Les coordonnées seraient (640 - 206, 480 - 112), car il faut retrancher la taille de notre rectangle pour qu'il s'affiche entièrement.

Mais... comment faire pour centrer le rectangle blanc ?
Si vous réfléchissez bien 2 secondes, c'est un simple petit calcul mathématique

Code : C

```c
position.x = (640 / 2) - (220 / 2); // La surface sera centrée
position.y = (480 / 2) - (180 / 2);
```

L'abscisse du rectangle sera la moitié de la largeur de l'écran (640 / 2). Mais, en plus de ça, il faut retrancher la moitié de la largeur du rectangle (220 / 2), car sinon ça ne sera pas parfaitement centré (essayez de ne pas le faire, vous verrez ce que je veux dire)
C'est la même chose pour l'ordonnée avec la hauteur de l'écran et du rectangle.

Résultat :

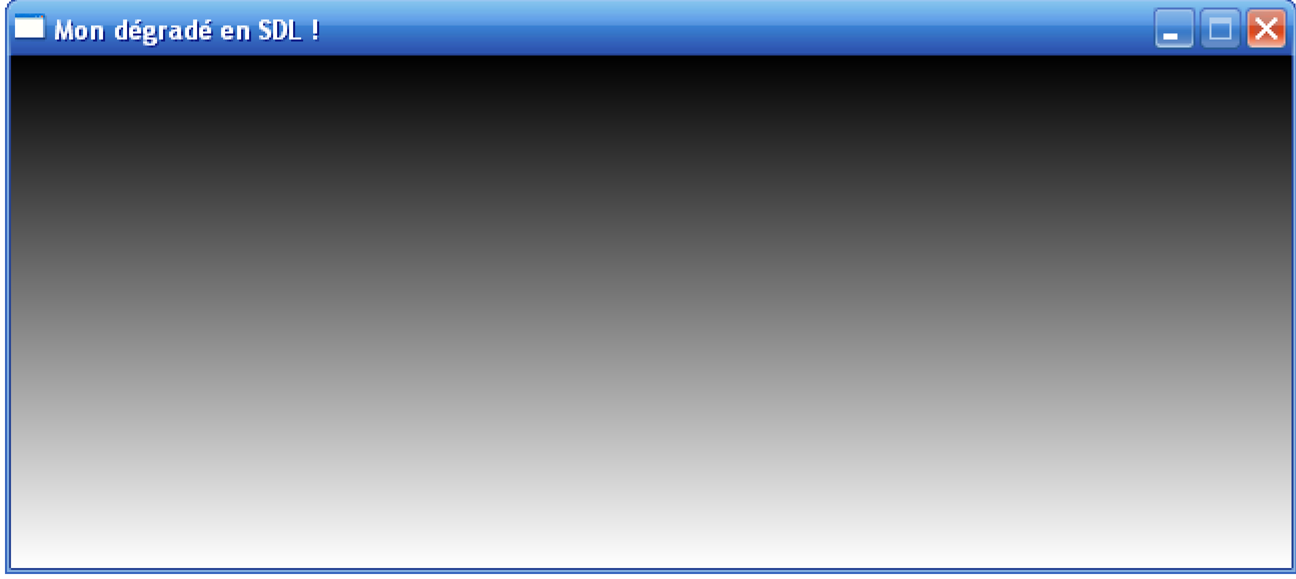
C'est pas magique, c'est mathématique !

(Exercice) Créer un dégradé

On va finir le chapitre par un petit exercice (corrigé) suivi d'une série d'autres exercices (non corrigés pour vous forcer à bosser)

L'exercice corrigé n'est vraiment pas difficile : on veut créer un dégradé vertical allant du noir au blanc.
Vous allez devoir créer 255 surfaces de 1 pixel de hauteur. Chacune aura une couleur différente, de plus en plus noire.

Voici ce que vous devez arriver à obtenir au final :



C'est mignon tout plein non ?
Et le pire c'est qu'il suffit de quelques petites boucles pour y arriver

Pour faire ça, on va devoir créer 256 surfaces (256 lignes) ayant les composantes rouge-vert-bleu suivantes :

Code : Autre

```
(0, 0, 0) // Noir
(1, 1, 1) // Gris très très proche du noir
(2, 2, 2) // Gris très proche du noir

...

(128, 128, 128) // Gris moyen (à 50%)

...

(253, 253, 253) // Gris très proche du blanc
(254, 254, 254) // Gris très très proche du blanc
(255, 255, 255) // Blanc
```

Tout le monde devrait avoir vu venir une boucle pour faire ça (on va pas faire 256 copier/collers, faut pas abuser !)
Vous allez devoir créer un tableau de type SDL_Surface* de 256 cases pour stocker chacune des lignes.

Allez au boulot, z'avez 5 minutes montre en main !

Correction !

Alors, facile ou facile ?

D'abord, il fallait créer notre tableau de 256 SDL_Surface*. On l'initialise à NULL :

Code : C

```
SDL_Surface *lignes[256] = {NULL};
```

On crée aussi une variable i dont on aura besoin pour nos for.

On change aussi la hauteur de la fenêtre pour qu'elle soit plus adaptée dans notre cas. On lui donne donc 256 pixels de hauteur (pour chacune des 256 lignes à afficher).

Ensuite, on fait un for pour allouer une à une chacune des 256 surfaces. Le tableau recevra 256 pointeurs vers chacune des surfaces créées :

Code : C

```
for (i = 0 ; i <= 255 ; i++)
    lignes[i] = SDL_CreateRGBSurface(SDL_HWSURFACE, 640, 1, 32, 0, 0, 0, 0); //
Allocation des 256 surfaces
```

Ensuite, on remplit et on blitte chacune de ces surfaces une par une.

Code : C

```
for (i = 0 ; i <= 255 ; i++)
{
    position.x = 0; // Les lignes sont à gauche (abscisse de 0)
    position.y = i; // La position verticale dépend du numéro de la ligne actuelle
    SDL_FillRect(lignes[i], NULL, SDL_MapRGB(ecran->format, i, i, i)); // Remplissage
    SDL_BlitSurface(lignes[i], NULL, ecran, &position); // Collage
}
```

Notez que j'utilise la même variable position tout le temps. Pas besoin d'en créer 256 en effet, car la variable ne sert que pour être envoyée à SDL_BlitSurface. On peut donc la réutiliser sans problème.
A chaque fois, je mets à jour l'ordonnée (y) pour blitter la ligne à la bonne hauteur. La couleur à chaque passage dans la boucle dépend de i (ce sera 0, 0, 0 la première fois, et 255, 255, 255 la dernière fois).

Mais pourquoi x est toujours à 0 ?
Comment se fait-il que toute la ligne soit remplie si x est tout le temps à 0 ?

Notre variable position indique à quel endroit est placé le coin en haut à gauche de notre surface (ici notre ligne). Elle n'indique pas la largeur de la surface, juste sa position sur l'écran.
Comme toutes nos lignes commencent à gauche de la fenêtre (le plus à gauche possible), on met une abscisse de 0.
Essayez de mettre une abscisse de 50 pour voir ce que ça fait : toutes les lignes seront décalées vers la droite.
Comme la surface fait 640 pixels de largeur, la SDL dessine 640 pixels vers la droite (de la même couleur) en partant des coordonnées indiquées dans la variable *position*.

Sur ce schéma je vous montre les coordonnées du point en haut à gauche de l'écran (position de la première ligne) et celui du point en bas à droite de l'écran (position de la dernière ligne) :

Mon dégradé en SDL !
(x = 0, y = 0)
(x = 0, y = 255)

Comme vous le voyez, de haut en bas l'abscisse ne change pas (x reste égal à 0 tout le long).

C'est seulement y qui change pour chaque nouvelle ligne, d'où le `position.y = i;`

Enfin, il ne faut pas oublier de libérer la mémoire pour chacune des 256 surfaces créées, le tout à l'aide d'une petite boucle bien entendu

Code : C

```
for (i = 0 ; i <= 255 ; i++) // N'oubliez pas de libérer chacune des 256 surfaces !
    SDL_FreeSurface(lignes[i]);
```

Résumé du main

Voici donc la fonction main au complet :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *lignes[256] = {NULL};
    SDL_Rect position;
    int i = 0;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 256, 32, SDL_HWSURFACE); // Hauteur de 256 pixels

    for (i = 0 ; i <= 255 ; i++)
        lignes[i] = SDL_CreateRGBSurface(SDL_HWSURFACE, 640, 1, 32, 0, 0, 0, 0); //
Allocation des 256 surfaces

    SDL_WM_SetCaption("Mon dégradé en SDL !", NULL);

    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));

    for (i = 0 ; i <= 255 ; i++)
    {
        position.x = 0; // Les lignes sont à gauche (abscisse de 0)
        position.y = i; // La position verticale dépend du numéro de la ligne actuelle
        SDL_FillRect(lignes[i], NULL, SDL_MapRGB(ecran->format, i, i, i)); // Remplissage
        SDL_BlitSurface(lignes[i], NULL, ecran, &position); // Collage
    }

    SDL_Flip(ecran);
    pause();

    for (i = 0 ; i <= 255 ; i++) // N'oubliez pas de libérer chacune des 256 surfaces !
        SDL_FreeSurface(lignes[i]);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

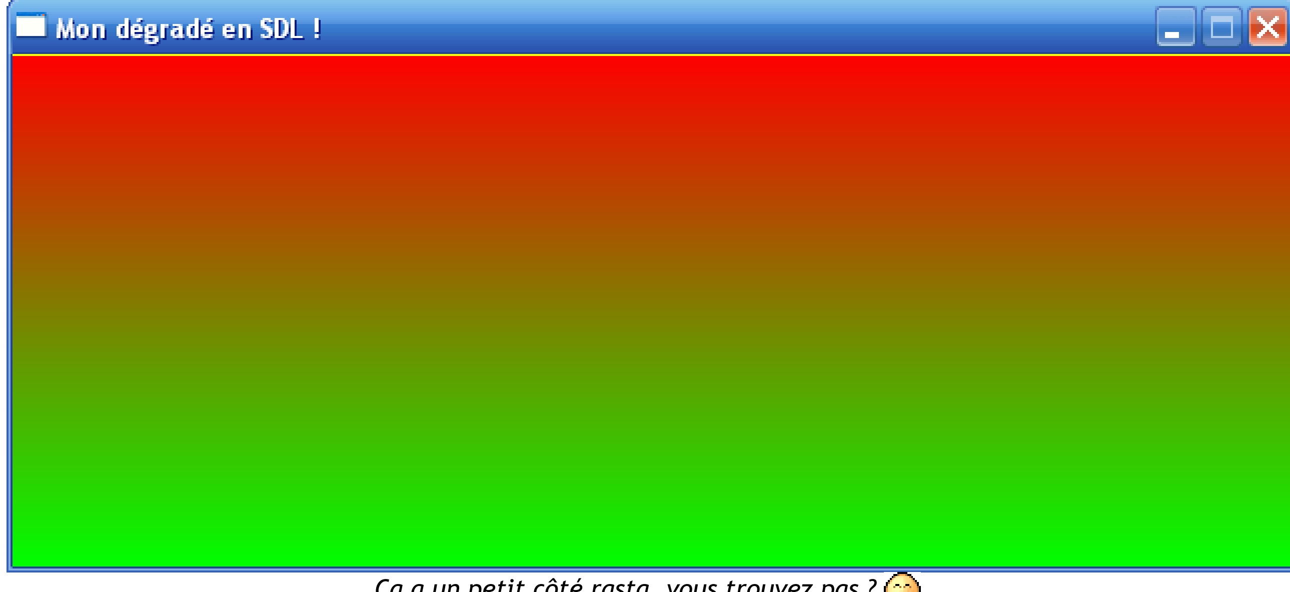
Je veux des exercices pour m'entraîner !

Pas de problème, je suis un véritable générateur d'exercices ambulant

- Créez le dégradé inverse, du blanc au noir. Il vous faudra peut-être réfléchir 10 secondes avant de trouver comment faire
- Vous pouvez aussi faire un double dégradé, en allant du noir au blanc comme on a fait ici puis du blanc au noir (la fenêtre fera alors le double de hauteur).

- Guère plus difficile, vous pouvez aussi vous entraîner à faire un dégradé horizontal au lieu d'un dégradé vertical
- Faites des dégradés en utilisant d'autres couleurs que le blanc et le noir. Essayez pour commencer du rouge au noir, du vert au noir et du bleu au noir, puis du rouge au blanc etc...
  Essayez de faire un dégradé pour d'autres couleurs allant du rouge au vert par exemple, ou encore du jaune au vert foncé etc.

  Ci-dessous un dégradé du rouge au vert :



Ca a un petit côté rasta, vous trouvez pas ?

Comme vous le voyez, notre travail ne consiste qu'à appeler les fonctions de la SDL.
Ce n'est pas bien difficile, il faut juste pratiquer et essayer de retenir le nom des principales fonctions et surtout l'ordre dans lequel on les appelle.

Alors ça vous plaît ?
Ohla ohla, mais c'est que le début en plus

Afficher des images

Dans notre premier chapitre de "vraie" pratique de la SDL, nous avons appris à charger la SDL, ouvrir une fenêtre et gérer les surfaces.

Or, pour le moment nous n'avons appris qu'à créer des surfaces unies, c'est-à-dire ayant la même couleur. Les surfaces unies ça va un moment, mais ce serait délicat de faire un jeu avec juste des carrés de couleur
Dans ce chapitre, nous allons justement apprendre à **charger des images** dans des surfaces, que ce soit des BMP, des PNG, des GIF, des JPG ou que sais-je encore

Charger une image BMP

Allons bon
Pourquoi est-ce qu'on ne va apprendre qu'à charger des images BMP ici ?

En fait, la SDL est une librairie très simple. Elle ne propose à la base que le chargement d'images de type Bitmap

(extension ".bmp"). Ne paniquez pas pour autant, car grâce à une extension de la SDL (la librairie SDL_Image) nous verrons qu'il est possible de charger de nombreux autres types.

Pour commencer déjà, nous allons nous contenter de ce que la SDL offre à la base. Nous allons donc étudier le chargement de BMP.

Le format BMP

Un BMP (abréviation de Bitmap) est un format d'image.
Les images que vous voyez sur votre ordinateur sont stockées dans des fichiers. Il existe plusieurs formats d'images, c'est-à-dire plusieurs façons de coder l'image dans un fichier. Selon le format, l'image prend plus ou moins d'espace disque et est de plus ou moins bonne qualité.

Le Bitmap est un format **non compressé** (contrairement aux JPG, PNG, GIF etc.)
Concrètement, cela signifie les choses suivantes :

- Le fichier est très rapide à lire, contrairement aux formats compressés doivent être décompressés, ce qui leur prend un peu plus de temps.
- La qualité de l'image est parfaite. Certains formats compressés comme le JPG détériorent la qualité de l'image, ce n'est pas le cas du BMP.
- Mais le fichier est aussi bien plus gros puisqu'il n'est pas compressé !

Bref, il y a des avantages et des défauts.
Pour la SDL, l'avantage c'est que ce type de fichier est simple et rapide à lire. Si vous avez souvent besoin de charger des images au cours de l'exécution de votre programme, il vaut mieux utiliser des BMP : certes le fichier est plus gros, mais il se chargera plus vite qu'un GIF par exemple.

Charger un bitmap

Téléchargement du pack d'images

Nous allons travailler avec plusieurs images dans ce chapitre. Si vous voulez faire les tests en même temps que vous lisez (et vous devriez !), je vous recommande fortement de télécharger ce pack qui contient toutes les images dont on va avoir besoin :

Télécharger le pack d'images (1 Mo)

Placez toutes les images dans le dossier de votre projet.
Nous allons commencer par travailler avec le fichier *lac_en_montagne.bmp*. C'est une scène 3D d'exemple livrée avec le logiciel de modélisation *Vue d'Esprit* (c'était à l'époque où j'espérais me découvrir un talent artistique, mais ça fait belle lurette que j'ai abandonné)

Charger l'image dans une surface

Nous allons utiliser une fonction qui va charger l'image BMP et la mettre dans une surface.
Cette fonction a pour nom **SDL_LoadBMP**. Vous allez voir c'est ridicule tellement c'est simple

Code : C

```c
maSurface = SDL_LoadBMP("image.bmp");
```

La fonction SDL_LoadBMP remplace 2 fonctions que vous connaissez :

- SDL_CreateRGBSurface : qui se chargeait d'allouer de la mémoire pour stocker une surface de la taille demandée (= malloc).
- SDL_FillRect : qui remplissait la structure d'une couleur unie.

Pourquoi est-ce que ça remplace ces 2 fonctions ? C'est très simple :

- La taille à allouer en mémoire pour la surface dépend de la taille de l'image : si l'image fait une taille de 250 x 300, alors votre surface aura une taille de 250 x 300.
- D'autre part, votre surface sera remplie pixel par pixel par le contenu de votre image BMP.

Codons sans plus tarder

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *imageDeFond = NULL;
    SDL_Rect positionFond;

    positionFond.x = 0;
    positionFond.y = 0;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Chargement d'images en SDL", NULL);

    /* Chargement d'une image Bitmap dans une surface */
    imageDeFond = SDL_LoadBMP("lac_en_montagne.bmp");
    /* On blitte par-dessus l'écran */
    SDL_BlitSurface(imageDeFond, NULL, ecran, &positionFond);

    SDL_Flip(ecran);
    pause();

    SDL_FreeSurface(imageDeFond); /* On libère la surface */
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

J'ai donc créé un pointeur vers une surface (*imageDeFond*) ainsi que les coordonnées correspondantes (*positionFond*).
La surface est créée en mémoire et remplie par la fonction SDL_LoadBMP.
On la blitte ensuite sur la surface *ecran* et c'est tout ! Admirez le résultat :

C'est aussi simple que cela

Donner une icône à son application

Maintenant que nous savons charger des images, nous pouvons voir comment donner une icône à notre programme. L'icône sera affichée en haut à gauche de la fenêtre (et aussi dans la barre des tâches). Pour le moment on a une icône par défaut.

Mais, les icônes des programmes ne sont pas des .ico normalement ?

Non pas forcément ! D'ailleurs les .ico n'existent que sous Windows, sur les autres OS on n'en trouve pas !
La SDL réconcilie tout le monde en utilisant un truc bien à elle : une surface !
Eh oui, l'icône d'un programme SDL n'est rien d'autre qu'une bête surface

Votre icône doit normalement être de taille 16x16 pixels.
Toutefois, sous Windows il faut que l'icône soit de taille 32x32 pixels sinon elle sera déformée. Ne vous en faites pas, la SDL "réduira" les dimensions de l'image pour qu'elle rentre dans 16x16 pixels (vous imaginez pas le temps que j'ai mis pour comprendre ça)

Pour ajouter l'icône à la fenêtre, on utilise la fonction **SDL_WM_SetIcon**

Cette fonction prend 2 paramètres : la surface qui contient l'image à afficher ainsi que des informations sur la transparence (NULL si on ne veut pas de transparence).

La gestion de la transparence d'une icône est un peu compliquée (il faut dire un à un quels sont les pixels transparents), nous ne l'étudierons donc pas.

On va combiner 2 fonctions en une :

Code : C

```c
SDL_WM_SetIcon(SDL_LoadBMP("sdl_icone.bmp"), NULL);
```

L'image est chargée en mémoire par *SDL_LoadBMP* et l'adresse de la surface est directement envoyée à *SDL_WM_SetIcon*.

La fonction SDL_WM_SetIcon doit être appelée avant que la fenêtre ne soit ouverte, c'est-à-dire qu'elle doit se trouver avant SDL_SetVideoMode dans votre code source.

Voici le code source complet (j'ai juste ajouté le SDL_WM_SetIcon par rapport à la dernière fois) :

Code : C

```c
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *imageDeFond = NULL;
    SDL_Rect positionFond;

    positionFond.x = 0;
    positionFond.y = 0;

    SDL_Init(SDL_INIT_VIDEO);

    /* Chargement de l'icône AVANT SDL_SetVideoMode */
    SDL_WM_SetIcon(SDL_LoadBMP("sdl_icone.bmp"), NULL);

    ecran = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Chargement d'images en SDL", NULL);

    imageDeFond = SDL_LoadBMP("lac_en_montagne.bmp");
    SDL_BlitSurface(imageDeFond, NULL, ecran, &positionFond);

    SDL_Flip(ecran);
    pause();

    SDL_FreeSurface(imageDeFond);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Résultat, l'icône est chargée

Chargement d'images en SDL

GESTION DE LA TRANSPARENCE

Le problème de la transparence

Nous avons chargé une jolie image bitmap tout à l'heure dans notre fenêtre.
Supposons que l'on veuille blitter une image par-dessus. Ca vous arrivera très fréquemment, car dans un jeu en général le personnage que l'on déplace est un bitmap, et il se déplace sur une image de fond.

On va blitter l'image de Zozor (la mascotte du Site du Zér0 !) sur la scène :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *imageDeFond = NULL, *zozor = NULL;
    SDL_Rect positionFond, positionZozor;

    positionFond.x = 0;
    positionFond.y = 0;
    positionZozor.x = 500;
    positionZozor.y = 260;

    SDL_Init(SDL_INIT_VIDEO);

    SDL_WM_SetIcon(SDL_LoadBMP("sdl_icone.bmp"), NULL);

    ecran = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Chargement d'images en SDL", NULL);

    imageDeFond = SDL_LoadBMP("lac_en_montagne.bmp");
    SDL_BlitSurface(imageDeFond, NULL, ecran, &positionFond);

    /* Chargement et blittage de zozor sur la scène */
    zozor = SDL_LoadBMP("zozor.bmp");
    SDL_BlitSurface(zozor, NULL, ecran, &positionZozor);

    SDL_Flip(ecran);
    pause();

    SDL_FreeSurface(imageDeFond);
    SDL_FreeSurface(zozor);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

On a juste rajouté une surface pour y stocker Zozor, que l'on blitte ensuite à un endroit sur la scène.
Le résultat est le suivant :

C'est moche hein ?

Bah oui, c'est parce que t'as mis un fond bleu tout moche sur l'image de Zozor !

Parce que vous croyez qu'avec un fond noir ou un fond marron derrière Zozor ça aurait été plus joli ?
Ben non, le problème ici c'est que notre image est forcément rectangulaire, donc si on la colle sur la scène on voit son fond et ça fait moche.

Heureusement, la SDL gère la transparence !

Rendre une image transparente

Etape 1 : préparer l'image

Pour commencer, il faut préparer l'image que vous voulez blitter sur la scène.
Les BMP ne sont pas des images gérant la transparence, contrairement aux GIF et PNG. Il va donc falloir utiliser une astuce.

Il faut mettre la même couleur de fond sur toute l'image. Celle-ci sera rendue transparente par la SDL au moment du blit. Observez à quoi ressemble mon zozor.bmp de plus près :

J'ai mis un fond bleu uni exprès. Notez que j'ai choisi le bleu au hasard, j'aurais très bien pu mettre un fond vert ou rouge par exemple. Ce qui compte, c'est que cette couleur soit unique et unie. J'ai choisi le bleu parce qu'il n'y en avait pas dans l'image de zozor. Si j'avais choisi le vert, j'aurais pris le risque que l'herbe que machouille Zozor (en bas à gauche de l'image) soit rendue transparente.

A vous donc de vous débrouiller avec votre logiciel de dessin (Paint, Photoshop, chacun ses goûts) pour donner un fond uni à votre image.

Etape 2 : indiquer la couleur transparente

Pour indiquer à la SDL quelle est la couleur qui doit être rendue transparente, vous devez utiliser la fonction **SDL_SetColorKey**. Cette fonction doit être appelée avant de blitter l'image.
Voici comment je m'en sers pour rendre le bleu derrière Zozor transparent :

Code : C

```c
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
```

Il y a 3 paramètres :

- La surface qui doit être rendue transparente (ici c'est zozor)
- Une liste de flags. Utilisez SDL_SRCCOLORKEY pour activer la transparence, 0 pour la désactiver.
- Indiquez ensuite la couleur qui doit être rendue transparente. J'ai utilité SDL_MapRGB pour créer la couleur au format nombre (Uint32) comme on l'a déjà fait par le passé. Comme vous le voyez, c'est le bleu pur (0, 0, 255) que je rends transparent.

En résumé, on charge d'abord l'image avec SDL_LoadBMP, on indique la couleur transparente avec SDL_SetColorKey, puis on peut blitter avec SDL_BlitSurface :

Code : C

```c
/* On charge l'image : */
zozor = SDL_LoadBMP("zozor.bmp");
/* On rend le bleu derrière Zozor transparent : */
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
/* On blitte l'image maintenant transparente sur le fond : */
SDL_BlitSurface(zozor, NULL, ecran, &positionZozor);
```

Résultat, Zozor est parfaitement intégré à la scène !

Ca, c'est LA technique de base que vous réutiliserez tout le temps dans vos futurs programmes. Apprenez à bien manier la transparence car c'est fondamental si on veut avoir un jeu un minimum réaliste

La transparence Alpha

C'est un autre type de transparence.
Jusqu'ici, on se contentait de définir UNE couleur de transparence (par exemple le bleu). Cette couleur n'apparaissait pas une fois l'image blittée.

La transparence Alpha c'est autre chose. Elle permet de réaliser un "mélange" entre une image et le fond. C'est une sorte de fondu.

La transparence Alpha d'une surface peut être activée par la fonction **SDL_SetAlpha** :

Code : C

```
SDL_SetAlpha(zozor, SDL_SRCALPHA, 128);
```

Il y a là encore 3 paramètres :

- La surface en question (zozor)
- Une liste de flags. Mettez SDL_SRCALPHA pour activer la transparence, 0 pour la désactiver.
- Très important : la valeur *Alpha* de la transparence. C'est un nombre compris entre 0 (image totalement transparente, donc invisible) et 255 (image totalement opaque, comme s'il n'y avait pas de transparence alpha)

Plus le nombre *Alpha* est petit, plus l'image est transparente et fondue.

Voici par exemple un code qui met une transparence alpha de 128 à notre Zozor :

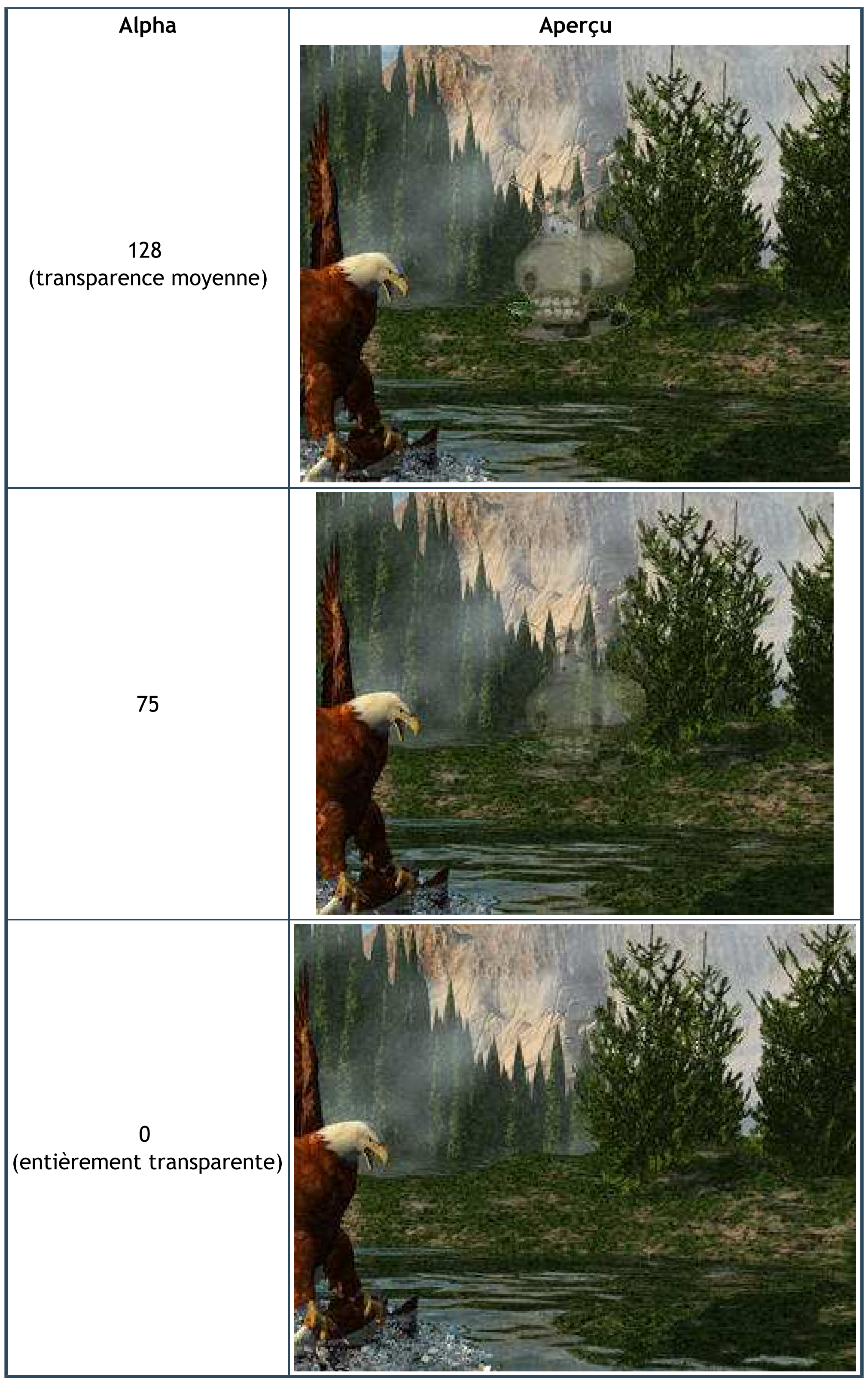
Code : C

```
zozor = SDL_LoadBMP("zozor.bmp");
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
/* Transparence Alpha moyenne (128) : */
SDL_SetAlpha(zozor, SDL_SRCALPHA, 128);
SDL_BlitSurface(zozor, NULL, ecran, &positionZozor);
```

Vous noterez que j'ai conservé la transparence de SDL_SetColorKey. Les 2 types de transparence sont en effet combinables.

Ce tableau vous montre à quoi ressemble Zozor selon la valeur de *Alpha* :

Alpha	Aperçu
255 (entièrement opaque)	
190	



Alpha	Aperçu
128 (transparence moyenne)	
75	
0 (entièrement transparente)	

La transparence Alpha 128 (transparence moyenne) est une valeur spéciale qui est optimisée par la SDL. Ce type de transparence est plus rapide à calculer pour votre ordinateur que les autres. C'est peut être bon à savoir si vous utilisez beaucoup de transparence Alpha dans votre programme.

CHARGER PLUS DE FORMATS D'IMAGE AVEC SDL_IMAGE

La SDL ne gère que les Bitmaps (BMP) comme on l'a vu.

A priori, ce n'est pas un très gros problème parce que la lecture des BMP est rapide pour la SDL, mais il faut reconnaître qu'aujourd'hui on est plutôt habitué à utiliser d'autres formats. En particulier, nous sommes habitués aux formats d'images "compressés" comme le PNG, le GIF et le JPEG. Ca tombe bien, il existe justement une librairie **SDL_Image** qui gère tous les formats suivants :

- TGA
- BMP
- PNM
- XPM
- XCF
- PCX
- GIF
- JPG
- TIF
- LBM
- PNG

Il est possible de rajouter des extensions à la SDL. Ce sont des librairies qui ont besoin de la SDL pour fonctionner. On peut voir ça comme des add-ons. SDL_Image est l'une d'entre elles.

Installer SDL_image

Téléchargement

Il y a une page spéciale sur **le site de la SDL** qui référence les librairies utilisant la SDL. Cette page s'intitule **Libraries**, vous trouverez un lien dans le menu de gauche.
Vous pourrez voir qu'il y a pas mal de librairies disponibles. Celles-ci ne proviennent pas des auteurs de la SDL généralement, ce sont plutôt des utilisateurs de la SDL qui proposent leurs librairies.
Certaines sont très bonnes et méritent le détour, d'autres sont moins bonnes et encore buggées. Il faut arriver à faire le tri

Cherchez **SDL_Image** dans la liste...
Bon allez, je suis de bonne humeur : voici un **lien direct vers la page de téléchargement de SDL_Image** pour vous aider

Téléchargez la version de SDL_Image qui vous correspond dans la section *Binary* (ne prenez PAS la source, on n'en a pas besoin !)
Si vous êtes sous Windows téléchargez **SDL_image-devel-1.2.4-VC6.zip** même si vous n'utilisez pas Visual C++ !

Installation

Dans ce zip vous trouverez :

- **SDL_image.h** : le seul header dont a besoin la librairie SDL_image. Placez-le dans mingw32/include/SDL/, c'est-à-dire à côté des autres headers de la SDL.
- **SDL_image.lib** : placez-le dans mingw32/lib. Oui, je sais, je vous ai dit que normalement les .lib étaient des fichiers réservés à Visual, mais ici exceptionnellement le .lib fonctionnera même avec le compilateur mingw.
- **Plusieurs DLL** : placez-les toutes dans le dossier de votre projet (à côté de SDL.dll donc).

Ensuite, vous devez modifier les options de votre projet pour "linker" avec le fichier SDL_image.lib.

Si vous êtes sous Code::Blocks par exemple, allez dans le menu Projects / Build options. Dans l'onglet Linker, cliquez sur le bouton "Add" et indiquez où se trouve le fichier SDL_image.lib
Si on vous demande "Keep as a relative path ?", répondez ce que vous voulez ça ne changera rien dans l'immédiat. Je recommande de répondre "Non" personnellement

Ensuite, vous n'avez plus qu'à inclure le header SDL_image.h dans votre code source comme ceci :

Code : C

```
#include <SDL/SDL_image.h>
```

Et voilà, la librairie SDL_image devrait maintenant fonctionner (à condition que vous ayez bien fait exactement tout ce que j'ai indiqué bien sûr)

Si vous êtes sous Dev-C++ ou Visual Studio, la manipulation est quasiment la même. Si vous avez réussi à installer la SDL, vous n'aurez aucun problème pour installer SDL_image

Charger les images

En fait, installer SDL_image est 100 fois plus compliqué que de l'utiliser, c'est vous dire la complexité du truc

Il y a UNE fonction à connaître : **IMG_Load**.
Elle prend un paramètre : le nom du fichier à ouvrir.

Ce qui est génial, c'est que cette fonction est capable d'ouvrir tous les types de fichiers que gère SDL_image (GIF, PNG, JPG, mais aussi BMP, TIF...). Elle détectera toute seule le type du fichier en fonction de son extension.

Comme SDL_image peut aussi ouvrir les BMP, vous pouvez même oublier la fonction SDL_LoadBMP maintenant et n'utiliser plus que IMG_Load pour le chargement de n'importe quelle image.

Autre bon point : si l'image que vous chargez gère la transparence (comme c'est le cas des PNG et des GIF), alors SDL_image activera automatiquement la transparence pour cette image ! Cela vous évite donc d'avoir à appeler *SDL_SetColorKey*

Voici le code source complet qui charge sapin.png et l'affiche.
Notez bien que j'inclue SDL/SDL_image.h et que je ne fais pas appel à *SDL_SetColorKey* car mon PNG est transparent.

Vous allez voir que j'utilise *IMG_Load* partout dans ce code en remplacement de *SDL_LoadBMP*.

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h> /* Inclusion du header de SDL_image */

void pause();

int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *imageDeFond = NULL, *sapin = NULL;
    SDL_Rect positionFond, positionSapin;

    positionFond.x = 0;
    positionFond.y = 0;
    positionSapin.x = 500;
    positionSapin.y = 260;

    SDL_Init(SDL_INIT_VIDEO);

    SDL_WM_SetIcon(IMG_Load("sdl_icone.bmp"), NULL);

    ecran = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Chargement d'images en SDL", NULL);

    imageDeFond = IMG_Load("lac_en_montagne.bmp");
    SDL_BlitSurface(imageDeFond, NULL, ecran, &positionFond);

    /* Chargement d'un PNG avec IMG_Load
    Celui-ci est automatiquement rendu transparent car les informations de
    transparence sont codées à l'intérieur du fichier PNG */
    sapin = IMG_Load("sapin.png");
    SDL_BlitSurface(sapin, NULL, ecran, &positionSapin);

    SDL_Flip(ecran);
    pause();

    SDL_FreeSurface(imageDeFond);
    SDL_FreeSurface(sapin);
    SDL_Quit();

    return EXIT_SUCCESS;
}

void pause()
{
    int continuer = 1;
    SDL_Event event;

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
        }
    }
}
```

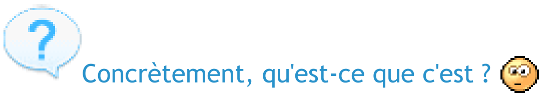
Résultat :

Quand même bien pratique cette petite librairie Vous êtes maintenant capables d'afficher de nombreux type d'images dans vos programmes SDL, et, surtout, de rendre ces images transparentes.
Avec ces fonctions pourtant très simples de la SDL, vous allez pouvoir réaliser à peu près tous les programmes que vous voulez.

Mais avant ça, il va falloir étudier un chapitre très important de la SDL : **les évènements**. Cela vous permettra de gérer le clavier, la souris et le joystick, rien de moins que ça !
Au lieu d'avoir un programme "passif" comme maintenant où vous vous contentez de regarder votre fenêtre, vous allez pouvoir contrôler un personnage au clavier, cliquer dans la fenêtre etc.

La gestion des évènements (Partie 1/2)

La gestion des évènements est une des fonctionnalités les plus importantes de la SDL.
C'est, je trouve, **intéressant et passionnant**. C'est à partir de là que vous allez vraiment être capables de tout faire.


Concrètement, qu'est-ce que c'est ?

Par exemple, quand l'utilisateur appuie sur une touche du clavier, on dit qu'il s'est produit un **évènement**. Mais ce n'est pas tout ! Il existe bien d'autres types d'évènements :

- Quand l'utilisateur clique avec la souris
- Quand il bouge la souris
- Quand il réduit la fenêtre
- Quand il demande à fermer la fenêtre etc.

Tout ça, ce sont des évènements. Ce sont des "signaux" envoyés à votre programme pour l'informer qu'il s'est passé quelque chose.

Le rôle de ce chapitre sera de vous apprendre à traiter ces évènements. Vous serez capables de dire à l'ordinateur " *Si l'utilisateur clique à cet endroit, fais ça, sinon fais cela... S'il bouge la souris, fais ceci. S'il appuie sur la touche Q, arrête le programme...*" etc.

Nous ne traiterons ici que les évènements du clavier et de la souris.
D'autres évènements un peu plus complexes, comme ceux générés par le joystick, seront vus plus tard.

Soyez plus que jamais attentifs, parce que ça vaut vraiment le coup

LE PRINCIPE DES ÉVÈNEMENTS

Pour nous habituer aux évènements, nous allons apprendre à traiter le plus simple d'entre eux : **la demande de fermeture du programme**.
C'est un évènement qui se produit lorsque l'utilisateur clique sur la croix pour fermer la fenêtre :

C'est vraiment l'évènement le plus simple. En plus, c'est un évènement que vous avez utilisé jusqu'ici sans vraiment le savoir, car il était situé dans la fonction *pause()* !
En effet, la fonction *pause* servait à attendre que l'utilisateur demande à fermer le programme. Si on n'avait pas fait cette fonction, la fenêtre se serait affichée et fermée en un éclair !

A partir de maintenant, vous pouvez oublier la fonction *pause*. Supprimez-la carrément de votre code source, car nous allons apprendre à la reproduire

La variable d'évènement

Pour traiter des évènements, vous aurez besoin de déclarer une variable (juste une seule rassurez-vous) de type SDL_Event. Appelez-la comme vous voulez, moi je vais l'appeler "event" (ce qui signifie "évènement" en anglais).

Code : C

```
SDL_Event event;
```

Nous allons nous contenter pour nos tests d'un *main* très basique qui affiche juste une fenêtre, comme on l'a vu il y a quelques chapitres. Voici à quoi doit ressembler votre *main* :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL;
    SDL_Event event; /* Cette variable servira plus tard à gérer les évènements */


    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Gestion des évènements en SDL", NULL);


    SDL_Quit();

    return EXIT_SUCCESS;
}
```

C'est donc un code très basique, il ne contient qu'une chose en plus : la déclaration de la variable *event* dont nous allons bientôt nous servir.
Testez ce code : comme prévu, la fenêtre va s'afficher et se fermer immédiatement après.

La boucle des évènements

Lorsqu'on veut attendre un évènement, on fait généralement une boucle. Cette boucle se répètera tant qu'on n'a pas eu l'évènement voulu.
On va avoir besoin d'utiliser un booléen qui indiquera si on doit continuer la boucle ou pas.
Créez donc ce booléen que vous appellerez par exemple *continuer* :

Code : C

```
int continuer = 1;
```

Ce booléen est mis à 1 au départ car on veut que la boucle se répète TANT QUE la variable continuer vaut 1 (vrai).
Dès qu'elle vaudra 0 (faux), alors on sortira de la boucle et le programme s'arrêtera.

Voici la boucle à créer :

Code : C

```
while (continuer)
{
    /* Traitement des évènements */
}
```

Voilà, on a pour le moment une boucle infinie qui ne s'arrêtera que si on met la variable *continuer* à 0.
C'est ce que nous allons écrire à l'intérieur de cette boucle qui est le plus intéressant

Récupération de l'évènement

Maintenant, faisons appel à une fonction de la SDL pour demander si un évènement s'est produit.
On dispose de 2 fonctions qui font cela, mais d'une manière différente :

- **SDL_WaitEvent** : elle attend qu'un évènement se produise. Cette fonction est dite bloquante car elle suspend l'exécution du programme tant qu'aucun évènement ne s'est produit.
- **SDL_PollEvent** : cette fonction fait la même chose mais n'est pas bloquante. Elle vous dit si un évènement s'est produit ou pas. Même si aucun évènement ne s'est produit, elle rend la main à votre programme de suite.

Ces 2 fonctions sont utiles, mais dans des cas différents.
Grosso modo, et pour résumer, si vous utilisez *SDL_WaitEvent* votre programme utilisera très peu de processeur car il attendra qu'un évènement se produise.
En revanche, si vous utilisez *SDL_PollEvent*, votre programme va parcourir votre boucle while et rappeler SDL_PollEvent indéfiniment jusqu'à ce qu'un évènement se soit produit. A tous les coups, vous utiliserez 100% du processeur.

Mais alors, il faut tout le temps utiliser SDL_WaitEvent si cette fonction utilise moins le processeur non ?

Non, car il y a des cas où SDL_PollEvent se révèle indispensable. C'est le cas des jeux dans lesquels l'écran se met à jour même quand il n'y a pas d'évènement.
Prenons par exemple Tetris : les blocs descendent tous seuls, il n'y a pas besoin que l'utilisateur crée d'évènement pour ça ! Si on avait utilisé SDL_WaitEvent, le programme serait resté "bloqué" dans cette fonction et vous n'auriez pas pu mettre à jour l'écran pour faire descendre les blocs !

Comment fait SDL_WaitEvent pour ne pas consommer de processeur ?
Après tout, la fonction est bien obligée de faire une boucle infinie pour tester tout le temps s'il y a un évènement ou pas non ?

C'est une question que je me posais il y a encore peu de temps. La réponse est un petit peu compliquée car ça concerne la façon dont l'OS gère les processus (les programmes).
Si vous voulez (mais je vous raconte ça rapidement), avec SDL_WaitEvent le processus de votre programme est mis "en pause". Votre programme n'est donc plus traité par le processeur.
Votre programme sera "réveillé" par l'OS au moment où il y aura un évènement. Du coup, le processeur se remettra à travailler sur votre programme à ce moment-là. Cela explique pourquoi votre programme ne consomme pas de processeur pendant qu'il attend l'évènement.

Bon je comprends que ce soit un peu abstrait pour vous pour le moment. Et à dire vrai vous n'avez pas besoin de comprendre ça maintenant. Vous comprendrez mieux toutes les différences plus loin en pratiquant.
Pour le moment, nous allons utiliser **SDL_WaitEvent** car notre programme reste très simple. Ces 2 fonctions s'utilisent de toute façon de la même manière.

Vous devez envoyer à la fonction l'adresse de votre variable *event* qui stocke l'évènement.
Comme cette variable n'est pas un pointeur (regardez la déclaration à nouveau), nous allons mettre le symbole & devant le nom de la variable afin de donner l'adresse :

Code : C

```
SDL_WaitEvent(&event);
```

Après appel de cette fonction, la variable *event* contient obligatoirement un évènement .

Ce n'aurait pas forcément été le cas si on avait utilisé SDL_PollEvent : cette fonction aurait pu renvoyer "Pas d'évènement".

Analyse de l'évènement

Maintenant, nous possédons une variable *event* qui contient des informations sur l'évènement qui s'est produit.
Il faut regarder la sous-variable *event.type* et faire un test sur sa valeur. Généralement on utilise un switch pour tester l'évènement.

Mais comment on sait quelle valeur correspond à l'évènement "Quitter" par exemple ?

La SDL nous fournit des constantes, ce qui simplifie grandement l'écriture du programme
Il existe de nombreuses constantes (autant qu'il y a d'évènements possibles). Nous les verrons au fur et à mesure plus loin dans ce chapitre.

Code : C

```
while (continuer)
{
    SDL_WaitEvent(&event); /* Récupèration de l'évènement dans event */
    switch(event.type) /* Test du type d'évènement */
    {
        case SDL_QUIT: /* Si c'est un évènement de type "Quitter" */
            continuer = 0;
            break;
    }
}
```

1. Dès qu'il y a un évènement, la fonction SDL_WaitEvent renvoie cet évènement dans *event*.
2. On analyse le type d'évènement grâce à un switch. Le type de l'évènement se trouve dans *event.type*
3. On teste à l'aide de "case" dans le switch le type de l'évènement. Pour le moment, on ne teste que l'évènement SDL_QUIT (demande de fermeture du programme) car c'est le seul qui nous intéresse.
    - Si c'est un évènement SDL_QUIT, c'est que l'utilisateur a demandé à quitter le programme. Dans ce cas on met le booléen continuer à 0. Au prochain tour de boucle, la condition sera fausse et donc la boucle s'arrêtera. Le programme s'arrêtera ensuite.
    - Si ce n'est pas un évènement SDL_QUIT, c'est qu'il s'est passé autre chose : l'utilisateur a appuyé sur une touche, a cliqué ou même a tout simplement bougé la souris dans la fenêtre. Comme ces autres évènements ne nous intéressent pas, on ne les traite pas. On ne fait donc rien : la boucle recommence et on attend à nouveau un évènement (on repart à l'étape 1).

Ce que je viens de vous expliquer est super important. Si vous avez compris ce code, vous avez tout compris et le reste du chapitre sera aussi facile qu'une partie de Mario Kart en mode 50cc (pour ceux qui ne connaissent pas Mario Kart, j'ai voulu dire que ça serait "un jeu d'enfant").

Le code complet

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL;
    SDL_Event event; /* La variable contenant l'évènement */
    int continuer = 1; /* Notre booléen pour la boucle */


    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Gestion des évènements en SDL", NULL);

    while (continuer) /* TANT QUE la variable ne vaut pas 0 */
    {
        SDL_WaitEvent(&event); /* On attend un évènement qu'on récupère dans event */
        switch(event.type) /* On teste le type d'évènement */
        {
            case SDL_QUIT: /* Si c'est un évènement QUITTER */
                continuer = 0; /* On met le booléen à 0, donc la boucle va s'arrêter */
                break;
        }
    }

    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Voilà le code complet. Il n'y a rien de bien difficile, si vous avez suivi jusqu'ici ça ne devrait pas vous surprendre. D'ailleurs, vous remarquerez qu'on n'a fait que reproduire ce que faisait la fonction *pause* (vérifiez le code, c'est le même, sauf qu'on a tout mis dans le *main*). Bien entendu, il est préférable de mettre ce code dans une fonction comme *pause* à part, car cela allège la fonction *main* et la rend plus lisible

LE CLAVIER

Nous allons maintenant étudier les évènements produits par le clavier.

Si vous avez compris le début du chapitre, vous n'aurez AUCUN problème pour traiter les autres types d'évènements. C'est tellement facile qu'on en pleurerait presque

Tellement facile que c'est pas la peine que je vous l'explique

(mais bon je vais vous l'expliquer quand même, je m'en voudrais sinon)

Pourquoi est-ce si simple ? Parce que maintenant que vous avez compris le coup de la boucle, tout ce que vous allez avoir à faire c'est rajouter d'autres "case" dans le "switch" pour traiter d'autres types d'évènements

C'est duuuur hein

Les évènements du clavier

Il existe 2 évènements différents qui peuvent être générés par le clavier :

- SDL_KEYDOWN : quand une touche du clavier est enfoncée
- SDL_KEYUP : quand une touche du clavier est relâchée

Pourquoi y a-t-il ces 2 évènements ?
Parce que quand vous appuyez sur une touche, en fait il se passe 2 choses : vous enfoncez la touche (SDL_KEYDOWN), puis vous la relâchez (SDL_KEYUP). La SDL vous permet de traiter ces 2 évènements, ce qui sera bien pratique vous verrez

Pour le moment, nous allons nous contenter de traiter l'évènement SDL_KEYDOWN (appui de la touche) :

Code : C

```c
while (continuer)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case SDL_QUIT:
            continuer = 0;
            break;
        case SDL_KEYDOWN: /* Si appui d'une touche */
            continuer = 0;
            break;
    }
}
```

Si on appuie sur une touche, le programme s'arrêtera
Testez, vous verrez

Récupérer la touche

Savoir qu'une touche a été enfoncée c'est bien, mais savoir laquelle, c'est quand même mieux

On peut récupérer la touche grâce à la sous-sous-sous-variable (ouf) : *event.key.keysym.sym*.
Cette variable contient la valeur de la touche qui a été enfoncée (elle fonctionne aussi lors d'un relâchement de la touche SDL_KEYUP).

L'avantage, c'est que la SDL permet de récupérer la valeur de toutes les touches du clavier. Il y a les lettres et les chiffres bien sûr (ABCDE0123...), mais aussi la touche Echap, ou encore Impr. Ecran, Suppr, Entrée etc...

Il y a une constante pour chacune des touches du clavier. Vous trouverez cette liste dans la documentation de la SDL, que vous avez très probablement téléchargée avec la librairie quand vous avez dû l'installer.
Si tel n'est pas le cas, je vous recommande fortement de retourner sur le site de la SDL et d'y télécharger la documentation, car elle est très utile.

Vous trouverez la liste des touches du clavier dans la section "**SDL Keysym definitions**".
Comme je suis un type super sympa (et modeste avec ça), je me suis permis de mettre à votre disposition sur le Site du Zér0 la page en question :

Liste des touches du clavier (keysyms)

Bien entendu, la documentation est en anglais donc la liste est en anglais. Si vous voulez vraiment programmer il est important d'être capable de lire l'anglais car toutes les documentations sont en anglais et vous ne pouvez pas vous en passer !

Il y a 2 tableaux dans cette liste : un grand (au début) et un petit (à la fin). Nous nous intéresserons au grand

tableau.
Dans la première colonne vous avez la constante, dans la seconde la représentation équivalente en ASCII (certaines touches comme "Shift" n'ont pas de valeur ASCII correspondante), et enfin dans la troisième colonne vous avez une description de la touche.

Prenons par exemple la touche "Echap" ("Escape" en anglais). On peut tester si la touche enfoncée est "Echap" comme ceci :

Code : C

```
switch (event.key.keysym.sym)
{
    case SDLK_ESCAPE: /* Appui sur la touche Echap, on arrête le programme */
        continuer = 0;
        break;
}
```

J'utilise un switch pour faire mon test mais j'aurais aussi bien pu utiliser un if.
J'ai toutefois plutôt tendance à me servir des switch quand je traite les évènements car je teste beaucoup de valeurs différentes (j'ai beaucoup de "case" dans un "switch" en pratique, contrairement à ici).

Voici une boucle d'évènement complète que vous pouvez tester :

Code : C

```
while (continuer)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case SDL_QUIT:
            continuer = 0;
            break;
        case SDL_KEYDOWN:
            switch (event.key.keysym.sym)
            {
                case SDLK_ESCAPE: /* Appui sur la touche Echap, on arrête le programme */
                    continuer = 0;
                    break;
            }
            break;
    }
}
```

Cette fois, le programme s'arrête si on appuie sur Echap ou si on clique sur la croix de la fenêtre

Maintenant que vous savez comment arrêter le programme en appuyant sur une touche, vous êtes autorisés à faire du plein écran si ça vous amuse (flag SDL_FULLSCREEN dans *SDL_SetVideoMode*).
Auparavant, je vous avais demandé d'éviter de le faire car on ne savait pas comment arrêter un programme en plein écran (y'a pas de croix sur laquelle cliquer pour arrêter)

(Exercice) Diriger Zozor au clavier

Vous êtes maintenant capables de déplacer une image dans la fenêtre à l'aide du clavier !

C'est un exercice très intéressant qui va d'ailleurs nous permettre de voir comment utiliser le double buffering et la répétition de touches.
De plus, ce que je vais vous apprendre là est la base de tous les jeux faits en SDL, donc ça vaut doublement le coup d'être très attentif

Charger l'image

Pour commencer, nous allons charger une image. On va faire simple : on va reprendre l'image de Zozor utilisée dans le chapitre précédent.
Créez donc la surface *zozor*, chargez l'image et rendez-la transparente (c'était un BMP je vous le rappelle)

Code : C

```c
zozor = SDL_LoadBMP("zozor.bmp");
SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));
```

Ensuite, et c'est certainement le plus important, vous devez créer une variable de type SDL_Rect pour retenir les coordonnées de Zozor :

Code : C

```c
SDL_Rect positionZozor;
```

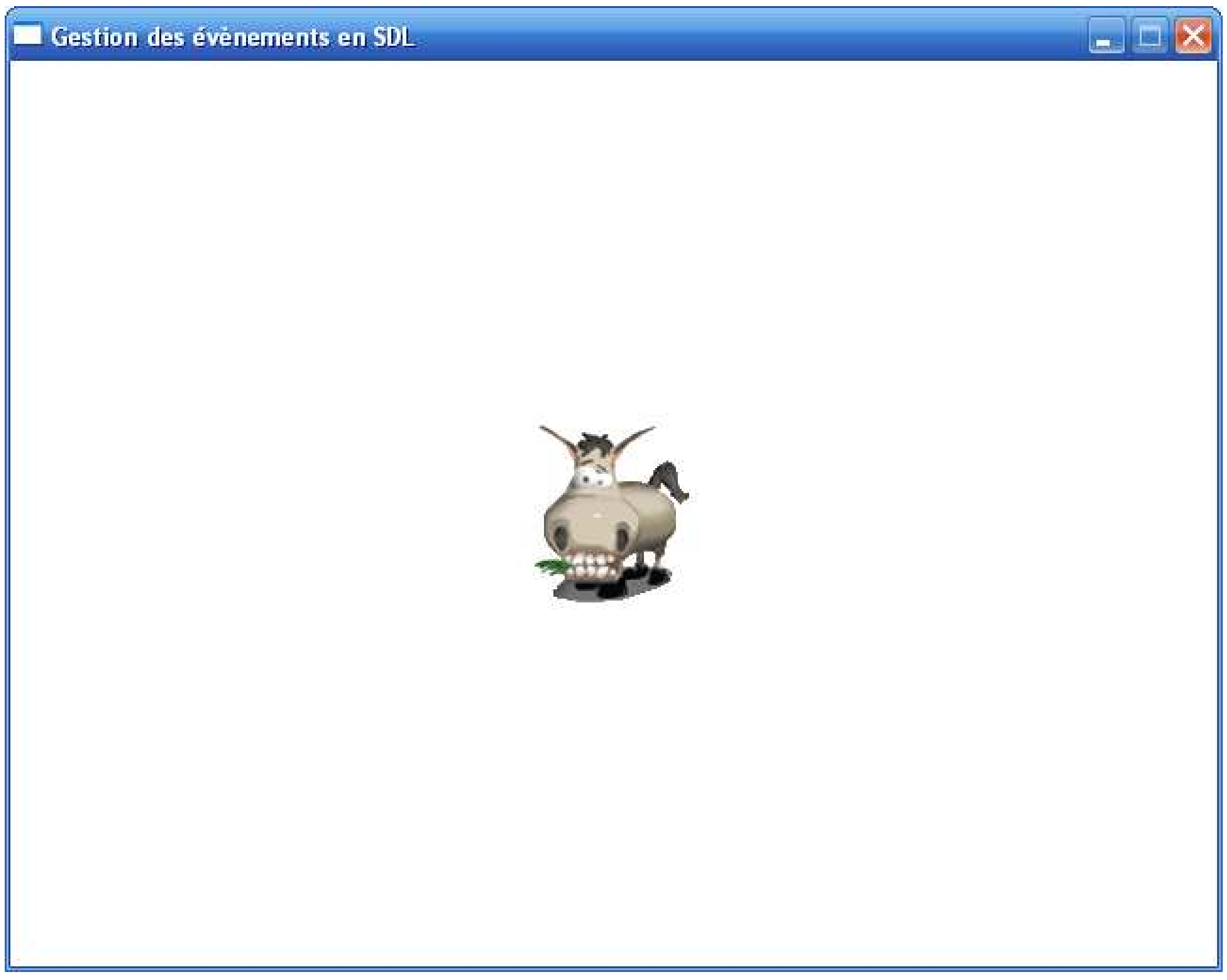
Je vous recommande d'initialiser les coordonnées, en mettant soit x = 0 et y = 0 (position en haut à gauche de la fenêtre), soit en centrant Zozor dans la fenêtre comme vous avez appris à le faire il n'y a pas si longtemps

Code : C

```c
/* On centre zozor à l'écran */
positionZozor.x = ecran->w / 2 - zozor->w / 2;
positionZozor.y = ecran->h / 2 - zozor->h / 2;
```

Vous devez initialiser *positionZozor* après avoir chargé les surfaces *ecran* et *zozor*. En effet, j'utilise la largeur (w) et la hauteur (h) de ces 2 surfaces pour calculer la position centrée de zozor à l'écran, il faut donc que ces surfaces aient été initialisées auparavant.

Si vous vous êtes bien débrouillés, vous devriez être arrivés à afficher Zozor au centre de l'écran :



J'ai choisi de mettre le fond en blanc cette fois (en faisant un *SDL_FillRect* sur *ecran*), mais ce n'est pas une obligation.
Si vous n'arrivez pas à reproduire cela, c'est que vous n'êtes pas au point et donc qu'il faut relire les chapitres précédents !

Schéma de la programmation évènementielle

Quand vous codez un programme qui réagit aux évènements (comme on va le faire là), vous devrez suivre la plupart du temps le même "schéma" de code.
Ce schéma est à connaître par coeur. Le voici :

Code : C

```
while (continuer)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case SDL_TRUC: /* Gestion des évènements de type TRUC */
        case SDL_BIDULE: /* Gestion des évènements de type BIDULE */
    }

    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255)); /* On efface l'écran (ici fond blanc) */
    /* On fait tous les SDL_BlitSurface nécessaires pour coller les surfaces à l'écran */
    SDL_Flip(ecran); /* On met à jour l'affichage */
}
```

Voilà en gros la forme de la boucle principale d'un programme SDL.

On boucle tant qu'on n'a pas demandé à arrêter le programme.

1. **On attend un évènement** (SDL_WaitEvent) OU BIEN on vérifie s'il y a un évènement mais on n'attend pas qu'il y en ait un (SDL_PollEvent). Pour le moment on se contente de SDL_WaitEvent.
2. On fait un (grand) switch pour savoir de quel type d'évènement il s'agit (évènement de type TRUC, de type BIDULE, comme ça vous chante). **On traite l'évènement qu'on a reçu** (on effectue certaines actions, certains calculs).
3. Une fois sortis du switch, on prépare un nouvel affichage :
    1. Première chose à faire : **on efface l'écran** en faisant un SDL_FillRect dessus. Si on ne le faisait pas, on aurait des "traces" de l'ancien écran qui resteraient, et forcément ça serait un peu moche
    2. Ensuite, on fait tous les **blits** nécessaires pour coller les surfaces sur l'écran.
    3. Enfin, une fois que c'est fait **on met à jour l'affichage** aux yeux de l'utilisateur, en appelant la fonction SDL_Flip(ecran).

Traiter l'évènement SDL_KEYDOWN

Voyons voir maintenant comment on va traiter l'évènement SDL_KEYDOWN.
Notre but est de diriger Zozor au clavier avec les flèches directionnelles. On va donc modifier ses coordonnées à l'écran en fonction de la flèche sur laquelle on appuie :

Code : C

```c
switch(event.type)
{
    case SDL_QUIT:
        continuer = 0;
        break;
    case SDL_KEYDOWN:
        switch(event.key.keysym.sym)
        {
            case SDLK_UP: // Flèche haut
                positionZozor.y--;
                break;
            case SDLK_DOWN: // Flèche bas
                positionZozor.y++;
                break;
            case SDLK_RIGHT: // Flèche droite
                positionZozor.x++;
                break;
            case SDLK_LEFT: // Flèche gauche
                positionZozor.x--;
                break;
        }
        break;
}
```

Comment j'ai trouvé ces constantes ? Dans la doc !
Je vous ai donné tout à l'heure un lien vers la page de la doc qui liste toutes les touches du clavier : c'est là que je me suis servi.

Ce qu'on fait là est très simple :

- Si on appuie sur la flèche haut : on diminue l'ordonnée (y) de la position de Zozor d'un pixel pour le faire "monter".

Notez qu'on n'est pas obligés de le déplacer d'un pixel, on pourrait très bien le déplacer 10 pixels par 10 pixels.

- Si on va vers le bas, on doit au contraire augmenter (incrémenter) l'ordonnée de Zozor (y).
- Si on va vers la droite, on augmente la valeur de l'abscisse (x).
- Si on va vers la gauche, on doit diminuer l'abscisse (x).

Et maintenant ?
En vous aidant du schéma de code donné précédemment, vous devriez être capables de diriger Zozor au clavier !

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *zozor = NULL;
    SDL_Rect positionZozor;
    SDL_Event event;
    int continuer = 1;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Gestion des évènements en SDL", NULL);

    /* Chargement de Zozor */
    zozor = SDL_LoadBMP("zozor.bmp");
    SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));

    /* On centre Zozor à l'écran */
    positionZozor.x = ecran->w / 2 - zozor->w / 2;
    positionZozor.y = ecran->h / 2 - zozor->h / 2;


    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
            case SDL_KEYDOWN:
                switch(event.key.keysym.sym)
                {
                    case SDLK_UP: // Flèche haut
                        positionZozor.y--;
                        break;
                    case SDLK_DOWN: // Flèche bas
                        positionZozor.y++;
                        break;
                    case SDLK_RIGHT: // Flèche droite
                        positionZozor.x++;
                        break;
                    case SDLK_LEFT: // Flèche gauche
                        positionZozor.x--;
                        break;
                }
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255)); /* On efface
l'écran */
        SDL_BlitSurface(zozor, NULL, ecran, &positionZozor); /* On place zozor à sa
nouvelle position */
        SDL_Flip(ecran); /* On met à jour l'affichage */
    }

    SDL_FreeSurface(zozor);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

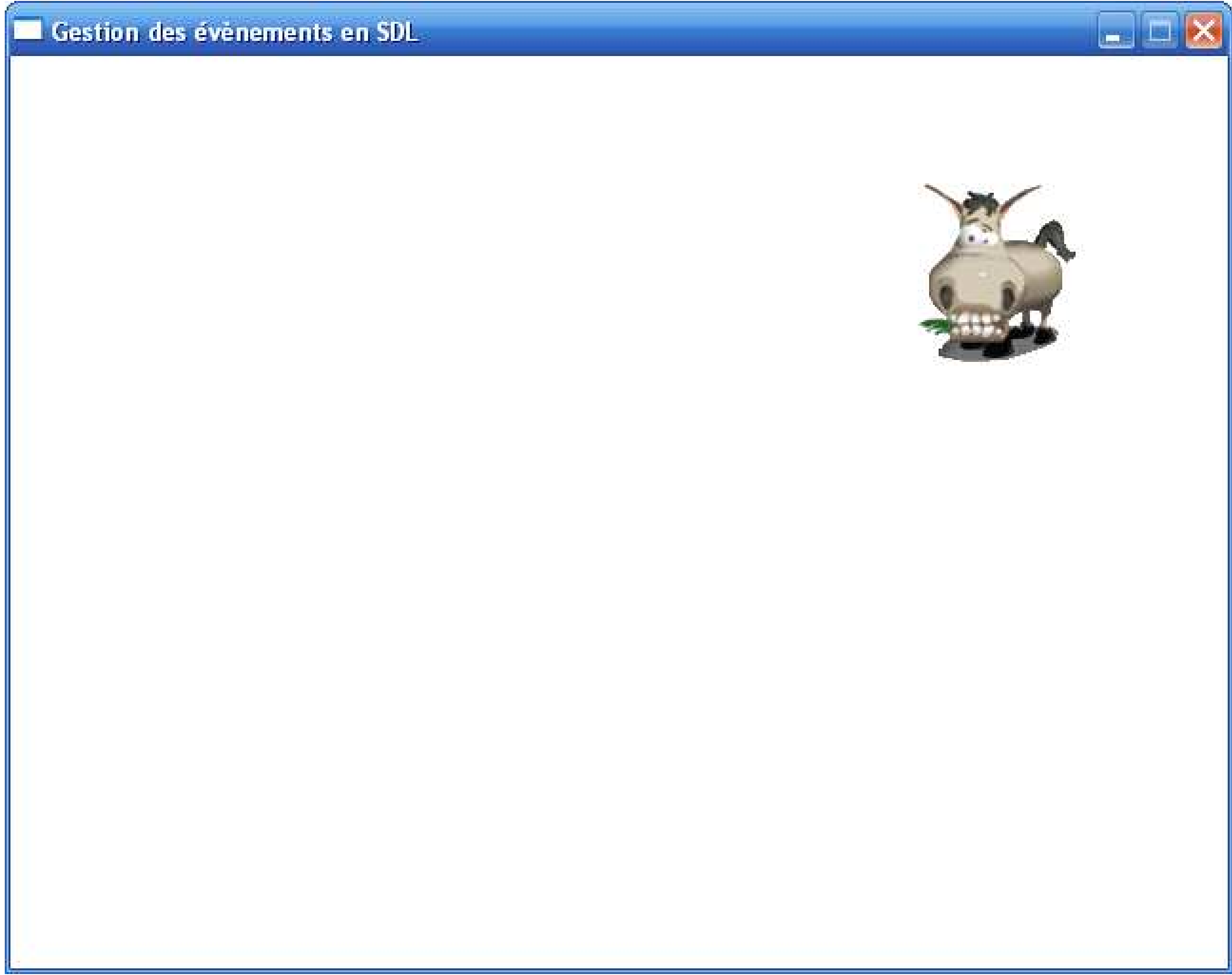
Il est primordial de bien comprendre comment est composée la boucle principale du programme. Il faut être capable de la refaire de tête. Relisez le schéma de code que vous avez vu plus haut au besoin.

Donc en résumé, on a une grosse boucle appelée "Boucle principale du programme". Elle ne s'arrêtera que si on le demande en mettant le booléen *continuer* à 0.
Dans cette boucle, on récupère d'abord un évènement à traiter. On fait un switch pour déterminer de quel type d'évènement il s'agit. En fonction de l'évènement, on effectue différentes actions. Ici, je mets à jour les coordonnées de Zozor pour donner l'impression qu'on le déplace.

Ensuite, après le switch vous devez mettre à jour votre écran :

1. Premièrement, vous effacez l'écran en faisant un SDL_FillRect (de la couleur de fond que vous voulez).
2. Ensuite, vous blittez vos surfaces sur l'écran. Ici, je n'ai eu besoin de blitter que Zozor car il n'y a que lui. Vous noterez, et c'est très important, que je blitte Zozor à *positionZozor* ! C'est là que la différence se fait : si j'ai mis à jour *positionZozor* auparavant, alors Zozor apparaîtra à un autre endroit et on aura l'impression qu'on l'a déplacé !
3. Enfin, toute dernière chose à faire : SDL_Flip. Cela ordonne la mise à jour de l'écran aux yeux de l'utilisateur.

On peut donc déplacer Zozor où l'on veut sur l'écran maintenant !



Quelques optimisations

Répétition des touches

Pour l'instant, on a un truc qui marche mais on ne peut se déplacer que d'un pixel à la fois, et on est obligés de rappuyer sur les flèches du clavier si on veut se déplacer encore d'un pixel.
Je sais pas vous, mais moi ça m'amuse moyennement de m'exciter frénétiquement sur la même touche du clavier juste pour déplacer le personnage de 200 pixels

Heureusement, il y a ~~Findus~~ *SDL_EnableKeyRepeat* !
Cette fonction permet d'activer la répétition des touches. Elle fait en sorte que la SDL regénère un évènement de type SDL_KEYDOWN si une touche est restée enfoncée un certain temps.

Cette fonction est à appeler quand vous voulez mais de préférence avant la boucle principale du programme. Elle prend 2 paramètres :

- La durée (en millisecondes) pendant laquelle une touche doit rester enfoncée avant d'activer la répétition des touches.
- Le délai (en millisecondes) entre chaque génération d'un évènement SDL_KEYDOWN une fois que la répétition a été activée.

En gros, le premier paramètre indique au bout de combien de temps on génère une répétition la première fois, et le second paramètre indique le temps qu'il faut ensuite pour que l'évènement se répète.
Personnellement, pour des raisons de fluidité, je mets la même valeur à ces 2 paramètres le plus souvent.

Essayez avec une répétition de 10ms :

Code : C

```
SDL_EnableKeyRepeat(10, 10);
```

Maintenant vous pouvez laisser une touche du clavier enfoncée.
Vous allez voir, c'est quand même mieux

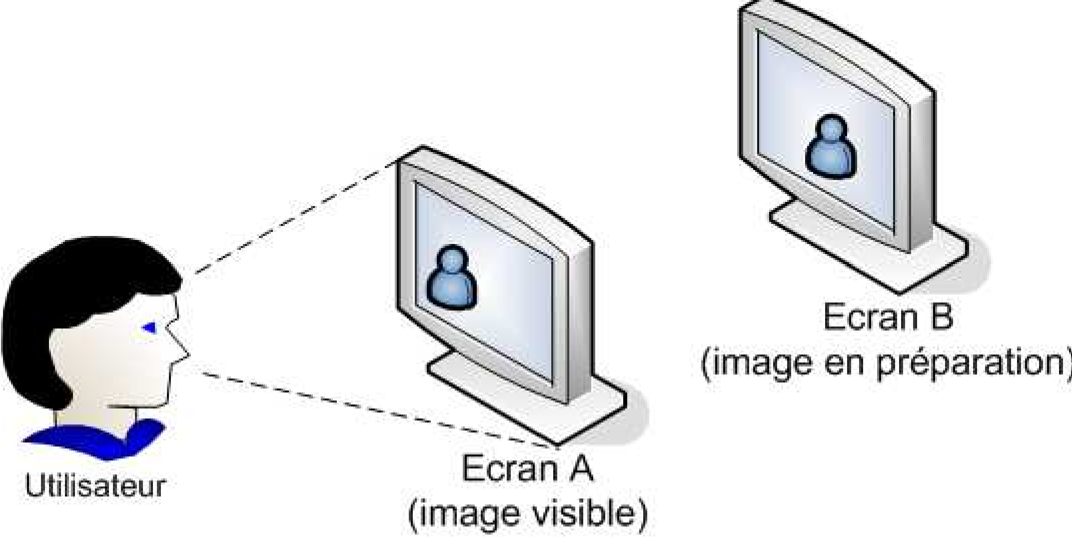
Travailler avec le double buffer

A partir de maintenant, il serait bon d'activer l'option de **double buffering** de la SDL.

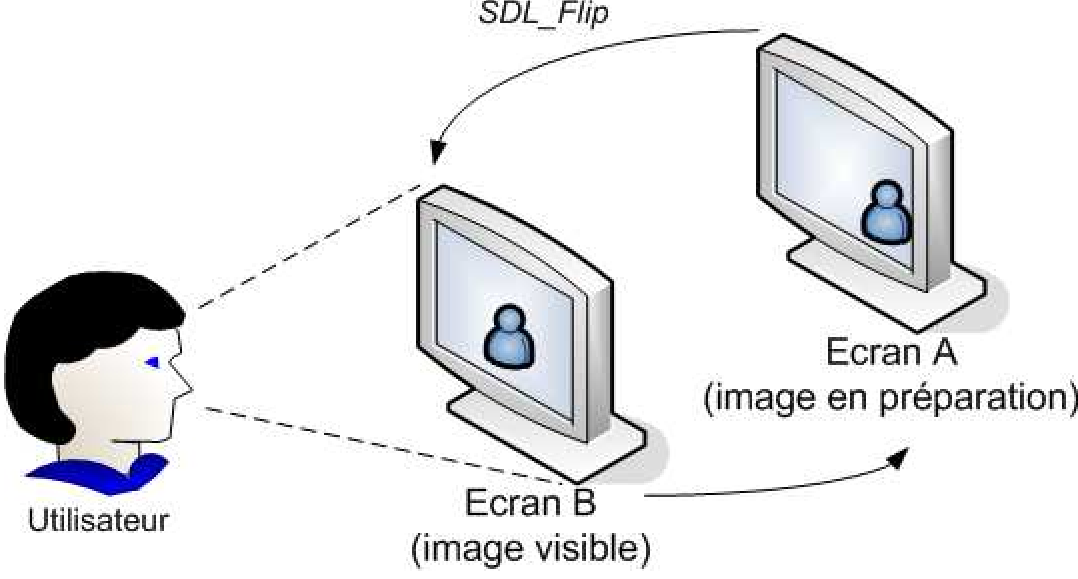
Le double buffering est une technique couramment utilisée dans les jeux. Elle permet d'éviter un scintillement de l'image.
Pourquoi l'image scintillerait-elle ? Parce que quand vous dessinez à l'écran, l'utilisateur "voit" quand vous dessinez et donc quand l'écran s'efface. Même si ça va très vite, notre cerveau perçoit un clignotement et c'est sacrément désagréable.

La technique du double buffering consiste à utiliser 2 "écrans" : l'un est réel (celui que l'utilisateur est en train de voir sur son moniteur), l'autre est virtuel (c'est une image que l'ordinateur est en train de construire en mémoire).

Ces 2 écrans alternent : l'écran A est affiché pendant que l'autre (l'écran B) en "arrière-plan" prépare l'image suivante.



Une fois que l'image en arrière-plan (l'écran B) a finie d'être dessinée, on intervertit les 2 écrans en appelant la fonction *SDL_Flip*.



L'écran A part en arrière-plan préparer l'image suivante, tandis que l'image de l'écran B s'affiche directement et instantanément aux yeux de l'utilisateur.
Résultat : aucun scintillement

C'est pas bien compliqué, vous avez juste à charger le mode vidéo en ajoutant le flag SDL_DOUBLEBUF :

Code : C

```
ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
```

Vous n'avez rien d'autre à changer dans votre code.

Le double buffering est une technique bien connue de votre carte graphique. C'est donc directement géré par le matériel et ça va très très vite

Vous vous demandez peut-être pourquoi on a déjà utilisé *SDL_Flip* auparavant ?
En fait cette fonction a 2 utilités :

- Si le double buffering est activé, elle sert à commander "l'échange" des écrans.
- Si le double buffering n'est pas activé, elle commande un rafraîchissement manuel de la fenêtre. Cette technique est valable dans le cas d'un programme qui ne bouge pas beaucoup, mais pour la plupart des jeux je recommande d'activer le double buffering.

Dorénavant, j'aurai toujours le double buffering activé dans mes codes sources (ça coûte pas plus cher et c'est mieux, de quoi se plaint-on ?)

Voici le code complet maintenant optimisé en double buffering avec la répétition des touches (seules 2 lignes ont changé par rapport à tout à l'heure) :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *zozor = NULL;
    SDL_Rect positionZozor;
    SDL_Event event;
    int continuer = 1;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF); /* Double
Buffering */
    SDL_WM_SetCaption("Gestion des évènements en SDL", NULL);

    zozor = SDL_LoadBMP("zozor.bmp");
    SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));

    positionZozor.x = ecran->w / 2 - zozor->w / 2;
    positionZozor.y = ecran->h / 2 - zozor->h / 2;

    SDL_EnableKeyRepeat(10, 10); /* Activation de la répétition des touches */

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
            case SDL_KEYDOWN:
                switch(event.key.keysym.sym)
                {
                    case SDLK_UP:
                        positionZozor.y--;
                        break;
                    case SDLK_DOWN:
                        positionZozor.y++;
                        break;
                    case SDLK_RIGHT:
                        positionZozor.x++;
                        break;
                    case SDLK_LEFT:
                        positionZozor.x--;
                        break;
                }
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
        SDL_BlitSurface(zozor, NULL, ecran, &positionZozor);
        SDL_Flip(ecran);
    }

    SDL_FreeSurface(zozor);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

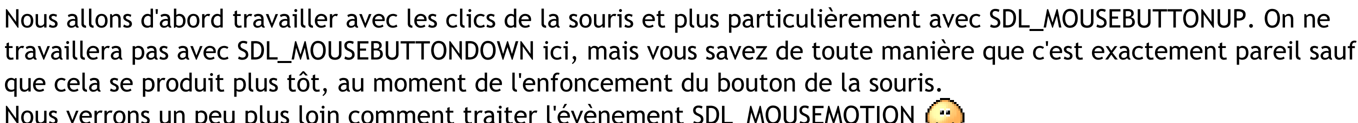
LA SOURIS

Après le clavier, attaquons maintenant la souris !

Vous vous dites peut-être que gérer la souris est plus compliqué que le clavier ?
Que nenni ! C'est même plus simple, vous allez voir

La souris peut générer 3 types d'évènements différents :

- **SDL_MOUSEBUTTONDOWN** : lorsqu'on clique avec la souris. Cela correspond au moment où le bouton de la souris est enfoncé.
- **SDL_MOUSEBUTTONUP** : lorsqu'on relâche le bouton de la souris. Tout cela fonctionne exactement sur le même principe que les touches du clavier : il y a d'abord un appui, puis un relâchement du bouton.
- **SDL_MOUSEMOTION** : lorsqu'on déplace la souris. A chaque fois que la souris bouge dans la fenêtre (ne serait-ce que d'un pixel !) un évènement SDL_MOUSEMOTION est généré !



Nous allons d'abord travailler avec les clics de la souris et plus particulièrement avec SDL_MOUSEBUTTONUP. On ne travaillera pas avec SDL_MOUSEBUTTONDOWN ici, mais vous savez de toute manière que c'est exactement pareil sauf que cela se produit plus tôt, au moment de l'enfoncement du bouton de la souris.
Nous verrons un peu plus loin comment traiter l'évènement SDL_MOUSEMOTION

Gérer les clics de la souris

Nous allons donc capturer un évènement de type SDL_MOUSEBUTTONUP (clic de la souris) puis voir quelles informations on peut récupérer.
Comme d'habitude, on va devoir rajouter un "case" dans notre switch de test, alors allons-y gaiement :

Code : C

```c
switch(event.type)
{
    case SDL_QUIT:
        continuer = 0;
        break;
    case SDL_MOUSEBUTTONUP: /* Clic de la souris */
        break;
}
```

Jusque-là, pas de difficulté majeure

Quelles informations peut-on récupérer lors d'un clic de la souris ? Il y en a 2 :

- **Le bouton de la souris** avec lequel on a cliqué (clic gauche ? clic droit ? clic bouton du milieu ?)
- **Les coordonnées de la souris** au moment du clic (x et y)

Récupérer le bouton de la souris

On va d'abord voir avec quel bouton de la souris on a cliqué.
Pour cela, il faut analyser la sous-variable *event.button.button* (non je ne bégaie pas) et comparer sa valeur avec l'une des 5 constantes :

- **SDL_BUTTON_LEFT** : clic avec le bouton gauche de la souris.
- **SDL_BUTTON_MIDDLE** : clic avec le bouton du milieu de la souris (tout le monde n'en a pas forcément un).
- **SDL_BUTTON_RIGHT** : clic avec le bouton droit de la souris.
- **SDL_BUTTON_WHEELUP** : molette de la souris vers le haut.

- **SDL_BUTTON_WHEELDOWN** : molette de la souris vers le bas.

Les 2 dernières constantes correspondent à faire tourner la molette de la souris vers le haut ou vers le bas. Elles ne correspondent pas à un "clic" avec la molette comme on pourrait le penser à tort

On va faire un test simple pour vérifier si on a fait un clic droit avec la souris. Si on a fait un clic droit, on arrête le programme (oui je sais c'est pas très original pour le moment mais ça permet de tester) :

Code : C

```
switch(event.type)
{
    case SDL_QUIT:
        continuer = 0;
        break;
    case SDL_MOUSEBUTTONUP:
        if (event.button.button == SDL_BUTTON_RIGHT) /* On arrête le programme si on a
fait un clic droit */
            continuer = 0;
        break;
}
```

Vous pouvez tester, vous verrez que le programme s'arrête si on fait un clic droit
Bref, c'est simple, c'est efficace, pas la peine de traîner 3 heures là-dessus

Récupérer les coordonnées de la souris

Voilà une information très intéressante : les coordonnées de la souris au moment du clic !
On les récupère à l'aide de 2 variables (pour l'abscisse et l'ordonnée) : *event.button.x* et *event.button.y*.

Amusons-nous un petit peu : on va blitter Zozor à l'endroit du clic de la souris.
Compliqué ? Pas du tout ! Essayez de le faire c'est un jeu d'enfant !

Voici la correction :

Code : C

```
while (continuer)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case SDL_QUIT:
            continuer = 0;
            break;
        case SDL_MOUSEBUTTONUP:
            positionZozor.x = event.button.x; /* On change les coordonnées de Zozor */
            positionZozor.y = event.button.y;
            break;
    }

    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
    SDL_BlitSurface(zozor, NULL, ecran, &positionZozor); /* On place zozor à sa nouvelle
position */
    SDL_Flip(ecran);
}
```

Ca ressemble à s'y méprendre à ce que je faisais avec les touches du clavier. Là c'est même encore plus simple : on met directement la valeur de x de la souris dans *positionZozor.x*, et de même pour y.
Ensuite on blitte Zozor à ces coordonnées-là, et voilà le travail

Gestion des évènements en SDL

Exercice trop facile pour être vrai : pour le moment, on déplace Zozor quel que soit le bouton de la souris utilisé pour le clic. Essayez de ne déplacer Zozor que si on fait un clic gauche avec la souris. Si on fait un clic droit, arrêtez le programme.
(ceux qui se plantent je viens les fouetter personnellement !)

Gérer le déplacement de la souris

Un déplacement de la souris génère un évènement de type SDL_MOUSEMOTION.
Notez bien qu'on génère autant d'évènements que de pixels dont on se déplace ! Si on bouge la souris de 100 pixels (ce qui n'est pas beaucoup), il y aura donc 100 évènements de générés.

Pas du tout, rassurez-vous il en a vu d'autres

Bon, que peut-on récupérer d'intéressant ici ?
Les coordonnées de la souris bien sûr ! On les trouve dans *event.motion.x* et *event.motion.y*

Faites attention : ce ne sont pas les mêmes variables que l'on utilise par rapport au clic de souris de toute à l'heure (avant c'était *event.button.x*). Les variables utilisées sont différentes en SDL en fonction de l'évènement.

On va placer Zozor aux mêmes coordonnées que la souris là encore. Vous allez voir, c'est rudement efficace et toujours aussi simple !

Code : C

```
while (continuer)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case SDL_QUIT:
            continuer = 0;
            break;
        case SDL_MOUSEMOTION:
            positionZozor.x = event.motion.x; /* On change les coordonnées de Zozor */
            positionZozor.y = event.motion.y;
            break;
    }

    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
    SDL_BlitSurface(zozor, NULL, ecran, &positionZozor); /* On place zozor à sa nouvelle
position */
    SDL_Flip(ecran);
}
```

Bougez votre Zozor à l'écran. Que voyez-vous ?
Il suit naturellement la souris où que vous alliez
C'est beau, c'est rapide, c'est fluide (vive le double buffering).

Quelques autres fonctions avec la souris

Nous allons voir 2 fonctions très simples en rapport avec la souris, puisque nous y sommes
Ces fonctions vous seront très probablement utiles bientôt.

Masquer la souris

On peut masquer le curseur de la souris très facilement.
Il suffit d'appeler *SDL_ShowCursor* et de lui envoyer un flag :

- **SDL_DISABLE** : masque le curseur de la souris
- **SDL_ENABLE** : réaffiche le curseur de la souris

Par exemple :

Code : C

```
SDL_ShowCursor(SDL_DISABLE);
```

Le curseur de la souris restera masqué tant qu'il sera à l'intérieur de la fenêtre.
Masquez de préférence le curseur avant la boucle principale du programme. Pas la peine en effet de le masquer à

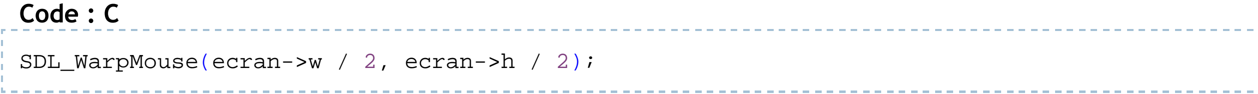
chaque tour de boucle, une seule fois suffit

Placer la souris à un endroit précis

On peut placer manuellement la souris aux coordonnées que l'on veut dans la fenêtre.
On utilise pour cela *SDL_WarpMouse* qui prend pour paramètres les coordonnées x et y où la souris doit être placée.

Par exemple, le code suivant place la souris au centre de l'écran :

Code : C



```c
SDL_WarpMouse(ecran->w / 2, ecran->h / 2);
```

Lorsque vous faites un WarpMouse, un évènement de type SDL_MOUSEMOTION sera généré. Ben oui, la souris a bougé ! Même si c'est pas l'utilisateur qui l'a fait, il y a quand même eu un déplacement

Il faut un peu de temps avant de se faire à la gestion des évènements. Passez donc le temps qu'il faudra pour être à l'aise, surtout avec la boucle de traitement des évènements.

Quelques exercices

Avant de vous laisser, voici quelques idées d'exercices que je vous recommande de faire pour vous entraîner :

Changement de la couleur de fond

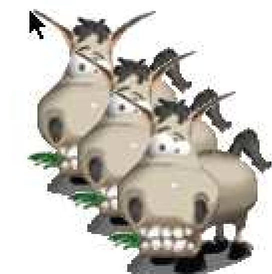
Faites un programme ouvrant une fenêtre initialement noire.
Si on appuie sur la flèche "haut" du clavier, le fond doit se blanchir progressivement : couleur 1, 1, 1, puis couleur 2, 2, 2, jusqu'à la couleur 255, 255, 255 (le blanc).
La flèche "bas" du clavier, elle, doit noircir l'écran.

Utilisez SDL_EnableKeyRepeat pour qu'on puisse changer la couleur de la fenêtre en laissant la touche enfoncée.

Attention : faites un test pour vérifier qu'on ne dépasse pas 255 et un autre pour vérifier si on ne va pas en-dessous de 0 ! Pour info, la couleur 300, 300, 300 n'existe pas

Traînée de Zozor

Faites un programme dans lequel 3 Zozors suivent la souris. Placez le premier à la position de la souris (comme on l'a fait dans le cours), puis le second 20 pixels en bas à droite, le troisième encore 20 pixels en bas à droite.



Vous n'avez besoin que d'une seule surface Zozor pour faire ça. Vous devrez juste la blitter 3 fois à des positions différentes sur l'écran, c'est tout.
Pour retenir 3 positions différentes, je vous recommande de créer un tableau de 3 SDL_Rect que vous mettrez à jour à des positions différentes à chaque évènement SDL_MOUSEMOTION.

Le tampon Zozor

Faites en sorte que lorsqu'on clique avec la souris sur l'écran, ça colle un Zozor à l'endroit indiqué.
Oui je sais, on l'a déjà fait pour étudier l'évènement "clic de la souris", mais cette fois je veux que l'on puisse "coller" à l'écran 10 Zozors maximum à la fois (alors qu'auparavant on ne pouvait en coller qu'un seul à la fois).

Cet exercice est en fait assez similaire au précédent : vous n'avez besoin que d'une surface Zozor, mais de plusieurs SDL_Rect (faites un tableau).
La difficulté sera de savoir comment initialiser ces positions, car il ne faut pas qu'il y ait de Zozors affichés à l'écran au départ. A vous de trouver une solution pour ne pas blitter de Zozor si, par exemple, les coordonnées sont (-1, -1).

La touche "Suppr" doit servir à effacer l'écran (il faudra réinitialiser toutes les coordonnées à (-1, -1) par exemple).

Si vous arrivez à faire tout ça, vous pouvez aller plus loin en faisant un "jeu de tampons" : on sélectionnera un tampon différent en appuyant sur une touche numérique (0 à 9) qu'on pourra ensuite coller à l'écran en cliquant avec la souris. On peut faire un mini-Paint pour les enfants, à condition d'avoir créé des tampons intéressants (vous avez déjà un Zozor et un sapin, à vous d'imaginer d'autres éléments comme un soleil, un nuage)

Allez au boulot les programmeurs en herbe

Que va-t-on faire maintenant ?

Si vous êtes un peu imaginatifs, vous devriez maintenant **être capables de réaliser de véritables jeux en SDL** ! Vous savez en effet pratiquement tout ce qu'il faut savoir sur le clavier et la souris et vous pouvez donc faire dès à présent un grand nombre de jeux intéressants.
Dans peu de temps nous ferons un TP pour rassembler tout ce qu'on a appris sur la SDL afin de créer un premier jeu (et un jeu sérieux attention, le temps du "Plus ou moins" est loin maintenant)

Bien entendu, il vous reste encore pas mal de choses à découvrir dans la SDL, comme la gestion du joystick, l'écriture de texte à l'écran, la gestion du son etc... Tout vient à point à qui sait attendre dit-on

Vu qu'il y a beaucoup de fonctions à retenir, **je vous conseille vraiment de ne pas aller trop vite sinon vous ne retiendrez rien**. "Ne mettez pas la charrue avant les boeufs" : commencez par faire des choses simples, puis compliquez-les au fur et à mesure. Ne commencez pas par quelque chose de compliqué dès le début, sinon c'est le ramassage de dents assuré !

La gestion des évènements (Partie 2/2)

Nous n'avons pas encore vu tous les évènements !
Comme il existe de nombreux évènements différents, je ne pouvais pas tout mettre dans un même chapitre. J'ai donc préféré le couper en 2

Dans ce chapitre, nous allons étudier les évènements :

- Liés au joystick
- Liés à la fenêtre

La gestion du joystick est un peu délicate vous allez voir. Toutefois, comme la SDL nous permet de manier le joystick

quel que soit le système d'exploitation (Windows, Mac, Linux)... je ne pouvais pas passer à côté !

INITIALISER LE JOYSTICK

En plus du clavier et de la souris, qui sont des outils de contrôle courants, la SDL gère aussi... **le joystick** !

Vous pensez peut-être que le joystick est plus difficile à gérer parce qu'il y a des boutons, un manche qu'on peut plus ou moins incliner etc... Eh bien, la manipulation du joystick nécessite quelques initialisations supplémentaires c'est vrai, mais à part ça c'est aussi facile à gérer que le clavier et la souris alors ne nous en privons pas
Grâce à ce que vous allez apprendre, vous serez capables de créer un jeu qu'on peut manipuler au clavier, à la souris ou au joystick (ou avec les 3 à la fois)

Initialiser le joystick

Si vous comptez utiliser un joystick dans votre programme, vous devrez effectuer quelques initialisations supplémentaires. Le clavier et la souris, eux, n'ont pas besoin d'être initialisés comme vous l'avez vu.

Initialisation du système de joystick de la SDL

Pour commencer, vous devez indiquer à *SDL_Init* que vous comptez utiliser le joystick dans votre programme.

Jusqu'ici, vous ne devriez avoir envoyé qu'un seul flag à cette fonction :

Code : C

```c
SDL_Init(SDL_INIT_VIDEO);
```

Tout ce que vous devez faire, c'est rajouter le flag SDL_INIT_JOYSTICK que vous combinez au précédent à l'aide du symbole "|" :

Code : C

```c
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK);
```

Voilà, la SDL sait maintenant qu'elle va devoir gérer les joysticks

Compter le nombre de joysticks

Contrairement au clavier et la souris qu'on ne branche pas en double sur un ordinateur, il se peut que vous ayez plusieurs joysticks branchés (que ce soit sur le port de jeu ou sur un port USB).

On va compter le nombre de joysticks installés sur votre système. Pour ce faire, il suffit d'appeler *SDL_NumJoysticks()* qui renvoie le nombre de joysticks. Cette fonction ne prend aucun paramètre.

Le problème, ça va être pour afficher la valeur. En SDL, il n'y a pas de fonction pour afficher du texte dans la fenêtre. Heureusement, nous apprendrons dans quelques chapitres qu'à l'aide d'une librairie supplémentaire (dans le même genre que SDL_Image) il sera possible d'afficher du texte dans une fenêtre SDL.
Mais... pour le moment, on n'en est pas là

On va utiliser la bonne vieille fonction *printf* :

Code : C

```
printf("Il y a %ld joysticks connectés\n", SDL_NumJoysticks());
```

Mais... s'il n'y a plus de console à l'écran où va s'afficher ce texte ?

- Si vous êtes sous Linux, la console est toujours en arrière-plan donc vous devriez voir le texte s'afficher dans la console
- Si vous êtes sous Windows, il ne peut pas y avoir de console et de fenêtre en même temps. Windows redirige tout le texte que vous écrivez à l'aide de *printf* dans un fichier stdout.txt situé dans le dossier de votre exécutable (tout comme stderr.txt)

Sous Windows vous devrez donc ouvrir le fichier stdout.txt qui sera automatiquement créé par le système d'exploitation lorsque vous utiliserez printf. Ce fichier reste sur le disque dur même après exécution du programme. Vous pourrez le supprimer manuellement sans crainte, il sera simplement recréé lors du prochain démarrage du programme.

Voici ce que printf a écrit chez moi :

Code : Console

```
Il y a 1 joysticks connectés
```

Cela signifie (je pense que vous l'aurez compris) que j'ai 1 joystick branché sur mon ordinateur.
Si le joueur a 0 joystick, il ne pourra pas manipuler votre programme au joystick (ça peut sembler logique) il faudra donc le rabattre sur une manipulation plus classique au clavier et à la souris.

Lister les noms des joysticks

La SDL nous permet d'avoir le nom des joysticks installés sur l'ordinateur. Cela devrait permettre de laisser au joueur le choix du joystick qu'il veut utiliser.

Je ne connais pas beaucoup de gens qui aient plusieurs joysticks branchés sur leur ordinateur, mais la SDL prévoit qu'il puisse y en avoir plusieurs ce qui n'est pas plus mal

Les joysticks sont numérotés de 0 à... autant qu'il y a de joysticks
Si vous n'avez qu'un joystick, ce sera donc le n°0.

On récupère le nom du joystick n°X en appelant : *SDL_JoystickName(X)*. Cette fonction prend donc un paramètre : le numéro du joystick dont on veut le nom.
Puisqu'on y est, on va faire une boucle pour lister tous les joysticks présents sur l'ordinateur :

Code : C

```
for (i = 0 ; i < SDL_NumJoysticks() ; i++ )
{
    printf("Joystick n°%ld : %s\n", i, SDL_JoystickName(i));
}
```

Voici le résultat que ça donne chez moi :

Code : Console

```
Il y a 1 joysticks connectés
-> Joystick n°0 : Boîtier de commandes Microsoft SideWinder
```

Comme vous le voyez, je possède un joystick appelé "Boîtier de commandes Microsoft SideWinder".

Charger le joystick désiré

Bien, maintenant on va dire à la SDL le numéro du joystick qu'on veut utiliser dans notre programme.
Dans mon cas ça ne sera pas très compliqué, je vais utiliser mon Joystick n°0

Si vous ne savez pas quel numéro de joystick choisir, prenez le n°0 (à condition qu'il y ait au moins un joystick sur l'ordinateur).
Plus tard, lorsque vous serez plus à l'aise avec la SDL, vous serez capables de demander au joueur le joystick qu'il veut utiliser.

On va avoir besoin de créer un pointeur de type SDL_Joystick pour manipuler le joystick. Vous pouvez donc rajouter la ligne suivante au début du *main* :

Code : C

```
SDL_Joystick *joystick = NULL;
```

Ensuite, vous allez devoir appeler 2 fonctions :

Code : C

```
SDL_JoystickEventState(SDL_ENABLE);
joystick = SDL_JoystickOpen(0);
```

La première active la gestion des évènements des joysticks.
La seconde "ouvre" (= charge) le joystick n°0, c'est-à-dire le premier joystick. Le résultat doit être stocké dans une variable *joystick* de type SDL_Joystick.

A la fin de votre programme (avant SDL_Quit), vous devrez "fermer" le joystick en appelant :

Code : C

```
SDL_JoystickClose(joystick);
```

LES ÉVÈNEMENTS DU JOYSTICK

Les propriétés du joystick

Nous avons vu comment charger un joystick.
Maintenant que c'est fait, essayons de récupérer un maximum d'infos sur ce joystick ouvert.

Il faut savoir qu'un joystick peut avoir 4 propriétés différentes, chacune pouvant générer des évènements :

- Les boutons

- Les axes du stick directionnel
- Les trackballs, s'il y en a.
- Les chapeaux ("hats" en anglais) s'il y en a. On en trouve sur certains joysticks un peu perfectionnés.

Je possède personnellement un joystick assez simple. C'est un Sidewinder de Microsoft, comme je vous l'ai dit précédemment :

Sur mon joystick, il n'y a que des boutons et un stick directionnel (donc des "axes"). Il n'y a ni trackballs ni chapeaux. Je doute d'ailleurs fortement qu'il existe un joystick avec tous ces éléments à la fois (ou alors c'est une vraie machine de guerre).

Bref, tout ça pour dire que je ne pourrai vous présenter que les évènements "boutons" et "axes". Heureusement, ce sont les plus courants et vous pourrez déjà tout faire avec ça. Le jour où vous aurez besoin de gérer des trackballs ou des chapeaux (ce qui n'est pas courant), vous serez déjà loin et bien meilleur que moi

Lister le nombre de boutons, d'axes etc...

Il y a 4 fonctions pour compter le nombre de chacun de ces éléments (boutons, axes, trackballs et chapeaux) présents sur le joystick.
Vous pouvez tester avec ce code :

Code : C

```
printf("----> %d boutons\n", SDL_JoystickNumButtons(joystick));
printf("----> %d axes\n", SDL_JoystickNumAxes(joystick));
printf("----> %d trackballs\n", SDL_JoystickNumBalls(joystick));
printf("----> %d chapeaux\n", SDL_JoystickNumHats(joystick));
```

Attention, le joystick doit avoir été ouvert au préalable. En effet, comme vous le voyez il faut envoyer à ces fonctions notre pointeur *joystick*.

Le résultat donné pour mon Sidewinder est le suivant :

Code : Console

```
----> 10 boutons
----> 2 axes
----> 0 trackballs
----> 0 chapeaux
```

Je possède donc 10 boutons et 2 axes : un axe de haut en bas, un autre axe de gauche à droite.
Je ne possède aucun trackball et aucun chapeau, comme prévu.

Les évènements du joystick

Comme je vous l'ai dit, un joystick peut générer de nombreux évènements parce qu'il y a plusieurs types de joysticks.
Je vais parler des évènements les plus courants ici : les boutons et les axes.

Les boutons

Tout d'abord, commençons par un évènement commun à tous les joysticks : **l'appui sur un bouton** !

Vous avez 2 évènements possibles :

- **SDL_JOYBUTTONDOWN** : appui sur un bouton
- **SDL_JOYBUTTONUP** : relâchement d'un bouton

Bref, classique

On récupère le numéro du bouton enfoncé à l'aide de la variable : *event.jbutton.button* :

Code : C

```c
switch(event.type)
{
    case SDL_QUIT:
        continuer = 0;
        break;
    case SDL_JOYBUTTONDOWN:
        if (event.jbutton.button == 0) /* Arrêt du programme si on appuie sur le 1er
bouton */
            continuer = 0;
        break;
}
```

Ici, le programme devrait s'arrêter si on appuie sur le bouton n°0 (le premier bouton)

Le stick directionnel (axe)

Un déplacement de l'axe du joystick génère un évènement de type SDL_JOYAXISMOTION.

Les évènements liés aux "axes" peuvent être générés soit par un stick directionnel (comme c'est le cas sur mon Sidewinder), soit par un manche comme c'est le cas sur les joysticks un peu plus chers que l'on utilise pour les jeux d'avion.

Quelle est la différence concrètement entre un stick directionnel et un manche ?

On peut contrôler plus finement l'inclinaison d'un manche que celle d'un stick directionnel. Un manche est donc

parfait pour un jeu d'avion. En revanche, le stick directionnel est toujours poussé "à fond", on ne peut pas contrôler son inclinaison.

Si je vous dis tout ça, c'est parce que la SDL nous renvoie l'inclinaison du manche. On la récupère dans event.jaxis.value. C'est un nombre négatif si on se déplace vers la gauche ou vers le haut, et un nombre positif si on se déplace vers la droite ou vers le bas.

En général, on teste l'inclinaison comme ceci :

Code : C

```
case SDL_JOYAXISMOTION:
    if (event.jaxis.value < -3200 || event.jaxis.value > 3200)
    {
        /* Le manche (ou le stick) a été déplacé de sa position initiale */
    }
    break;
```

Si on rentre dans le if, c'est que le manche a été poussé (mais on ne sait pas encore dans quelle direction).
En temps normal, le manche est en position centrale et event.jaxis.value vaut 0.

Sur mon Sidewinder, comme il n'y a qu'un stick directionnel, des valeurs extrêmes sont générées pour *value* : -32768 et 32767 (selon le sens dans lequel on se dirige). Si vous avez un manche, vous pouvez utiliser *value* pour gérer plus finement le déplacement (comme l'inclinaison d'un vaisseau).

Bon, maintenant qu'on sait que le manche a été un minimum déplacé, on veut connaître la direction.
Pour ça, on doit regarder la variable *event.jaxis.axis*. Elle vaut :

- 0 si c'est un mouvement dans l'axe gauche-droite.
- 1 si c'est un mouvement dans l'axe haut-bas.

En combinant intelligemment *event.jaxis.axis* et *event.jaxis.value*, on peut donc savoir dans quelle direction on veut se déplacer :

Code : C

```
case SDL_JOYAXISMOTION:
    if (event.jaxis.axis == 0 && event.jaxis.value < -3200) /* Vers la gauche */
        positionZozor.x--;
    else if (event.jaxis.axis == 0 && event.jaxis.value > 3200) /* Vers la droite */
        positionZozor.x++;
    else if (event.jaxis.axis == 1 && event.jaxis.value < -3200) /* Vers le haut */
        positionZozor.y--;
    else if (event.jaxis.axis == 1 && event.jaxis.value > 3200) /* Vers le bas */
        positionZozor.y++;
    break;
```

Et voilà le travail !

LES ÉVÈNEMENTS DE LA FENÊTRE

Il ne nous reste plus qu'à voir les évènements liés à la fenêtre et nous aurons vu pratiquement tous les évènements de la SDL !
Je sais, ça demande du travail, mais c'est vraiment la base de tout programme écrit en SDL, donc autant mettre le paquet

Quels évènements peuvent être générés par la fenêtre ?

- Lorsque la fenêtre est redimensionnée.
- Lorsque la fenêtre est réduite en barre des tâches ou restaurée.
- Lorsque la souris se trouve à l'intérieur de la fenêtre ou lorsqu'elle en sort.
- Lorsque la fenêtre est active (au premier plan) ou lorsqu'elle n'est plus active.

Y'a de quoi faire !
Commençons par l'évènement généré lors du redimensionnement de la fenêtre.

Le redimensionnement de la fenêtre

Par défaut, une fenêtre SDL ne peut pas être redimensionnée par l'utilisateur.
Je vous rappelle que pour changer ça, il faut ajouter le flag SDL_RESIZABLE dans la fonction SDL_SetVideoMode

Code : C

```
ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF | SDL_RESIZABLE);
```

Lorsque l'utilisateur redimensionne la fenêtre, un évènement SDL_VIDEORESIZE est généré.

Vous pouvez récupérer :

- La nouvelle largeur dans *event.resize.w*
- La nouvelle hauteur dans *event.resize.h*

Code : C

```
case SDL_VIDEORESIZE:
    positionZozor.x = event.resize.w / 2 - zozor->w / 2;
    positionZozor.y = event.resize.h / 2 - zozor->h / 2;
    break;
```

Avec ce code, Zozor sera toujours centré dans la fenêtre quelle que soit sa taille

Visibilité de la fenêtre

L'évènement SDL_ACTIVEEVENT est généré lorsque la visibilité de la fenêtre change.
Cela peut être dû à de nombreuses choses :

- La fenêtre est réduite en barre des tâches ou restaurée.
- La souris se trouve à l'intérieur de la fenêtre ou en sort.
- La fenêtre est active (au premier plan) ou n'est plus active.

En programmation, on parle de **focus**.
Lorsqu'on dit qu'une application a le focus, c'est que le clavier ou la souris de l'utilisateur se trouve dedans.
Tous les clics ou appuis des touches du clavier que vous ferez seront envoyés à la fenêtre qui a la focus et pas aux autres.
Une seule fenêtre peut avoir le focus à la fois (vous ne pouvez pas avoir 2 fenêtres au premier plan en même temps !).

Vu que plusieurs choses peuvent s'être passées, il faut regarder dans des variables pour en savoir plus :

- **event.active.gain** : indique si l'évènement est un gain (1) ou une perte (0). Par exemple, si la fenêtre est passée en arrière-plan c'est une perte (0), si elle est remise au premier plan c'est un gain (1).
- **event.active.state** : c'est une combinaison de flags indiquant le type d'évènement qui s'est produit. Voici la liste des flags possibles :
  - **SDL_APPMOUSEFOCUS** : la souris vient de rentrer ou de sortir de la fenêtre.
    Il faut regarder la valeur de *event.active.gain* pour savoir si elle est rentrée (gain = 1) ou sortie (gain = 0) de la fenêtre.
  - **SDL_APPINPUTFOCUS** : l'application vient de recevoir le focus du clavier ou de le perdre. Cela signifie grosso modo que votre fenêtre vient d'être mise au premier plan ou en arrière-plan.
    Il faut regarder la valeur de *event.active.gain* pour savoir si la fenêtre a été mise au premier plan (gain = 1) ou en arrière-plan (gain = 0).
  - **SDL_APPACTIVE** : l'applicaton a été iconifiée, c'est-à-dire réduite dans la barre des tâches (gain = 0), ou restaurée dans son état normal (gain = 1).

Vous suivez toujours ?
Il faut donc bien comparer les valeurs des 2 variables pour savoir exactement ce qui s'est produit.

Tester la valeur d'une combinaison de flags

Le seul problème, c'est que *event.active.state* est une combinaison de flags. Cela signifie que dans un évènement il peut se produire 2 choses à la fois (par exemple si on réduit la fenêtre dans la barre des tâches, on perd aussi le focus du clavier et de la souris).
Il va donc falloir faire un test un peu plus compliqué que (par exemple) : `if (event.active.state == SDL_APPACTIVE)`

Pourquoi est-ce que c'est plus compliqué ?

Parce que c'est une combinaison de bits. Je ne vais pas vous faire un cours sur les opérations logiques bit à bit ici, ça serait un peu trop pour le cours et vous n'avez pas besoin de connaître le détail.
Je vais vous donner le code qu'il faut utiliser pour tester si un flag est présent dans une variable sans rentrer dans les détails

Pour tester s'il y a eu un changement de focus de la souris, on doit taper le code suivant :

Code : C
```c
if ((event.active.state & SDL_APPMOUSEFOCUS) == SDL_APPMOUSEFOCUS)
```

Il n'y a pas d'erreur. Attention c'est précis : il faut un seul & et deux =, et il faut bien mettre les parenthèses comme je l'ai fait

Ca marche de la même manière pour les autres évènements. Par exemple :

Code : C

```
if ((event.active.state & SDL_APPACTIVE) == SDL_APPACTIVE)
```

Tester l'état et le gain à la fois

Dans la pratique, vous voudrez sûrement tester l'état et le gain à la fois. Vous pourrez ainsi savoir exactement ce qui s'est passé.

Supposons que vous ayez un jeu qui fait faire beaucoup de calculs à l'ordinateur. Vous voulez que le jeu se mette en pause automatiquement lorsque la fenêtre est réduite et qu'il se relance lorsque la fenêtre est réagrandie. Cela évite que le jeu ne continue pendant que le joueur n'est plus dessus, et cela évite aussi au processeur de faire trop de calculs par la même occasion.

Le code ci-dessous met en pause le jeu en activant par exemple un booléen *pause* à 1. Il remet en marche le jeu en désactivant le booléen à 0.

Code : C

```
if ((event.active.state & SDL_APPACTIVE) == SDL_APPACTIVE)
{
    if (event.active.gain == 0) /* La fenêtre a été réduite en barre des tâches */
        pause = 1;
    else if (event.active.gain == 1) /* La fenêtre a été restaurée */
        pause = 0;
}
```

Bien sûr ce code n'est pas complet. Ce sera à vous d'écrire le code pour tester l'état de la variable *pause* pour savoir s'il faut effectuer les calculs ou pas.

Ce qui compte, c'est que vous compreniez l'idée générale

Je vous laisse faire d'autres tests pour par exemple vérifier si la souris est à l'intérieur ou à l'extérieur de la fenêtre. Vous n'avez qu'à faire bouger Zozor vers la droite lorsque la souris rentre dans la fenêtre et le faire bouger vers la gauche lorsqu'elle en sort Vous comprenez un peu mieux maintenant pourquoi j'ai tenu à séparer le chapitre sur les évènements en 2. Il y avait vraiment beaucoup à dire !

En gros, on peut résumer les 2 chapitres comme ceci :

- **La partie 1/2** : elle portait sur le clavier et la souris. Ce sont des évènements que vous aurez souvent à gérer (pour ne pas dire tout le temps). Ils sont faciles à manipuler.
- **La partie 2/2** : en revanche, dans ce chapitre nous avons vu les évènements liés au joystick et à la fenêtre qui sont un peu plus rares. Ce sont des évènements un peu plus compliqués à manipuler : pour le joystick il faut faire pas mal de vérifications et d'ouvertures préalables, et pour la fenêtre il faut tester plusieurs variables à la fois et savoir vérifier si une valeur se trouve dans une combinaison de flags !

Personnellement je n'ai jamais créé de jeu utilisant le joystick à l'heure actuelle.
Enfin bon, tout est faisable comme vous l'avez vu

Le chapitre suivant sera un TP. Ce TP vous montrera la création d'un jeu complet (Mario Sokoban) de A à Z en SDL. Nous n'utiliserons pas le joystick ni les évènements de la fenêtre que nous avons étudiés dans ce chapitre. Toutefois, si vous désirez ajouter la possibilité d'utiliser le joystick dans le jeu, n'hésitez pas ! Ca fera un bon entraînement !

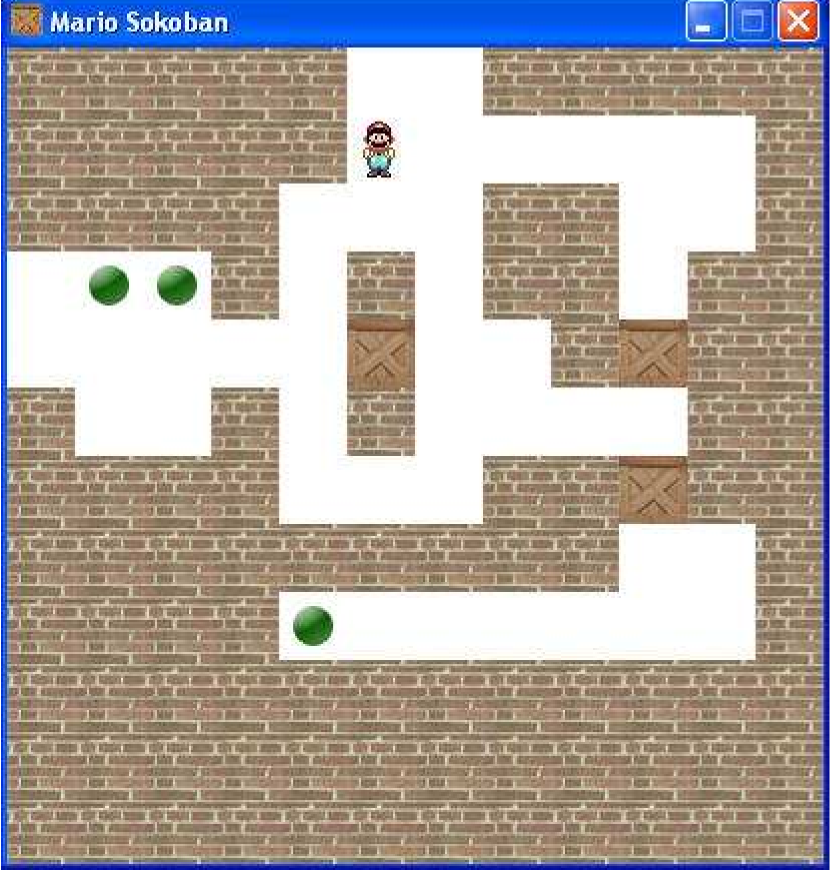
TP : Mario Sokoban

Attaquons maintenant **notre premier TP utilisant la SDL**

Cette fois, je crois vous l'aurez compris, notre programme ne sera pas une console mais bel et bien une fenêtre

Quel va être le sujet du TP aujourd'hui ? Il va s'agir d'un jeu de **Sokoban** !
Peut-être que ce nom ne vous dit rien, mais le jeu je suis à peu près certain que vous le connaissez, c'est un classique des casse-têtes.

Voici une capture d'écran du programme que nous allons réaliser :



Alors, ça vous dit quelque chose maintenant ?

Cahier des charges du Sokoban

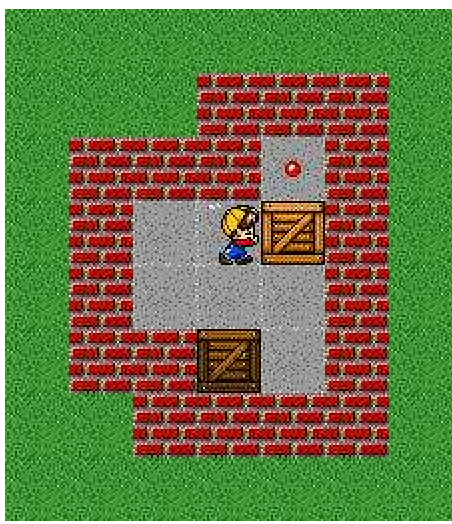
A propos du Sokoban

"Sokoban" est un terme japonais qui signifie "Magasinier".
Il s'agit d'un casse-tête inventé dans les années 80 par Hiroyuki Imabayashi. Le jeu a remporté un concours de programmation à l'époque.

Le but du jeu

Il est simple à comprendre : vous dirigez un personnage dans un labyrinthe. Il doit pousser des caisses pour les amener à des endroits précis. Le joueur ne peut pas déplacer 2 caisses à la fois.

Si le principe se comprend vite et bien, cela ne veut pas dire pour autant que le jeu est toujours facile. Il est en effet possible de réaliser des casse-têtes vraiment... prise de tête



Pourquoi avoir choisi ce jeu ?

Parce que c'est un jeu populaire, que je trouve intéressant et qu'on peut réaliser rien qu'avec ce qu'on a appris dans les cours de ce site.
Alors bien sûr, ça demande de l'organisation. La difficulté n'est pas vraiment dans le codage mais dans l'organisation. Il va en effet falloir découper notre programme en plusieurs fichiers .c intelligement, essayer de trouver les bonnes fonctions à créer etc.

C'est aussi pour cette raison que j'ai décidé cette fois de ne pas construire le TP comme les précédents : je ne vais pas vous donner des indices et puis une correction à la fin. Au contraire : je vais vous montrer comment je réalise tout le projet de A à Z.

Pas de problème ! Allez-y lancez-vous, c'est même très bien !
Il vous faudra certainement un peu de temps : personnellement ça m'a pris une bonne petite journée, et encore c'est parce que j'ai un peu l'habitude de programmer et que j'évite certains pièges courants. Ca ne m'a pas empêché de me prendre la tête à 2 ou 3 reprises quand même

Sachez qu'un tel jeu peut être codé de nombreuses façons différentes. Je vais vous montrer ma façon de faire : ce n'est pas la meilleure, mais ce n'est pas la plus mauvaise non plus
Le TP se terminera par une série de suggestions d'améliorations, et je vous proposerai de télécharger le code source complet bien entendu.

Encore une fois : **je vous conseille d'essayer de vous y lancer par vous-mêmes.** Passez-y 2 ou 3 jours et faites de votre mieux. Il est important que vous pratiquiez.

Le cahier des charges

Le cahier des charges, c'est un document dans lequel on écrit tout ce que le programme doit savoir faire.
En l'occurence, que veut-on que notre jeu soit capable de faire ? C'est le moment de se décider !

Voici ce que je propose :

- Le joueur doit pouvoir se déplacer dans un labyrinthe et pousser des caisses.
- Il ne peut pas pousser 2 caisses à la fois.
- Une partie est considérée comme gagnée lorsque toutes les caisses sont sur des objectifs
- Les niveaux de jeu seront enregistrés dans un fichier (par exemple "niveaux.lvl").
- Un éditeur sera intégré au programme pour que n'importe qui puisse créer ses propres niveaux (bah oui tant qu'on y est hein)

Voilà ça fait déjà pas mal je crois
Il y a des choses que notre programme ne saura pas faire, ça aussi il faut le dire :

- Notre programme ne pourra gérer qu'un seul niveau à la fois. Si vous voulez coder une "aventure" avec une suite de niveaux, vous n'aurez qu'à le faire vous-mêmes à la fin de ce tp
- Il n'y aura pas de gestion du temps écoulé (on sait pas faire ça encore) ni du score.

En fait, avec tout ce qu'on veut faire déjà (notamment l'éditeur de niveaux), y'en a pour un petit moment.

Je vous indiquerai à la fin du TP une liste d'idées pour améliorer le programme. Et ce ne seront pas des paroles en l'air, car ce sont des idées que j'aurai moi-même implémentées dans une version plus complète du programme que je vous proposerai de télécharger.
En revanche, je ne vous donnerai pas les codes sources de la version "complète" pour vous forcer à travailler (faut pas abuser)

Récupérer les sprites du jeu

Dans la plupart des jeux 2D (que ce soit des jeux de plateforme ou de casse-tête comme ici), on appelle les images qui composent le jeu des **sprites**.
Dans notre cas, j'ai décidé qu'on créerait un Sokoban mettant en scène Mario (d'où le nom "Mario Sokoban"). Comme Mario est un personnage populaire dans le monde de la 2D, on n'aura pas trop de mal à trouver des sprites de Mario.

Il faudra aussi trouver des sprites pour les murs de briques, les caisses, les objectifs etc.

Si vous faites une recherche sur Google de "sprites", vous trouverez de nombreuses réponses. En effet, il y a pas mal de sites qui proposent de télécharger des sprites de jeux 2D auxquels vous avez sûrement joué par le passé

Personnellement, je me suis servi sur : **http://www.panelmonkey.org/**
Vous en trouverez aussi sur : **http://www.videogamesprites.net/**

... et bien entendu **Google** vous en donnera d'autres

Voici les sprites que j'ai récupérés. J'ai fait un peu de retouches sur certains pour qu'ils soient à la bonne dimension :

Sprite	Description
	Un mur
	Une caisse
	Une caisse placée sur un objectif. J'ai juste légèrement "rougi" la caisse sous Photoshop.
	Un objectif (où l'on doit mettre une caisse).
	Le joueur (Mario) orienté vers le bas
	Mario vers la droite
	Mario vers la gauche
	Mario vers le haut

Télécharger toutes les images

Voilà, avec ça on a tout ce qu'il faut
Comme vous le voyez, les images ont parfois eu besoin de légères retouches. Ce ne sont pas des retouches difficiles à faire (même moi qui suis un gros nul sous Photoshop je m'en suis sorti, alors pourquoi pas vous)

Notez que j'aurais très bien pu n'utiliser qu'un sprite pour le joueur. J'aurais pu faire en sorte que Mario soit toujours orienté vers le bas, mais le fait de pouvoir le diriger dans les 4 directions ajoute un peu plus de réalisme je trouve. Ca ne fera qu'un petit défi de plus à relever

Ah, et j'ai aussi fait une petite image qui pourra servir de menu d'accueil au lancement du programme :

Cette image se trouve dans le pack d'images que je vous ai fait télécharger plus haut.

Important : vous noterez que ces images sont dans différents formats. Il y a des GIF, des PNG et des JPEG. Nous allons donc avoir besoin de la librairie SDL_Image.
Pensez à configurer votre projet pour qu'il gère la **SDL** et **SDL_Image**. Si vous avez oublié comment faire, revoyez les chapitres précédents.

Si vous ne configurez pas votre projet correctement, on vous dira que les fonctions que vous utilisez (comme *IMG_Load*) n'existent pas !

LE MAIN ET LES CONSTANTES

A chaque fois qu'on commence un projet assez important, il est nécessaire de **bien s'organiser dès le départ**.
En général, je commence par me créer un fichier de constantes `constantes.h` ainsi qu'un fichier `main.c` qui contiendra la fonction *main* (et uniquement la fonction *main*). Ce n'est pas une règle : c'est juste ma façon de fonctionner. Chacun a sa propre façon de faire.

Les différents fichiers du projet

Je propose de créer dès à présent tous les fichiers du projet (même s'ils restent vides au départ).
Voici les fichiers que je crée :

- **constantes.h** : les définitions de constantes globales à tout le programme
- **main.c** : le fichier qui contient la fonction main (fonction principale du programme).
- **jeu.c** : fonctions gérant une partie de Sokoban
- **jeu.h** : prototypes des fonctions de jeu.c
- **editeur.c** : fonctions gérant l'éditeur de niveaux
- **editeur.h** : prototypes.
- **fichiers.c** : fonctions gérant la lecture et l'écriture de fichiers de niveaux (niveaux.lvl par exemple).
- **fichiers.h** : prototypes.

Allons-y donc, on va commencer par créer le fichier des constantes.

Les constantes : constantes.h

Voici le contenu de mon fichier de constantes :

Code : C

```c
/*
constantes.h
------------

Par mateo21, pour Le Site du Zér0 (www.siteduzero.com)

Rôle : définit des constantes communes à tout le programme (taille de la fenêtre...)
*/

#ifndef DEF_CONSTANTES
#define DEF_CONSTANTES

    #define TAILLE_BLOC         34 // Taille d'un bloc (carré) en pixels
    #define NB_BLOCS_LARGEUR    12
    #define NB_BLOCS_HAUTEUR    12
    #define LARGEUR_FENETRE     TAILLE_BLOC * NB_BLOCS_LARGEUR
    #define HAUTEUR_FENETRE     TAILLE_BLOC * NB_BLOCS_HAUTEUR


    enum {HAUT, BAS, GAUCHE, DROITE};
    enum {VIDE, MUR, CAISSE, OBJECTIF, MARIO, CAISSE_OK};

#endif
```

Vous noterez plusieurs choses :

- Le fichier commence par un **commentaire d'en-tête**. Je recommande de mettre ce type de commentaire au début de chacun de vos fichiers (que ce soient des .h ou des .c). Généralement, un commentaire d'en-tête contient :
    - Le nom du fichier (constantes.h)
    - L'auteur (mateo21)
    - Le rôle du fichier (à quoi servent les fonctions à l'intérieur)
    - Je ne l'ai pas fait là, mais on met aussi généralement la date de création et la date de dernière modification. Ca permet de s'y retrouver, surtout dans les gros projets à plusieurs.
- Le fichier est **protégé contre les inclusions infinies**. Il utilise la technique que l'on a apprise à la **fin du chapitre sur le préprocesseur** dans la partie II. Ici cette protection ne sert à rien, mais j'ai pris l'habitude de faire ça pour chacun de mes fichiers .h sans exception.
- Enfin, le coeur du fichier. Vous avez une série de **defines**. J'indique la taille d'un petit bloc en pixels (tous mes sprites sont des carrés de 34px). J'indique que ma fenêtre comportera 12 x 12 blocs de largeur. Je calcule comme ça les dimensions de la fenêtre par une simple multiplication des constantes. Ce que je fais là n'est pas obligatoire, mais ça un énorme avantage : si plus tard je veux changer la taille du jeu, je n'aurai qu'à éditer ce fichier (et à recompiler) et tout mon code source s'adaptera en conséquence.
- Enfin, j'ai défini d'autres constantes via des **énumérations anonymes**. C'est légèrement différent de ce qu'on a appris dans le **chapitre sur les types de variables personnalisés**. Ici, je ne crée pas un type personnalisé, je définis juste des constantes.

Oui, à une différence près : c'est l'ordinateur qui attribue automatiquement un nombre à chacune des valeurs (en commençant par 0). Ainsi, on a HAUT = 0, BAS = 1, GAUCHE = 2 etc. Cela permettra de rendre notre code source beaucoup plus clair par la suite, vous verrez !

En résumé, j'utilise :

- Des **defines** lorsque je veux attribuer une valeur précise à une constante (par exemple "34 pixels").
- Des **énumérations** lorsque la valeur attribuée à une constante ne m'importe pas. Ici, je m'en fous que HAUT vale 0 (ça pourrait aussi bien valoir 150 ça ne changerait rien), tout ce qui compte pour moi c'est que cette constante ait une valeur différente de BAS, GAUCHE et DROITE

Inclure les définitions de constantes

Le principe, c'est que ce fichier de constantes sera inclus dans chacun de mes fichiers .c.
Ainsi, partout dans mon code je pourrai utiliser les constantes que je viens de définir.

Il faudra donc taper la ligne suivante au début de chacun des fichiers .c :

Code : C

```c
#include "constantes.h"
```

Le main : main.c

La fonction main est extrêmement simple, je peux même me permettre de vous la donner d'un coup comme ça. En effet, elle n'est pas plus difficile que ce qu'on a appris dans les chapitres précédents, donc si vous avez suivi récemment ça ne vous posera aucun problème de compréhension :

Code : C

```
/*
main.c
------

Par mateo21, pour Le Site du Zér0 (www.siteduzero.com)

Rôle : menu du jeu. Permet de choisir entre l'éditeur et le jeu lui-même.
*/

#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>

#include "constantes.h"
#include "jeu.h"
#include "editeur.h"



int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *menu = NULL;
    SDL_Rect positionMenu;
    SDL_Event event;

    int continuer = 1;

    SDL_Init(SDL_INIT_VIDEO);

    SDL_WM_SetIcon(IMG_Load("caisse.jpg"), NULL); // L'icône doit être chargée avant
SDL_SetVideoMode
    ecran = SDL_SetVideoMode(LARGEUR_FENETRE, HAUTEUR_FENETRE, 32, SDL_HWSURFACE |
SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Mario Sokoban", NULL);

    menu = IMG_Load("menu.jpg");
    positionMenu.x = 0;
    positionMenu.y = 0;

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
            case SDL_KEYDOWN:
                switch(event.key.keysym.sym)
                {
                    case SDLK_ESCAPE: // Veut arrêter le jeu
                        continuer = 0;
                        break;
                    case SDLK_KP1: // Demande à jouer
                        jouer(ecran);
                        break;
                    case SDLK_KP2: // Demande l'éditeur de niveaux
                        editeur(ecran);
                        break;
                }
                break;
        }

        // Effacement de l'écran
        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));
        SDL_BlitSurface(menu, NULL, ecran, &positionMenu);
        SDL_Flip(ecran);
    }

    SDL_FreeSurface(menu);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

La fonction *main* se charge d'effectuer les initialisations de la SDL, de donner un titre à la fenêtre ainsi qu'une icône. A la fin de la fonction, *SDL_Quit()* est appelé pour arrêter la SDL proprement.

La fonction affiche un menu. Le menu est chargé en utilisant la fonction *IMG_Load* de SDL_Image :

Code : C

```
menu = IMG_Load("menu.jpg");
```

Vous remarquerez que pour donner les dimensions de la fenêtre j'utilise les constantes LARGEUR_FENETRE et HAUTEUR_FENETRE qu'on a définies dans `constantes.h`

La boucle des évènements

La boucle infinie gère les évènements suivants :

- **SDL_QUIT** : si on demande à fermer le programme (clic sur la croix en haut à droite de la fenêtre), alors on passe le booléen continuer à 0, et la boucle s'arrêtera. Bref, classique.
- **Appui sur la touche Echap** : arrêt du programme (comme SDL_QUIT)
- **Appui sur la touche 1 du pavé numérique** : lancement du jeu (appel de la fonction *jouer*)
- **Appui sur la touche 2 du pavé numérique** : lancement de l'éditeur (appel de la fonction *editeur*)

Comme vous le voyez, c'est vraiment très simple. Si on appuie sur 1, le jeu est lancé. Une fois que le jeu est terminé, la fonction *jouer* s'arrête et on retourne dans le main dans lequel on refait un tour de boucle. Le main boucle à l'infini tant qu'on ne demande pas à arrêter le jeu.

Grâce à cette petite organisation très simple, on peut donc gérer le menu dans le main, et laisser des fonctions spéciales (comme *jouer*, ou *editeur*) gérer les différentes parties du programme.

LE JEU

Attaquons maintenant le gros du sujet : la fonction *jouer* !
Cette fonction est la plus importante du programme, aussi soyez attentifs car c'est vraiment là qu'il faut comprendre.
Vous verrez après que créer l'éditeur de niveaux n'est pas si compliqué en fait

Les paramètres envoyés à la fonction

La fonction *jouer* a besoin d'un paramètre : la surface *ecran*. En effet, la fenêtre a été ouverte dans le main, et pour que la fonction jouer puisse dessiner dedans il faut qu'elle récupère le pointeur sur *ecran* !

Si vous regardez le main à nouveau, vous voyez qu'on appelle *jouer* en lui envoyant *ecran* :

Code : C

```
jouer(ecran);
```

Le prototype de la fonction, que vous pouvez mettre dans jeu.h, est donc le suivant :

Code : C

```
void jouer(SDL_Surface* ecran);
```

La fonction ne renvoie aucune valeur (d'où le "void"), mais on pourrait en renvoyer une si on voulait. On pourrait par exemple renvoyer un booléen pour dire si oui ou non on a gagné.

Les déclarations de variables

Cette fonction va avoir besoin de nombreuses variables.
Je n'ai pas pensé à toutes les variables dont j'ai eu besoin du premier coup. Il y en a donc certaines que j'ai rajoutées par la suite.

Variables de type défini par la SDL

Voici pour commencer toutes les variables de types définis par la SDL dont j'ai besoin :

Code : C

```
SDL_Surface *mario[4] = {NULL}; // 4 surfaces pour chacune des directions de mario
SDL_Surface *mur = NULL, *caisse = NULL, *caisseOK = NULL, *objectif = NULL, *marioActuel
= NULL;
SDL_Rect position, positionJoueur;
SDL_Event event;
```

J'ai créé un tableau de SDL_Surface appelé *mario*. C'est un tableau de 4 cases qui stockera Mario dans chacune des directions (un vers le bas, un autre vers la gauche, vers le haut et vers la droite).

Il y a ensuite plusieurs surfaces correspondant à chacun des sprites que je vous ai faits télécharger plus haut : mur, caisse, caisseOK (une caisse sur un objectif) et objectif.

A quoi sert marioActuel ?

C'est un pointeur vers une surface. Il pointe sur la surface correspondant au Mario orienté dans la direction actuelle. C'est donc marioActuel qu'on blittera à l'écran. Si vous regardez tout en bas de la fonction *jouer*, vous verrez justement :

Code : C

```
SDL_BlitSurface(marioActuel, NULL, ecran, &position);
```

On ne blitte donc pas un élément du tableau *mario*, mais le pointeur marioActuel.
Ainsi, en blittant marioActuel, on blitte soit le mario vers le bas, soit vers le haut etc. Le pointeur marioActuel pointe vers une des cases du tableau *mario*.

Quoi d'autre à part ça ?
Une variable *position* de type SDL_Rect dont on se servira pour définir la position des éléments à blitter (on s'en servira pour tous les sprites, c'est pas la peine de créer un SDL_Rect pour chaque surface !).

positionJoueur est en revanche un peu différent : il indique à quelle case sur la carte se trouve se trouve actuellement le joueur.

Enfin, la variable *event* traitera les évènements, là il n'y a rien de particulièrement nouveau.

Variables plus "classiques"

J'ai aussi besoin de me créer des variables un peu plus classiques de type int (entier).

Code : C

```
int continuer = 1, objectifsRestants = 0, i = 0, j = 0;
int carte[NB_BLOCS_LARGEUR][NB_BLOCS_HAUTEUR] = {0};
```

continuer et objectifsRestants sont des booléens.
i et j sont des petites variables qui vont me permettre de parcourir le tableau.

Le tableau *carte* défini juste en-dessous !
C'est là que les choses deviennent vraiment intéressantes. J'ai en effet créé **un tableau à 2 dimensions**. Je ne vous ai pas parlé de ce type de tableaux auparavant (pas eu l'occasion d'en parler), mais c'est justement le moment idéal pour vous apprendre ce que c'est. Ce n'est pas bien compliqué vous allez voir.

Regardez la définition de plus près :

Code : C

```
int carte[NB_BLOCS_LARGEUR][NB_BLOCS_HAUTEUR] = {0};
```

En fait, il s'agit d'un tableau d'int (entiers) qui a la particularité d'avoir 2 paires de crochets [ ].
Si vous vous souvenez bien de `constantes.h`, vous savez que NB_BLOCS_LARGEUR et NB_BLOCS_HAUTEUR sont des constantes qui valent toutes les deux 12.

Ce tableau sera donc à la compilation créé comme ceci :

Code : C

```
int carte[12][12] = {0};
```

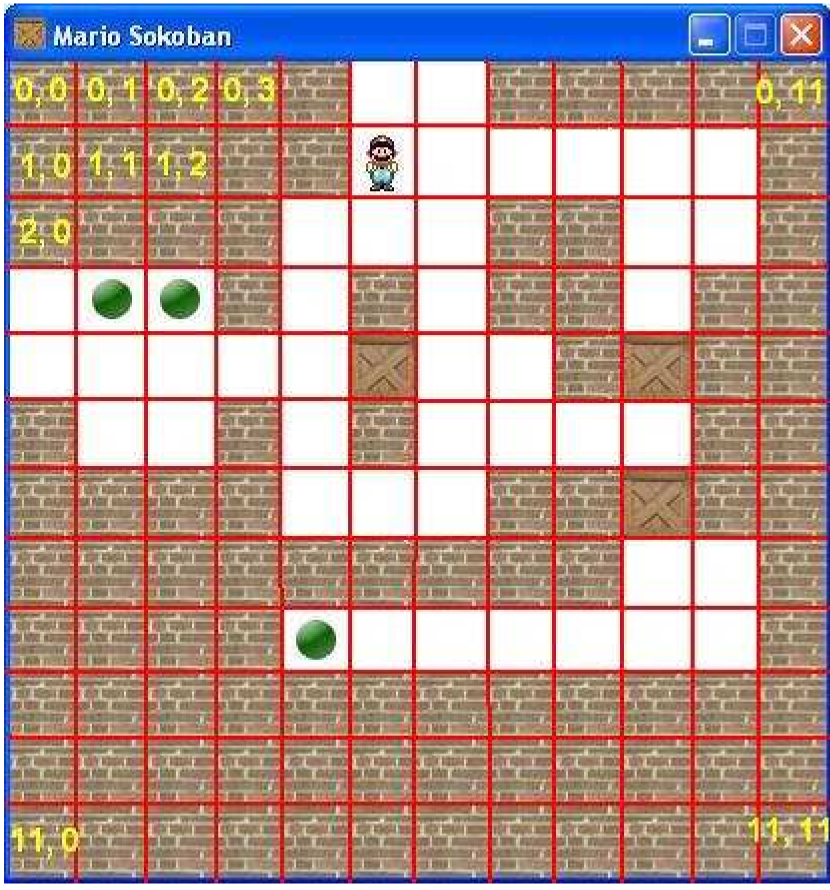
Ca veut dire que pour chaque "case" de carte, il y a 12 sous-cases.
Il y aura donc les variables suivantes :

Code : Autre

```
carte[0][0]
carte[0][1]
carte[0][2]
carte[0][3]
carte[0][4]
carte[0][5]
carte[0][6]
carte[0][7]
carte[0][8]
carte[0][9]
carte[0][10]
carte[0][11]
carte[1][0]
carte[1][1]
carte[1][2]
carte[1][3]
carte[1][4]

...


carte[11][8]
carte[11][9]
carte[11][10]
carte[11][11]
```

C'est donc un tableau de 12 * 12 = 144 cases !
Chacune des ces cases représente une case de la carte.

Voici comment on peut représenter la carte, vous allez comprendre en voyant ça :



(Enfin vous avez intérêt à comprendre parce que j'ai passé un bon moment à tracer toutes les lignes une à une sous Paint là)

Ainsi, la case en haut à gauche est stockée dans `carte[0][0]`.

La case en haut à droite est stockée dans `carte[0][11]`.
La case en bas à droite (la toute dernière) est stockée dans `carte[11][11]`.

Selon la valeur de la case (qui est un nombre entier), on sait si la case contient un mur, une caisse, un objectif etc...). C'est justement là que va servir notre énumération de tout à l'heure !

Code : C

```
enum {VIDE, MUR, CAISSE, OBJECTIF, MARIO, CAISSE_OK};
```

Si la case vaut VIDE (0) on sait que cette partie de l'écran devra rester blanche. Si elle vaut MUR (1), on sait qu'il faudra blitter une image de mur etc...

Initialisations

Chargement des surfaces

Maintenant qu'on a passé en revue toutes les variables de la fonction jouer, on peut commencer à faire quelques initialisations :

Code : C

```
// Chargement des sprites (décors, personnage...)
mur = IMG_Load("mur.jpg");
caisse = IMG_Load("caisse.jpg");
caisseOK = IMG_Load("caisse_ok.jpg");
objectif = IMG_Load("objectif.png");
mario[BAS] = IMG_Load("mario_bas.gif");
mario[GAUCHE] = IMG_Load("mario_gauche.gif");
mario[HAUT] = IMG_Load("mario_haut.gif");
mario[DROITE] = IMG_Load("mario_droite.gif");
```

Rien de sorcier là-dedans, on charge tout grâce à IMG_Load.
Le petit truc intéressant, c'est le chargement de mario. On charge en effet Mario dans chacune des directions dans le tableau "mario" en utilisant les constantes HAUT, BAS, GAUCHE, DROITE. Le fait d'utiliser les constantes rend ici comme vous le voyez le code plus clair. On aurait très bien pu écrire `mario[0]`, mais c'est quand même plus lisible d'avoir `mario[HAUT]` par exemple !

Orientation initiale du Mario (marioActuel)

On initialise ensuite marioActuel pour qu'il ait une direction au départ :

Code : C

```
marioActuel = mario[BAS]; // Mario sera dirigé vers le bas au départ
```

J'ai trouvé plus logique de commencer la partie avec un Mario qui regarde vers le bas (c'est-à-dire vers nous). Si vous voulez, vous pouvez changer cette ligne et mettre :

Code : C

```
marioActuel = mario[DROITE];
```

Vous verrez que Mario sera alors orienté vers la droite au début du jeu

Chargement de la carte

Maintenant, il va falloir remplir notre tableau à 2 dimensions *carte*. Pour l'instant, ce tableau ne contient que des 0. Il faut lire le niveau qui est stocké dans le fichier niveaux.lvl.

Code : C

```
// Chargement du niveau
if (!chargerNiveau(carte))
    exit(EXIT_FAILURE); // On arrête le jeu si on n'a pas pu charger le niveau
```

J'ai choisi de faire gérer le chargement (et l'enregistrement) de niveaux par des fonctions situées dans `fichiers.c`
Ici, on appelle donc la fonction *chargerNiveau*. On l'étudiera plus en détail plus loin (elle n'est pas très compliquée de toute manière). Tout ce qui nous intéresse ici c'est de savoir que notre niveau a été chargé dans le tableau *carte*.

Si le niveau n'a pas pu être chargé (parce que niveaux.lvl n'existe pas), la fonction renverra faux. Sinon, elle renverra vrai.

On teste donc le résultat du chargement dans un if.
Si le résultat est négatif (d'où le point d'exclamation qui sert à exprimer la négation), on arrête tout : on appelle *exit*.
Sinon, c'est que tout va bien et on peut continuer.

Recherche de la position de départ de Mario

Il faut maintenant initialiser la variable positionJoueur.
Cette variable, de type SDL_Rect, est un peu particulière. On ne s'en sert pas pour stocker des coordonnées en pixels. On s'en sert pour stocker des coordonnées en "cases" sur la carte. Ainsi, si on a :

```
positionJoueur.x == 11
positionJoueur.y == 11
```

... c'est que le joueur se trouve dans la toute dernière case en bas à droite de la carte
Reportez-vous au schéma de la carte qu'on a vu plus haut pour bien voir à quoi ça correspond si vous avez (déjà) oublié

Bref, on doit parcourir notre tableau *carte* à 2 dimensions à l'aide d'une double boucle. On utilise la petite variable i pour parcourir le tableau verticalement, et la variable j pour le parcourir horizontalement :

Code : C

```
// Recherche de la position de Mario au départ
for (i = 0 ; i < NB_BLOCS_HAUTEUR ; i++)
{
    for (j = 0 ; j < NB_BLOCS_LARGEUR ; j++)
    {
        if (carte[i][j] == MARIO) // Si Mario se trouve à cette position sur la carte
        {
            positionJoueur.x = i;
            positionJoueur.y = j;
            carte[i][j] = VIDE;
        }
    }
}
```

A chaque case, on teste si elle contient MARIO (c'est-à-dire le départ du joueur sur la carte). Si c'est le cas, on stocke les coordonnées actuelles (situées dans i et j) dans la variable positionJoueur.
On efface aussi la case en la mettant à VIDE pour qu'elle soit considérée comme une case vide par la suite.

Activation de la répétition des touches

Dernière chose, très simple : on active la répétition des touches pour qu'on puisse se déplacer sur la carte en laissant une touche enfoncée.

Code : C

```
// Activation de la répétition des touches
SDL_EnableKeyRepeat(100, 100);
```

La boucle principale

Pfiou !
Nos initialisations sont faites, on peut maintenant attaquer la boucle principale.

C'est une boucle classique qui fonctionne sur le même schéma que celles qu'on a vues jusqu'ici. Elle est juste un peu plus grosse et un peu plus complète (faut c'qui faut)

Regardons de plus près le switch qui teste l'évènement :

Code : C

```
switch(event.type)
{
    case SDL_QUIT:
        continuer = 0;
        break;
    case SDL_KEYDOWN:
        switch(event.key.keysym.sym)
        {
            case SDLK_ESCAPE:
                continuer = 0;
                break;
            case SDLK_UP:
                marioActuel = mario[HAUT];
                deplacerJoueur(carte, &positionJoueur, HAUT);
                break;
            case SDLK_DOWN:
                marioActuel = mario[BAS];
                deplacerJoueur(carte, &positionJoueur, BAS);
                break;
            case SDLK_RIGHT:
                marioActuel = mario[DROITE];
                deplacerJoueur(carte, &positionJoueur, DROITE);
                break;
            case SDLK_LEFT:
                marioActuel = mario[GAUCHE];
                deplacerJoueur(carte, &positionJoueur, GAUCHE);
                break;
        }
        break;
}
```

Si on fait Echap, le jeu s'arrêtera et on retournera au menu principal.

Comme vous le voyez, il n'y a pas 36 évènements différents à gérer : on teste juste si le joueur appuie sur haut, bas, gauche ou droite sur son clavier.
Selon la touche enfoncée, on change la direction de mario. C'est là qu'intervient marioActuel ! Si on appuie vers le haut, alors :

Code : C

```
marioActuel = mario[HAUT];
```

Si on appuie vers le bas, alors :

Code : C

```
marioActuel = mario[BAS];
```

marioActuel pointe donc sur la surface représentant Mario dans la position actuelle. C'est ainsi qu'en blittant marioActuel tout à l'heure, on sera sûrs de blitter Mario dans la bonne direction

Maintenant, chose très importante : on appelle une fonction *deplacerJoueur*. Cette fonction va déplacer le joueur sur la carte s'il a le droit de le faire.

- Par exemple, on ne peut pas faire monter Mario d'un cran vers le haut s'il se trouve déjà tout en haut de la carte.
- On ne peut pas non plus le faire monter s'il y a un mur au-dessus de lui.
- On ne peut pas le faire monter s'il y a 2 caisses au-dessus de lui.
- Par contre, on peut le faire monter s'il y a juste une caisse au-dessus de lui.
- Mais attention, on ne peut pas le faire monter s'il y a une caisse au-dessus de lui et que le caisse se trouve au bord de la carte !

C'est ce qu'on appelle **la gestion des collisions**. Si ça peut vous rassurer, ici c'est une gestion des collisions extrêmement simple vu que le joueur se déplace par "cases" et dans seulement 4 directions possibles à la fois.

Quoi, vous croyiez tout de même pas qu'un jeu se codait en claquant des doigts hein ?
Dans un jeu 2D où on peut se déplacer dans toutes les directions pixel par pixel, c'est donc une gestion des collisions bien plus complexe (tout ça ne serait-ce que pour faire un RPG 2D !).

Mais il y a pire : la 3D. Je n'ai jamais expérimenté la gestion des collisions dans un jeu 3D, mais d'après ce que j'ai entendu dire, c'est la bête noire des programmeurs
Heureusement, il existe des librairies de gestion des collisions en 3D qui font le gros du travail à notre place.

Bon je divague là.
Revenons à la fonction deplacerJoueur. On lui envoie 3 paramètres :

- La carte (pour qu'il puisse la lire mais aussi la modifier si on déplace une caisse par exemple)
- La position du joueur (là aussi la fonction devra lire et éventuellement modifier la position du joueur)
- La direction dans laquelle on demande à aller. On utilise là encore les constantes HAUT, BAS, GAUCHE, DROITE pour plus de lisibilité.

Nous étudierons la fonction deplacerJoueur plus loin. J'aurais très bien pu mettre tous les tests dans le switch, mais ça serait devenu énorme et illisible. **C'est là que découper son programme en fonctions prend tout son intérêt.**

Blittons, blittons, la queue du cochon

Notre switch est terminé, à ce stade la carte a changé (ou pas), et le joueur a changé de position et de direction (ou pas).
Quoi qu'il en soit, c'est l'heure du blit !

On commence par effacer l'écran en lui mettant une couleur de fond blanche :

Code : C

```
// Effacement de l'écran
SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
```

Et maintenant, on parcourt tout notre tableau à 2 dimensions *carte* pour savoir quel élément blitter à quel endroit sur l'écran.
On effectue une double boucle comme on l'a vu plus tôt pour parcourir toutes les 144 cases du tableau :

Code : C

```
// Placement des objets à l'écran
objectifsRestants = 0;

for (i = 0 ; i < NB_BLOCS_LARGEUR ; i++)
{
    for (j = 0 ; j < NB_BLOCS_HAUTEUR ; j++)
    {
        position.x = i * TAILLE_BLOC;
        position.y = j * TAILLE_BLOC;

        switch(carte[i][j])
        {
            case MUR:
                SDL_BlitSurface(mur, NULL, ecran, &position);
                break;
            case CAISSE:
                SDL_BlitSurface(caisse, NULL, ecran, &position);
                break;
            case CAISSE_OK:
                SDL_BlitSurface(caisseOK, NULL, ecran, &position);
                break;
            case OBJECTIF:
                SDL_BlitSurface(objectif, NULL, ecran, &position);
                objectifsRestants = 1;
                break;
        }
    }
}
```

Pour chacune des cases, on prépare la variable *position* (de type SDL_Rect) pour placer l'élément actuel à la bonne position sur l'écran.
Le calcul est très simple :

Code : C

```
position.x = i * TAILLE_BLOC;
position.y = j * TAILLE_BLOC;
```

Il suffit de multiplier i par TAILLE_BLOC pour avoir position.x.
Ainsi, si on se trouve à la 3ème case, c'est que i vaut 2 (n'oubliez pas que i commence à 0 !). On fait donc le calcul 2 * 34 = 68. On blittera donc l'image 68 pixels vers la droite sur *ecran*.
On fait la même chose pour les ordonnées y.

Ensuite, on fait un switch sur la case de la carte qu'on est en train d'analyser.
Là encore, avoir fait des constantes est vraiment pratique et ça rend les choses plus lisibles
On teste donc si la case vaut MUR, dans ce cas on blitte un mur. De même pour les caisses et les objectifs.

Test de victoire

Vous remarquerez qu'avant la double boucle on initialise le booléen objectifsRestants à 0.
Ce booléen sera mis à 1 dès qu'on aura détecté un objectif sur la carte. S'il ne reste plus d'objectifs, c'est que toutes les caisses sont sur des objectifs (il n'y a plus que des CAISSE_OK).

Il suffit de tester si le booléen vaut FAUX (= il ne reste plus d'objectifs).
Dans ce cas, on met la variable continuer à 0 pour arrêter la partie :

Code : C

```
// Si on n'a trouvé aucun objectif sur la carte, c'est qu'on a gagné
if (!objectifsRestants)
    continuer = 0;
```

Le joueur

Le joueur, on le blitte juste après. Justement, venons-y :

Code : C

```
// On place le joueur à la bonne position
position.x = positionJoueur.x * TAILLE_BLOC;
position.y = positionJoueur.y * TAILLE_BLOC;
SDL_BlitSurface(marioActuel, NULL, ecran, &position);
```

On calcule sa position (en pixels cette fois) en faisant une simple multiplication entre positionJoueur et TAILLE_BLOC. On blitte ensuite le joueur à la position indiquée.

Flip !

Bon ben on a tout fait je crois, on peut afficher le nouvel écran au joueur

Code : C

```
SDL_Flip(ecran);
```

Fin de la fonction : déchargements

Après la boucle principale, on doit faire quelques FreeSurface pour libérer la mémoire des sprites qu'on a chargés.
On désactive aussi la répétition des touches en envoyant les valeurs 0 à la fonction SDL_EnableKeyRepeat :

Code : C

```
// Désactivation de la répétition des touches (remise à 0)
SDL_EnableKeyRepeat(0, 0);

// Libération des surfaces chargées
SDL_FreeSurface(mur);
SDL_FreeSurface(caisse);
SDL_FreeSurface(caisseOK);
SDL_FreeSurface(objectif);
for (i = 0 ; i < 4 ; i++)
    SDL_FreeSurface(mario[i]);
```

La fonction deplacerJoueur

La fonction deplacerJoueur se trouve elle aussi dans jeu.c.
C'est une fonction... très chiante à écrire

C'est peut-être là la principale difficulté du codage du jeu de Sokoban.

Rappel : la fonction deplacerJoueur vérifie si on a le droit de déplacer le joueur dans la direction demandée. Elle met à jour la position du joueur (positionJoueur) et aussi la carte si une caisse a été déplacée.

Voici le prototype de la fonction :

Code : C

```
void deplacerJoueur(int carte[][NB_BLOCS_HAUTEUR], SDL_Rect *pos, int direction);
```

Ce prototype est un peu particulier. Vous voyez que j'envoie le tableau carte, et que je précise la taille de la deuxième dimension (NB_BLOCS_HAUTEUR).
Pourquoi cela ?

C'est un problème qui m'a pas mal embêté pendant longtemps et je ne comprenais pas pourquoi. La réponse est un peu compliquée pour que je la développe au milieu de ce cours. Grosso modo, le C ne devine pas qu'il s'agit d'un tableau à 2 dimensions, et il faut au moins donner la taille de la seconde dimension pour que ça fonctionne.
Donc, lorsque vous envoyez un tableau à 2 dimensions à une fonction, vous devez indiquer la taille de la seconde dimension dans le prototype. C'est comme ça, c'est obligatoire.

Autre chose : vous noterez que positionJoueur s'appelle en fait "pos" dans cette fonction. J'ai choisi de raccourcir le nom parce que c'est plus court à écrire, et vu qu'on va avoir besoin de l'écrire de nombreuses fois autant pas se fatiguer

Bon on commence par tester la direction dans laquelle on veut aller via un grand :

Code : C

```
switch(direction)
{
    case HAUT:
    /* etc. */
```

Et c'est parti pour des tests de folie !

On peut commencer à faire tous les tests, en essayant tant qu'à faire de n'oublier aucun cas
Voici comment je procède : je teste toutes les possibilités de collision cas par cas, et dès que je détecte une collision (qui fait que le joueur ne peut pas bouger), je fais un `break;` pour sortir du switch et donc empêcher le déplacement.

Voici par exemple toutes les possibilités de collision qui existent pour un joueur qui veut se déplacer vers le haut :

- Le joueur est déjà tout en haut de la carte
- Il y a un mur au-dessus du joueur
- Il y a 2 caisses au-dessus du joueur (et il ne peut pas déplacer 2 caisses à la fois, rappelez-vous).

Si tous ces tests sont ok, alors je me permet de déplacer le joueur.

Je vais vous montrer les tests pour un déplacement vers le haut. Pour les autres sens, il suffira d'adapter un petit peu le code.

Code : C

```
if (pos->y - 1 < 0) // Si le joueur dépasse l'écran, on arrête
    break;
```

On commence par vérifier si le joueur est déjà tout en haut de l'écran. En effet, si on essayait d'appeler carte[5][-1] par exemple, ce serait le plantage de programme assuré !
On commence donc par vérifier qu'on ne va pas "déborder" de l'écran.

Ensuite :

Code : C

```
if (carte[pos->x][pos->y - 1] == MUR) // S'il y a un mur, on arrête
    break;
```

Là encore c'est simple. On vérifie s'il n'y a pas un mur au-dessus du joueur. Si tel est le cas, on arrête (break).

Ensuite (attention ça devient hardcore) :

Code : C

```
// Si on veut pousser une caisse, il faut vérifier qu'il n'y a pas de mur derrière (ou une autre caisse, ou la limite du monde)
if ((carte[pos->x][pos->y - 1] == CAISSE || carte[pos->x][pos->y - 1] == CAISSE_OK) &&
    (pos->y - 2 < 0 || carte[pos->x][pos->y - 2] == MUR ||
    carte[pos->x][pos->y - 2] == CAISSE || carte[pos->x][pos->y - 2] == CAISSE_OK))
    break;
```

Ce méga test de bourrin peut se traduire comme ceci :
SI au-dessus du joueur il y a une caisse (ou une caisse_ok, c'est-à-dire une caisse bien placée)
ET SI au-dessus de cette caisse il y a : soit le vide (on déborde du niveau car on est tout en haut), soit une autre caisse, soit une caisse_ok) :
ALORS on ne peut pas se déplacer : break.

Si on arrive jusque-là, on a le droit de déplacer le joueur !
On appelle d'abord une fonction qui va déplacer une caisse si nécessaire :

Code : C

```
// Si on arrive là, c'est qu'on peut déplacer le joueur !
// On vérifie d'abord s'il y a une caisse à déplacer
deplacerCaisse(&carte[pos->x][pos->y - 1], &carte[pos->x][pos->y - 2]);
```

Le déplacement de caisse : deplacerCaisse

J'ai choisi de gérer le déplacement de caisse dans une autre fonction car c'est le même code pour les 4 directions. On doit juste s'être assuré avant qu'on a le droit de se déplacer (ce qu'on vient de faire).
On envoie à la fonction 2 paramètres : le contenu de la case dans laquelle on veut aller, et le contenu de la case d'après.

Code : C

```
void deplacerCaisse(int *premiereCase, int *secondeCase)
{
    if (*premiereCase == CAISSE || *premiereCase == CAISSE_OK)
    {
        if (*secondeCase == OBJECTIF)
            *secondeCase = CAISSE_OK;
        else
            *secondeCase = CAISSE;

        if (*premiereCase == CAISSE_OK)
            *premiereCase = OBJECTIF;
        else
            *premiereCase = VIDE;
    }
}
```

Cette fonction met à jour la carte car elle prend des pointeurs sur les cases concernées en paramètre.
Je vous laisse la lire, c'est assez simple à comprendre. Il ne faut pas oublier que si on déplace une CAISSE_OK, il faut remplacer la case où elle se trouvait par un OBJECTIF. Sinon, si c'est une simple CAISSE, alors on remplace la case en question par du VIDE.

Déplacer le joueur

On retourne dans la fonction deplacerJoueur.
Cette fois c'est la bonne, on peut déplacer le joueur.

Comment on fait ?
C'est supra-simple

Code : C

```
pos->y--; // On peut enfin faire monter le joueur (oufff !)
```

Il suffit de diminuer l'ordonnée y (car le joueur veut monter).

Résumé

En guise de résumé, voici tous les tests pour le cas HAUT :

Code : C

```c
switch(direction)
{
    case HAUT:
        if (pos->y - 1 < 0) // Si le joueur dépasse l'écran, on arrête
            break;
        if (carte[pos->x][pos->y - 1] == MUR) // S'il y a un mur, on arrête
            break;
        // Si on veut pousser une caisse, il faut vérifier qu'il n'y a pas de mur derrière (ou une autre caisse, ou la limite du monde)
        if ((carte[pos->x][pos->y - 1] == CAISSE || carte[pos->x][pos->y - 1] == CAISSE_OK) &&
            (pos->y - 2 < 0 || carte[pos->x][pos->y - 2] == MUR ||
            carte[pos->x][pos->y - 2] == CAISSE || carte[pos->x][pos->y - 2] == CAISSE_OK))
            break;

        // Si on arrive là, c'est qu'on peut déplacer le joueur !
        // On vérifie d'abord s'il y a une caisse à déplacer
        deplacerCaisse(&carte[pos->x][pos->y - 1], &carte[pos->x][pos->y - 2]);

        pos->y--; // On peut enfin faire monter le joueur (oufff !)
        break;
```

Je vous laisse le soin de faire du copier-coller pour les autres cas (attention, il faudra adapter le code, ce n'est pas exactement pareil à chaque fois !).

Et voilà, on vient de finir de coder le jeu
Enfin presque, il nous reste à coder la fonction de chargement (et de sauvegarde) de niveau.
On verra ensuite comment créer l'éditeur (rassurez-vous, ça ira bien plus vite)

Chargement et enregistrement de niveaux

`fichiers.c` contient 2 fonctions :

- chargerNiveau
- sauvegarderNiveau

Commençons par le chargement de niveau

chargerNiveau

Cette fonction prend un paramètre : la carte. Là encore, il faut préciser la taille de la seconde dimension car il s'agit d'un tableau à 2 dimensions.
La fonction renvoie un booléen : vrai si le chargement a réussi, faux si c'est un échec.

Le prototype est donc :

Code : C

```c
int chargerNiveau(int niveau[][NB_BLOCS_HAUTEUR]);
```

Voyons le début de la fonction :

Code : C

```
FILE* fichier = NULL;
char ligneFichier[NB_BLOCS_LARGEUR * NB_BLOCS_HAUTEUR + 1] = {0};
int i = 0, j = 0;

fichier = fopen("niveaux.lvl", "r");
if (fichier == NULL)
    return 0;
```

On crée un tableau de char pour stocker le résultat du chargement du niveau temporairement.
On ouvre le fichier en lecture seule ("r"). On arrête la fonction en renvoyant 0 (faux) si l'ouverture a échoué. Classique.

Le fichier niveaux.lvl contient une ligne qui est une suite de nombres. Chaque nombre représente une case du niveau. Par exemple :

11111001111111111400000111110001100103310101101100000200121110010100001111100011211111111111001111130000000111

On va donc lire cette ligne avec un fgets :

Code : C

```
fgets(ligneFichier, NB_BLOCS_LARGEUR * NB_BLOCS_HAUTEUR + 1, fichier);
```

On va analyser le contenu de ligneFichier. On sait que les 12 premiers caractères représentent la première ligne, les 12 suivants la seconde ligne etc.

Code : C

```
for (i = 0 ; i < NB_BLOCS_HAUTEUR ; i++)
{
    for (j = 0 ; j < NB_BLOCS_LARGEUR ; j++)
    {
        switch (ligneFichier[(i * NB_BLOCS_LARGEUR) + j])
        {
            case '0':
                niveau[j][i] = 0;
                break;
            case '1':
                niveau[j][i] = 1;
                break;
            case '2':
                niveau[j][i] = 2;
                break;
            case '3':
                niveau[j][i] = 3;
                break;
            case '4':
                niveau[j][i] = 4;
                break;
        }
    }
}
```

Par un simple petit calcul, on prend le caractère qui nous intéresse dans ligneFichier et on analyse sa valeur.

Ce sont des "lettres" qui sont stockées dans le fichier. Je veux dire par là que '0' est stocké comme le caractère ASCII '0', et sa valeur n'est pas 0 !
Pour analyser le fichier, il faut tester avec `case '0'` et non avec `case 0` ! Attention aux erreurs là

Bref, le switch fait la conversion '0' => 0, '1' => 1 etc... Et place tout dans le tableau carte (qui s'appelle *niveau* dans

la fonction d'ailleurs, mais ça ne change rien)

Une fois que c'est fait, on peut fermer le fichier et renvoyer 1 pour dire que tout s'est bien passé :

Code : C

```
fclose(fichier);
return 1;
```

Résumé de la fonction chargerFichier

Code : C

```
int chargerNiveau(int niveau[][NB_BLOCS_HAUTEUR])
{
    FILE* fichier = NULL;
    char ligneFichier[NB_BLOCS_LARGEUR * NB_BLOCS_HAUTEUR + 1] = {0};
    int i = 0, j = 0;

    fichier = fopen("niveaux.lvl", "r");
    if (fichier == NULL)
        return 0;

    fgets(ligneFichier, NB_BLOCS_LARGEUR * NB_BLOCS_HAUTEUR + 1, fichier);

    for (i = 0 ; i < NB_BLOCS_HAUTEUR ; i++)
    {
        for (j = 0 ; j < NB_BLOCS_LARGEUR ; j++)
        {
            switch (ligneFichier[(i * NB_BLOCS_LARGEUR) + j])
            {
                case '0':
                    niveau[j][i] = 0;
                    break;
                case '1':
                    niveau[j][i] = 1;
                    break;
                case '2':
                    niveau[j][i] = 2;
                    break;
                case '3':
                    niveau[j][i] = 3;
                    break;
                case '4':
                    niveau[j][i] = 4;
                    break;
            }
        }
    }

    fclose(fichier);
    return 1;
}
```

Ca reste assez simple, le seul piège à éviter c'était de bien penser à convertir la valeur ASCII '0' en un nombre 0 (et de même pour 1, 2, 3, 4...).

sauvegarderNiveau

Cette fonction est là encore simple :

Code : C

```
int sauvegarderNiveau(int niveau[][NB_BLOCS_HAUTEUR])
{
    FILE* fichier = NULL;
    int i = 0, j = 0;

    fichier = fopen("niveaux.lvl", "w");
    if (fichier == NULL)
        return 0;

    for (i = 0 ; i < NB_BLOCS_HAUTEUR ; i++)
    {
        for (j = 0 ; j < NB_BLOCS_LARGEUR ; j++)
        {
            fprintf(fichier, "%d", niveau[j][i]);
        }
    }

    fclose(fichier);
    return 1;
}
```

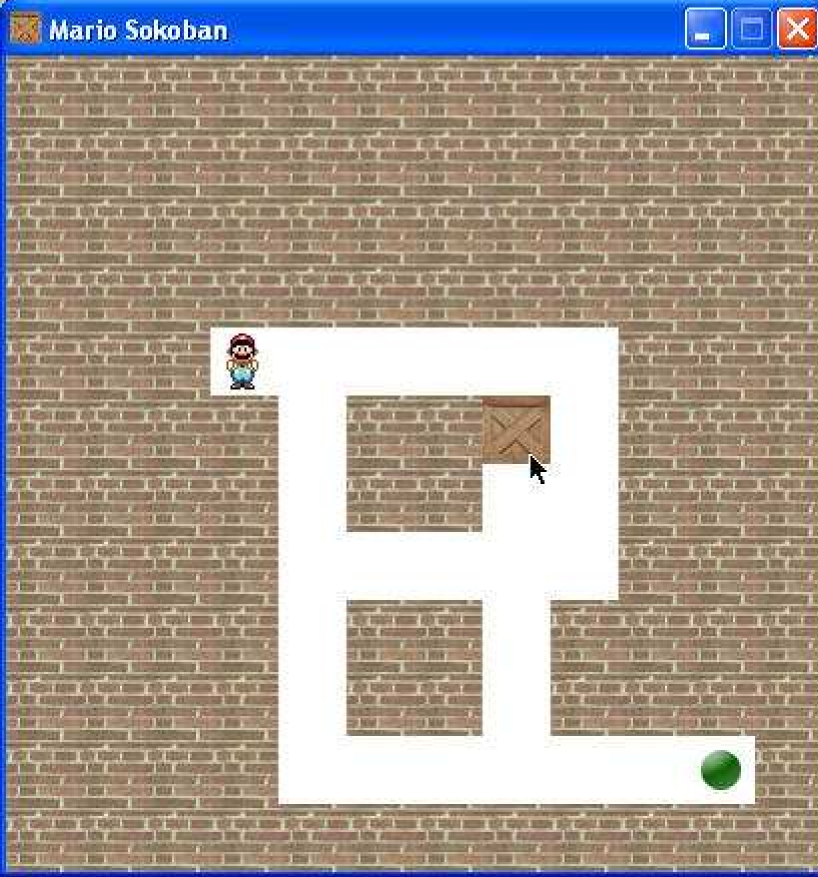
J'utilise fprintf pour "traduire" les nombres du tableau *niveau* en caractères ASCII.
C'était là encore la seule difficulté, il ne faut pas écrire 0 mais '0'

L'ÉDITEUR DE NIVEAUX

L'éditeur de niveau est plus facile à créer qu'on ne pourrait l'imaginer.
En plus c'est une fonctionnalité qui va considérablement allonger la durée de vie de notre jeu, alors pourquoi s'en priver

Voilà comment l'éditeur va fonctionner :

- On utilise la souris pour placer les blocs qu'on veut sur l'écran.
- Un clic droit efface le bloc sur lequel se trouve la souris.
- Un clic gauche place un objet. Cet objet est mémorisé : par défaut, on pose des murs avec le clic gauche. On peut changer l'objet en cours en appuyant sur les touches du pavé numérique :
    - 1 : mur
    - 2 : caisse
    - 3 : objectif
    - 4 : départ du joueur Mario
- En appuyant sur S, le niveau sera sauvegardé.
- On peut revenir au menu principal en appuyant sur Echap.



Edition d'un niveau avec l'éditeur

Initialisations

Globalement, la fonction ressemble à celle du jeu. J'ai d'ailleurs commencé à la créer en faisant un simple copier-coller de la fonction de jeu, puis en enlevant ce qui ne servait plus et en ajoutant de nouvelles fonctionnalités.

Le début y ressemble pas mal déjà :

Code : C

```
void editeur(SDL_Surface* ecran)
{
    SDL_Surface *mur = NULL, *caisse = NULL, *objectif = NULL, *mario = NULL;
    SDL_Rect position;
    SDL_Event event;

    int continuer = 1, clicGaucheEnCours = 0, clicDroitEnCours = 0;
    int objetActuel = MUR, i = 0, j = 0;
    int carte[NB_BLOCS_LARGEUR][NB_BLOCS_HAUTEUR] = {0};

    // Chargement des objets et du niveau
    mur = IMG_Load("mur.jpg");
    caisse = IMG_Load("caisse.jpg");
    objectif = IMG_Load("objectif.png");
    mario = IMG_Load("mario_bas.gif");

    if (!chargerNiveau(carte))
        exit(EXIT_FAILURE);
```

Là, vous avez les définitions de variables et les initialisations.
Vous remarquerez que je ne charge qu'un Mario (celui dirigé vers le bas). En effet, on ne va pas diriger Mario au clavier là, on a juste besoin d'un sprite représentant la position de départ de Mario.

La variable objetActuel retient l'objet actuellement sélectionné par l'utilisateur. Par défaut, c'est un MUR. Le clic gauche créera donc un mur au départ, mais cela pourra être changé par l'utilisateur en appuyant sur 1, 2, 3 ou 4.

Très important : les booléens *clicGaucheEnCours* et *clicDroitEnCours* qui, comme leur nom l'indique, permettent de mémoriser si un clic est en cours (si le bouton de la souris est enfoncé). Cela nous permettra de poser des objets à l'écran en laissant le bouton de la souris enfoncé (sinon on est obligés de cliquer frénétiquement avec la souris pour placer plusieurs fois le même objet à différents endroits).
Je vous expliquerai le principe un peu plus loin.

Enfin, la carte actuellement sauvegardée dans niveaux.lvl est chargée. Ce sera notre point de départ.

La gestion des évènements

Cette fois, on va devoir gérer un nombre important d'évènements différents.
Allons-y, un par un

SDL_QUIT

Code : C

```
case SDL_QUIT:
    continuer = 0;
    break;
```

Si on clique sur la croix la boucle s'arrête et on revient au menu principal.

Notez que ce n'est pas pratique pour l'utilisateur ça : lui il s'attend à ce que le programme s'arrête quand on clique sur la croix, or ce n'est pas ce qu'il se passe ici. Il faudrait peut-être trouver un moyen d'arrêter le programme en renvoyant une valeur spéciale à la fonction main par exemple (je vous laisse réfléchir à une solution)

SDL_MOUSEBUTTONDOWN

Code : C

```
case SDL_MOUSEBUTTONDOWN:
    if (event.button.button == SDL_BUTTON_LEFT)
    {
        // On met l'objet actuellement choisi (mur, caisse...) à l'endroit du clic
        carte[event.button.x / TAILLE_BLOC][event.button.y / TAILLE_BLOC] = objetActuel;
        clicGaucheEnCours = 1; // On active un booléen pour retenir qu'un bouton est
enfoncé
    }
    else if (event.button.button == SDL_BUTTON_RIGHT) // Le clic droit sert à effacer
    {
        carte[event.button.x / TAILLE_BLOC][event.button.y /TAILLE_BLOC] = VIDE;
        clicDroitEnCours = 1;
    }
    break;
```

On commence par tester le bouton qui est enfoncé (on vérifie si c'est le clic gauche ou le clic droit) :

- Si c'est un clic gauche, on place l'*objetActuel* sur la carte à la position de la souris.
- Si c'est un clic droit, on efface ce qu'il y a à cet endroit sur la carte (on met VIDE comme je vous avais dit).

Ca se trouve par un petit calcul. Il suffit de prendre les coordonnées de la souris (event.button.x par exemple) et de diviser cette valeur par la taille d'un bloc (TAILLE_BLOC).
C'est une division de nombre entiers. Comme en C une division de nombre entiers donne un nombre entier, on est sûrs d'avoir une valeur qui corresponde à une des cases de la carte.

Exemple : si je suis au 75ème pixel sur la carte (sur l'axe des abscisses x), je divise ce nombre par TAILLE_BLOC qui vaut ici 34.

```
75 / 34 = 2
```

N'oubliez pas que le reste est ignoré. On ne garde que la partie entière de la division en C car il s'agit d'une divison de nombre entiers.
On sait donc qu'on se trouve sur la case n°2 (c'est-à-dire la 3ème case, car un tableau commence à 0 souvenez vous).

Autre exemple : si je suis au 10ème pixel (c'est-à-dire très proche du bord), ça va donner le calcul suivant :

```
10 / 34 = 0
```

On est donc à la case n°0 !

C'est comme ça qu'un simple petit calcul nous permet de savoir sur quelle case de la carte on se situe

Code : C

```c
carte[event.button.x / TAILLE_BLOC][event.button.y / TAILLE_BLOC] = objetActuel;
```

Autre chose très importante : on met un booléen clicGaucheEnCours (ou clicDroit selon le cas) à 1. Cela nous permettra de savoir lors d'un évènement MOUSEMOTION si un bouton de la souris est enfoncé pendant le déplacement !

SDL_MOUSEBUTTONUP

Code : C

```c
case SDL_MOUSEBUTTONUP: // On désactive le booléen qui disait qu'un bouton était enfoncé
    if (event.button.button == SDL_BUTTON_LEFT)
        clicGaucheEnCours = 0;
    else if (event.button.button == SDL_BUTTON_RIGHT)
        clicDroitEnCours = 0;
    break;
```

L'évènement MOUSEBUTTONUP sert simplement à remettre le booléen à 0. On sait que le clic est terminé et donc qu'il n'y a plus de "clic en cours".

SDL_MOUSEMOTION

Code : C

```
case SDL_MOUSEMOTION:
    if (clicGaucheEnCours) // Si on déplace la souris et que le bouton gauche de la
souris est enfoncé
    {
        carte[event.motion.x / TAILLE_BLOC][event.motion.y / TAILLE_BLOC] = objetActuel;
    }
    else if (clicDroitEnCours) // Pareil pour le bouton droit de la souris
    {
        carte[event.motion.x / TAILLE_BLOC][event.motion.y / TAILLE_BLOC] = VIDE;
    }
    break;
```

C'est là que nos booléens prennent toute leur importance. On vérifie quand on bouge la souris si un clic est en cours. Si tel est le cas, on place sur la carte un objet (ou du vide si c'est un clic droit).
Cela nous permet donc de placer plusieurs objets du même type d'affilée sans avoir à cliquer plusieurs fois. On a juste à déplacer la souris en maintenant le bouton de la souris enfoncé !

En clair, à chaque fois qu'on bouge la souris (ne serait-ce que d'un pixel), on vérifie si un des booléens est activé. Si tel est le cas, alors on pose un objet sur la carte. Sinon, on ne fait rien.

Résumé : je résume la technique, car vous vous en servirez certainement dans d'autres programmes.
Cette technique permet de savoir si un bouton de la souris est enfoncé lorsqu'on la déplace. On peut s'en servir pour coder un glisser / déplacer.

1. Lors d'un **MOUSEBUTTONDOWN** : on met un booléen clicEnCours à 1.
2. Lors d'un **MOUSEMOTION** : on teste si le booléen clicEnCours vaut vrai. S'il vaut vrai, on sait qu'on est en train de faire une sorte de glisser / déplacer avec la souris.
3. Lors d'un **MOUSEBUTTONUP** : on remet le booléen clicEnCours à 0, car le clic est terminé (relâchement du bouton de la souris).

SDL_KEYDOWN

Les touches du clavier permettent de charger / sauvgarder le niveau ainsi que de changer l'objet actuellement sélectionné pour le clic gauche de la souris.

Code : C

```
case SDL_KEYDOWN:
    switch(event.key.keysym.sym)
    {
        case SDLK_ESCAPE:
            continuer = 0;
            break;
        case SDLK_s:
            sauvegarderNiveau(carte);
            break;
        case SDLK_c:
            chargerNiveau(carte);
            break;
        case SDLK_KP1:
            objetActuel = MUR;
            break;
        case SDLK_KP2:
            objetActuel = CAISSE;
            break;
        case SDLK_KP3:
            objetActuel = OBJECTIF;
            break;
        case SDLK_KP4:
            objetActuel = MARIO;
            break;
    }
    break;
```

Ce code est très simple. On change l'objetActuel si on appuie sur une des touches numériques, on enregistre le niveau si on appuie sur S, ou on charge le dernier niveau enregistré si on appuie sur C.

Blit time !

Voilà, on a passé en revue tous les évènements.
Maintenant on n'a plus qu'à blitter chacun des éléments de la carte à l'aide d'une double boucle. C'est quasiment le même code que celui de la fonction de jeu. Je vous le redonne, mais pas la peine de vous le réexpliquer

Code : C

```
// Effacement de l'écran
SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));

// Placement des objets à l'écran
for (i = 0 ; i < NB_BLOCS_LARGEUR ; i++)
{
    for (j = 0 ; j < NB_BLOCS_HAUTEUR ; j++)
    {
        position.x = i * TAILLE_BLOC;
        position.y = j * TAILLE_BLOC;

        switch(carte[i][j])
        {
            case MUR:
                SDL_BlitSurface(mur, NULL, ecran, &position);
                break;
            case CAISSE:
                SDL_BlitSurface(caisse, NULL, ecran, &position);
                break;
            case OBJECTIF:
                SDL_BlitSurface(objectif, NULL, ecran, &position);
                break;
            case MARIO:
                SDL_BlitSurface(mario, NULL, ecran, &position);
                break;
        }
    }
}

// Mise à jour de l'écran
SDL_Flip(ecran);
```

Il ne faut pas oublier après la boucle principale de faire les SDL_FreeSurface qui s'imposent :

Code : C

```
SDL_FreeSurface(mur);
SDL_FreeSurface(caisse);
SDL_FreeSurface(objectif);
SDL_FreeSurface(mario);
```

C'est touuuut

RÉSUMÉ ET AMÉLIORATIONS

Bien, on a vu assez de code je crois (en fait on a tout vu)
L'heure est au résumé.

Alors résumons !

Et quel meilleur résumé pourrait-on imaginer que **le code source complet du programme avec les commentaires** ?
C'est gratuit, c'est cadeau, c'est offert par le Site du Zér0 (faut vraiment que j'arrête les rimes stupides...)

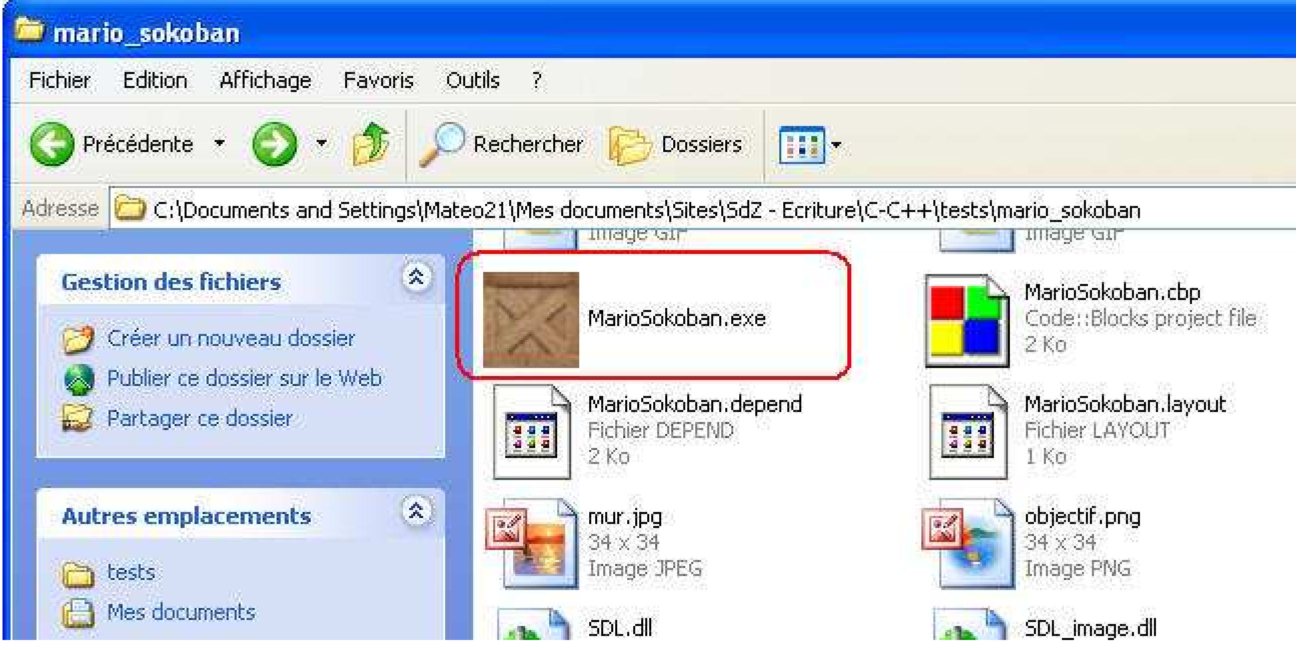
Télécharger le programme + les sources (436 Ko)

Ce fichier zip contient :

- L'exécutable pour Windows (si vous êtes sous un autre OS, il suffira de recompiler)
- Les DLL de la SDL et de SDL_Image
- Toutes les images dont a besoin le programme (je vous les ai fait télécharger plus tôt dans le pack "sprites").

- Les sources complètes du programme
- Le fichier .cbp de projet Code::Blocks (oui j'ai fait ça sous Code::Blocks). Si vous voulez ouvrir le projet sous un autre IDE, créez un nouveau projet de type SDL (configurez-le correctement pour la SDL) et ajoutez-y manuellement tous les fichiers .c et .h. Ce n'est pas bien compliqué vous verrez.

Vous noterez que le projet contient, en plus des .c et des .h, un fichier ressources.rc.
C'est un fichier qui peut être ajouté au projet (uniquement sous Windows) permettant d'intégrer des fichiers dans l'exécutable. Ici, je me sers du fichiers de ressources pour intégrer une icône dans l'exécutable. Cela aura pour effet de donner une icône à l'exécutable, visible dans l'explorateur Windows :



Avouez que c'est quand même plus sympa que d'avoir l'icône par défaut de Windows pour les exécutables

Vous trouverez plus d'infos sur cette technique dans l'annexe **Créer une icône pour son programme**.
Je vous rappelle que cela ne concerne que Windows.

Améliorez !

Ce programme n'est pas parfait, loin de là !
Vous voulez des idées pour l'améliorer ? J'en ai plein !

- Il manque un **mode d'emploi**. Affichez un écran d'explications juste avant le lancement d'une partie et avant le lancement de l'éditeur. Indiquez en particulier les touches à utiliser.
- Dans l'éditeur de niveaux, on ne sait pas quel est l'objet actuellement sélectionné. Ce qui serait bien, c'est qu'on ait **l'objet actuellement sélectionné** qui suive le curseur de la souris. Comme ça, l'utilisateur verrait ce qu'il s'apprête à mettre sur la carte. C'est facile à faire : on a déjà fait un Zozor qui suit le curseur de la souris dans le chapitre précédent !
- Dans l'éditeur de niveaux, il serait bien qu'on puisse choisir **l'objet CAISSE_OK** (une caisse bien placée sur un objectif dès le départ). En effet, je me suis rendu compte par la suite qu'il y a de nombreux niveaux qui commencent avec des caisses bien placées dès le départ (ça ne veut pas dire que le niveau est plus facile, loin de là)
- Dans l'éditeur toujours, il faudrait **empêcher que l'on puisse placer plus d'un départ de joueur sur une même carte** !

- Lorsqu'on réussit un niveau, on retourne immédiatement au menu. C'est un peu brut. Que diriez-vous d'**afficher un message "Bravo"** au centre de l'écran quand on gagné ?
- Enfin, il serait bien que le programme puisse gérer plus d'un niveau à la fois. Il faudrait que l'on puisse créer **une véritable petite aventure d'une vingtaine de niveaux** par exemple. C'est un petit peu plus compliqué à coder mais faisable. Il faudra adapter le jeu et l'éditeur de niveaux en conséquence.

Je vous suggère de mettre un niveau par ligne dans niveaux.lvl

Comme promis, pour vous prouver que c'est faisable... je l'ai fait
Je ne vous donne pas le code source en revanche (je crois que je vous en ai déjà assez donné jusqu'ici !), mais je vous donne le programme complet pour Windows.

Le programme comporte une aventure de 20 niveaux (de très très facile... à super difficile).
Pour la plupart de ces niveaux, je me suis servi sur le **site d'un passionné de Sokoban**. Certains niveaux sont vraiment tordus, je ne suis pas capable de créer des trucs aussi compliqués, donc merci à l'auteur pour tous les niveaux qu'il propose

Puisque j'y suis, comme c'est un "vrai" programme, je me suis permis de créer une installation. C'est un véritable programme d'installation qui va vous demander où vous voulez mettre le programme et si vous voulez l'exécuter à la fin. Il mettra même des raccourcis dans le menu démarrer ou sur le bureau si vous le souhaitez

Téléchargez l'installation du Mario Sokoban amélioré (665 Ko)

Ou bien : **Téléchargez la version compilée pour linux au format .tar.gz (64 Ko)**

Pour créer l'installation, je me suis servi de l'excellent programme de création d'installations **Inno Setup**. Il est entièrement gratuit et très puissant.
Si vous voulez apprendre à créer des installations de ce type vous aussi, je vous invite à lire l'annexe **Créer une installation**. C'est facile, rapide, professionnel et rudement efficace vous verrez !

Pfiou !
Sacré boulot hein ? Tout ça pour coder un jeu de Sokoban

Remarquez, le jeu est bien complet (surtout dans sa version améliorée) et peut facilement être distribué à vos amis (ou même vendu). Bon les Sokoban c'est un marché saturé depuis un moment, c'est pas ce qui manque, donc n'espérez pas faire fortune avec ça

Comme vous pouvez le constater, programmer un jeu c'est faisable mais... ça demande du travail ! Vous n'imaginiez peut-être pas en commençant le cours que ça serait autant de travail, mais c'est pourtant la réalité : il faut tout dire à l'ordinateur (sinon il sait pas faire !).

Ne soyez pas impressionnés par la quantité de code source et ne vous laissez pas submerger. Moi aussi, comme vous, j'ai du mal à "rentrer" dans le code source de quelqu'un d'autre. Le mieux est certainement de **créer vous-même votre code source de A à Z.** L'expérience vient en pratiquant, je crois que je l'ai déjà dit pas mal de fois

Bon codage les Zér0s !

Maîtrisez le temps !

Ce chapitre est d'une importance capitale : il va vous apprendre à **gérer le temps en SDL**.

Maîtriser le temps est quasiment indispensable... j'ai bien dit quasiment car, comme vous l'avez vu dans le TP **Mario**

Sokoban, on peut très bien faire un jeu sans manipuler le temps.
Mais c'est rare. Pour un nombre pratiquement incalculable de jeux, la gestion du temps est fondamentale.

Tenez par exemple, comment coderiez-vous un **Tetris** ou un **Snake** (jeu du serpent) ? Il faut bien que les blocs bougent toutes les X secondes, et ça vous ne savez pas faire. Du moins, vous ne savez pas encore le faire

Le Delay et les Ticks

Dans un premier temps, nous allons apprendre à utiliser 2 fonctions très simples :

- **SDL_Delay** : permet de mettre en pause le programme un certain nombre de millisecondes.
- **SDL_GetTicks** : retourne le nombre de millisecondes écoulées depuis le lancement du programme.

Ces 2 fonctions sont peut-être super simples comme nous allons le voir, mais bien savoir les utiliser n'est pas évident... alors ouvrez grand vos ~~oreilles~~ yeux

SDL_Delay

Comme je l'ai précédemment dit, cette fonction effectue une pause sur le programme durant un certain temps.
Pendant que le programme est en pause, on dit qu'il dort ("sleep" en anglais) : il n'utilise pas le processeur.

SDL_Delay peut donc être utile pour réduire l'utilisation du processeur. Notez que j'abrègerai processeur par **CPU** désormais, ce qui signifie Central Processing Unit, soit "*Unité centrale de calcul*".
Grâce à SDL_Delay, vous pourrez rendre votre programme moins gourmand en ressources processeur. Il fera donc moins "ramer" votre PC si SDL_Delay est utilisée intelligemment.

Tout dépend du programme que vous faites : parfois on aimerait bien que notre programme utilise le moins de CPU possible pour que l'utilisateur puisse faire autre chose en même temps, comme c'est le cas pour un lecteur MP3 qui tourne en fond pendant que vous naviguez sur Internet.
Mais... d'autres fois on se moque complètement que notre programme utilise tout le temps 100% de CPU. C'est le cas de la quasi-totalité des jeux. En effet : quand vous jouez, en théorie vous ne faites que ça, donc on peut se permettre d'utiliser tout le temps le CPU. Quand vous jouez à Far Cry ou Half-Life 2, votre PC travaille donc à 100% tout le temps !

Bien, revenons à la fonction qui nous intéresse. Son prototype est d'une simplicité affligeante :

Code : C

```c
void SDL_Delay(Uint32 ms);
```

En clair, vous envoyez à la fonction le nombre de millisecondes pendant lesquelles votre programme doit "dormir".
C'est une simple mise en pause.

Par exemple, si vous voulez que votre programme se mette en pause 1 seconde, vous devrez taper :

Code : C

```c
SDL_Delay(1000);
```

N'oubliez pas que ce sont des millisecondes :
1000 millisecondes = 1 seconde
500 millisecondes = 1/2 seconde
250 millisecondes = 1/4 seconde
etc etc.

Attention : vous ne pouvez rien faire dans votre programme pendant qu'il est en pause ! Un programme qui "dort" ne peut rien faire puisqu'il n'est pas actif pour l'ordinateur.

Le problème de la granularité du temps

Non rassurez-vous, je ne vais pas vous faire un traité de physique quantique au beau milieu d'un chapitre SDL
Toutefois, il y a quelque chose que j'estime que vous devez savoir : SDL_Delay n'est pas une fonction "parfaite". Et ce n'est pas sa faute, c'est la faute à votre OS (comme Windows).

Qu'est-ce que vient faire l'OS là-dedans ?

Oh mais l'OS a tout à faire là-dedans : c'est lui qui contrôle les programmes qui tournent !
Votre programme va donc dire à l'OS : "*Je dors, réveille-moi dans 1 seconde*". Mais l'OS ne va pas forcément le réveiller au bout d'une seconde exactement.

En effet, il aura peut-être un peu de retard (un retard de 10ms en moyenne environ, ça dépend des PC). Pourquoi ?
Parce que votre CPU ne peut travailler que sur un programme à la fois. Le rôle de l'OS est de dire au CPU ce sur quoi il doit travailler : "*Alors, pendant 40ms tu vas travailler sur firefox.exe, puis pendant 110ms tu vas travailler sur explorer.exe, et ensuite pendant 80ms tu vas travailler sur programme_sdl.exe, puis tu vas retravailler sur firefox.exe pendant 65ms*" etc etc...
L'OS est le véritable chef d'orchestre de l'ordinateur !

Maintenant, imaginez qu'au bout d'une seconde un autre programme soit encore en train de travailler : il faudra qu'il ait fini de travailler pour que votre programme puisse "reprendre la main" comme on dit, c'est-à-dire être traité à nouveau par le CPU.

Tout ça pour dire quoi ?

Ooops, excusez-moi j'étais en train de dériver
En gros, j'essayais de vous expliquer que votre CPU ne pouvait pas gérer plus d'un programme à la fois. Pour donner l'impression que l'on peut faire tourner plusieurs programmes en même temps sur un ordinateur, l'OS "découpe" le temps et autorise les programmes à travailler au tour par tour.

Or, cette gestion des programmes est très complexe et on ne peut donc pas avoir la garantie que notre programme sera réveillé au bout d'une seconde exactement.

Toutefois, ça dépend des PC comme je vous l'ai dit plus haut. Chez moi, la fonction SDL_Delay est assez précise.

A cause de ce problème de "granularité du temps", vous ne pouvez donc pas mettre en pause votre programme pendant un temps trop court. Par exemple, si vous faites :

Code : C

```
SDL_Delay(1);
```

Vous pouvez être sûrs que votre programme ne sera pas mis en pause 1ms mais un peu plus (peut-être 9-10ms).

En résumé : SDL_Delay c'est cool, mais ne lui faites pas trop confiance. Elle ne mettra pas en pause votre programme pendant le temps *exact* que vous indiquez.
Ce n'est pas parce que la fonction est mal codée, c'est parce que le fonctionnement d'un ordinateur est très complexe et ne permet pas d'être très précis à ce niveau.

SDL_GetTicks

Cette fonction renvoie le nombre de millisecondes écoulées depuis le lancement du programme.

Au contraire, c'est un indicateur de temps indispensable. Cela vous permet de vous **repérer dans le temps**, vous allez voir !

Voici le prototype :

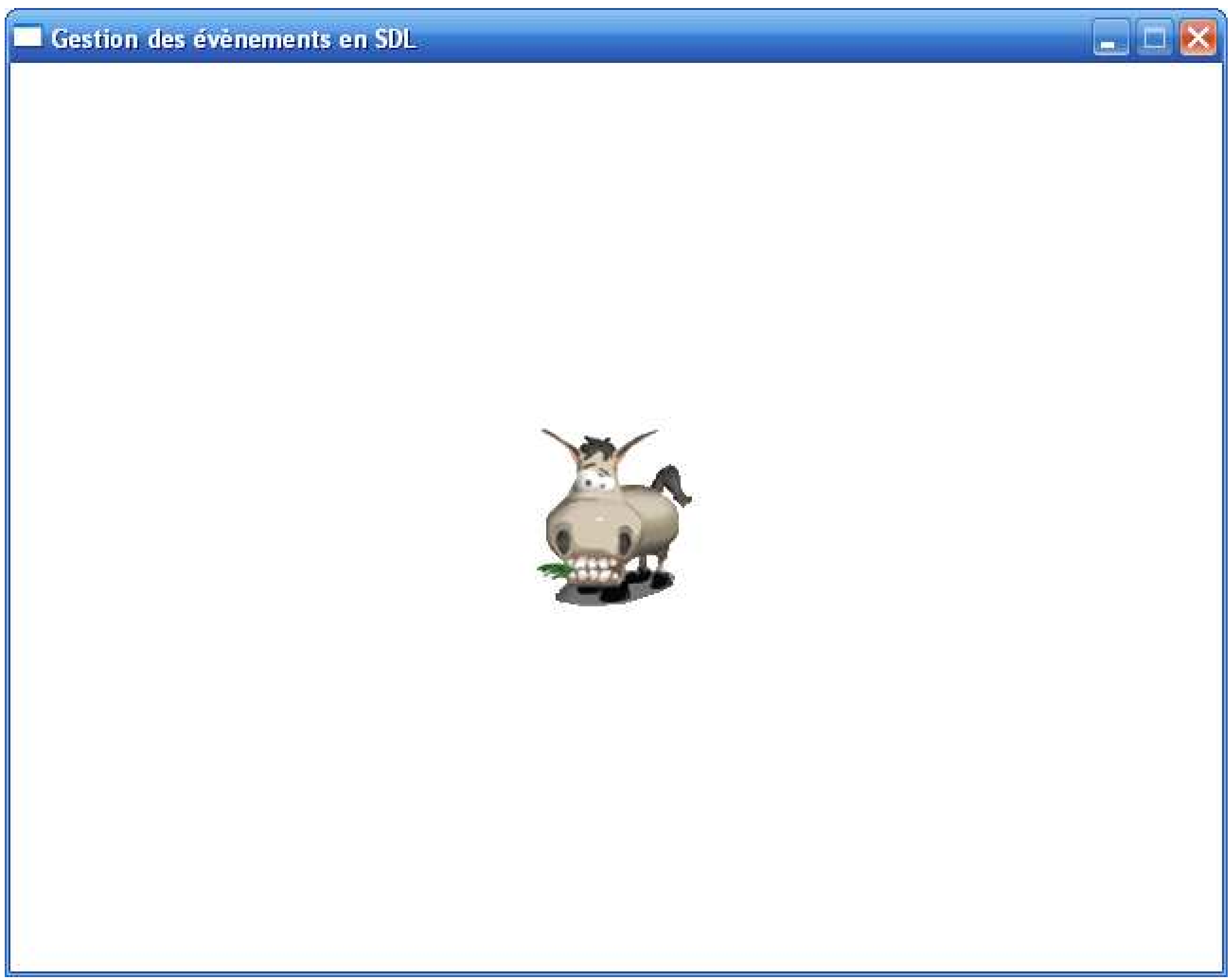
Code : C

```
Uint32 SDL_GetTicks(void);
```

La fonction n'attend aucun paramètre, elle renvoie juste le nombre de millisecondes écoulées.
Ce nombre augmente au fur et à mesure du temps, inlassablement. Pour info, la doc de la SDL indique que le nombre atteint son maximum et est réinitialisé au bout de 49 jours ! A priori votre programme SDL devrait tourner moins longtemps que ça, donc pas de souci de ce côté-là

Utiliser SDL_GetTicks pour gérer le temps

Si SDL_Delay est assez facile à comprendre et à utiliser, ce n'est pas le cas de SDL_GetTicks. Il est temps d'apprendre à bien s'en servir...
Voici un exemple ! Nous allons reprendre notre bon vieux programme avec la fenêtre affichant Zozor à l'écran.
Souvenez-vous :



Cette fois, au lieu de le diriger au clavier ou à la souris, nous allons faire en sorte qu'il bouge tout seul sur l'écran ! Pour faire simple, on va le faire bouger horizontalement sur la fenêtre.

On reprend pour commencer exactement le même code source que celui qu'on avait utilisé dans le chapitre sur les évènements. Cette fois, j'ai enlevé la partie gérant les évènements au clavier vu qu'on ne va plus bouger Zozor au clavier :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *zozor = NULL;
    SDL_Rect positionZozor;
    SDL_Event event;
    int continuer = 1;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du temps en SDL", NULL);

    zozor = SDL_LoadBMP("zozor.bmp");
    SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));

    positionZozor.x = ecran->w / 2 - zozor->w / 2;
    positionZozor.y = ecran->h / 2 - zozor->h / 2;

    SDL_EnableKeyRepeat(10, 10);

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
        SDL_BlitSurface(zozor, NULL, ecran, &positionZozor);
        SDL_Flip(ecran);
    }

    SDL_FreeSurface(zozor);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Bon, si vous testez ce code, vous devriez voir un Zozor au centre de la fenêtre. Rien de bien palpitant encore.

Nous voulons le faire bouger. Pour cela, le mieux est d'utiliser **SDL_GetTicks**. On va avoir besoin de 2 variables : *tempsPrecedent* et *tempsActuel*. Elles vont stocker le temps retourné par **SDL_GetTicks** à des moments différents. Il nous suffira de faire la différence entre *tempsActuel* et *tempsPrecedent* pour voir le temps qui s'est écoulé. Si le temps écoulé est supérieur à 30ms par exemple, alors on change les coordonnées de Zozor.

Commencez donc par créer ces 2 variables dont on va avoir besoin :

Code : C

```
int tempsPrecedent = 0, tempsActuel = 0;
```

Maintenant, dans notre boucle infinie, nous allons rajouter le code suivant :

Code : C

```
tempsActuel = SDL_GetTicks();
if (tempsActuel - tempsPrecedent > 30) /* Si 30 ms se sont écoulées */
{
    positionZozor.x++; /* On bouge Zozor */
    tempsPrecedent = tempsActuel; /* Le temps "actuel" devient le temps "precedent" pour
nos futurs calculs */
}
```

Attention, là c'est <u>super important</u>. C'est là qu'on comprend le truc (ou pas)

1. On prend le temps actuel grâce à **SDL_GetTicks**
2. On compare au temps précedemment enregistré. Si il y a un écart de 30 ms au moins, alors...
3. ... alors on bouge Zozor, car on veut qu'il se déplace toutes les 30 ms. Ici, on le décale juste vers la droite toutes les 30 ms.

   Il faut vérifier si le temps est <u>supérieur</u> à 30ms, et non égal à 30ms ! En effet, il faut vérifier si au moins 30ms se sont écoulées. Rien ne vous garantit que l'instruction sera exécutée pile poil toutes les 30ms

4. Puis (et c'est vraiment le truc à pas oublier), on place le temps "actuel" dans le temps "précédent". En effet, imaginez au prochain tour de boucle : le temps "actuel" aura changé, et on pourra le comparer au temps précédent. A nouveau, on pourra vérifier si 30 ms se seront écoulées et bouger Zozor

Et que se passe-t-il si la boucle va plus vite que 30 ms ?

Lisez mon code : il ne se passe rien
On ne rentre pas dans le "if", on ne fait donc rien. On attend le prochain tour de boucle où on vérifiera à nouveau si 30 ms se seront écoulées depuis la dernière fois qu'on a fait bouger Zozor.

Ce code est court, mais il faut le comprendre ! Relisez mes explications autant de fois que nécessaire, parce que c'était ça le passage le plus important du chapitre

Un changement dans la gestion des évènements

Notre code est presque bon à un détail près : la fonction **SDL_WaitEvent**.
Elle était très pratique jusqu'ici, puisqu'on n'avait pas à gérer le temps. Cette fonction mettait en "pause" le programme (un peu à la manière de *SDL_Delay*) tant qu'il n'y avait pas d'évènement.

Or ici, on n'a pas besoin d'attendre un évènement pour faire bouger Zozor ! <u>Il doit bouger tout seul</u>.
Vous n'allez quand même pas bouger la souris juste pour générer des évènements et donc faire sortir le programme de la fonction SDL_WaitEvent

La solution ? **SDL_PollEvent**.
Je vous avais déjà présenté cette fonction : contrairement à SDL_WaitEvent, elle renvoie une valeur qu'il y ait eu un évènement ou pas. On dit que la fonction n'est pas "bloquante" : elle ne met pas en pause le programme, la boucle infinie va donc tourner indéfiniment tout le temps.

Code complet

Voici le code final que vous pouvez tester :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *zozor = NULL;
    SDL_Rect positionZozor;
    SDL_Event event;
    int continuer = 1;
    int tempsPrecedent = 0, tempsActuel = 0;

    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du temps en SDL", NULL);

    zozor = SDL_LoadBMP("zozor.bmp");
    SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));

    positionZozor.x = ecran->w / 2 - zozor->w / 2;
    positionZozor.y = ecran->h / 2 - zozor->h / 2;

    SDL_EnableKeyRepeat(10, 10);


    while (continuer)
    {
        SDL_PollEvent(&event); /* On utilise PollEvent et non WaitEvent pour ne pas
bloquer le programme */
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
        }

        tempsActuel = SDL_GetTicks();
        if (tempsActuel - tempsPrecedent > 30) /* Si 30 ms se sont écoulées depuis le
dernier tour de boucle */
        {
            positionZozor.x++; /* On bouge Zozor */
            tempsPrecedent = tempsActuel; /* Le temps "actuel" devient le temps
"precedent" pour nos futurs calculs */
        }


        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
        SDL_BlitSurface(zozor, NULL, ecran, &positionZozor);
        SDL_Flip(ecran);
    }

    SDL_FreeSurface(zozor);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Vous devriez voir Zozor bouger tout seul sur l'écran. Il se décale vers la droite
Essayez de changer le temps de 30ms en 15ms par exemple : Zozor devrait se déplacer 2 fois plus vite ! En effet, il se déplacera une fois toutes les 15ms au lieu d'une fois toutes les 30ms auparavant

Consommer moins de CPU

Actuellement, notre programme tourne en boucle indéfiniment à la vitesse de la lumière (enfin presque). Il consomme donc 100% du CPU.
Si vous faites CTRL + ALT + SUPPR (onglet "Processus"), vous verrez ça sous Windows :

Comme vous pouvez le voir, notre CPU est utilisé à 100% par notre programme testsdl.exe.
Je vous l'ai dit plus tôt : si vous codez un jeu (surtout un jeu plein écran), ce n'est pas grave si vous utilisez 100% du CPU. Mais si c'est un jeu dans une fenêtre par exemple, il vaut mieux qu'il utilise le moins de CPU possible pour que l'utilisateur puisse faire autre chose sans que son PC ne "rame".

La solution ? On va reprendre exactement le même code que ci-dessus mais on va lui ajouter en plus un SDL_Delay pour patienter le temps qu'il faut afin que ça fasse 30ms.

On va juste rajouter un SDL_Delay dans un "else" :

Code : C

```c
tempsActuel = SDL_GetTicks();
if (tempsActuel - tempsPrecedent > 30)
{
    positionZozor.x++;
    tempsPrecedent = tempsActuel;
}
else /* Si ça fait moins de 30ms depuis le dernier tour de boucle, on endort le programme le temps qu'il faut */
{
    SDL_Delay(30 - (tempsActuel - tempsPrecedent));
}
```

Comment ça fonctionne cette fois ?
C'est simple, il y a 2 possibilités (d'après le if) :

- Soit ça fait plus de 30ms qu'on n'a pas bougé Zozor, dans ce cas on le bouge.
- Soit ça fait moins de 30 ms, dans ce cas on fait dormir le programme avec SDL_Delay le temps qu'il faut pour que ça fasse 30ms environ.
  D'où mon petit calcul `30 - (tempsActuel - tempsPrecedent)`. Si la différence entre le temps actuel et le temps précédent est de 20ms par exemple, alors on endormira le programme (30 - 20) = 10ms afin que ça fasse environ 30ms

Rappelez-vous que SDL_Delay mettra peut-être quelques millisecondes de plus que prévu. Chez moi, comme je vous l'ai dit, c'est assez précis.

Du coup, notre programme va "dormir" la plupart du temps et donc consommer très peu de CPU. Regardez !

En moyenne, le programme utilise 0-1% de CPU... Parfois il utilise légèrement plus, mais il retombe rapidement à 0% de CPU.

Contrôler le nombre d'images par seconde

Vous vous demandez certainement comment on peut limiter (fixer) le nombre d'images par seconde (FPS) affichées par l'ordinateur.

Eh bien c'est exactement ce qu'on est en train de faire ! Ici, on affiche une nouvelle image toutes les 30ms en moyenne.
Sachant qu'une seconde vaut 1000ms, pour trouver le nombre de FPS (images par seconde), il suffit de faire une bête division : 1000 / 30 = **33 images par seconde** environ.

Pour l'oeil humain, une animation est fluide si elle contient au moins 25 images / seconde. Avec 33 images / seconde, notre animation sera donc tout à fait fluide, elle n'apparaîtra pas "saccadée".

Si vous voulez plus d'images par seconde, il faut réduire la limite de temps entre 2 images. Passez de 30 à 20ms, et ça vous fera du 1000 / 20 = 50 FPS

Exercices

La manipulation du temps n'est pas évidente, il serait bien de vous entraîner un peu, qu'en dites-vous ?
Voici quelques exercices justement :

- Pour le moment, Zozor se décale vers la droite puis disparaît de l'écran. Ce serait mieux s'il repartait dans l'autre sens une fois arrivé tout à droite non ? **Ca donnerait l'impression qu'il rebondit**
  Pour faire ça, je vous conseille de créer un booléen *versLaDroite* qui vaut vrai si Zozor se déplace vers la droite (et faux s'il va vers la gauche). Si le booléen vaut vrai, vous décalez donc Zozor vers la droite, sinon vous le décalez vers la gauche.
  Surtout, n'oubliez pas de changer la valeur du booléen lorsque Zozor atteint le bord droit ou le bord gauche ! Eh oui, il faut qu'il reparte dans l'autre sens
- Plutôt que de faire rebondir Zozor de droite à gauche, faites le rebondir en diagonale sur l'écran ! Il vous suffira de modifier *positionZozor.x* et *positionZozor.y* simultanément. Vous pouvez essayer de voir ce que ça fait si on augmente x et si on diminue y en même temps, ou bien si on augmente les 2 en même temps etc.
- Faites en sorte qu'un appui sur **la touche P empêche Zozor de se déplacer**, et qu'un rappui à nouveau sur la touche P relance le déplacement de Zozor. C'est un bête booléen à activer / désactiver

Voilà, ce sont quelques petits exos comme ça qui devraient vous occuper quelques minutes et vous permettre ainsi d'améliorer le programme

LES TIMERS

L'utilisation des Timers est un peu complexe. Elle fait intervenir une notion qu'on n'a pas vue jusqu'ici : les pointeurs de fonctions.
Il n'est pas indispensable d'utiliser les Timers : si vous ne le sentez pas, vous pouvez donc passer votre chemin sans souci

Les Timers sont une autre façon de réaliser ce qu'on vient de faire avec SDL_GetTicks.
C'est une technique un peu particulière. Certains la trouveront pratique, d'autres pas. Ca dépend donc des goûts du programmeur

Qu'est-ce qu'un Timer ?
C'est un système qui permet de demander à la SDL d'**appeler une fonction toutes les X millisecondes**. Vous pourriez ainsi créer une fonction `bougerEnnemi()` que la SDL appellerait automatiquement toutes les 50ms afin que l'ennemi se déplace à intervalles réguliers.

Comme je viens de vous le dire, cela est aussi faisable avec SDL_GetTicks en utilisant la technique qu'on a vue plus haut.
Quel avantage alors ? Eh bien disons que les Timers nous obligent à mieux structurer notre programme en fonctions.

Initialiser le système de Timers

Pour pouvoir utiliser les Timers, vous devez d'abord initialiser la SDL avec un flag spécial : SDL_INIT_TIMER.
Vous devriez donc avoir une fonction SDL_Init appelée comme ceci :

Code : C

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER);
```

Voilà, la SDL est maintenant prête à utiliser les Timers

Ajouter un Timer

Il existe 2 fonctions permettant d'ajouter un Timer en SDL : SDL_AddTimer et SDL_SetTimer.
Quelle est la différence entre les 2 ? En fait, elles sont quasiment identiques. Cependant, SDL_SetTimer est une fonction "ancienne" qui existe toujours pour des raisons de compatibilité. Aujourd'hui, si on veut bien faire les choses, on nous recommande donc d'utiliser **SDL_AddTimer**

Alors attention, c'est là que ça se corse. Voici le prototype de SDL_AddTimer :

Code : C

```
SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback callback, void *param);
```

On envoie 3 paramètres à la fonction :

- **L'intervalle de temps** (en ms) entre chaque appel de la fonction
- **Le nom de la fonction à appeler**. On appelle cela un callback : le programme se charge de rappeler cette fonction de callback régulièrement.
- **Les paramètres** à envoyer à votre fonction de callback.

QUOIII ? Un nom de fonction peut servir de paramètre ? Je croyais qu'on ne pouvait envoyer que des variables !?

En fait, les fonctions sont aussi stockées en mémoire au chargement du programme. Elles ont donc elles aussi une adresse.
Du coup, on peut créer des... **pointeurs de fonctions** ! Il suffit d'écrire le nom de la fonction à appeler pour indiquer l'adresse de la fonction. Ainsi, la SDL saura à quelle adresse en mémoire elle doit se rendre pour appeler votre fonction de callback.

Si vous voulez en savoir plus sur les pointeurs de fonctions, je vous conseille de lire **le tuto rédigé par mleg à ce sujet.**

SDL_AddTimer renvoie un numéro de Timer (un "ID"). Vous devez stocker ce résultat dans une variable de type *SDL_TimerID*. Cela vous permettra par la suite de désactiver le Timer : il vous suffira d'indiquer l'ID du Timer à arrêter.

La SDL vous permet d'activer plusieurs Timers en même temps. Cela explique l'intérêt de stocker un ID de Timer : on peut ainsi savoir par la suite quel est le Timer qu'on demande à arrêter

On va donc créer un ID de Timer :

Code : C

```
SDL_TimerID timer; /* Variable pour stocker le numéro du Timer */
```

... puis on va créer notre Timer :

Code : C

```
timer = SDL_AddTimer(30, bougerZozor, &positionZozor); /* Démarrage du Timer */
```

Ici, je crée un Timer qui a les propriétés suivantes :

- Il sera appelé toutes les 30ms
- Il appellera la fonction de callback bougerZozor
- Il lui enverra comme paramètre un pointeur sur la position de Zozor pour qu'il puisse la modifier.

Vous l'aurez compris : le rôle de la fonction bougerZozor sera de changer la position de Zozor toutes les 30ms

Création de la fonction de callback

Attention là il ne faut pas se planter. Votre fonction de callback doit obligatoirement avoir le prototype suivant :

Code : C

```
Uint32 nomDeLaFonction(Uint32 intervalle, void *parametre);
```

Pour créer le callback bougerZozor, je devrai donc écrire la fonction comme ceci :

Code : C

```
Uint32 bougerZozor(Uint32 intervalle, void *parametre);
```

Voici maintenant le contenu de ma fonction bougerZozor (ouvrez grand les yeux c'est très intéressant) :

Code : C

```c
/* Fonction de callback (sera appelée toutes les 30ms) */
Uint32 bougerZozor(Uint32 intervalle, void *parametre)
{
    SDL_Rect* positionZozor = parametre; /* Conversion de void* en SDL_Rect* */
    positionZozor->x++;

    return intervalle;
}
```

La fonction bougerZozor sera donc automatiquement appelée toutes les 30ms par la SDL.
La SDL lui enverra toujours 2 paramètres (ni plus, ni moins) :

- **L'intervalle de temps** qui sépare 2 appels de la fonction (ici ça sera 30)
- **Le paramètre "personnalisé"** que vous avez demandé à envoyer à la fonction. Remarquez, et c'est très important, que ce paramètre est un pointeur sur void. Cela signifie que c'est un pointeur qui peut pointer sur n'importe quoi : un int, une structure personnalisée, ou comme ici un SDL_Rect (positionZozor)

Le problème, c'est que ce paramètre est un pointeur de type "inconnu" (void) pour la fonction. Il va donc falloir dire à l'ordinateur que ce paramètre est un SDL_Rect* (un pointeur sur SDL_Rect).

Pour faire ça, je crée un pointeur sur SDL_Rect dans ma fonction qui prend comme valeur... le pointeur *parametre*.

Quel intérêt d'avoir créé un DEUXIEME pointeur qui contient la même adresse ?

L'intérêt, c'est que *positionZozor* est de type SDL_Rect* contrairement à *parametre* qui était de type void*.

Vous pourrez donc accéder à *positionZozor->x* et *positionZozor->y*.
Si vous aviez fait *parametre->x* ou *parametre->y*, le compilateur vous aurait jeté parce que le type void ne contient pas de sous-variable x et y

Après, la ligne suivante est simple : on modifie la valeur de positionZozor->x pour décaler Zozor vers la droite.

Dernière chose, très importante : **vous devez retourner la variable *intervalle***. Cela indiquera à la SDL qu'on veut continuer à ce que la fonction soit appelée toutes les 30ms.
Si vous voulez changer l'intervalle d'appel, il suffit de renvoyer une autre valeur (mais bien souvent, on ne change pas l'intervalle d'appel)

Arrêter le Timer

Pour arrêter le Timer, c'est super simple :

Code : C

```c
SDL_RemoveTimer(timer); /* Arrêt du Timer */
```

Il suffit d'appeler SDL_RemoveTimer en indiquant l'ID du Timer à arrêter.
Ici, j'arrête le Timer juste après la boucle infinie (au même endroit que les SDL_FreeSurface quoi).

Code complet d'exemple

Allez, on résume tout ça pour mettre nos idées au clair. Toutes les lignes intéressantes sont commentées :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>

/* Prototype de la fonction de callback */
Uint32 bougerZozor(Uint32 intervalle, void *parametre);

int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *zozor = NULL;
    SDL_Rect positionZozor;
    SDL_Event event;
    int continuer = 1;
    SDL_TimerID timer; /* Variable pour stocker le numéro du Timer */

    SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du temps en SDL", NULL);

    zozor = SDL_LoadBMP("zozor.bmp");
    SDL_SetColorKey(zozor, SDL_SRCCOLORKEY, SDL_MapRGB(zozor->format, 0, 0, 255));

    positionZozor.x = ecran->w / 2 - zozor->w / 2;
    positionZozor.y = ecran->h / 2 - zozor->h / 2;

    SDL_EnableKeyRepeat(10, 10);

    timer = SDL_AddTimer(30, bougerZozor, &positionZozor); /* Démarrage du Timer */

    while (continuer)
    {
        SDL_PollEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
        }


        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
        SDL_BlitSurface(zozor, NULL, ecran, &positionZozor);
        SDL_Flip(ecran);
    }

    SDL_RemoveTimer(timer); /* Arrêt du Timer */

    SDL_FreeSurface(zozor);
    SDL_Quit();

    return EXIT_SUCCESS;
}

/* Fonction de callback (sera appelée toutes les 30ms) */
Uint32 bougerZozor(Uint32 intervalle, void *parametre)
{
    SDL_Rect* positionZozor = parametre; /* Conversion de void* en SDL_Rect* */
    positionZozor->x++;

    return intervalle;
}
```

Vous devriez voir Zozor bouger toutes les 30ms. Cette fois, le déplacement est géré par un Timer qui appelle une fonction de callback à intervalle de temps régulier (ouah ça fait pro cette phrase vous trouvez pas 😁)

Sur un petit exemple comme ça, il peut sembler plus simple d'utiliser SDL_GetTicks. Je suis bien d'accord. Mais sur de gros programmes, vous aurez très probablement des fonctions de callback plus longues et complexes à créer, ce qui

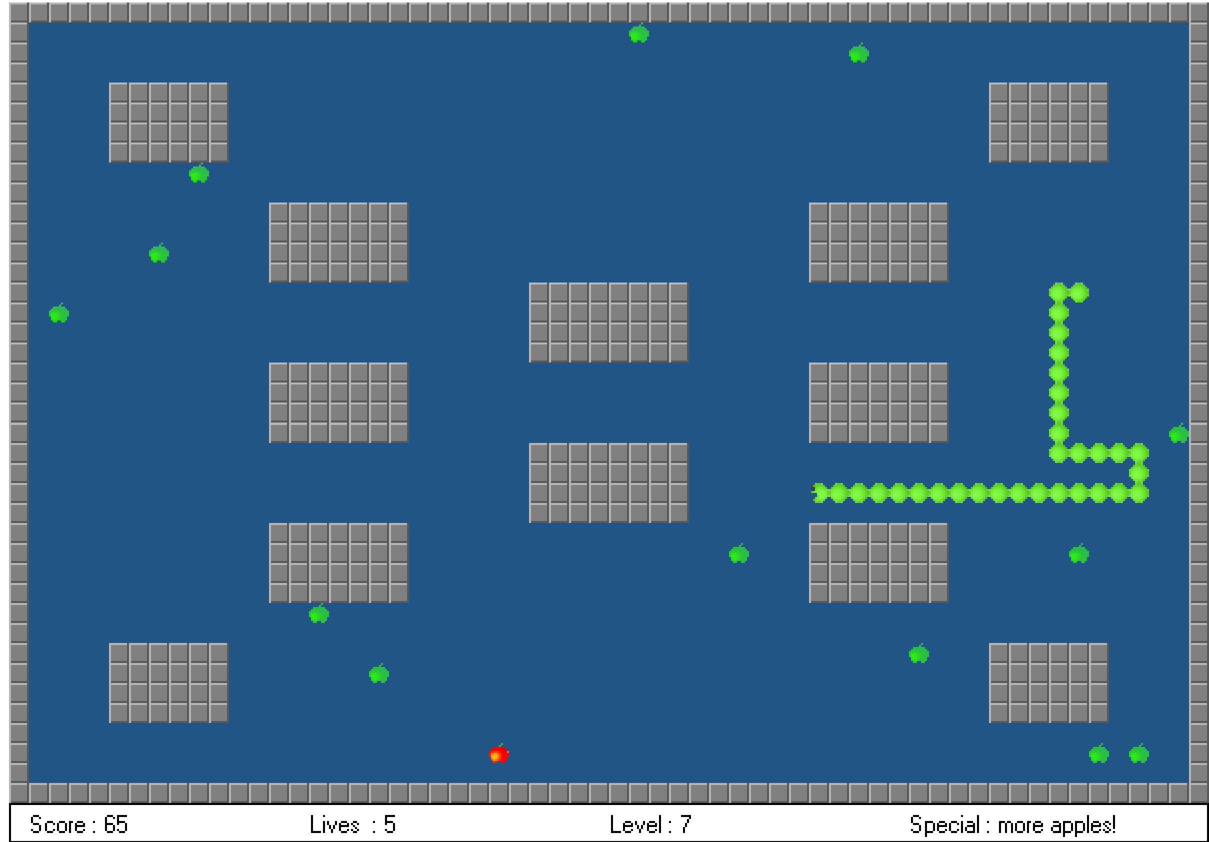
rendra l'utilisation de Timers plus intéressante

Au fait... On ne peut envoyer qu'un seul paramètre à la fonction de callback ?

Oui. Comme vous le voyez, le seul paramètre qu'on peut envoyer est : `void *parametre`.
Ce void* peut être un pointeur sur int, sur SDL_Rect, sur ce que vous voulez quoi

Si vous voulez envoyer plusieurs variables à la fois, c'est tout à fait possible : créez une structure personnalisée qui contiendra les variables que vous voulez. C'est d'ailleurs ce qu'on fait avec notre positionZozor qui contient les sous-variables x et y ! Il n'y a pas beaucoup de fonctions à connaître pour gérer le temps en SDL. Par contre, bien savoir les utiliser est une autre paire de manches : il va falloir que vous pratiquiez pour vous sentir à l'aise.
Grâce à ces fonctions, on peut faire bouger automatiquement des personnages à l'écran (comme des ennemis) mais aussi gérer les FPS (nombre d'images par seconde) pour optimiser son programme. Bref, on peut faire plein de choses ! A vous de bien vous organiser

Je vous conseille vivement de vous entraîner : je vous ai déjà donné quelques idées d'exercices plus haut, inventez-en d'autres !
Vous pourriez par exemple créer **un jeu du snake** maintenant. Vous savez, ce petit serpent qui doit manger des pommes dans un labyrinthe sans se toucher lui-même
En fonction du délai entre 2 déplacements du serpent, la vitesse sera plus ou moins élevée !



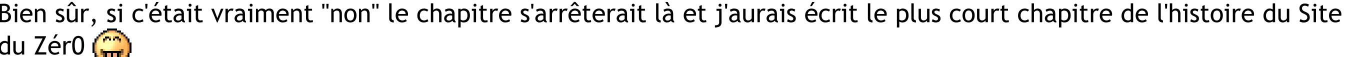
Aperçu d'un jeu de Snake

Bon codage

Ecrire du texte avec SDL_ttf

Je suis persuadé que la plupart d'entre vous se sont déjà posés cette question : "*Mais bon sang il n'y a aucune*

fonction pour écrire du texte dans une fenêtre SDL ?"

Il est temps de vous apporter la réponse : c'est **non**

Bien sûr, si c'était vraiment "non" le chapitre s'arrêterait là et j'aurais écrit le plus court chapitre de l'histoire du Site du Zér0
Alors qu'en est-il ?

En fait, la SDL ne propose vraiment aucune fonction pour écrire du texte dans la fenêtre, je ne vous ai pas menti.
Mais il y a quand même toujours moyen d'y arriver, il suffit d'utiliser... la ruse ! Et pour cela vous avez 2 solutions :

- Vous ne pouvez pas écrire de texte, mais vous avez le droit de **blitter des images**. On peut donc créer un bitmap pour chacun des lettres d'alphabet (de A à Z). Vous n'auriez qu'à coder une fonction qui assemblerait ces bitmaps dans une SDL_Surface en fonction du texte que vous lui envoyez en paramètre.
- Cependant, cette première solution est un peu lourde à mettre en place. Il y a plus simple : utiliser la librairie **SDL_ttf**. C'est une librairie qui vient s'ajouter par-dessus la SDL, tout comme SDL_image. Son rôle est de créer une SDL_Surface contenant le texte que vous lui envoyez.

Nous allons donc dans ce chapitre apprendre à manier SDL_ttf pour pouvoir écrire du texte dans la fenêtre

INSTALLER SDL_TTF

Il faut savoir que, comme SDL_image, SDL_ttf est une librairie qui nécessite que la SDL soit installée.
Bon, si à ce stade du cours vous n'avez toujours pas installé la SDL c'est grave, donc je vais supposer que ça c'est fait

Tout comme SDL_image, SDL_ttf est une des librairies liées à la SDL les plus populaires (c'est-à-dire qu'elle est très téléchargée). Comme vous allez pouvoir le constater, cette librairie est effectivement bien faite. Une fois que vous aurez appris à l'utiliser, vous ne pourrez plus vous en passer !

Comment fonctionne SDL_ttf ?

SDL_ttf n'utilise pas des images bitmap pour générer du texte dans des SDL_Surface. C'est une méthode en effet assez lourde à mettre en place et on n'aurait pu utiliser qu'une seule police.
En fait, SDL_ttf fait appel à une autre librairie : **Freetype**. C'est une librairie capable de lire les fichiers de police (`.ttf`) et d'en sortir l'image. SDL_ttf récupère donc cette "image" et la convertit pour la SDL en créant une SDL_Surface.

Point important : SDL_ttf a donc besoin de la librairie Freetype pour fonctionner, sinon elle ne sera pas capable de lire les fichiers `.ttf`.

- Si vous êtes sous **Windows** et que vous prenez, comme je le fais, la version "compilée" de la librairie, vous n'aurez pas besoin de télécharger quoi que ce soit de Freetype car cette librairie sera incluse dans la DLL SDL_ttf.dll. Bref, vous n'avez rien à faire.
- Si vous êtes sous **Linux ou Mac OS** et que vous devez recompiler la librairie, il vous faudra en revanche Freetype pour compiler. Rendez-vous sur **la page de téléchargement de Freetype** pour récupérer les fichiers pour développeurs.

Installer SDL_ttf

Rendez-vous sur **la page de téléchargement de SDL_ttf**.
Là, choisissez le fichier qu'il vous faut dans la section "Binary".

Sous Windows, vous remarquerez qu'il n'y a que deux fichiers zip ayant le suffixe "win32" et "VC6".
Le premier (win32) contient la DLL que vous aurez besoin de livrer avec votre exécutable. Vous aurez aussi besoin de mettre cette DLL dans le dossier de votre projet pour pouvoir tester votre programme évidemment
Le second (VC6) contient les .h et .lib dont vous allez avoir besoin pour programmer. On pourrait penser d'après le nom que ça n'est fait que pour Visual-C++. C'est vrai. Mais en fait, rassurez-vous, exceptionnellement le fichier .lib livré ici marche aussi avec mingw32, donc il fonctionnera sous Code::Blocks et Dev-C++.

Le fichier ZIP contient comme d'habitude un dossier include et un dossier lib. Mettez le contenu du dossier include dans mingw32/include/SDL et le contenu du dossier lib dans mingw32/lib (c'est un fichier .lib et non un .a, mais comme je vous ai dit exceptionnellement ce fichier marche avec le compilateur mingw).

Vous devez copier le fichier SDL_ttf.h dans le dossier mingw32/include/SDL et non pas dans mingw32/include tout court.
Attention aux erreurs !

Configurer un projet pour SDL_ttf

Eh bien c'est long hein !
Il nous reste une dernière petite chose à faire : **configurer notre projet** pour qu'il utilise bien SDL_ttf. Il va falloir modifier les options du linker pour qu'il compile bien votre programme en utilisant la librairie SDL_ttf.

Vous avez déjà appris à faire cette opération pour la SDL et pour SDL_image, je vais donc aller plus vite.
Comme je travaille sous Code::Blocks, je vais vous donner la procédure avec cet IDE. Ce n'est pas bien différent avec les autres IDE :

- Rendez-vous dans le menu Project / Build Options
- Dans l'onglet "Linker", cliquez sur le petit bouton "Add".
- Indiquez où se trouve le fichier SDL_ttf.lib (chez moi c'est dans C:\Program Files\CodeBlocks\mingw32\lib)
- On vous demande "Keep this as a relative path ?". Peu importe ce que vous répondez, ça marchera dans les deux cas. Je vous conseille quand même de répondre Non, car sinon votre projet ne fonctionnera plus si vous le déplacez de dossier.
- Validez en cliquant sur OK : c'est bon

Mais... On n'a pas besoin de linker avec la librairie Freetype ?

Non, car comme je vous l'ai dit Freetype est incluse dans la DLL de SDL_ttf. Vous n'avez pas à vous préoccuper de Freetype, c'est SDL_ttf qui gère ça maintenant

La documentation

Maintenant que vous commencez à devenir des programmeurs aguerris, vous devriez vous demander immédiatement : "*Mais où est la doc ?*"
Si vous ne vous êtes pas encore posé cette question, c'est que vous n'êtes pas encore un programmeur aguerri

Bien sûr, vous vous dites : "*Il y a toujours les tutos de tonton M@teo pour apprendre à s'en servir*". C'est vrai : je vais vous apprendre à vous en servir dans ce chapitre. Toutefois :

- Je ne vais pas faire un tuto pour toutes les librairies qui existent (même en y passant ma vie je n'aurais pas le temps). Il va donc falloir tôt ou tard lire une doc, et mieux vaut commencer à apprendre à le faire maintenant !
- D'autre part, une librairie est en général assez complexe et contient beaucoup de fonctions. Je ne peux pas présenter toutes ces fonctions dans un tuto, ce serait bien trop long !

En clair : une doc c'est complet mais un peu dur à comprendre quand on n'a pas l'habitude, un tuto c'est pas complet mais ça aide bien à démarrer, surtout quand on débute

Je vous conseille donc de mettre dans vos favoris l'adresse suivante :

http://jcatki.no-ip.org/SDL_ttf/

C'est l'adresse de la doc de SDL_ttf. Elle est disponible en plusieurs formats : HTML en ligne, HTML zippé, PDF etc.
Prenez la version qui vous arrange le plus

Vous verrez que SDL_ttf est une librairie très simple : il y a peu de fonctions.
Environ 40-50 fonctions, c'est peu (je vous dis pas combien il y en a dans la SDL ou dans des librairies plus complexes que nous étudierons plus tard comme QT !). Bref, ça devrait être signe (pour le programmeur aguerri que vous êtes) que cette librairie est simple et que vous saurez la manier assez vite.

Allez, il est temps d'apprendre à utiliser SDL_ttf maintenant

Chargement de SDL_ttf

L'include

Avant toute chose, il faut ajouter l'include suivant en haut de votre fichier .c :

Code : C

```
#include <SDL/SDL_ttf.h>
```

Si vous avez des erreurs de compilation à ce stade, vérifiez si vous avez bien placé le fichier SDL_ttf.h dans le dossier mingw32/include/SDL et non dans mingw32/include tout court.

Démarrage de SDL_ttf

Tout comme la SDL, SDL_ttf a besoin d'être démarrée et arrêtée.
Il y a donc des fonctions très similaires à la SDL :

- **TTF_Init** : démarre SDL_ttf.
- **TTF_Quit** : arrête SDL_ttf.

Il n'est pas nécessaire que la SDL soit démarrée avant SDL_ttf.

Pour démarrer SDL_ttf (on dit aussi "initialiser"), on doit donc appeler la fonction TTF_Init().
Aucun paramètre n'est nécessaire. La fonction renvoie -1 s'il y a eu une erreur.

Vous pouvez donc démarrer SDL_ttf très simplement comme ceci :

Code : C

```
TTF_Init();
```

(dur de faire plus simple avouez)

Si vous voulez vérifier s'il y a une erreur et être ainsi plus rigoureux, utilisez ce code à la place :

Code : C

```
if(TTF_Init() == -1)
{
    fprintf(stderr, "Erreur d'initialisation de TTF_Init : %s\n", TTF_GetError());
    exit(EXIT_FAILURE);
}
```

S'il y a eu une erreur au démarrage de SDL_ttf, un fichier stderr.txt sera créé (sous Windows du moins) contenant un message explicatif de l'erreur.
Pour ceux qui se poseraient la question : la fonction TTF_GetError() renvoie le dernier message d'erreur de SDL_ttf. C'est pour cela qu'on l'utilise dans le fprintf.

Arrêt de SDL_ttf

Pour arrêter SDL_ttf, on appelle TTF_Quit(). Là encore, pas de paramètre, pas de prise de tête
Vous pouvez appeler TTF_Quit avant ou après SDL_Quit, peu importe.

Code source (attention c'est du haut niveau !) :

Code : C

```
TTF_Quit();
```

Ca va vous suivez toujours ?

Chargement d'une police

Bon tout ça c'est bien beau mais c'est pas assez compliqué, c'est pas rigolo.
Passons aux choses sérieuses si vous le voulez bien !

Maintenant que SDL_ttf est chargée, nous devons charger une police.
Une fois que cela sera fait, nous pourrons enfin voir comment écrire du texte !

Là encore il y a 2 fonctions :

- **TTF_OpenFont** : ouvre un fichier de police (.ttf)
- **TTF_CloseFont** : ferme une police ouverte.

TTF_OpenFont doit stocker son résultat dans une variable de type TTF_Font. Vous devez créer un pointeur de TTF_Font, comme ceci :

Code : C

```
TTF_Font *police = NULL;
```

Le pointeur *police* contiendra donc les informations sur la police une fois qu'on l'aura ouverte.

La fonction TTF_OpenFont prend 2 paramètres :

- Le nom du fichier de police (au format .ttf) à ouvrir. L'idéal c'est de mettre le fichier de police dans le répertoire de votre projet. Exemple de fichier : `arial.ttf` (pour la police Arial).
- La taille de la police à utiliser. Vous pouvez par exemple utiliser une taille de 22.
  Ce sont les mêmes tailles que celles que vous avez dans un logiciel de texte comme Word.

Où trouver des polices .ttf ?

Sur votre ordinateur

Vous en avez déjà sur votre ordinateur !
Si vous êtes sous Windows, vous en trouverez déjà plein dans le dossier C:/Windows/Fonts
Vous n'avez qu'à copier le fichier de police qui vous plaît dans le dossier de votre projet.

Attention : si le nom contient des caractères "bizarres" comme des espaces, des accents ou même des majuscules, je vous conseille de le renommer. Pour être sûr de n'avoir aucun problème, n'utilisez que des minuscules.
Exemple de nom incorrect : TIMES NEW ROMAN.TTF
Exemple de nom correct : times.ttf

Sur Internet

Autre possibilité : récupérer une police sur Internet. Vous trouverez pas mal de sites proposant des polices gratuites et originales à télécharger.
Testez un coup de **Google "polices"**, vous allez voir le résultat. Il y a de quoi faire !

Je connais de nombreux bons sites, et si personnellement je devais n'en retenir qu'un ce serait **dafont.com**. C'est bien classé, très bien fourni et varié. Que demande le peuple

Voici un aperçu de polices que vous pourrez trouver très facilement là-bas :

C'est que du bonheur

Bon, retour à la programmation

Allez, on va utiliser la police **Angelina** elle me plaît bien

On ouvre la police comme ceci :

Code : C

```
police = TTF_OpenFont("angelina.ttf", 65);
```

La police utilisée sera angelina.ttf. J'ai bien pris soin de mettre le fichier dans le dossier de mon projet et de le renommer pour qu'il soit tout en minuscules.
La police sera de taille 65. Ca paraît gros mais visiblement c'est une police qu'il faut écrire en gros pour qu'on puisse la voir

Ce qui est très important, c'est que TTF_OpenFont stocke le résultat dans la variable police. Vous allez réutiliser cette variable tout à l'heure en écrivant du texte. Elle permettra d'indiquer la police que vous voulez utiliser pour écrire du texte.

Vous pouvez ouvrir plusieurs polices à la fois.
Vous n'avez pas besoin d'ouvrir une même police à chaque fois que vous écrivez du texte : ouvrez la police juste une fois au début du programme et fermez-la à la fin.

Fermer la police

Il faut penser à fermer chaque police ouverte avant l'appel à TTF_Quit().
Dans mon cas, ça donnera donc le code suivant :

Code : C

```
TTF_CloseFont(police); /* Doit être avant TTF_Quit() */
TTF_Quit();
```

Et voilà l'travail

Code source pour résumer

On résume tout ce qu'on vient d'apprendre sur le "chargement" de SDL_ttf à l'aide d'un petit code source complet histoire de se situer. Les lignes intéressantes sont commentées :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_ttf.h> /* Ne pas oublier l'include ! */

int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL;
    SDL_Event event;
    TTF_Font *police = NULL; /* Stockera les informations de police */
    int continuer = 1;



    SDL_Init(SDL_INIT_VIDEO);
    TTF_Init(); /* Initialisation (peut être avant ou après SDL_Init) */

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du texte avec SDL_ttf", NULL);

    police = TTF_OpenFont("angelina.ttf", 65); /* Ouverture de la police au début */


    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
        SDL_Flip(ecran);
    }

    TTF_CloseFont(police); /* Fermeture de la police avant TTF_Quit */
    TTF_Quit(); /* Arrêt de SDL_ttf (peut être avant ou après SDL_Quit, peu importe) */

    SDL_Quit();


    return EXIT_SUCCESS;
}
```

Les différentes méthodes d'écriture

Maintenant que SDL_ttf est chargée et qu'on a une variable *police* chargée elle aussi, plus rien ni personne ne nous empêchera d'écrire du texte dans notre fenêtre SDL !

Bon, écrire du texte c'est bien, mais avec quelle fonction ?
J'ai été un peu surpris la première fois que j'ai vu la doc de SDL_ttf : 12 fonctions pour écrire du texte, ça en fait du choix !

En fait, il y a 3 façons différentes pour SDL_ttf de dessiner du texte :

- **Solid** : c'est la technique la plus rapide. Le texte sera rapidement écrit dans une SDL_Surface. La surface sera transparente mais n'utilisera qu'un niveau de transparence (on a appris ça il y a quelques chapitres). C'est pratique, mais le texte ne sera pas très joli, pas très "arrondi" surtout s'il est écrit gros. Utilisez cette technique lorsque vous devez souvent changer le texte (par exemple pour afficher le temps qui s'écoule ou le nombre de FPS d'un jeu).
- **Shaded** : cette fois, le texte sera joli. Les lettres seront antialiasées, le texte apparaîtra plus lisse. Il y a un défaut par contre : le fond doit être d'une couleur unie. Pas moyen de rendre le fond de la SDL_Surface transparente en Shaded.
- **Blended** : c'est la technique la plus puissante, mais elle est lente. En fait, elle met autant de temps que Shaded à créer la SDL_Surface. La seule différence avec Shaded, c'est que vous pouvez blitter le texte sur une

image et la transparence sera respectée (contrairement à Shaded qui imposait un fond uni). Attention : le calcul du blit sera plus lent que pour Shaded.
C'est là que la différence de lenteur avec Shaded se fait : au moment du blit et non au moment de la création de la SDL_Surface.

J'ai mis un peu de temps à m'habituer à ces 3 types d'écriture du texte.
Pour vous aider à vous faire une idée, voici des screenshots d'un même texte écrit avec ces différentes techniques :

Solid
Mode d'écriture très rapide mais pas très beau (texte non lissé).

Shaded
Mode d'écriture lent mais plus joli car antialisé. Fond obligatoirement uni.

Blended
Mode d'écriture lent (et blit lent) mais très beau car antialisé et fonctionne sur un fond non uni.

En résumé :

- Si vous avez un texte qui change souvent, comme un compte à rebours, utilisez **Solid**.
- Si votre texte ne change pas très souvent et que vous voulez blitter votre texte sur un fond uni, utilisez **Shaded**.
- Si votre texte ne change pas très souvent mais que vous voulez blitter sur un fond non uni (comme une image), utilisez **Blended**.

Voilà, vous devriez déjà être un peu plus familier avec ces 3 types d'écriture de SDL_ttf.

Je vous avais dit qu'il y avait 12 fonctions en tout.
En effet, pour chacun de ces 3 types d'écriture, il y a 4 fonctions. Chaque fonction écrit le texte à l'aide d'un charset différent, c'est-à-dire d'une palette de caractères différentes. Il y en a 4 :

- Latin1
- UTF8
- Unicode
- Unicode Glyph

L'idéal est d'utiliser l'Unicode car c'est un charset gérant la quasi-totalité des caractères existant sur Terre (eh ouais ça en fait avec toutes les langues).
Toutefois, utiliser l'Unicode n'est pas toujours forcément simple (un caractère prend plus que la taille d'un char en mémoire), nous ne verrons pas comment l'utiliser ici.

A priori, si votre programme est écrit en français le mode Latin1 suffit amplement, vous pouvez vous contenter de celui-là

Les 3 fonctions utilisant le charset Latin1 sont :

- TTF_RenderText_Solid
- TTF_RenderText_Shaded
- TTF_RenderText_Blended

Vous savez tout, ou presque
Nous allons voir comment écrire un texte en Blended (c'est le plus joli)

Exemple d'écriture de texte en Blended

Pour spécifier une couleur à SDL_ttf, on ne va pas utiliser le même type qu'avec la SDL (un Uint32 créé à l'aide de la fonction SDL_MapRGB).
Au contraire, nous allons utiliser une structure toute prête de la SDL : SDL_Color. Cette structure comporte 3 sous-variables : la quantité de rouge, de vert et de bleu.

Si vous voulez créer une variable *couleurNoire*, vous devrez donc écrire :

Code : C

```
SDL_Color couleurNoire = {0, 0, 0};
```

Attention à ne pas confondre avec les couleurs qu'utilise habituellement la SDL !
La SDL utilise des Uint32 créés à l'aide de SDL_MapRGB.
SDL_ttf utilise des SDL_Color.

On va écrire un texte en noir dans une SDL_Surface *texte* :

Code : C

```
texte = TTF_RenderText_Blended(police, "Salut les Zér0s !", couleurNoire);
```

Vous voyez dans l'ordre les paramètres à envoyer : la police (de type TTF_Font), le texte à écrire, et enfin la couleur (de type SDL_Color).
Le résultat est stocké dans une SDL_Surface. SDL_ttf calcule automatiquement la taille nécessaire à donner à la surface en fonction de la taille du texte et du nombre de caractères que vous avez voulu écrire.

Comme toute SDL_Surface, notre pointeur texte contient les sous-variables *w* et *h* indiquant respectivement sa largeur et sa hauteur.
C'est donc un bon moyen de connaître les dimensions du texte une fois que celui-ci a été écrit dans la SDL_Surface. Vous n'aurez qu'à faire :

Code : C

```
texte->w /* Donne la largeur */
texte->h /* Donne la hauteur */
```

Code complet d'écriture de texte

Voilà vous savez tout
Voici un code complet montrant l'écriture de texte en mode Blended :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
#include <SDL/SDL_ttf.h>

int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *texte = NULL, *fond = NULL;
    SDL_Rect position;
    SDL_Event event;
    TTF_Font *police = NULL;
    SDL_Color couleurNoire = {0, 0, 0};
    int continuer = 1;


    SDL_Init(SDL_INIT_VIDEO);
    TTF_Init();

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du texte avec SDL_ttf", NULL);

    fond = IMG_Load("moraira.jpg");

    /* Chargement de la police */
    police = TTF_OpenFont("angelina.ttf", 65);
    /* Ecriture du texte dans la SDL_Surface "texte" en mode Blended (optimal) */
    texte = TTF_RenderText_Blended(police, "Salut les Zér0s !", couleurNoire);

    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));

        position.x = 0;
        position.y = 0;
        SDL_BlitSurface(fond, NULL, ecran, &position); /* Blit du fond */

        position.x = 60;
        position.y = 370;
        SDL_BlitSurface(texte, NULL, ecran, &position); /* Blit du texte par-dessus */
        SDL_Flip(ecran);
    }

    TTF_CloseFont(police);
    TTF_Quit();

    SDL_FreeSurface(texte);
    SDL_Quit();


    return EXIT_SUCCESS;
}
```

Et le résultat :

Sympa n'est-ce pas ?

Si vous voulez changer de mode d'écriture pour tester, il n'y a qu'une ligne à changer : celle créant la surface (avec l'appel à la fonction *TTF_RenderText_Blended*).

La fonction TTF_RenderText_Shaded prend un 4ème paramètre contrairement aux 2 autres. Ce dernier paramètre est la couleur de fond à utiliser. Vous devrez donc créer une autre variable de type SDL_Color pour indiquer une couleur de fond (par exemple le blanc).

Attributs d'écriture du texte

Il est possible aussi de spécifier des attributs d'écriture, comme gras, italique et souligné.

Il faut d'abord que la police soit chargée. Vous devriez donc avoir une variable *police* valide.
Vous pouvez alors faire appel à la fonction **TTF_SetFontStyle** qui va modifier la police pour qu'elle soit en gras, italique ou souligné selon vos désirs.

La fonction prend 2 paramètres :

- La police à modifier
- Une combinaison de flags pour indiquer le style à donner : gras, italique ou souligné.

Pour les flags, vous devez utiliser ces constantes :

- **TTF_STYLE_NORMAL** : normal.
- **TTF_STYLE_BOLD** : gras.
- **TTF_STYLE_ITALIC** : italique.
- **TTF_STYLE_UNDERLINE** : souligné.

Comme c'est une liste de flags, vous pouvez les combiner à l'aide du symbole | comme on a appris à le faire.

Testons :

Code : C

```
/* Chargement de la police */
police = TTF_OpenFont("angelina.ttf", 65);
/* Le texte sera écrit en italique et souligné */
TTF_SetFontStyle(police, TTF_STYLE_ITALIC | TTF_STYLE_UNDERLINE);
/* Ecriture du texte en italique et souligné */
texte = TTF_RenderText_Blended(police, "Salut les Zér0s !", couleurNoire);
```

Résultat, le texte est écrit en italique et souligné :

Pour restaurer une police à son état normal, il suffit de refaire appel à TTF_SetFontStyle en utilisant cette fois le flag TTF_STYLE_NORMAL.

Exercice : le compteur

Cet exercice va cumuler ce que vous avez appris dans ce chapitre et dans le chapitre sur la gestion du temps.
Votre mission, si vous l'acceptez, consistera à créer un compteur qui s'incrémentera tous les dixièmes de seconde.

Ce compteur va donc progressivement afficher :
0
100
200
300
400
etc.

Au bout d'une seconde je veux voir affiché 1000, au bout d'une seconde et demie je veux voir affiché 1500 etc.

Astuce pour écrire dans une chaîne

Pour réaliser cet exercice, vous aurez besoin de savoir comment écrire dans une chaîne de caractères en mémoire.
En effet, vous devez donner un char* à TTF_RenderText mais vous ce que vous aurez c'est un nombre (un int par exemple). Comment convertir un nombre en chaîne de caractères ?

On peut utiliser la fonction **sprintf**.
Elle marche de la même manière que fprintf, sauf qu'au lieu d'écrire dans un fichier elle écrit dans une chaîne (le "s" de sprintf signifie "string", c'est-à-dire "chaîne" en anglais).
Le premier paramètre que vous lui donnerez sera donc un pointeur sur un tableau de char.

Veillez à réserver suffisamment d'espace pour le tableau de char si vous ne voulez pas déborder en mémoire !

Exemple :

Code : C

```
sprintf(temps, "Temps : %d", compteur);
```

Ici, *temps* est un tableau de char (20 caractères), et *compteur* est un int qui contient le temps.
Après cette instruction, la chaîne *temps* contiendra par exemple "Temps : 500"

Allez au boulot !

Correction

Voici une correction possible de l'exercice :

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *texte = NULL;
    SDL_Rect position;
    SDL_Event event;
    TTF_Font *police = NULL;
    SDL_Color couleurNoire = {0, 0, 0}, couleurBlanche = {255, 255, 255};
    int continuer = 1;
    int tempsActuel = 0, tempsPrecedent = 0, compteur = 0;
    char temps[20] = ""; /* Tableau de char suffisamment grand */


    SDL_Init(SDL_INIT_VIDEO);
    TTF_Init();

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du texte avec SDL_ttf", NULL);


    /* Chargement de la police */
    police = TTF_OpenFont("angelina.ttf", 65);

    /* Initialisation du temps et du texte */
    tempsActuel = SDL_GetTicks();
    sprintf(temps, "Temps : %d", compteur);
    texte = TTF_RenderText_Shaded(police, temps, couleurNoire, couleurBlanche);

    while (continuer)
    {
        SDL_PollEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));


        tempsActuel = SDL_GetTicks();
        if (tempsActuel - tempsPrecedent >= 100) /* Si 100ms au moins se sont écoulées */
        {
            compteur += 100; /* On rajoute 100ms au compteur */
            sprintf(temps, "Temps : %d", compteur); /* On écrit dans la chaîne "temps" le
nouveau temps */
            SDL_FreeSurface(texte); /* On supprime la surface précédente de la mémoire
avant d'en charger une nouvelle (IMPORTANT) */
            texte = TTF_RenderText_Shaded(police, temps, couleurNoire, couleurBlanche);
/* On écrit la chaine temps dans la SDL_Surface */
            tempsPrecedent = tempsActuel; /* On met à jour le tempsPrecedent */
        }

        position.x = 180;
        position.y = 210;
        SDL_BlitSurface(texte, NULL, ecran, &position); /* Blit du texte contenant le
temps */
        SDL_Flip(ecran);

    }

    TTF_CloseFont(police);
    TTF_Quit();

    SDL_FreeSurface(texte);
    SDL_Quit();


    return EXIT_SUCCESS;
}
```

Voici ce que ça donne au bout de 13,9 secondes très exactement

Gestion du texte avec SDL_ttf

Temps : 13900

Télécharger le projet (437 Ko)

Ce n'est pas parfait, on pourrait par exemple utiliser SDL_Delay pour éviter d'utiliser 100% du CPU.

Pour aller plus loin

Si vous voulez améliorer ce petit bout de programme, vous pouvez essayer d'en faire un jeu où il faut cliquer le plus de fois possible dans la fenêtre avec la souris dans un temps imparti. Un compteur s'incrémentera à chaque clic de la souris.
Un compte à rebours doit s'afficher. Lorsqu'il atteint 0, on récapitule le nombre de clics effectués et on demande si on veut faire une nouvelle partie.

Vous pouvez aussi gérer les meilleurs scores en les enregistrant dans un fichier. Ca vous fera travailler à nouveau la gestion des fichiers en C

Bon courage ! Vous savez maintenant tout ce qu'il faut pour écrire du texte dans une fenêtre SDL

Mine de rien, vous commencez à en savoir beaucoup sur la SDL. Nous aurons bientôt fini d'en faire le tour.
Nous devons toutefois apprendre à gérer le son (nos programmes sont muets pour le moment) et faire encore quelques TP pour bien comprendre comment on s'y prend pour créer des jeux en C.

Lorsque nous en aurons fini avec la SDL, nous passerons au C++. En effet, nous n'avons fait que du C jusqu'ici, et nous ne connaissons toujours rien du langage C++ ! Nous y viendrons donc dans quelques chapitres, lorsque la partie III sur la SDL sera terminée.

Jouer du son avec FMOD

Depuis le début de la partie III, nous avons appris à placer des images dans la fenêtre, à faire interagir l'utilisateur par tous les moyens possibles et imaginables (clavier, souris, joystick...), à écrire du texte, mais il manque clairement un élément : **le son** !

Ce chapitre va combler ce manque. Nous allons découvrir une librairie spécialisée dans le son : **FMOD**.

Faites chauffer les enceintes, c'est DJ M@teo qui mixe ce soir

INSTALLER FMOD

Pourquoi FMOD ?

Vous le savez maintenant : la SDL n'est pas seulement une librairie graphique. Elle permet aussi de gérer le son via un module appelé *SDL_audio*. Alors que vient faire une librairie externe qui n'a rien à voir comme FMOD dans ce chapitre ?

C'est en fait un choix que j'ai fait après de nombreux tests. J'aurais pu vous expliquer comment gérer le son en SDL mais j'ai préféré ne pas le faire.
Je m'explique.

Pourquoi j'ai évité SDL_audio

La gestion du son en SDL est "bas niveau". Trop à mon goût. Il faut effectuer plusieurs manipulations très précises pour jouer du son. C'est donc complexe et je ne trouve pas ça amusant. Il y a bien d'autres librairies qui proposent de jouer du son simplement.

Petit rappel : une librairie "bas niveau" est une librairie proche de l'ordinateur. On doit donc connaître un peu le fonctionnement interne de l'ordinateur pour s'en servir et il faut généralement plus de temps pour arriver à faire la même chose qu'avec une librairie "haut niveau".
N'oubliez pas que tout est relatif : il n'y a pas les librairies bas niveau d'un côté et les librairies haut niveau de l'autre. Certaines sont juste plus ou moins haut niveau que d'autres. Par exemple FMOD est plus haut niveau que le module SDL_audio de la SDL.

Autre détail important, la SDL ne permet de jouer que des sons au format WAV. Le format WAV est un format de son non compressé. Une musique de 3 minutes dans ce format prend plusieurs dizaines de Mo, contrairement à un format compressé comme MP3 ou Ogg qui occupe beaucoup moins d'espace (2-3 Mo).

En fait, si on y réfléchit bien, c'était un peu pareil avec les images. La SDL ne gère que les BMP (images non compressées) à la base. On a dû installer une librairie supplémentaire (SDL_image) pour pouvoir lire d'autres images comme les JPEG, PNG, GIF, etc.

Eh bien figurez-vous qu'il y a une librairie équivalente pour le son : SDL_mixer. Elle est capable de lire un grand nombre de formats audio, parmi lesquels les MP3, les Ogg, les Midi... Et pourtant, j'ai évité de vous parler de cette librairie là encore. Pourquoi ?

Pourquoi j'ai évité SDL_mixer

SDL_mixer est une librairie qu'on ajoute en plus de la SDL à la manière de SDL_image. Elle est simple à utiliser et lit beaucoup de formats audio différents. Toutefois, après mes tests, il s'est avéré que la librairie comportait des bugs gênants en plus d'être relativement limitée en fonctionnalités.

C'est donc pour cela que je me suis ensuite penché sur FMOD, une librairie qui n'a certes rien à voir avec la SDL, mais

qui a l'avantage d'être puissante et réputée.

Télécharger FMOD

Si je vous raconte tout ça, c'est pour vous expliquer que le choix de FMOD n'est pas anodin. C'est tout simplement parce que c'est la meilleure librairie gratuite que j'ai pu trouver.
Elle est aussi simple à utiliser que SDL_mixer, avec un avantage non négligeable : elle n'est pas buggée

FMOD permet en outre de réaliser plusieurs effets intéressants que SDL_mixer ne propose pas, comme des effets sonores 3D.

Pour télécharger la librairie, il vous faut l'adresse du site de FMOD. La voici :

http://www.fmod.org

FMOD est une librairie gratuite mais pas sous license LGPL, contrairement à la SDL. Cela signifie que vous pouvez l'utiliser gratuitement du temps que vous ne réalisez pas de programme payant avec. Si vous voulez faire payer votre programme, il faudra payer une redevance à l'auteur (je vous laisse consulter les prix sur le site de FMOD).

Rendez-vous sur la page des téléchargements (Downloads) et prenez FMOD 3.
Attention, il y a plusieurs versions de FMOD, en particulier une plus récente et appelée FMOD Ex (il s'agit en fait de FMOD 4). Toutefois, FMOD Ex est une librairie C++, contrairement à FMOD 3 qui est une librairie C.

FMOD Ex, bien qu'écrite en C++, peut être utilisée en C en faisant quelques légères manipulations. Ce n'est toutefois pas la librairie que nous utiliserons ici. FMOD 3 est toujours supportée par ses créateurs et suffit amplement à nos besoins.

Téléchargez donc FMOD 3, il s'agit de la section "FMOD 3.75 Programmers API" sur la page Downloads (il peut s'agir d'une version plus récente si FMOD 3 a évolué). Prenez la version correspondant à votre système d'exploitation : Windows 32 / 64 bits si vous êtes sous Windows.
Vous remarquerez au passage que FMOD fonctionne sur un très grand nombre de plateformes, en particulier sur des consoles de jeux comme la XBOX, la PS2, la PS3, la PSP, la Wii, etc. Cette librairie a été utilisée par plusieurs jeux professionnels parus sur console, ça devrait vous rassurer quant à sa qualité

Installer FMOD

Le fichier que vous avez téléchargé est un fichier ZIP (si c'est un exécutable c'est que vous avez pris FMOD Ex et non FMOD 3 !). L'installation se déroule de la même manière qu'avec les autres librairies qu'on a vues jusqu'ici.

1. Vous avez un dossier "Api" avec la DLL de FMOD (fmod.dll) à placer dans le répertoire de votre projet.

   fmod64.dll, que vous trouverez dans le même dossier, est la DLL adaptée aux systèmes

d'exploitation 64 bits, pour ceux qui ont des processeurs 64 bits.
Si vous n'êtes pas sûr de la DLL à prendre, utilisez **fmod.dll**, il y a 99% de chances pour que ce soit la bonne. Si vous êtes sous Windows XP par exemple, c'est que vous n'êtes pas sous un OS 64 bits, vous pouvez donc prendre **fmod.dll**.

2. Dans le dossier "api/inc", vous trouverez les .h. Placez-les à côté des autres .h dans le dossier de votre IDE. Par exemple "Code Blocks/mingw32/include/FMOD" (j'ai créé un dossier spécial pour FMOD comme pour SDL).
3. Dans le dossier "api/lib", récupérez le fichier qui correspond à votre compilateur.
   - Si vous utilisez Code Blocks ou Dev-C++, donc le compilateur mingw, copiez "libfmod.a" dans le dossier "lib" de votre IDE.
     Dans le cas de Code Blocks, c'est le dossier "CodeBlocks/mingw32/lib".
   - Si vous utilisez Visual C++, récupérez le fichier "fmodvc.lib"
4. Enfin, et c'est peut-être le plus important, dans le dossier "documentation" vous trouverez un fichier d'aide qui correspond à la documentation de FMOD. Je vous conseille de le placer quelque part sur votre disque dur et d'en faire un raccourci bien visible, par exemple dans le menu Démarrer.
   En effet, la documentation liste toutes les possibilités de FMOD dans le détail. Dans ce chapitre, on ne pourra voir que les principales. Si vous voulez allez plus loin, il faudra donc vous plonger dans cette doc

La doc est aussi accessible en ligne à l'adresse : **http://www.fmod.org/docs.**

Il reste à configurer notre projet. Là encore, c'est comme les autres fois : vous ouvrez votre projet avec votre IDE favori et vous ajoutez le fichier .a (ou .lib) à la liste des fichiers que le linker doit récupérer.
Sous Code Blocks (j'ai l'impression de me répéter), menu *Project / Build Options*, onglet *Linker*, cliquez sur "Add" et indiquez où se trouve le fichier .a. Si on vous demande "Keep as a relative path ?", je vous conseille de répondre non, mais dans les deux cas de toute manière ça devrait marcher.

Voilà qui est bien
FMOD est installé, voyons voir rapidement de quoi il est constitué.

Les différentes sections de FMOD

FMOD 3 est en fait la combinaison de 2 librairies :

- **FSOUND** : cette partie gère tous les sons de type PCM. Il s'agit tout simplement de sons "réels" enregistrés, cela comprend aussi bien les formats compressés que non compressés : WAV, MP3, OGG, etc. Ces sons peuvent être des musiques ou des courts son comme un bruit de pas, un bruit de balle... D'ailleurs, FMOD distingue 2 types de sons :
  - **Les sons courts** (bruit de pas, bruit de balle) destinés à être répétés souvent.
  - **Les sons longs**, comme une musique qui dure 3 minutes par exemple (ça peut être la musique de fond de votre jeu).
- **FMUSIC** : cette section gère les musiques au format binaire. Cette fois, il n'y a pas de son enregistré, juste les notes de musique. Le format binaire le plus connu est probablement le MIDI. Vous savez probablement que les MIDI sont des fichiers audio très petits : c'est justement parce qu'ils enregistrent seulement les notes de musique (il ne peut donc pas y avoir de "paroles" avec un tel format de fichier). Cette section peut être très utile pour jouer des vieilles musiques type Gameboy ou SuperNES, comme par exemple la musique de Super Mario, de Tetris, etc.

Dans ce chapitre, nous verrons les trois types de sons car ils se chargent et se lisent avec des fonctions différentes :

- FSOUND : sons courts
- FSOUND : sons longs (musiques)
- FMUSIC : musique type MIDI

INITIALISER ET LIBÉRER FMOD

Comme la plupart des librairies écrites en C, il faut charger FMOD (on dit aussi "initialiser") et le libérer quand on n'en a plus besoin. Ca ne devrait pas beaucoup vous changer de la SDL à ce niveau

Inclure le header

Avant toute chose, vous avez dû le faire instinctivement maintenant mais ça ne coûte rien de le préciser, il faut inclure le fichier .h de FMOD.

Code : C

```
#include <FMOD/fmod.h>
```

J'ai placé ce fichier dans un sous-dossier "FMOD". Adaptez cette ligne en fonction de la position du fichier si chez vous c'est différent.

Initialiser FMOD

On initialise FMOD avec la fonction *FSOUND_Init*.
Elle prend 3 paramètres :

- **La fréquence d'échantillonnage** : comptez pas sur moi pour vous faire un cours de physique ici
  En gros, ce paramètre permet d'indiquer la qualité de son que doit gérer FMOD. Plus la fréquence est élevée, meilleur est le son (mais la puissance demandée est plus grande). Voici quelques exemples de fréquences pour vous aider à faire votre choix :
  - Qualité CD : 44 100 Hz
  - Qualité radio : 22 050 Hz
  - Qualité téléphonique : 11 025 Hz

  Dans tout ce chapitre, nous utiliserons une fréquence de 44 100 Hz.

  Si le son que vous utilisez est de mauvaise qualité à la base, FMOD ne l'améliorera pas. Par contre, si vous avez un son de fréquence 44 100 Hz et que FMOD utilise une fréquence de 22 050 Hz, sa qualité sera diminuée.
- **Le nombre maximal de canaux** que devra gérer FMOD. En d'autres termes, c'est le nombre maximal de sons qui pourront être joués en même temps. Tout dépend de la puissance de votre carte son. On conseille généralement une valeur de 32 (ce sera suffisant pour la plupart des petits jeux). Pour info, FMOD peut théoriquement gérer jusqu'à 1024 canaux différents, mais à ce niveau ça risque de beaucoup faire travailler votre ordinateur !
- Enfin, on peut indiquer **des flags**. Il n'y a rien de bien intéressant à mettre en général, donc on se contentera d'envoyer 0 (pas de flags).

Nous pouvons donc initialiser FMOD comme ceci :

Code : C

```
FSOUND_Init(44100, 32, 0);
```

Ce qui signifie : fréquence de 44 100 Hz (qualité CD au mieux), 32 canaux et pas d'options particulières (flag = 0).

Libérer FMOD

On arrête FMOD de la manière la plus simple qui soit :

Code : C

```c
FSOUND_Close();
```

Est-ce que j'ai vraiment besoin de commenter ce code ?

Les sons courts

Nous commencerons par étudier les sons courts.
Un "son court", comme je l'appelle, est un son qui dure généralement quelques secondes (parfois moins d'une seconde) et qui est généralement destiné à être joué régulièrement.

Quelques exemples de sons courts :

- Un bruit de balle
- Un bruit de pas
- Un tic-tac (pour faire stresser le joueur avant la fin d'un compte à rebours)
- Des applaudissements
- etc etc

Bref, ça correspond à tous les sons qui ne sont pas des musiques.
Généralement, ces sons sont tellements courts qu'on ne prend pas la peine de les compresser. On les trouve donc le plus souvent au format WAV non compressé.

Trouver des sons courts

Avant de commencer, il serait bien de connaître quelques sites qui proposent des banques de sons. En effet, tout le monde ne veut pas forcément enregistrer les sons chez soi (pour les bruits de balle par exemple, vous avez intérêt à être armé !).

Ca tombe bien, le net regorge de sons courts, généralement au format WAV.
Où les trouver ? Ca peut paraître bête, on n'y pense pas forcément (et pourtant on devrait), mais Google est notre ami. Au hasard, je tape "**Free Sounds**", ce qui signifie "sons gratuits" en anglais, et boum... des centaines de millions de résultats !

Rien qu'avec les sites de la première page, vous devriez trouver votre bonheur.
Personnellement, j'ai retenu **FindSounds.com**, un moteur de recherche pour sons. Je ne sais pas si c'est le meilleur, mais en tout cas il est bien complet.

Si vous ne savez pas quels mots-clés taper pour votre recherche, rendez-vous sur la page des **exemples de recherche.**
Certes, il faut connaître quelques mots d'anglais (mais si vous voulez programmer de toute façon, comment voulez-vous faire sans être au moins capable de lire l'anglais ?).

En recherchant "gun", on trouve des tonnes de sons de tir de fusil, en tapant "footsteps" on trouve des bruits de pas, etc etc.

FindSounds - Search Results - Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

http://www.findsounds.com/ISAPI/search.dll?keywords=door

FindSounds
Search the Web for Sounds

Search for door Search Help

Need Examples?

File Formats	Number of Channels	Minimum Resolution	Minimum Sample Rate	Maximum File Size
AIFF	mono	8-bit	8000 Hz	2 MB
AU	stereo			
WAVE				

Sounds 1-10 of 200 labelled "door"

1. http://sep800.mine.nu/files/sounds/jaildoorclose2.wav
close jail door
8k, mono, 8-bit, 8000 Hz, 1.1 seconds (show page | e-mail this sound)

2. http://www.littlemusicclub.com/doors/DoorOpen.WAV
open door
31k, mono, 16-bit, 11025 Hz, 1.4 seconds (show page | e-mail this sound)

3. http://users2.fdn.com/~pkjax/Blindtrails/Sounds/DOORCLOS.WAV
close door
21k, mono, 16-bit, 22050 Hz, 0.5 seconds (show page | e-mail this sound)

4. http://www.master-of-web.net/klopfen.wav
door knocks
62k, mono, 16-bit, 11025 Hz, 2.9 seconds (show page | e-mail this sound)

Terminé GP Adbloc

Le site FindSounds.com, un moteur de recherche de sons courts (ici, des bruits de porte)

Bonne pioche et bons téléchargements

Les étapes à suivre pour jouer un son

La première étape consiste à charger en mémoire le son que vous voulez jouer.
Il est conseillé de charger tous les sons qui seront fréquemment utilisés dans le jeu dès le début du programme. Vous les libérerez à la fin. En effet, une fois que le son est chargé en mémoire, sa lecture est très rapide.

Le pointeur

Première étape : créer un pointeur de type FSOUND_SAMPLE qui représentera notre son.

Code : C

```
FSOUND_SAMPLE *tir = NULL;
```

Charger le son

Deuxième étape : charger le son avec la fonction *FSOUND_Sample_Load*. Elle prend... 5 paramètres :

- Le **numéro de la *sample pool*** dans laquelle FMOD doit garder une trace du son. Je m'explique
  La *sample pool*, que j'ai pas trop essayé de traduire mais on pourrait dire que c'est la "piscine à samples", c'est une sorte de tableau dans lequel FMOD garde une copie des pointeurs vers chacun des sons courts chargés. Cela lui permet de libérer automatiquement la mémoire lorsqu'on appelle *FSOUND_Close()* (la fonction d'arrêt de FMOD) : il suffit à FMOD de lire ce tableau et de libérer chacun des éléments qui s'y trouvent. Toutefois, plutôt que de faire confiance à FMOD, il est mieux de penser à appeler nous-mêmes la fonction de libération de mémoire (*FSOUND_Sample_Free()*) que nous allons découvrir dans quelques instants.

  Bref, tout ce blabla pour dire que ce paramètre sert au fonctionnement interne de FMOD pour qu'il s'assure de libérer tous les sons à la fin au cas où vous auriez oublié de le faire.
  Pour indiquer un numéro de la sample pool, le mieux est d'envoyer FSOUND_FREE à la fonction. Elle se chargera alors de prendre le premier emplacement libre de la sample pool qu'elle trouvera. Ne vous posez donc pas trop de questions ce n'est pas bien utile ici, envoyez juste FSOUND_FREE
- Le **nom du fichier** son à charger. Il peut être de format WAV, MP3, OGG, etc. Toutefois, il vaut mieux charger des sons courts (quelques secondes maximum) plutôt que des sons longs. En effet, la fonction chargera et décodera tout le son en mémoire, ce qui peut prendre de la place si le son est une musique !
- Le troisième paramètre ne nous intéresse pas, il permet de préciser **les caractéristiques du fichier** qu'on veut charger (fréquence d'échantillonnage, etc.). Or, dans le cas des WAV, MP3, OGG et cie, ces informations sont inscrites dans le fichier. On va donc envoyer la valeur 0 pour ne rien préciser.
- **L'offset** où doit commencer la lecture. Cela permet de commencer la lecture du son à un moment précis. Mettez 0 pour commencer du début.
- **La longueur** : si vous précisez un offset, il faudra aussi donner la longueur de son à lire. On mettra là encore 0 car on veut tout lire.

La fonction renvoie l'adresse mémoire où a été chargé le son.
Voici un exemple de chargement :

Code : C

```
tir = FSOUND_Sample_Load(FSOUND_FREE, "pan.wav", 0, 0, 0);
```

Ici, je charge le son "pan.wav" et je le place dans le premier canal libre. Le pointeur *tir* fera référence à ce son par la suite.
Vous remarquerez qu'en règle générale on laisse les 3 derniers paramètres à 0.

Si vous voulez faire les tests en même temps que moi (et si vous ne voulez pas vous devriez), voici le fichier son **pan.wav** sur lequel nous allons travailler par la suite.

La fonction renvoie NULL si le fichier n'a pas été chargé. Vous avez tout intérêt à vérifier si le chargement a réussi ou s'il a échoué.

Jouer le son

Vous voulez jouer le son ? Pas de problème avec la fonction *FSOUND_PlaySound* !
Il suffit de lui donner un numéro de canal sur lequel jouer ainsi que le pointeur sur le son. Pour le numéro de canal, ne vous prenez pas la tête et envoyez FSOUND_FREE pour laisser FMOD gérer ça

Code : C

```
FSOUND_PlaySound(FSOUND_FREE, tir);
```

Libérer le son de la mémoire

Lorsque vous n'avez plus besoin du son, vous devez le libérer de la mémoire.
Il n'y a rien de plus simple, il suffit d'indiquer le pointeur à libérer avec la fonction *FSOUND_Sample_Free*

Code : C

```
FSOUND_Sample_Free(tir);
```

Exemple : un jeu de tir

Le mieux maintenant est de résumer tout ce qu'on a vu dans un cas concret de programme écrit en SDL.
Il n'y avait rien de compliqué et, normalement, vous ne devriez avoir aucune difficulté à réaliser cet exercice.

Le sujet

Votre mission est simple : créer un jeu de tir.
Bon, on ne va pas réaliser un jeu complet ici, mais déjà juste la gestion du viseur. Je vous ai justement fait un petit viseur sous Paint

Oui je sais...
J'aurais dû faire les Beaux Arts

Bref, voilà les objectifs :

- Fond de fenêtre : noir.
- Pointeur de la souris : invisible.
- L'image du viseur est blittée à la position de la souris lorsqu'on la déplace. Attention : il faut que le CENTRE de l'image soit placé au niveau du pointeur de la souris.
- Quand on clique, le son pan.wav doit être joué.

Ca peut être le début d'un jeu de tir.
Trop facile ? Ok alors à vous de jouer

La correction

Voici le code complet :

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
#include <FMOD/fmod.h>

int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *viseur = NULL;
    SDL_Event event;
    SDL_Rect position;
    int continuer = 1;
    FSOUND_SAMPLE *tir = NULL;

    /* Initialisation de FMOD */
    FSOUND_Init(44100, 32, 0);

    /* Chargement du son et vérification du chargement */
    tir = FSOUND_Sample_Load(FSOUND_FREE, "pan.wav", 0, 0, 0);
    if (tir == NULL)
    {
        fprintf(stderr, "Impossible de lire pan.wav\n");
        exit(EXIT_FAILURE);
    }

    /* Initialisation de la SDL */
    SDL_Init(SDL_INIT_VIDEO);

    SDL_ShowCursor(SDL_DISABLE);
    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du son avec FMOD", NULL);

    viseur = IMG_Load("viseur.png");


    while (continuer)
    {
        SDL_WaitEvent(&event);


        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
            case SDL_MOUSEBUTTONDOWN:
                /* Lorqu'on clique, on joue le son */
                FSOUND_PlaySound(FSOUND_FREE, tir);
                break;
            case SDL_MOUSEMOTION:
                /* Lorsqu'on déplace la souris, on place le centre du viseur à la
position de la souris
                ... D'où notamment le "viseur->w / 2" pour réussir à faire cela */
                position.x = event.motion.x - (viseur->w / 2);
                position.y = event.motion.y - (viseur->h / 2);
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));
        SDL_BlitSurface(viseur, NULL, ecran, &position);
        SDL_Flip(ecran);
    }

    /* On ferme la SDL */
    SDL_Quit();

    /* On libère le son et on ferme FMOD */
    FSOUND_Sample_Free(tir);
    FSOUND_Close();

    return EXIT_SUCCESS;
}
```

Aperçu du mini-jeu :

Le jeu avec le viseur. Quand on clique avec la souris, on entend "PAN !"

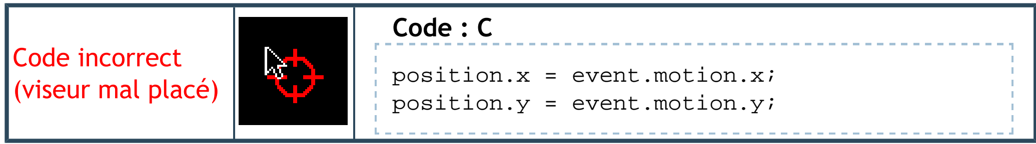
Le mieux est encore de voir le résultat en vidéo avec le son !

Gestion du son avec FMOD : le viseur (111 Ko)

Ici, j'ai chargé FMOD avant la SDL et je l'ai libéré après la SDL. Il n'y a pas de règles au niveau de l'ordre (j'aurais tout aussi bien pu faire l'inverse). J'ai choisi de charger la SDL et d'ouvrir la fenêtre après le chargement de FMOD pour que le jeu soit prêt à être utilisé dès que la fenêtre s'ouvre (sinon il aurait peut-être fallu attendre quelques millisecondes le temps que FMOD se charge).
Bref, vous faites comme vous voulez c'est un peu du détail ça de toute manière

Le code est, je pense, suffisamment commenté. Il n'y a pas de piège particulier, pas de nouveauté fracassante.

On notera la "petite" difficulté qui consistait à blitter le centre du viseur au niveau du pointeur de la souris. Le calcul de la position de l'image est fait en fonction.
Je vous fais un petit tableau pour ceux qui n'auraient pas encore compris la différence. Pour l'occasion, j'ai réactivé l'affichage du pointeur de la souris pour qu'on voie comment est placé le viseur par rapport au pointeur.



Code incorrect (viseur mal placé)		**Code : C**
		`position.x = event.motion.x;`
		`position.y = event.motion.y;`

Code correct (viseur bien placé)		Code : C
		`position.x = event.motion.x - (viseur->w / 2);` `position.y = event.motion.y - (viseur->h / 2);`

Idées d'amélioration

Ce code est la base d'un jeu de shoot. Vous avez le viseur, le bruit de tir, il ne vous reste plus qu'à faire apparaître ou défiler des ennemis et à marquer le score du joueur.
Comme d'hab, c'est à vous de jouer. Vous vouliez faire un jeu ? Qu'à cela ne tienne, vous avez le niveau maintenant et même un code de base pour démarrer un jeu de tir ! Qu'est-ce que vous attendez franchement ?

Bien sûr, les forums du Site du Zéro sont toujours là pour vous aider si vous êtes bloqués à un moment de la création de votre jeu. Il est normal de rencontrer des difficultés, quel que soit le niveau qu'on ait

LES MUSIQUES (MP3, OGG, WMA...)

En théorie, la fonction *FSOUND_Sample_Load* permet de charger n'importe quel type de son, y compris les formats compressés MP3, OGG, WMA. Le problème concerne les sons "longs", c'est-à-dire les musiques.
En effet, une musique dure en moyenne 3 à 4 minutes. Or, la fonction *FSOUND_Sample_Load* charge tout le fichier en mémoire (et c'est la version décompressée qui est mise en mémoire, donc ça prend beaucoup de place !).

Si vous avez un son long (on va parler de "musique" dorénavant), il est préférable de le charger en **streaming**, c'est-à-dire d'en charger des petits bouts au fur et à mesure de la lecture. C'est ce que font tous les lecteurs audio pour info.

Trouver des musiques

Là, on rentre en terrain miné, épineux, explosif (comme vous préférez).
En effet, la plupart des musiques et chansons que l'on connaît sont soumises au droit d'auteur. Même si vous ne faites qu'un petit programme, il faut verser une redevance ~~à l'auteur~~ à la SACEM (du moins en France c'est l'organisation qui s'occupe de ça).

Ne comptez pas sur moi pour vous expliquer comment télécharger ces chansons, tout le monde sait que c'est illégal (ce qui n'empêche pas tout le monde de le faire).

Donc, mis à part les MP3 soumis à droit d'auteur, que nous reste-t-il ?
Heureusement, il y a des chansons libres de droit ! Les auteurs vous autorisent à diffuser librement leurs chansons, il n'y a donc aucun problème pour que vous les utilisiez dans vos programmes.

Si votre programme est payant, il faudra en parler à l'artiste à moins que celui-ci n'autorise explicitement une utilisation commerciale de son oeuvre.
Une chanson libre de droit peut être téléchargée, copiée et écoutée librement, mais ça ne veut pas dire qu'on vous autorise à vous faire de l'argent sur le dos des artistes !

Bon, la question maintenant est : où trouver des musiques libres de droit ?
On pourrait faire une recherche de **Free Music** sur Google, mais là, pour le coup, Google n'est pas notre ami
En effet, allez savoir pourquoi, on a beau taper le mot "Free", on tombe quand même sur des sites qui nous proposent d'acheter des musiques !

Il existe heureusement des sites (à connaître !) qui sont dédiés à la musique libre de droit. Là, je vous recommande Jamendo qui est un très bon site, mais ce n'est pas le seul qui existe dans le domaine.

http://www.jamendo.com

Les chansons sont classées par style. Vous avez beaucoup de choix. On y trouve du bon, du moins bon, du très très bon, du très très nul... En fait, tout dépend de vos goûts et de votre récéptivité aux différents styles de musique
De préférence, prenez une chanson qui peut servir de musique de fond et qui correspond bien à l'univers de votre jeu.

Personnellement, j'ai flâné sur le site en écoutant des musiques dans les styles que j'aime bien (Rock, Pop Rock, Punk...) et je suis tombé sur une petite perle alors je vais utiliser cette chanson dans la suite de ce chapitre. L'artiste en question est un groupe français, Hype, et l'album s'appelle **Lies and Speeches**.

Jamendo.com propose des musiques libre de droit (ici le groupe Hype)

Je suis parfaitement conscient que les goûts et les couleurs ne se discutent pas. N'ayez donc pas peur de prendre une autre musique si celle-ci ne vous plaisait pas.

J'ai donc téléchargé l'album et je vais utiliser la chanson "Home" au format MP3.
Vous pouvez la **télécharger directement depuis le Site du Zéro (5,2 Mo)** si vous voulez faire des tests en même temps que moi. C'est un des avantages de la musique libre : on peut la copier / distribuer librement, donc ne nous gênons pas

Les étapes à suivre pour jouer une musique

Comme d'habitude, il faut que FMOD soit chargé avec *FSOUND_Init* et déchargé avec *FSOUND_Close*

Le pointeur

Cette fois, le pointeur doit être de type FSOUND_STREAM.

Code : C

```
FSOUND_STREAM *musique = NULL;
```

Charger le son

Comme je vous ai dit, le son sera chargé progressivement (on dit "en streaming"). Toutefois, il faut quand même ouvrir le fichier, car pour l'instant notre pointeur *musique* vaut toujours NULL je vous rappelle

On utilise ici la fonction *FSOUND_Stream_Open*. Elle prend 4 paramètres, ce sont les 4 mêmes derniers paramètres que ceux de la fonction *FSOUND_Sample_Load* qu'on a vue tout à l'heure.
En clair, indiquez le nom du fichier à ouvrir dans le premier paramètre, et laissez les 3 autres paramètres à 0.

La fonction retourne une adresse mémoire qu'on récupère avec notre pointeur *musique*.

Code : C

```
musique = FSOUND_Stream_Open("Hype_Home.mp3", 0, 0, 0);
```

Il est là encore fortement conseillé de vérifier si le fichier a bien été chargé. En cas d'échec, le pointeur vaut NULL.

Jouer la musique

C'est très simple, on fait appel à *FSOUND_Stream_Play*.
Elle prend 2 paramètres :

- Le numéro du canal sur lequel jouer le son (envoyez *FSOUND_FREE* et FMOD se débrouillera tout seul pour trouver un canal libre)
- Le pointeur vers le fichier à lire (dans notre cas il s'appelle *musique*).

On peut donc jouer notre musique avec :

Code : C

```
FSOUND_Stream_Play(FSOUND_FREE, musique);
```

Et voilà le travail

Mais ce n'est pas tout. Dans le cas d'une musique, il peut être bien de savoir modifier le volume, gérer les répétitions de la chanson, la mettre en pause ou même l'arrêter. C'est ce genre de choses que nous allons voir maintenant.

Modifier le volume

Avec la fonction *FSOUND_SetVolume*, vous pouvez changer le volume d'un canal.

Code : C

```
FSOUND_SetVolume(FSOUND_ALL, 120);
```

Il faut envoyer 2 paramètres :

- Le numéro du canal dont on doit changer le volume (pour changer le volume de tous les canaux, envoyez FSOUND_ALL)
- Le nouveau volume : mettez un nombre de 0 (silencieux) à 255 (volume maximal)

Cette fonction permet aussi de changer le volume des sons courts, et pas seulement celui des sons streamés (longs).

Répétition de la chanson

On a souvent besoin de répéter la musique de fond. C'est justement ce que propose la fonction *FSOUND_Stream_SetLoopCount*. Elle prend 2 paramètres :

- Le pointeur vers la chanson
- Le nombre de fois qu'elle doit être répétée. Si vous mettez 1, la chanson sera donc répétée une seule fois. Si vous mettez un nombre négatif (comme -1), la chanson sera répétée à l'infini.

Avec ce code source, notre musique sera donc répétée à l'infini :

Code : C

```
FSOUND_Stream_SetLoopCount(musique, -1);
```

Mettre en pause la chanson

Il y a ici 2 fonctions à connaître :

- **FSOUND_GetPaused(numero_du_canal)** : indique si la chanson jouée sur le canal indiqué est en pause ou pas. Elle renvoie vrai si la chanson est en pause, faux si elle est en train d'être jouée.
- **FSOUND_SetPaused(numero_du_canal, etat)** : met en pause ou réactive la lecture de la chanson sur le canal indiqué. Envoyez 1 (vrai) pour mettre en pause, 0 (faux) pour réactiver la lecture.

Ce bout de code de fenêtre SDL met en pause la chanson si on appuie sur P, et la réactive si on appuie à nouveau sur P ensuite.

Code : C

```
case SDL_KEYDOWN:
    if (event.key.keysym.sym == SDLK_p) // Si on appuie sur P
    {
        if (FSOUND_GetPaused(1)) // Si la chanson est en pause
            FSOUND_SetPaused(FSOUND_ALL, 0); // On enlève la pause
        else // Sinon, elle est en cours de lecture
            FSOUND_SetPaused(FSOUND_ALL, 1); // On met en pause
    }
    break;
```

Stopper la lecture

Il suffit d'appeler *FSOUND_Stream_Stop*. On lui envoie le pointeur vers la chanson à arrêter.

Code : C

```
FSOUND_Stream_Stop(musique);
```

Et bien d'autres choses

On peut faire beaucoup d'autres choses, mais je ne vais pas vous les énumérer toutes ici, autant répéter la doc ! Je vous invite donc à la lire si vous cherchez des fonctions supplémentaires

Libérer la mémoire

Pour décharger la musique de la mémoire, appelez *FSOUND_Stream_Close* et donnez-lui le pointeur.

Code : C

```
FSOUND_Stream_Close(musique);
```

Code complet de lecture du MP3

Le code ci-dessous vous montre un programme jouant la musique "Home" qu'on a récupéré sur Jamendo.
La musique est jouée dès le début du programme. On peut la mettre en pause en appuyant sur P.

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *pochette = NULL;
    SDL_Event event;
    SDL_Rect position;
    int continuer = 1;
    FSOUND_STREAM *musique = NULL;

    FSOUND_Init(44100, 32, 0);
    musique = FSOUND_Stream_Open("Hype_Home.mp3", 0, 0, 0); /* On ouvre la musique */
    if (musique == NULL) /* On vérifie si elle a bien été ouverte (IMPORTANT) */
    {
        fprintf(stderr, "Impossible de lire Hype_Home.mp3\n");
        exit(EXIT_FAILURE);
    }

    FSOUND_Stream_SetLoopCount(musique, -1); /* On active la répétition de la musique à
l'infini */
    FSOUND_Stream_Play(FSOUND_FREE, musique); /* On joue la musique */

    SDL_Init(SDL_INIT_VIDEO);


    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du son avec FMOD", NULL);
    pochette = IMG_Load("hype_liesandspeeches.jpg");
    position.x = 0;
    position.y = 0;


    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
            case SDL_KEYDOWN:
                if (event.key.keysym.sym == SDLK_p) //Si on appuie sur P
                {
                    if (FSOUND_GetPaused(1)) // Si la chanson est en pause (sur le canal
1)
                        FSOUND_SetPaused(1, 0); // On enlève la pause
                    else // Sinon, elle est en cours de lecture
                        FSOUND_SetPaused(1, 1); // On active la pause
                }
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));
        SDL_BlitSurface(pochette, NULL, ecran, &position);
        SDL_Flip(ecran);
    }

    FSOUND_Stream_Close(musique); /* On libère la mémoire */
    FSOUND_Close();

    SDL_FreeSurface(pochette);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Histoire d'avoir autre chose qu'une fenêtre noire, j'ai mis la pochette de l'album en image de fond. Bien entendu, ce qui nous intéresse, c'est comment est jouée la musique hein, pas la pochette

Donc bon, en image ça donne juste une fenêtre SDL avec la pochette :

L'intérêt dans cette image, c'est le son qui se joue derrière
(euh oui certes, les images ne produisent pas de son...)

Bon, allez, je crois qu'une vidéo sera plus convaincante

Une musique jouée avec FMOD (730 Ko)

Les musiques (MIDI)

Les musiques de type MIDI sont très différentes des musiques de type MP3, OGG ou WMA qu'on vient d'étudier. En effet, au lieu d'enregistrer la musique (avec un micro), cette fois la musique est créée de toutes pièces sur l'ordinateur. On n'enregistre que des notes de musique, ce qui explique pourquoi on ne peut pas enregistrer la voix.

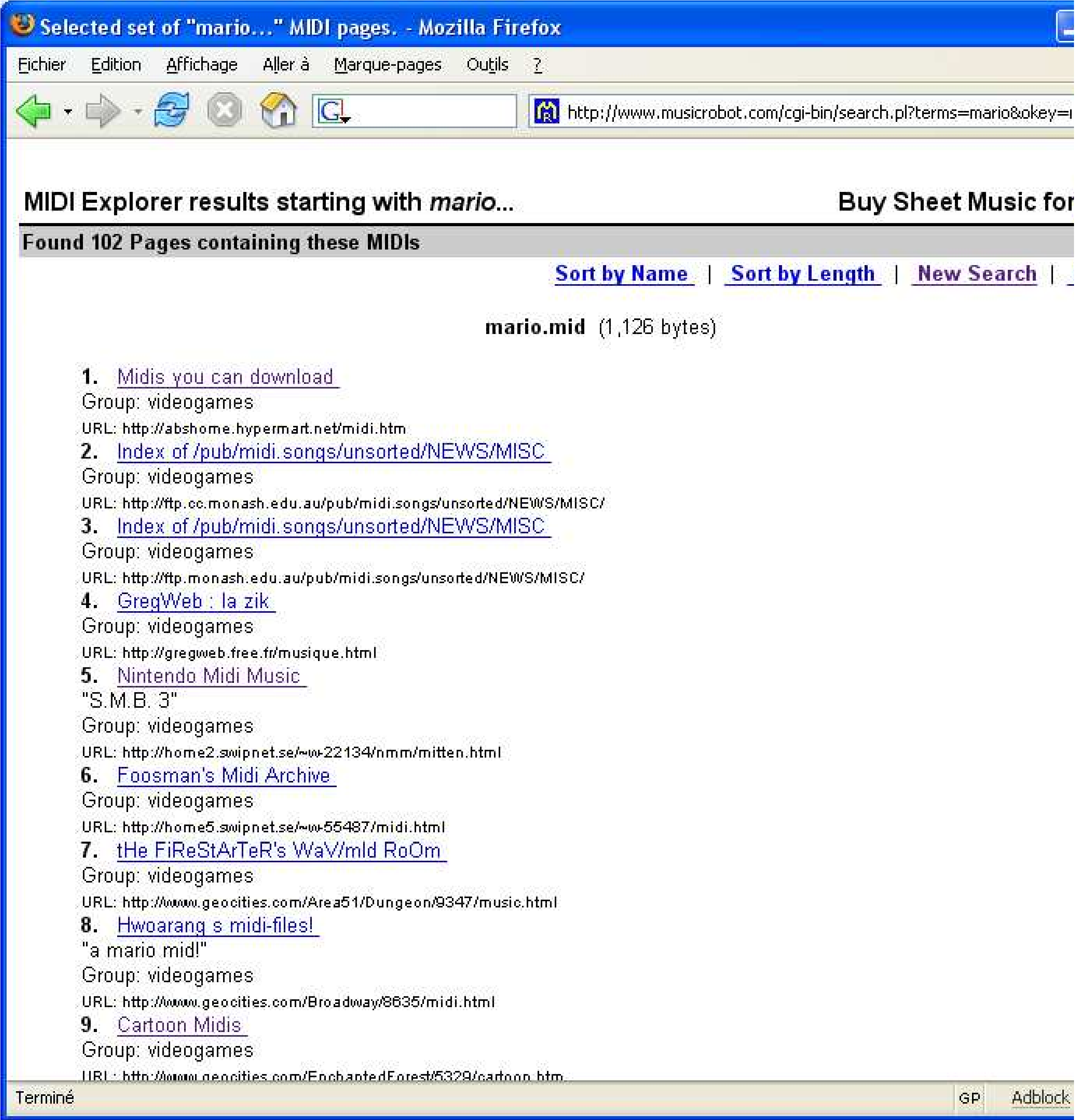
L'avantage ? En enregistrant uniquement les notes, on obtient des fichiers très très petits. Vous avez peut-être déjà remarqué que les MIDI étaient de tous petits fichiers.
Le défaut ? Eh bien on ne peut pas enregistrer de voix et les effets autorisés par le format, bien que nombreux, sont limités.

Ce format est donc inadapté pour enregistrer des musiques qui passent à la radio par exemple (mais certains essaient de les recréer !), en revanche il est tout à fait adapté pour jouer de vieilles musiques de l'époque de la Super-NES, GameBoy, MegaDrive, etc.

Trouver des MIDI

Google "Free Midi"
Etonnant non ?

On trouve des tooooonnes de MIDI sur le net. Je m'en fais pas pour vous, vous trouverez votre bonheur !
Personnellement, j'ai retenu MusicRobot.com, un moteur de recherche pour fichiers MIDI.



Le moteur de recherche de fichiers MIDI MusicRobot.com
(ici, à la recherche d'un MIDI de Mario)

Personnellement, j'ai récupéré la musique de Mario (ah les souvenirs). Vous pouvez la télécharger pour vos tests si vous le voulez.

Les étapes à suivre pour jouer un MIDI

Les fonctions pour jouer des MIDI commencent par le préfixe FMUSIC au lieu de FSOUND.
Toutefois, les fonctions de chargement et de déchargement de FMOD à utiliser restent les mêmes, et elles ont bien

le préfixe FSOUND.

Bon vous commencez à avoir l'habitude alors je vais aller un peu plus vite maintenant dans le listing des fonctions

Le pointeur

Code : C

```
FMUSIC_MODULE *musique = NULL;
```

Charger un MIDI

Code : C

```
musique = FMUSIC_LoadSong("mario.mid");
```

La fonction prend un seul paramètre, comme vous le voyez c'est encore plus simple.
Elle renvoie NULL si le fichier n'a pas pu être chargé.

Jouer un MIDI

Code : C

```
FMUSIC_PlaySong(musique);
```

Répéter un MIDI

Code : C

```
FMUSIC_SetLooping(musique, 1);
```

Cette fois, c'est un peu différent. Il faut envoyer 1 (VRAI) pour que la musique soit répétée à l'infini.

Mettre en pause un MIDI

La fonction *FMUSIC_GetPaused* indique si la chanson est en pause ou pas.

Code : C

```
FMUSIC_GetPaused(musique);
```

La fonction *FMUSIC_SetPaused* met en pause ou réactive la lecture de la chanson.

Code : C

```
FMUSIC_SetPaused(musique, 1);
```

Envoyez 1 pour mettre en pause, 0 pour relancer la lecture.

Exemple de code gérant la pause si on appuie sur P :

Code : C

```
case SDL_KEYDOWN:
    if (event.key.keysym.sym == SDLK_p) //Si on appuie sur P
    {
        if (FMUSIC_GetPaused(musique)) // Si la chanson est en pause
            FMUSIC_SetPaused(musique, 0); // On enlève la pause
        else // Sinon, elle est en cours de lecture
            FMUSIC_SetPaused(musique, 1); // On active la pause
    }
```

Attention : bien que similaire, ce code est différent du code de pause qu'on a vu tout à l'heure. En particulier, il n'y a pas de canal à indiquer ici.

Modifier le volume

Code : C

```
FMUSIC_SetMasterVolume(musique, 150);
```

Le second paramètre correspond au volume.

- 0 = silencieux
- 256 = volume maximal

Stopper la lecture

Code : C

```
FMUSIC_StopSong(musique);
```

Libérer la musique MIDI

Code : C

```
FMUSIC_FreeSong(musique);
```

Code d'exemple pour résumer

Code : C

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *niveau = NULL;
    SDL_Event event;
    SDL_Rect position;
    int continuer = 1;
    FMUSIC_MODULE *musique = NULL;

    FSOUND_Init(44100, 32, 0);
    musique = FMUSIC_LoadSong("mario.mid"); // Chargement de la chanson
    if (musique == NULL)
    {
        fprintf(stderr, "Impossible de lire mario.mid\n");
        exit(EXIT_FAILURE);
    }
    FMUSIC_SetLooping(musique, 1); // Répétition infinie
    FMUSIC_PlaySong(musique); // On joue la chanson


    SDL_Init(SDL_INIT_VIDEO);

    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Gestion du son avec FMOD", NULL);
    niveau = IMG_Load("mario_niveau.jpg"); // Je me permets de mettre une petite image de
fond ^^
    position.x = 0;
    position.y = 0;


    while (continuer)
    {
        SDL_WaitEvent(&event);
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
            case SDL_KEYDOWN:
                if (event.key.keysym.sym == SDLK_p) //Si on appuie sur P
                {
                    if (FMUSIC_GetPaused(musique)) // Si la chanson est en pause
                        FMUSIC_SetPaused(musique, 0); // On enlève la pause
                    else // Sinon, elle est en cours de lecture
                        FMUSIC_SetPaused(musique, 1); // On active la pause
                }
                break;
        }

        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));
        SDL_BlitSurface(niveau, NULL, ecran, &position);
        SDL_Flip(ecran);
    }

    FMUSIC_FreeSong(musique); // Déchargement de la chanson
    FSOUND_Close();

    SDL_FreeSurface(niveau);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Ce code reprend les fonctions principales qu'on vient de voir.
J'ai même mis une image de fond pour l'ambiance

Oui oui il y a la musique de Mario en fond !
(mais n'allez pas croire que c'est un jeu, c'est juste une image !)

Bien sûr, c'est mieux quand on a la musique en fond derrière
Je vous propose donc de voir ce que ça donne en vidéo.

La musique MIDI de Mario (240 Ko)
(la compression Flash a un peu déterioré le son désolé)

Bien sûr, ce n'est pas un vrai jeu, rien ne bouge, mais on s'y croirait ! Et puis, rien n'empêche de le coder, ce jeu
Ce chapitre devrait vous avoir permis de démarrer dans la manipulation du son dans vos programmes (du moins je l'espère). Que vous vouliez créer un jeu, un lecteur MP3 ou même un simple programme utilitaire, vous aurez la plupart du temps besoin de faire appel à une librairie comme FMOD pour gérer le son.

Il faut reconnaître que tout cela n'est pas bien compliqué. Il faut juste savoir quelles fonctions utiliser dans le bon ordre et savoir gérer la mémoire correctement (c'est-à-dire utiliser les fonctions de lecture streamées sur des sons longs par exemple).

Toutefois, nous n'avons pas vu toutes les fonctionnalités de FMOD, loin de là ! La librairie gère de nombreux effets audio (écho, distorsion, effets de son 3D...), l'enregistrement, et bien d'autres choses encore ! Comme je ne peux pas tout vous détailler, je vous recommande de lire la documentation de FMOD.
A votre stade, il est vraiment primordial que vous commenciez à être capable de lire des documentations. N'ayez pas peur, la doc ne vous mangera pas Vous y découvrirez toutes les fonctions que propose la librairie, ce qui vous permettra de vous perfectionner encore plus

---

TP : visualisation spectrale du son

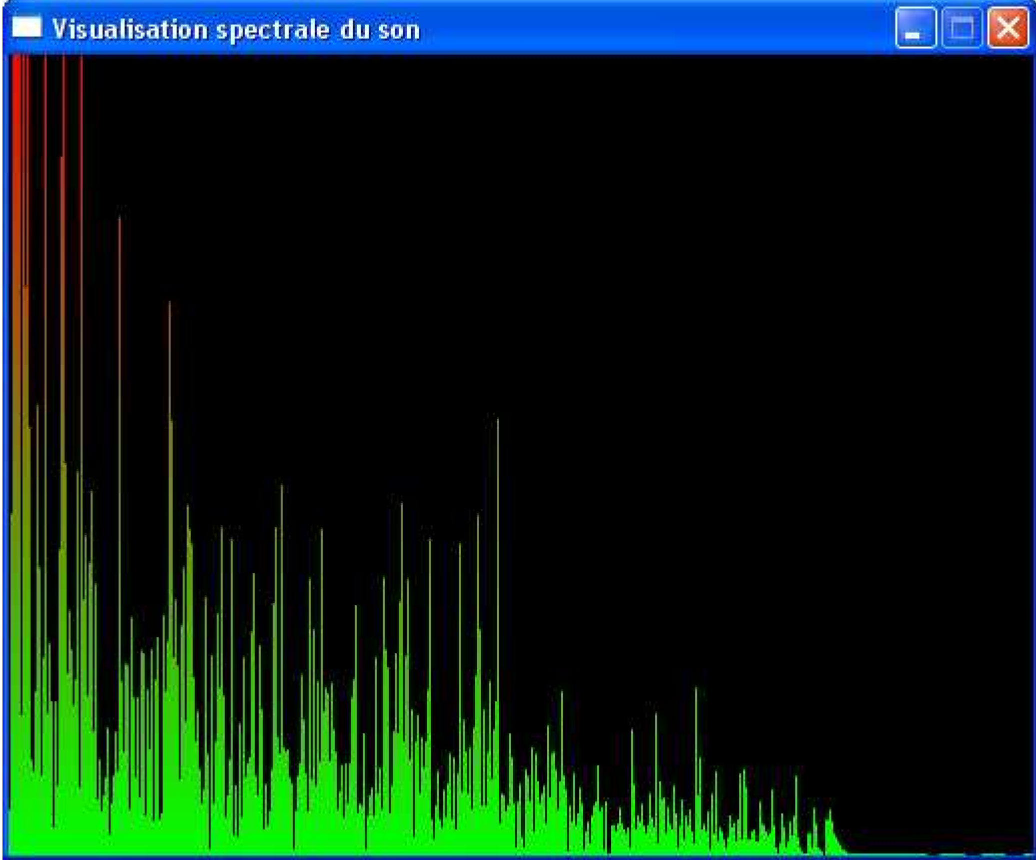
Bienvenue dans ce **TP SDL & FMOD** !
Eh oui, nous utiliserons dans ce TP les librairies SDL et FMOD simultanément pour réaliser notre programme

Qu'on se le dise : nous n'allons pas créer de jeu ici ! Certes, la SDL est tout particulièrement adaptée aux jeux, mais ça ne veut pas dire qu'elle est faite uniquement pour ça. Ce chapitre va vous prouver qu'elle peut servir à autre chose

Qu'allons-nous faire ici ? Nous allons réaliser une visualisation du spectre sonore en SDL.
Nooooon ne partez pas, vous allez voir ce n'est pas si compliqué que ça, et en plus ça nous fera travailler :

- La gestion du temps
- La librairie FMOD
- La modification d'une surface pixel par pixel (on n'a pas encore découvert ça, mais ce TP sera l'occasion idéale !)

Voici un aperçu de ce que nous allons réaliser ici :



C'est le genre de visualisation qu'on peut retrouver dans les lecteurs audio comme Winamp, Windows Media Player ou encore AmaroK.
Et pour ne rien gâcher, ce n'est pas bien difficile à faire. D'ailleurs, contrairement au TP Mario Sokoban, cette fois c'est vous qui allez travailler. Ca vous fera un très bon exercice

LES CONSIGNES

Les consignes sont simples. Suivez-les pas à pas dans l'ordre, et vous n'aurez pas d'ennuis

1/ Lire un MP3

Pour commencer, vous devez créer un programme qui lit un fichier MP3. Vous n'avez qu'à reprendre la chanson

"Home" du groupe Hype que nous avons utilisée dans le chapitre sur FMOD pour illustrer le fonctionnement de la lecture d'une musique. Si vous avez bien suivi ce chapitre, il ne vous faudra pas plus de quelques minutes pour arriver à le faire
Je vous conseille au passage de placer le MP3 dans le dossier de votre projet.

2/ Activer le module DSP de FMOD

Je vous avais dit dans le chapitre sur FMOD que nous n'avions pas vu toutes les possibilités de cette librairie. Elle peut lire aussi des CD, effectuer des enregistrements, des modifications sur le son, etc etc...
Nous allons ici nous intéresser à un de ses modules, appelé DSP.

A quoi servent les fonctions du module DSP ?
Eh bien ce sont justement elles qui nous donnent des informations sur le son, afin que nous puissions réaliser notre visualisation spectrale.

Il va falloir activer ce module.

Pourquoi ce module n'est-il pas activé par défaut ?

En fait, cela demande pas mal de calculs supplémentaires à l'ordinateur. Comme on n'en a pas toujours besoin, le module DSP est désactivé par défaut. Heureusement, l'activer est très simple :

Code : C

```
FSOUND_DSP_SetActive(FSOUND_DSP_GetFFTUnit(), 1);
```

Le premier paramètre doit toujours être le résultat renvoyé par la fonction *FSOUND_DSP_GetFFTUnit()*.

Quant au second paramètre, c'est un booléen :

- Si on envoie VRAI (1), le module DSP est activé.
- Si on envoie FAUX (0), le module DSP est désactivé.

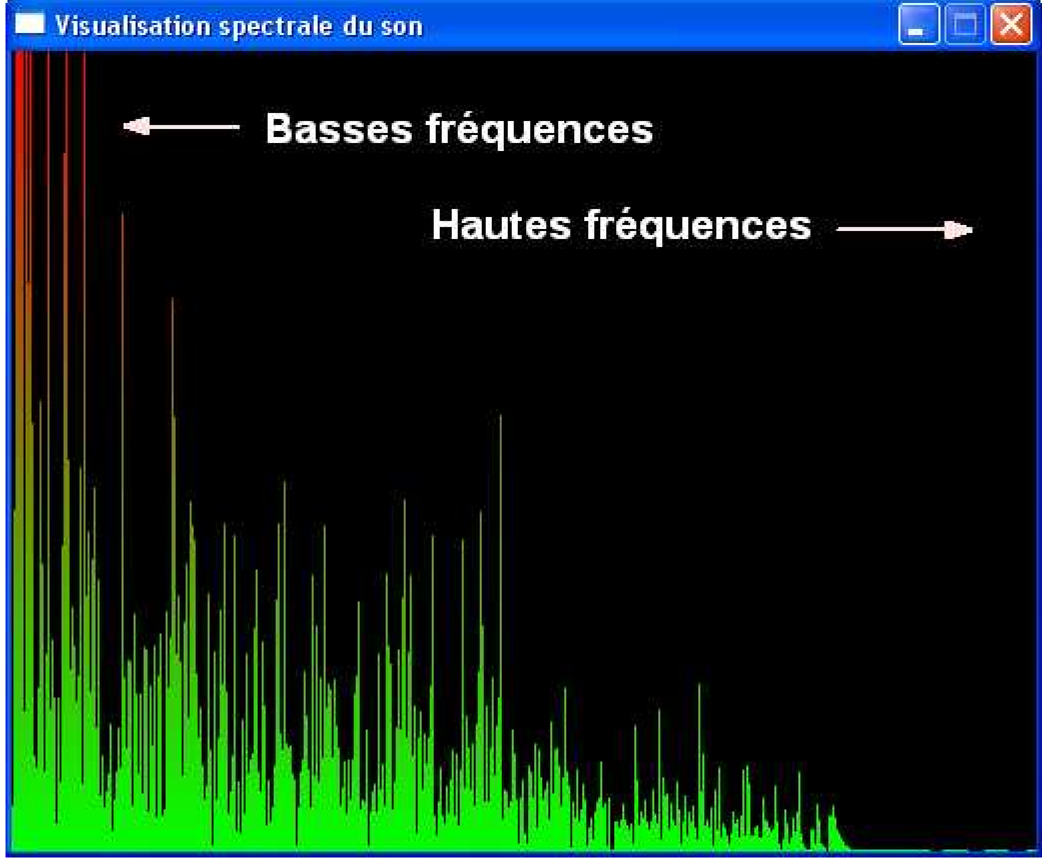
Par conséquent, si on veut désactiver le module DSP juste avant la fin du programme, il suffit d'écrire :

Code : C

```
FSOUND_DSP_SetActive(FSOUND_DSP_GetFFTUnit(), 0);
```

3/ Récupérer les données spectrales du son

Pour comprendre comment la visualisation spectrale fonctionne, il est indispensable que je vous explique un peu comment ça marche à l'intérieur (rapidement hein, pas dans le détail, sinon ça va se transformer en cours de maths).



Un son peut être découpé en fréquences. Certaines fréquences sont basses, d'autres moyennes, et d'autres hautes. Ce que nous allons faire dans notre visualisation, c'est afficher la quantité de chacune de ces fréquences sous formes de barres. Plus la barre est grande, plus la fréquence est utilisée.

Sur la gauche de la fenêtre, nous faisons donc apparaître les basses fréquences, et sur la droite les hautes fréquences.

Mais comment récupérer les quantités de chaque fréquence ?

FMOD nous mâche le travail
Maintenant que le module DSP est activé, on peut faire appel à la fonction *FSOUND_DSP_GetSpectrum()*.
Il n'y a aucun paramètre à lui donner. En revanche, elle vous renvoie quelque chose de très intéressant : un pointeur vers un tableau de 512 float.

Rappel : le type *float* est un type décimal, au même titre que *double*. La différence entre les deux vient du fait que *double* est plus précis que *float*, mais dans notre cas le type *float* est suffisant. C'est celui utilisé par FMOD ici, donc c'est celui que nous devrons utiliser nous aussi.

Il vous faudra donc déclarer un pointeur vers float. La fonction *FSOUND_DSP_GetSpectrum()* vous renvoie le pointeur sur le tableau, **donc sur le premier élément**.

En clair, on déclare le pointeur :

Code : C
```c
float *spectre = NULL;
```

Et lorsque la musique est en train d'être jouée et que le module DSP est activé, on peut récupérer l'adresse du tableau de 512 float que nous a créé FMOD :

Code : C

```c
spectre = FSOUND_DSP_GetSpectrum();
```

On peut ensuite parcourir ce tableau pour obtenir les valeurs de chacune des fréquences :

Code : C

```c
spectre[0] // Fréquence la plus basse (à gauche)
spectre[1]
spectre[2]
...
spectre[509]
spectre[510]
spectre[511] // Fréquence la plus haute (à droite)
```

Chaque fréquence est un nombre décimal compris entre 0 (rien) et 1 (maximum). Votre travail va consister à afficher une barre plus ou moins grande en fonction de la valeur que contient chaque case du tableau.

Par exemple, si la valeur est 0.5, vous devrez tracer une barre dont la hauteur correspondra à la moitié de la fenêtre.
Si la valeur est 1, elle devra faire toute la hauteur de la fenêtre.

Généralement, les valeurs sont assez faibles (plutôt proches de 0 que de 1). Je recommande de multiplier par 4 toutes les valeurs pour mieux voir le spectre.
ATTENTION : si vous faites ça, vérifiez que vous ne dépassez pas 1 (arrondissez à 1 s'il le faut). Si vous vous retrouvez avec des valeurs supérieures à 1, vous risquez d'avoir des problèmes pour tracer les barres verticales par la suite !

Mais les barres doivent bouger au fur et à mesure du temps non ? Comme le son change tout le temps, il faut mettre à jour le graphique. Comment faire ?

Bonne question. En effet, le tableau de 512 floats que vous renvoie FMOD **change toutes les 25 ms** (pour être à jour par rapport au son actuel). Il va donc falloir dans votre code que vous relisiez le tableau de 512 floats (en refaisant appel à *FSOUND_DSP_GetSpectrum()*) toutes les 25 ms, puis que vous mettiez à jour votre graphique en barres.

Relisez le chapitre sur la gestion du temps en SDL pour vous rappeler comment faire
Vous avez le choix entre une solution à base de GetTicks ou à base de callbacks. Faites ce qui vous paraît le plus facile.

4/ Réaliser le dégradé

Dans un premier temps, vous pouvez réaliser des barres de couleur unie. Vous pourrez donc créer des surfaces. Il devra y avoir 512 surfaces : une pour chaque barre. Chaque surface fera donc 1 pixel de large et la hauteur des surfaces variera en fonction de l'intensité de chaque fréquence.

Toutefois, je vous propose ensuite d'effectuer une amélioration : la barre doit fondre vers le rouge lorsque le son devient de plus en plus intense. En clair, la barre doit être verte en bas et rouge en haut.

Mais... une surface ne peut avoir qu'une seule couleur si on utilise *SDL_FillRect()*. On ne peut pas faire de dégradé !

En effet. On pourrait certes créer des surfaces de 1 pixel de large et 1 pixel de haut pour chaque couleur du dégradé, mais ça ferait vraiment beaucoup de surfaces à gérer et ce ne serait pas très optimisé !

Comment fait-on pour dessiner pixel par pixel ?
Je ne vous l'ai pas appris auparavant car cette technique ne méritait pas un chapitre entier. Vous allez voir en effet que ce n'est pas bien compliqué.

En fait, la SDL ne propose aucune fonction pour dessiner pixel par pixel. Mais on a le droit de l'écrire nous-même

Pour ce faire, il faut suivre ces étapes méthodiquement dans l'ordre :

1. Faites appel à la fonction *SDL_LockSurface* pour annoncer à la SDL que vous allez modifier la surface manuellement. Cela "bloque" la surface pour la SDL et vous êtes le seul à y avoir accès tant que la surface est bloquée.
   Ici, je vous conseille de ne travailler qu'avec une seule surface : l'écran. Si vous voulez dessiner un pixel à un endroit précis de l'écran, vous devrez donc bloquer la surface *ecran* :

   Code : C
   ```
   SDL_LockSurface(ecran);
   ```

2. Vous pouvez ensuite modifier le contenu de chaque pixel de la surface. Comme la SDL ne propose aucune fonction pour faire ça, il va falloir l'écrire nous-même dans notre programme.
   Cette fonction, je vous la donne. Je la tire de la documentation de la SDL. Elle est un peu compliquée car elle travaille sur la surface directement et gère toutes les profondeurs de couleurs (bits par pixel) possibles. Pas besoin de la retenir ou de la comprendre, faites simplement un copier-coller dans votre programme pour pouvoir l'utiliser :

   Code : C

```
void setPixel(SDL_Surface *surface, int x, int y, Uint32 pixel)
{
    int bpp = surface->format->BytesPerPixel;

    Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * bpp;

    switch(bpp) {
    case 1:
        *p = pixel;
        break;

    case 2:
        *(Uint16 *)p = pixel;
        break;

    case 3:
        if(SDL_BYTEORDER == SDL_BIG_ENDIAN) {
            p[0] = (pixel >> 16) & 0xff;
            p[1] = (pixel >> 8) & 0xff;
            p[2] = pixel & 0xff;
        } else {
            p[0] = pixel & 0xff;
            p[1] = (pixel >> 8) & 0xff;
            p[2] = (pixel >> 16) & 0xff;
        }
        break;

    case 4:
        *(Uint32 *)p = pixel;
        break;
    }
}
```

Elle est simple à utiliser. Envoyez les paramètres suivants :

- Le pointeur vers la surface à modifier (cette surface doit préalablement avoir été bloquée avec SDL_LockSurface)
- La position en abscisse du pixel à modifier dans la surface (x).
- La position en ordonnée du pixel à modifier dans la surface (y).
- La nouvelle couleur à donner à ce pixel. Cette couleur doit être au format Uint32, vous pouvez donc la générer à l'aide de la fonction *SDL_MapRGB()* que vous connaissez bien maintenant

3. Enfin, lorsque vous avez fini de travailler sur la surface, il ne faut pas oublier de la débloquer en appelant *SDL_UnlockSurface*.

Code : C

```
SDL_UnlockSurface(ecran);
```

Code résumé d'exemple

Si on résume, vous allez voir que c'est tout simple.
Ce code dessine un pixel rouge au milieu de la surface *ecran* (donc au milieu de la fenêtre).

Code : C

```
SDL_LockSurface(ecran); /* On bloque la surface */
setPixel(ecran, ecran->w / 2, ecran->h / 2, SDL_MapRGB(ecran->format, 255, 0, 0)); /* On
dessine un pixel rouge au milieu de l'écran */
SDL_UnlockSurface(ecran); /* On débloque la surface*/
```

Débrouillez-vous avec ça pour réaliser les dégradés du vert au rouge.
Un indice : il faut utiliser des boucles

(quoi vous aviez deviné tout seul ?)

LA SOLUTION

Alors, comment vous avez trouvé le sujet ?
Il est pas bien difficile à appréhender, il faut juste faire quelques calculs surtout pour la réalisation du dégradé. C'est du niveau de tout le monde, il faut juste réfléchir un petit peu.

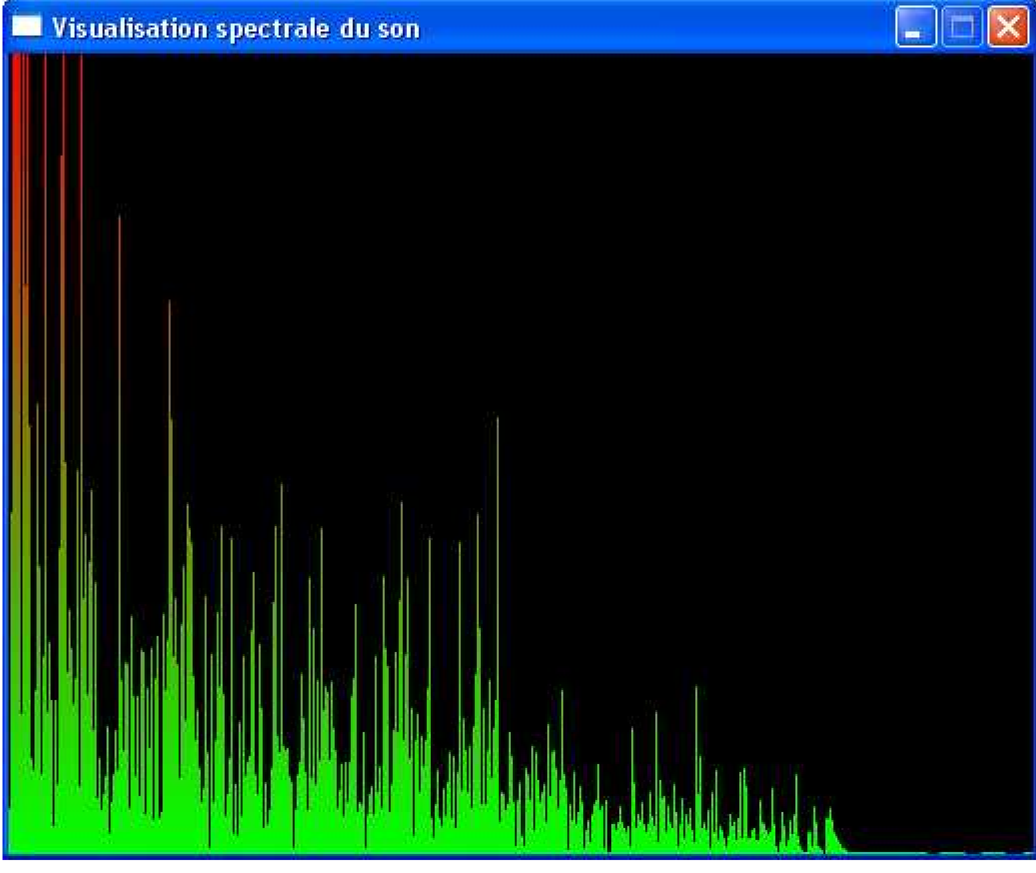
Certains mettent plus de temps que d'autres pour trouver la solution. Si vous avez du mal, ce n'est pas bien grave. Ce qui compte c'est de finir par y arriver.
Quel que soit le projet dans lequel vous vous lancez, vous aurez forcément des petits moments où il ne suffit pas de savoir programmer, il faut aussi **être logique et bien réfléchir**.

Je vous donne le code complet ci-dessous. Il est suffisamment commenté.

Code : C

```
#include <stdlib.h>
#include <stdio.h>
#include <SDL/SDL.h>
#include <FMOD/fmod.h>

#define LARGEUR_FENETRE         512 /* DOIT rester à 512 impérativement car il y a 512
barres (correspondant aux 512 floats) */
#define HAUTEUR_FENETRE         400 /* Vous pouvez la faire varier celle-là par contre
:o) */
#define RATIO                   (HAUTEUR_FENETRE / 255.0)
#define DELAI_RAFRAICHISSEMENT  25 /* Temps en ms entre chaque mise à jour du graphe. 25
ms est la valeur minimale. */

void setPixel(SDL_Surface *surface, int x, int y, Uint32 pixel);


int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL, *ligne = NULL;
    SDL_Event event;
    SDL_Rect position;
    int continuer = 1, hauteurBarre = 0, tempsActuel = 0, tempsPrecedent = 0, i = 0, j =
0;
    float *spectre = NULL;

    /* Initialisation de FMOD
       ----------------------

    On charge FMOD, la musique, on active le module DSP et on lance la lecture
    de la musique */

    FSOUND_Init(44100, 4, 0);
    FSOUND_STREAM* musique = FSOUND_Stream_Open("Hype_Home.mp3", 0, 0, 0);
    if (musique == NULL)
    {
        fprintf(stderr, "Impossible d'ouvrir la musique");
        exit(EXIT_FAILURE);
    }
    FSOUND_DSP_SetActive(FSOUND_DSP_GetFFTUnit(), 1);
    FSOUND_Stream_Play(FSOUND_FREE, musique);


    /* Initialisation de la SDL
       ------------------------

    On charge la SDL, on ouvre la fenêtre et on écrit dans sa barre de titre.
    On récupère au passage un pointeur vers la surface ecran, qui sera la seule
    surface utilisée dans ce programme */

    SDL_Init(SDL_INIT_VIDEO);
    ecran = SDL_SetVideoMode(LARGEUR_FENETRE, HAUTEUR_FENETRE, 32, SDL_SWSURFACE |
SDL_DOUBLEBUF);
    SDL_WM_SetCaption("Visualisation spectrale du son", NULL);



    /* Boucle principale */

    while (continuer)
    {
        SDL_PollEvent(&event); /* On doit utiliser PollEvent car il ne faut pas attendre
d'évènement
                                de l'utilisateur pour mettre à jour la fenêtre*/
        switch(event.type)
        {
            case SDL_QUIT:
                continuer = 0;
                break;
        }

        /* On efface l'écran à chaque fois avant de dessiner le graphe (fond noir */
        SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));


        /* Gestion du temps
           -----------------
        On compare le temps actuel par rapport au temps précédent (dernier passage dans
```

Vous devriez obtenir un résultat correspondant à la capture d'écran que je vous avais montrée au début du chapitre :



Bien entendu, il vaut mieux une animation pour voir le résultat. C'est donc ce que je vous propose ci-dessous

Voir l'animation "Visualisation spectrale du son" (4,3 Mo)

Notez que la compression a réduit la qualité du son et le nombre d'images par seconde.
Le mieux est encore de télécharger le programme complet (avec son code source) pour tester chez soi. Vous pourrez ainsi apprécier le programme dans les meilleures conditions

Télécharger le projet Code::Blocks complet + l'exécutable Windows (335 Ko)

Il faut impérativement que le fichier "Hype_Home.mp3" soit placé dans le dossier du programme pour qu'il fonctionne (sinon il s'arrêtera de suite).
Si vous n'avez toujours pas la chanson en question, vous pouvez **la télécharger** ou bien récupérer une autre chanson à vous et la renommer en "Hype_Home.mp3".

IDÉES D'AMÉLIORATION

~~Comment améliorer un truc parfait ?~~

Il est toujours possible d'améliorer un programme. Ici, j'ai par exemples des tonnes d'idées d'extensions qui pourraient aboutir à la création d'un véritable petit lecteur MP3.

- Il serait bien qu'on puisse choisir le MP3 qu'on veut lire. Il faudrait par exemple lister tous les .mp3 présents

dans le dossier du programme. On n'a pas vu comment faire ça, mais vous pouvez le découvrir par vous-même Indice : utilisez la librairie dirent (il faudra inclure dirent.h). A vous de chercher des informations sur le net pour savoir comment l'utiliser. L'idéal est d'être capable de **lire la doc à ce sujet**.
- Si votre programme était capable de lire et gérer les playlists, ça serait encore mieux. Il existe plusieurs formats de playlist, le plus connu est le format M3U.
  Exemple de fichier playlist au format M3U :

  Code : Autre
  ```
  #EXTM3U
  #EXTINF:0,01 - Home.mp3
  01 - Home.mp3

  #EXTINF:0,02 - My Innocence.mp3
  02 - My Innocence.mp3

  #EXTINF:0,03 - Scream.mp3
  03 - Scream.mp3

  #EXTINF:0,04 - Anybody here.mp3
  04 - Anybody here.mp3

  #EXTINF:0,05 - Get there.mp3
  05 - Get there.mp3

  #EXTINF:0,06 - Spirits above.mp3
  06 - Spirits above.mp3
  ```

- Vous pourriez afficher le nom du MP3 en cours de lecture dans la fenêtre (il faudra utiliser SDL_ttf)
- Vous pourriez afficher un indicateur pour qu'on sache où en est de la lecture du morceau, comme cela se fait sur la plupart des lecteurs MP3.
- Vous pourriez aussi proposer de modifier le volume de lecture
- etc etc...

Bref, il y a beaucoup à faire.
Vous avez la possibilité de créer de beaux lecteurs, il ne tient plus qu'à vous de les coder S'il y a une chose à retenir de ce TP, c'est que la difficulté quand on programme n'est pas toujours de savoir *quelle fonction utiliser pour tel effet*. La preuve : je vous ai donné toutes les fonctions nécessaires dès le début de ce TP.

Non, la difficulté consiste à réfléchir correctement pour arriver à résoudre un problème. C'est quelque chose que vous rencontrerez quel que soit le langage utilisé. On dit que c'est un problème d'algorithmique : il faut arriver à écrire le meilleur code source pour arriver au résultat que l'on veut.

Ici, il y avait 2 problèmes algorithmiques :

- Le dessin des barres verticales, qu'il fallait adapter en fonction de la hauteur de la fenêtre. Vous avez dû faire un petit calcul pour transformer un nombre entre 0 et 1 en un nombre entre 0 et HAUTEUR_FENETRE pour pouvoir tracer les barres.
- Le dégradé du vert au rouge. Il fallait là encore faire quelques calculs pas bien compliqués sur une variable pour l'adapter en une valeur comprise entre 0 et 255

Lire la solution n'a que peu d'intérêt. D'ailleurs pour être franc, je crois que si j'avais lu cette solution avant d'avoir réfléchi au problème, ça m'aurait découragé. Il est en effet toujours plus difficile de lire le code source d'un autre que de l'écrire.
D'où l'intérêt de prendre le temps qu'il faut, mais de le faire soi-même

N'ayez donc pas peur si vous ne comprenez pas instantanément tous les codes sources que vous lisez. Personne ne peut se vanter de savoir faire ça.
Ce qui compte, c'est d'y arriver tout seul, quel que soit le temps que vous mettez. Au fur et à mesure, vous deviendrez habitué à ce genre de problèmes d'algorithmique et il vous faudra de moins en moins de temps pour les résoudre La partie sur la SDL est terminée, mais il est fort probable que des TP supplémentaires fassent leur

apparition dans le futur.
Cette partie n'était qu'une *application pratique* de ce que vous avez appris dans les parties I et II. Vous n'avez en fait rien découvert de nouveau sur le langage C, mais vous avez vu comment *concrétiser* vos connaissances en travaillant sur une librairie intéressante, la SDL.

S'il y en a parmi vous qui sont intéressés par la 3D, je vous recommande vivement de lire le cours sur OpenGL rédigé par Kayl. C'est une librairie graphique 3D dont vous avez sûrement déjà entendu parler. Kayl a plus d'expérience que moi dans le domaine de la 3D, il sait de quoi il parle et vous apprendrez une foule de choses intéressantes avec lui !
Notez que pour suivre son cours il faut avoir lu tout mon cours de C / C++ jusqu'à la partie III sur la SDL incluse (Kayl utilise la SDL et OpenGL en même temps, vous verrez)

Bien, attaquons maintenant la partie IV sur le C++ si vous le voulez bien

PARTIE 4 : [LANGAGE C++] LA PROGRAMMATION ORIENTÉE OBJET

Après avoir découvert le langage C dans les parties précédentes, nous nous intéressons maintenant au C++
Le langage C++ est basé sur le C : ce que vous avez donc appris jusqu'ici va vous resservir, pour ne pas dire vous être indispensable !

Les modifications entre le C et le C++ sont nombreuses. La plus importante d'entre elles est l'introduction de la **Programmation Orientée Objet**, que l'on abrège couramment **POO**. On en entend souvent parler, mais qu'est-ce que c'est concrètement ?
La réponse se trouve dans cette partie du cours

Introduction au C++

Le C++, *enfin* on y arrive !

La partie IV du cours démarre maintenant. J'espère que vous êtes encore en forme, parce qu'on n'en est qu'à la moitié du cours ! Il vous reste des tonnes de choses à apprendre (et pas des plus faciles). Mais courage, si vous êtes arrivés là, vous serez capable de réussir à comprendre la suite ça ne fait aucun doute

Cette partie marque un tournant dans le cours parce qu'à partir de maintenant nous allons programmer dans un autre langage : le C++. Ce langage ressemble au C en apparence, mais vous allez vite vous rendre compte qu'il est en fait bien différent.

Ce chapitre servira d'introduction. Nous allons commencer par voir ce qui différencie le C du C++ et quels sont les défauts du C qui ont conduit à la réalisation du C++.

POURQUOI AVOIR CRÉÉ LE C++ ?

Je vous l'ai dit dès le tout début du premier chapitre du cours, et je ne vous ai pas menti : le langage C n'est pas limité. Vous pouvez faire tout ce que vous voulez avec, des fenêtres (avec GTK+ par exemple), de la 2D (avec SDL), de la 3D (avec OpenGL)... Le tout est de trouver la librairie qui propose ce dont vous avez besoin.

Alors du coup on peut se demander... pourquoi un gars s'est levé un jour et a dit : "Aujourd'hui je vais inventer un nouveau langage" ?
Le type en question s'appelle **Bjarne Stroustrup**, il est danois et il est l'auteur d'un des langages de programmation les plus utilisés dans le monde aujourd'hui. Et il n'est pas fou : il n'a pas fait ça pour le plaisir (enfin j'espère) !

Le C a des avantages

Maintenant que vous programmez en C, vous avez pu vous rendre compte certainement que celui-ci a un grand

nombre d'avantages :

- C'est un langage **très rapide** car assez bas niveau. Si on a des calculs importants à faire, un programme écrit en C les exécute en moins de temps qu'il n'en faut pour dire "ouf" (sauf si vous avez codé avec les pieds bien entendu)
- C'est un langage **portable**. Le même code source peut être compilé aussi bien sous Windows, Linux, Mac OS et ne dépend d'aucun type de processeur particulier.
- Le langage est **libre**, ce qui permet à n'importe qui de (très) motivé d'écrire son propre compilateur. Cela explique donc la grande diversité des compilateurs aujourd'hui : GCC, mingw, MS Visual C++, Borland, et j'en passe.

Bref, autant de raisons valables d'aimer le C

Mais le C a aussi des défauts !

Si le C n'avait que des avantages, ce serait le langage parfait. Or, les programmeurs savent très bien que le langage parfait n'existe pas. Tout langage a des défauts.

Donc le C n'est pas parfait. Que lui reproche-t-on ?
Déjà, certains concepts de programmation plus récents manquent :

- Les **références**, qui permettent d'éviter quelques prises de tête avec les pointeurs.
- Les **exceptions**, une technique puissante pour gérer les erreurs de ses programmes.
- ... et il y en a d'autres mais ça ne sert à rien de rentrer dans le détail de suite.

Mais le *vrai* problème du langage C est qu'il n'est pas prévu pour faire de la **Programmation Orientée Objet**, une technique de programmation particulièrement efficace apparue récemment.

LA PROGRAMMATION ORIENTÉE QUOI ?

Non, ce n'est pas une insulte.
Bon d'accord, il faut avouer que quand quelqu'un nous dit : "Je fais de la programmation orientée objet", on a tendance à s'éloigner un peu de peur que ça soit contagieux

Beaucoup de gens parlent ou ont entendu parler de Programmation Orientée Objet (que j'abrègerai maintenant tout le temps POO, ça va plus vite). Mais concrètement, la POO c'est quoi ?

La POO est un concept de programmation, une façon de programmer. Ce n'est pas un langage.
Le C++, lui, est un langage. Il a été principalement inventé pour faciliter l'utilisation de la POO.

La POO n'est pas utilisée qu'en C++. De nombreux langages, encore plus récents, exploitent au maximum les concepts de la POO. Je pense en particulier à Java et Python, mais il y en a bien d'autres. Par ailleurs, il est possible de faire de la POO en C, mais c'est assez compliqué.

Bon, à quoi ça sert la POO ?

C'est une façon de programmer qui permet de rendre un code source plus facilement réutilisable, plus facile à modifier.

Ah non, je n'ai pas dit ça
Simplement, quand le programme devient gros, il faut avouer qu'on finit assez facilement par se perdre dans toutes les fonctions qu'on a créées. La POO nous permet de mieux organiser notre code source, de lui donner une certaine *logique*. Les avantages de cette meilleure organisation sont nombreux, vous les découvrirez progressivement.

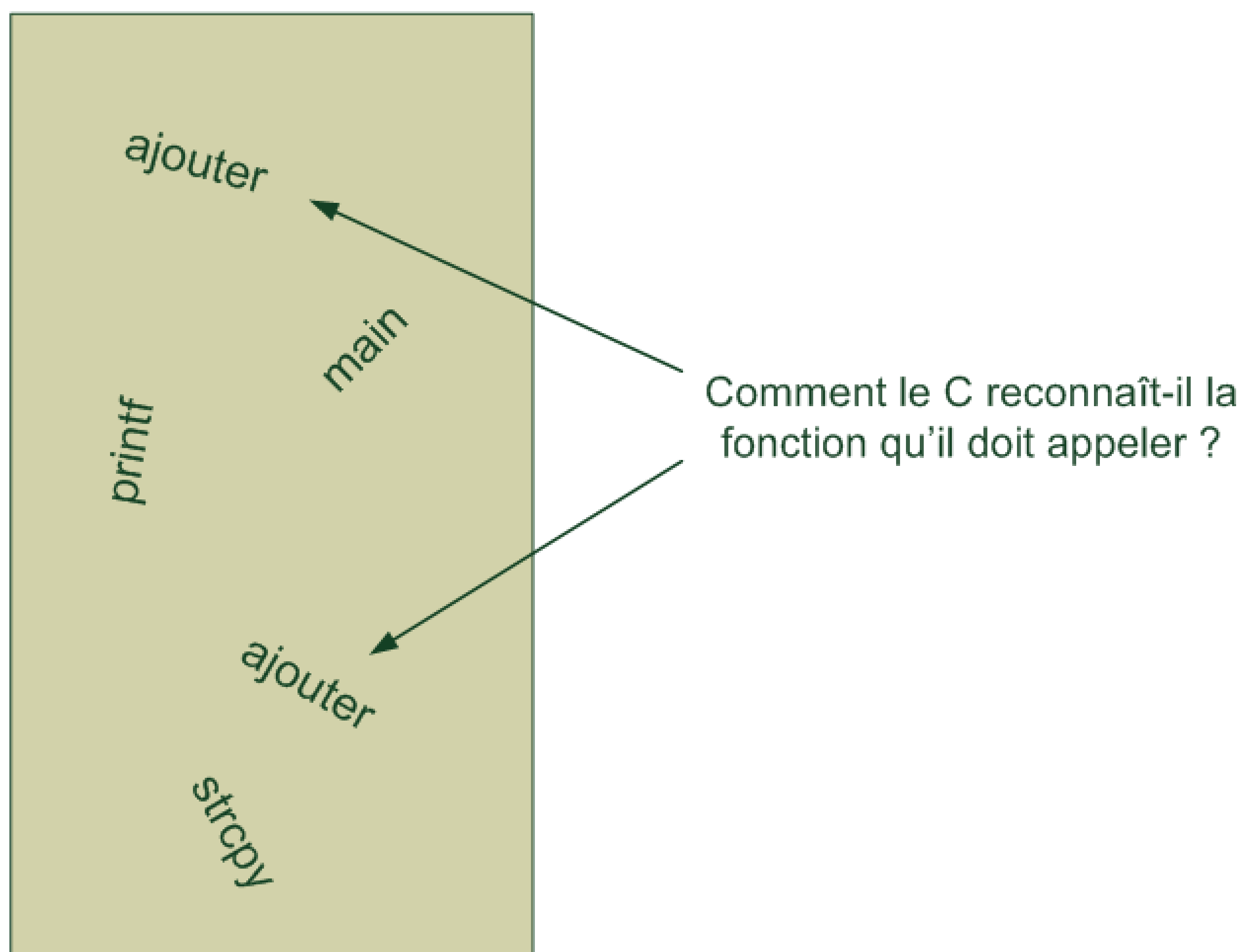
En fait, c'est un peu comme les pointeurs : vous n'en avez pas forcément compris l'intérêt tout de suite, mais je suis sûr que maintenant vous ne pouvez plus vous en passer !

L'idée à la base de la POO

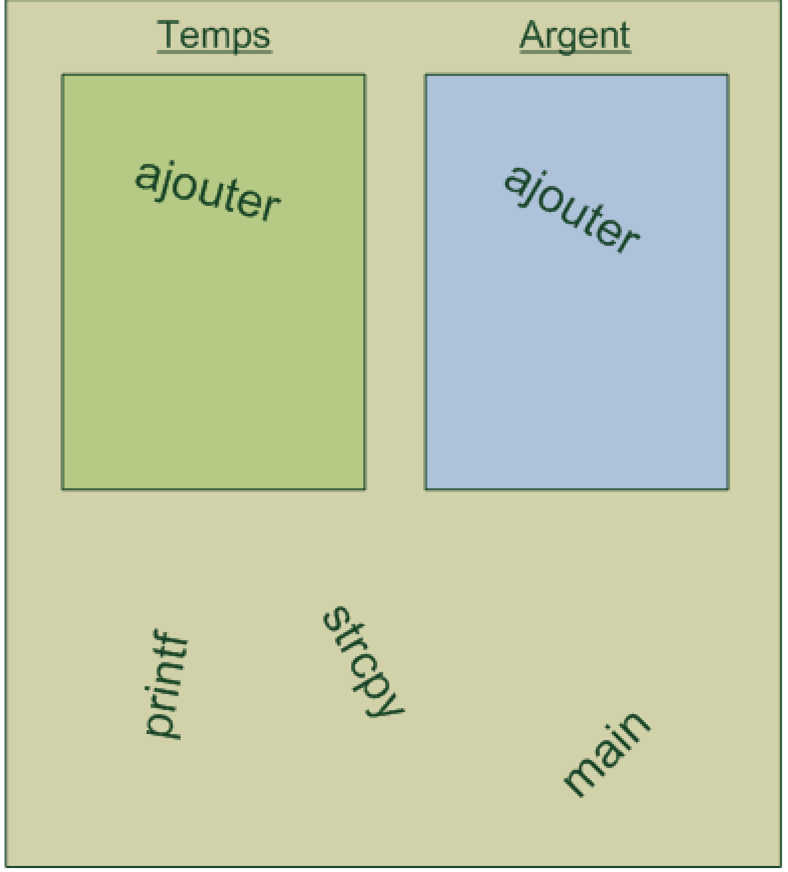
En C, vos programmes ne sont au final qu'un ensemble de fonctions accessibles de partout qui manipulent des tonnes de pointeurs. Même si vous pouvez faire plusieurs fichiers, cela ne suffit pas toujours à organiser correctement votre programme.

Par exemple, **vous ne pouvez pas avoir deux fonctions nommées `ajouter` dans votre programme**. Même si ces fonctions sont dans deux fichiers différents !
Ainsi, si vous avez une fonction `ajouter` permettant d'ajouter des heures et ailleurs une autre fonction `ajouter` permettant d'ajouter des euros, le langage C sera perdu et vous dira qu'il ne sait pas quelle fonction appeler quand vous demandez la fonction `ajouter`.

Imaginez que toutes les fonctions d'un programme en C nagent dans une seule et même grande piscine. C'est ce qu'on appelle **l'espace global**. Les fonctions évoluent toutes dans un même espace.



Imaginez maintenant si on séparait ces fonctions. On place la fonction `ajouter` spécialisée dans les heures dans une piscine, et on place la fonction `ajouter` spécialisée dans les euros dans une autre piscine. Du coup, vous n'avez plus qu'à vous rendre devant la piscine qui vous intéresse (par exemple la piscine spécialisée dans les heures) et vous pouvez alors appeler la fonction `ajouter`. Il n'y a pas de risque de conflit cette fois, parce que les fonctions sont confinées dans des espaces différents ! On ne les mélange plus



En C++, les fonctions sont compartimentées dans des espaces différents

Dans le schéma ci-dessus, j'ai créé 2 espaces pour séparer les fonctions `ajouter`. J'ai nommé ces espaces pour les identifier. On a maintenant :

- L'espace spécialisé dans la gestion du temps,
- L'espace spécialisé dans la gestion de l'argent.

Vous noterez que les autres fonctions comme *main* nagent encore dans l'espace global. Il est en effet toujours possible de mettre des fonctions dans l'espace global en C++.
D'autres langages interdisent carrément ce genre de choses (je pense à Java par exemple). Java est un langage complètement orienté objet : toutes les fonctions sont obligatoirement confinées dans des espaces particuliers et non dans l'espace global. Le C++ autorise encore de mettre des fonctions dans l'espace global, et c'est d'ailleurs en partie ce que certains lui reprochent (je vous rappelle que le langage parfait n'existe pas).

Allons un peu plus loin

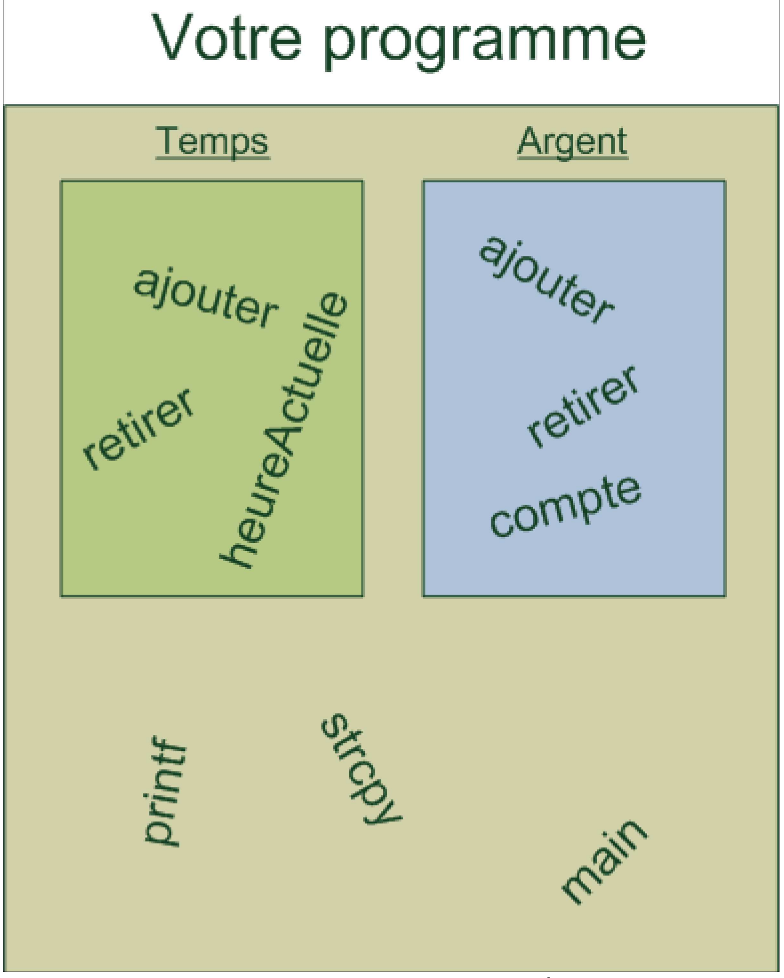
Est-ce qu'on ne peut mettre qu'une seule fonction par espace ?

Non, bien sûr ! Le but, c'est de regrouper les fonctions de notre programme par "thème".

Prenons par exemple le thème *Temps*.
On peut ajouter des heures, mais on pourrait aussi créer la fonction qui retire des heures, une autre qui donne l'heure actuelle, etc.

De même pour le thème *Argent*. Disons que ça c'est l'argent que vous avez.
Vous pouvez aussi retirer de l'argent, créer la fonction qui vous donne la quantité d'argent que vous avez sur votre compte en banque, etc.

Ajoutons ces fonctions dans notre schéma :



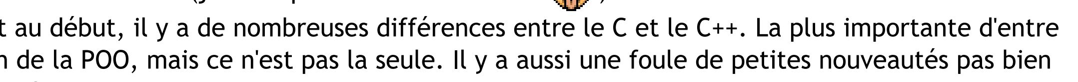
Regroupez vos fonctions par thème !

Vous noterez qu'il y a 2 fonctions *retirer* : là encore ça ne pose pas de problème car elles sont dans 2 espaces différents.

Eh bien, croyez-moi si vous voulez, mais si vous avez compris ce que je viens d'expliquer à l'instant, vous avez déjà une bonne petite idée de ce qu'est la **programmation orientée objet** !

Maintenant je vous rassure : ça c'est vraiment la base de la base. La POO implique pas mal de règles, et l'organisation est parfois un peu déroutante. On va parler de POO dans pratiquement toute cette partie IV du cours, vous aurez donc le temps de vous rendre compte à quel point le sujet est riche Comme la POO est un vaaaaste sujet, nous

n'allons pas l'aborder immédiatement (je veux pas vous tuer de suite).



Comme je vous l'ai dit au début, il y a de nombreuses différences entre le C et le C++. La plus importante d'entre elles est l'introduction de la POO, mais ce n'est pas la seule. Il y a aussi une foule de petites nouveautés pas bien compliquées à comprendre.

Voilà ce qu'on va faire :

1. Dans un premier temps nous allons découvrir toutes ces petites nouveautés qui n'ont *aucun* rapport avec la POO (ça prendra environ 3 chapitres).
2. Ensuite nous attaquerons la POO et je vous en ferai manger jusqu'à la fin de la partie IV

On attaque notre premier programme C++ dès le chapitre suivant. Nous y découvrirons la notion de flux d'entrée / sortie. Une sorte de... mise en bouche quoi

Premier programme C++ avec cout et cin

Après un bref chapitre d'introduction, nous pouvons commencer à coder nos premières lignes de C++

Ce chapitre sera assez simple : nous verrons quelles techniques on utilise en C++ pour afficher du texte à l'écran (dans une console) et comment on récupère du texte saisi au clavier. Vous allez voir que c'est assez différent ce qu'on connaissait en C avec *printf* et *scanf*.

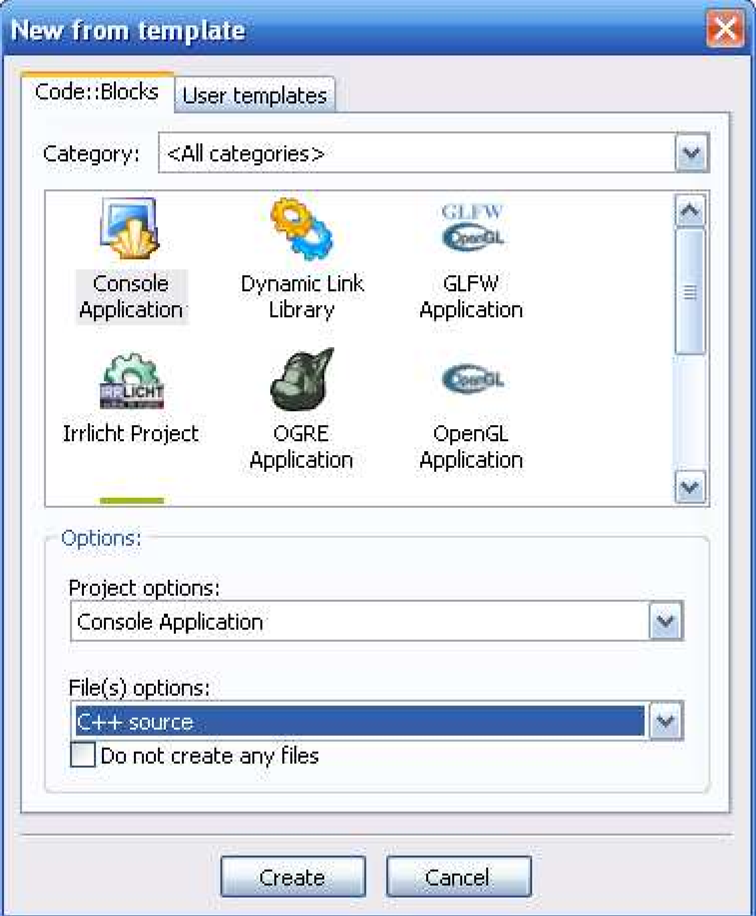
Nous réutiliserons cela dans toute la partie IV sur le C++. Soyez donc attentifs, et ça ne devrait pas vous poser de problème

Configurer l'IDE pour le C++

Jusqu'ici, vous n'avez créé dans votre IDE que des projets en C.
Or, en C++ on utilise un autre compilateur. Par exemple, il y a gcc qui est un compilateur C, et g++ qui est un compilateur C++. Il va donc falloir dire à votre IDE que vous allez faire du C++, sinon il appellera le mauvais compilateur (et là ça risque pas de marcher).

Lancez donc votre IDE favori. Pour ma part, vous l'aurez compris dans les chapitres précédents, je travaille principalement sous Code::Blocks. Si vous avez un autre IDE comme Visual C++ ou Dev C++ ça marche aussi

Créez un nouveau projet de type console (eh oui, on retourne à la console pour faire nos expériences) et pensez à bien sélectionner C++.



Création d'un nouveau projet. Veillez à bien sélectionner C++

Si vous avez Dev C++ la manipulation est la même, je ne vous refais pas de screenshot vous êtes grands maintenant
Sous Visual C++, le projet est par défaut compilé en C++, vous n'aurez donc pas besoin de spécifier quoi que ce soit.

Cliquez sur "Create" pour créer le nouveau projet.
Code::Blocks crée un premier fichier nommé `main.cpp` dans le projet avec quelques premières lignes de code C++.

En C++, vos fichiers .c ont l'extension .cpp. Les fichiers .h, eux, gardent, l'extension .h.
Certains trouvent cela illogique et ont choisi à la place d'utiliser .cc (pour les sources C++) et .hh (pour les headers C++). Ne soyez donc pas surpris par ce type de notation

Voici le code de base que nous propose Code::Blocks dans notre nouveau projet :

Code : C++

```
#include <iostream>

int main()
{
	std::cout << "Hello world!" << std::endl;
	return 0;
}
```

Si vous avez un autre IDE, supprimez le code qui a été généré et utilisez celui-ci à la place pour qu'on soit sûrs de travailler sur le même code

ANALYSE DU PREMIER CODE SOURCE C++

Intéressons-nous maintenant à chacune de ces lignes de code et voyons ce qui change pour le moment par rapport au C.

Include

Code : C++

```
#include <iostream>
```

On reconnaît là une bonne vieille directive de préprocesseur.

2 choses :

- Ce qui choque tout d'abord, c'est qu'**il n'y a pas d'extension .h**. En effet, en C++ les fichiers d'en-tête standard ne possèdent plus d'extension .h, mais vous verrez qu'il y a des exceptions (des gens qui ont pas encore fait l'évolution).
- D'autre part, la directive d'inclusion n'est plus la même. Ici, elle s'appelle iostream, ce qui signifie "flux d'entrée-sortie". Oubliez stdlib et stdio, ce sont des en-têtes du C, on ne les utilise plus en C++.

On inclut donc ici la librairie iostream qui contient les outils nécessaires pour afficher du texte dans la console et récupérer la saisie au clavier.

Fonction main()

Code : C++

```
int main()
{
    // ...
    return 0;
}
```

On retrouve notre fonction main habituelle. Cela fonctionne comme en C : tout programme commence par la fonction `main()`. Rien de choquant ici
La fonction retourne 0, ce qui est là encore logique puisque la fonction doit retourner un int.

Pour info, 2 formes de *main* sont possibles. Celle-ci :
`int main()`
... mais aussi cette forme un peu plus compliquée que vous connaissez aussi :
`int main(int argc, char *argv[])`
La seconde forme permet de récupérer les arguments d'appel du programme, ce qu'on ne fait pas toujours.
Vous pouvez donc vous contenter de la première forme qui est surtout plus simple à retenir

cout

Code : C++

```
std::cout << "Hello world!" << std::endl;
```

La première ligne du *main* est la plus intéressante, c'est d'ailleurs la seule qui doit vraiment vous surprendre.
En effet, ça ne ressemble pas à un appel de fonction, il y a plein de signes bizarres, pas de parenthèses comme on a l'habitude dans un appel de fonction. Bon sang de bonsoir qu'est-ce que c'est ?

On va découvrir ça maintenant plus en détails

LE FLUX DE SORTIE COUT

cout n'est pas une fonction mais un flux, un élément nouveau introduit en C++. Notez que ça n'a rien à voir avec la POO pour le moment

Rassurez-vous, les fonctions existent toujours en C++ (d'ailleurs vous avez vu qu'il y a un *main()*), mais vous vous rendrez compte petit à petit qu'on ne les utilise plus de la même manière.

Le flux *cout* est l'équivalent de la fonction printf... mais en mieux, en plus simple

Revoyons cette fameuse ligne de code :

Code : C++

```
std::cout << "Hello world!" << std::endl;
```

Il y a deux mots-clé particuliers dans cette ligne : cout et endl. Vous noterez qu'ils ont tous les deux le préfixe `std::`
Tous les mots-clé de la librairie standard du C++ utilisent ce préfixe. Théoriquement, on est obligé de le mettre à chaque fois, mais c'est un peu lourd.
On a heureusement une solution pour se simplifier la vie. Rajoutez cette ligne de code avant le *main()* :

Code : C++

```
using namespace std;
```

Du coup, vous pouvez virer tous les préfixes `std::`
Ca rend déjà notre code un peu plus facile à lire :

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
        cout << "Hello world!" << endl;
        return 0;
}
```

Voilà un premier point de réglé. Nous ne rentrerons pas dans le détail de ce "using namespace" pour le moment (pour ne pas compliquer inutilement les choses).

Intéressons-nous de plus près à cette ligne avec cout, désormais plus lisible :

Code : C++

```
cout << "Hello world!" << endl;
```

Vous voyez qu'il y a un nouveau symbole : le chevron <
On le rencontre d'ailleurs toujours par paire, comme ceci : <<

Imaginez que ces chevrons représentent en fait des flèches. Du coup, la ligne se lit de droite à gauche :

1. On prend le mot-clé endl, qui signifie *retour à la ligne*.
2. On l'insère à la fin de la chaîne "Hello world!"
3. On insère le résultat obtenu dans cout.

cout est la contraction de "c-out" (console out = sortie console). En bon français, vous devez prononcer cela "Ci Aoute"
cout représente la sortie en C++ (out = sortie). La sortie d'un programme, ben c'est tout simplement l'écran. Donc cout représente l'écran

Résultat des courses, ce code signifie que le texte "Hello world!" suivi d'un retour à la ligne est envoyé vers l'écran. Il faut bien imaginer que les chevrons << indiquent le sens dans lequel les données sont envoyées.

Cette ligne de code commande donc un affichage de texte à l'écran. Compilez et exécutez le programme pour voir :

Code : Console

```
Hello world!
```

Revenons un peu sur **endl**.
endl est un mot-clé qui signifie "fin de ligne" (*end line* en anglais). En fait, c'est tout bêtement un mot qui remplace le \n que vous connaissez du C, qu'on utilisait pour faire des sauts de ligne.

Ah bon, on ne peut plus utiliser \n en C++ ?

Si si. En fait, le mot-clé endl a été entre autres introduit pour améliorer la lisibilité du code source (pour ne pas qu'on mélange le "n" avec le texte à afficher).
Vous pouvez d'ailleurs tester, vous verrez que l'\n fonctionne toujours :

Code : C++

```
cout << "Hello world!\n";
```

Le résultat à l'écran est exactement le même :

Code : Console

```
Hello world!
```

Désormais, j'utiliserai *endl* à la place de \n dans la suite du cours.

L'intérêt de cout

Pour le moment, vous devez vous dire que cout ressemble étrangement à la fonction printf, avec juste des symboles << en plus pour vous embrouiller l'esprit

En fait, c'est encore plus facile à utiliser que printf. L'intérêt se voit notamment lorsqu'on veut afficher le contenu d'une variable.
Regardez ce code, c'est super simple :

Code : C++

```
int main()
{
    int age = 21;
    cout << "Salut, j'ai " << age << " ans" << endl;
    return 0;
}
```

Code : Console

```
Salut, j'ai 21 ans
```

Voilà, c'est assez intuitif pour que je n'aie pas besoin de vous expliquer comment ça marche

Ce qui est génial, c'est qu'on n'a plus besoin de s'embêter avec la syntaxe des printf : %d, %ld, %lf, %s, %c etc... Ici, le langage est plus intelligent, il reconnaît le type de variable qui lui est envoyé.

Sceptiques ? Ok, essayez d'ajouter une variable de type chaîne de caractère dans ce cout :

Code : C++

```
int main()
{
    int age = 21;
    char pseudo[] = "M@teo21";
    cout << "Salut, j'ai " << age << " ans et je m'appelle " << pseudo << endl;
    return 0;
}
```

Code : Console

```
Salut, j'ai 21 ans et je m'appelle M@teo21
```

Vous voyez, on a envoyé d'un coup à cout un entier et une chaîne de caractères, sans préciser le type de variable, et il n'a pas bronché

Pour rappel, en C on aurait dû faire :

Code : C

```
printf("Salut, j'ai %ld ans et je m'appelle %s\n", age, pseudo);
```

Non seulement il fallait se souvenir des codes %ld et %s, mais en plus les variables utilisées sont indiquées à la fin. En C++ avec cout, comme vous avez pu le constater, la variable est placée au milieu, ce qui rend le code plus facile à lire

Bien entendu, vous n'êtes pas limité à un seul cout par programme
Vous pouvez tout à fait faire plusieurs cout si vous le voulez :

Code : C++

```
cout << "Salut, j'ai " << age << " ans" << endl;
cout << "Je m'appelle " << pseudo << endl;
```

Résultat :

Code : Console

```
Salut, j'ai 21 ans
Je m'appelle M@teo21
```

Le endl à la fin de chaque cout n'est pas obligatoire. Si vous l'enlevez, il n'y aura juste pas de retour à la ligne.
Entraînez-vous un peu avec cout, vous devriez vous y habituer rapidement !

LE FLUX D'ENTRÉE CIN

Après avoir vu comment afficher du texte, voyons voir maintenant comment récupérer du texte saisi au clavier.
Là encore, ça fonctionne avec un système de flux, et vous allez voir que c'est un vrai régal de simplicité

Le mot-clé à connaître ici est **cin**. Il vient en remplacement de la pratique (mais complexe) fonction scanf du langage C.
cin est la contraction de "c-in", ce qui signifie *entrée console*. Prononcez ça comme il faut siouplaît : "Ci in"

cin représente l'entrée en C++. Et qu'est-ce que l'entrée ? C'est le clavier ! Eh oui, c'est par le clavier qu'on entre les données.

cin représente donc le clavier et permet de récupérer du texte saisi par l'utilisateur.
Tenez, on va faire un truc super original : on va lui demander son âge

Regardez comment ça fonctionne :

Code : C++

```cpp
cin >> age;
```

Si vous êtes un peu observateur, 2 choses doivent vous avoir choqué :

- **Les flèches ont changé de sens !** Eh oui, la lecture se fait ici de gauche à droite. cin représente le clavier et *envoie* les données dans la variable age. Il faut donc imaginer que les données transitent du clavier vers la variable age. C'est ce qu'on appelle un flux
- **Il n'y a pas de symbole & devant age !** En C, on aurait dû écrire &age pour envoyer l'adresse de la variable à la fonction pour qu'elle sache où écrire en mémoire. En C++, c'est plus la peine ! Il y a en effet un mécanisme qui remplace un peu les pointeurs qu'on appelle les références. On étudiera ça dans le prochain chapitre plus en détails.

Ne confondez pas le sens des flèches entre cout et cin. Le principe c'est que les données transitent dans un sens précis : de la mémoire vers l'écran (cout) ou du clavier vers la mémoire (cin).
Les flèches sont donc dans le sens de déplacement des informations. Avec un peu de bon sens, vous ne devriez pas vous tromper

Programmes de test de cin

Testons maintenant cin dans un petit programme. Voici le code complet :

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    int age = 0;
    cout << "Quel age avez-vous ?" << endl;
    cin >> age;
    cout << "Ah ! Vous avez donc " << age << " ans !" << endl;
    return 0;
}
```

Code : Console

```
Quel age avez-vous ?
21
Ah ! Vous avez donc 21 ans !
```

Désolé si je me répète, mais je trouve cela d'une simplicité effarante
Finis les %ld qui nous agacent, finis les oublis de symbole & dans les scanf, finis
Bon d'accord, cette nouvelle syntaxe surprend un peu quand on a fait pas mal de C avant, mais on s'y fait vite rassurez-vous.

On peut aussi s'entraîner à demander le pseudonyme de l'utilisateur :

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    char pseudo[50];
    cout << "Quel est votre pseudo ?" << endl;
    cin >> pseudo;
    cout << "Salut " << pseudo << endl;
    return 0;
}
```

Code : Console

```
Quel est votre pseudo ?
M@teo21
Salut M@teo21
```

Alors, premières impressions du C++ ?

"Ca change" ? Ah ben oui un peu, c'est sûr
En effet, la syntaxe des flux cout et cin surprend un peu... mais je suis sûr que vous serez convaincu comme moi que grâce à ce système de nombreuses choses sont simplifiées !

On va continuer notre tour d'horizon des ajouts-au-C++-qui-n'ont-rien-à-voir-avec-la-POO dans le chapitre suivant. Au programme, nous allons découvrir les changements au niveau de la gestion des variables. Nous verrons en particulier ce que sont ces mystérieuses **références** dont vous avez entendu parler.

Nouveautés pour les variables

Nous continuons notre tour d'horizon des nouveautés du C++ dans ce chapitre. Nous n'allons pas encore voir ici la POO, mais patience, ça ne saurait tarder

Nous nous intéresserons aux nouveautés relatives aux variables, c'est-à-dire à la gestion de la mémoire. Nous découvrirons entre autres le type bool, les modifications par rapport aux définitions de variables, les allocations dynamiques en C++ et les références.

Rien de bien difficile au programme donc, mais il s'agit de nouveautés importantes, donc à ne pas négliger.

LE TYPE BOOL

On a découvert dès le début du cours de C qu'il existait un grand nombre de types de variable différents :

- int
- long
- float
- double
- char
- etc.

Le problème s'est posé lorsqu'on a voulu stocker des **booléens**. Faute d'avoir un type de donnée spécialisé dans le stockage des booléens, on a fait comme la plupart des programmeurs C font : on a utilisé un type entier, comme int.

> Rappel. Un booléen est une variable qui peut prendre 2 valeurs : vrai *ou* faux. Par exemple, "majeur" est un booléen : soit on est majeur, soit on ne l'est pas
> On dit que le nombre 0 représente "Faux", tandis que le nombre 1 représente "Vrai" (en fait, tout nombre différent de 0 signifie "Vrai").

Or, si on utilise int pour les booléens, on risque de les confondre avec des variables destinées à stocker des nombres, puisque int est à la base fait pour stocker des nombres !

Petit exemple tout simple :

Code : C

```
int majeur = 1;
int age = 21;
```

La variable *majeur* est un booléen, car elle signifie soit vrai soit faux.
La variable *age*, elle, est un nombre. Elle peut valoir par exemple 21.

Mais comment fait-on pour savoir laquelle de ces variables est un booléen et laquelle est un nombre ?
On peut se baser sur le nom de la variable, c'est sûr, mais il aurait été plus pratique et plus clair d'avoir un type spécial pour les booléens.

Ca tombe bien ! Il y a justement en C++ un nouveau type de base : le type **bool**. Toute variable de ce type peut prendre 2 valeurs :

- **true**, qui signifie vrai.
- **false**, qui signifie faux.

(je vous conseille de retenir ces 2 valeurs par coeur, vous en aurez besoin *)*

Du coup, le code qu'on a vu plus haut s'écrirait comme ceci en C++ :

Code : C++

```
bool majeur = true;
int age = 21;
```

Voilà une bonne chose qui nous permettra d'éviter des ambigüités dans nos programmes

Il n'y a pas de guillemets autour de true, car c'est un mot-clé du langage C++. Ce n'est pas une chaîne de caractères !

Rappel : les booléens dans les conditions

Je tiens juste à vous faire un petit rappel. Si vous avez bien suivi le cours de C, ça ne devrait pas vous choquer

En théorie, on peut tester un booléen comme ceci dans une condition :

Code : C++

```
if (majeur == true) // S'il est majeur (forme longue)
{
    // ...
}
```

Mais en général, si la variable a un nom clair, on préfèrera enlever la partie `== true`. C'est tout à fait possible et l'ordinateur le comprend très bien :

Code : C++

```
if (majeur) // S'il est majeur (forme courte)
{
    // ...
}
```

Ce code est plus lisible et plus court que le précédent. On comprend bien que la condition est " *S'il est majeur*".

Par ailleurs, le point d'exclamation sert à exprimer la négation. Dans notre cas, ce code signifierait " *S'il n'est pas majeur*" :

Code : C++

```
if (!majeur) // S'il n'est PAS majeur
{
    // ...
}
```

Ce n'est pas une nouveauté du C++ car ça existait déjà en C, mais je tenais juste à vous informer que cette technique fonctionnait toujours avec le type **bool**

LES DÉCLARATIONS DE VARIABLES

En C, les variables devaient être déclarées (= créées) au début des fonctions. Vous avez vu cela dans le chapitre sur

les variables au tout début du cours

Vous deviez donc faire toutes vos déclarations avant de commencer les instructions :

Code : C

```
void maFonction()
{
    // D'abord on déclare les variables
    double prixOrigine = 0.0;
    double prixAchat = 0.0;
    double difference = 0.0;
    FILE* fichier = NULL;

    // Ensuite on peut exécuter des instructions, des appels de fonction, etc.
    fichier = fopen("exemple.txt", "r");

    if (fichier != NULL)
    {
        fonction();

        fscanf(fichier, "%lf", &prixOrigine);
        fscanf(fichier, "%lf", &prixAchat);

        difference = prixAchat - prixOrigine;
        // etc.
    }
}
```

La nouveauté en C++, c'est que l'on peut désormais déclarer des variables *n'importe où* dans une fonction. C'est plus pratique lorsqu'on programme, ça nous évite d'avoir à remonter au début de la fonction si on n'a besoin d'une variable qu'à un moment de la fonction. Cela peut aussi améliorer la lisibilité du code surtout dans de grosses fonctions.

On pourrait donc écrire le code précédent comme ceci en C++ :

Code : C++

```
void maFonction()
{
    FILE* fichier = NULL;
    fichier = fopen("exemple.txt", "r");

    if (fichier != NULL)
    {
        fonction();

        double prixOrigine = 0.0; // Déclaration au milieu
        fscanf(fichier, "%lf", &prixOrigine);

        double prixAchat = 0.0; // Autre déclaration au milieu
        fscanf(fichier, "%lf", &prixAchat);

        double difference = prixAchat - prixOrigine; // Encore autre déclaration au
milieu
        // etc.
    }
}
```

Avec une version récente du langage C, il est aussi possible de déclarer une variable en plein milieu d'une fonction. Cependant, les programmeurs C préfèrent en général continuer à déclarer leurs variables au début des fonctions.

Précision importante : les variables ainsi créées sont *locales* aux blocs où elles ont été déclarées. Je m'explique.
On dit que les accolades { et } délimitent des **blocs**. Dans le code ci-dessus, vous devriez en voir deux : la fonction et le bloc if. Comme la variable *prixAchat* a été déclarée dans le bloc if, elle sera supprimée à la fin du bloc if. Si elle avait été déclarée au début de la fonction en revanche, elle aurait été accessible dans toute la fonction.

Voilà, c'est assez simple à comprendre mais il faut le savoir ! La variable est détruite à la fin du bloc dans lequel elle a été déclarée.

Code : C++

```
void maFonction()
{
    FILE* fichier = NULL;
    fichier = fopen("exemple.txt", "r");

    if (fichier != NULL)
    {
        fonction();

        double prixOrigine = 0.0; // Création de prixOrigine
        fscanf(fichier, "%lf", &prixOrigine);

        double prixAchat = 0.0; // Création de prixAchat
        fscanf(fichier, "%lf", &prixAchat);

        double difference = prixAchat - prixOrigine; // Création de difference
    } // Destruction automatique de prixOrigine, prixAchat et difference
} // Destruction automatique de fichier
```

Déclaration dans une boucle

Dans le même ordre d'idée, il y a une nouveauté vraiment très pratique (comprenez : je m'en sers tout le temps). On peut déclarer une variable directement *dans* une instruction for.

Prenons un exemple. Vous codez votre programme, tout va bien. Puis à un moment, pour une raison ou une autre, vous avez besoin de faire une boucle qui se répète 10 fois. Vous allez sûrement faire un for. Mais pour boucler 10 fois, vous aurez besoin d'une variable de boucle qui va retenir le nombre de tours de boucle (quand on n'est pas inspiré on appelle en général cette variable **i**).

En C, c'est un peu embêtant parce qu'il faut remonter au début de la fonction pour rajouter la déclaration de la variable. En plus, on ne sait pas trop quand elle sera utilisée en lisant la déclaration :

Code : C

```
void maFonction()
{
    int i = 0;

    /* Plein de code
    ....
    ....
    */

    for (i = 0 ; i < 10 ; i++)
    {

    }
}
```

La nouveauté en C++, c'est que vous pouvez déclarer votre variable i directement dans l'instruction for. Elle sera détruite à la fin de la boucle, quand vous n'en aurez plus besoin.
Avantages : vous n'avez pas à remonter au début de la fonction pour déclarer la variable, et celle-ci est automatiquement détruite à la fin de la boucle. Pas d'utilisation inutile de la mémoire.

Le code C++ ressemblera donc à cela :

Code : C++

```
void maFonction()
{
    /* Plein de code
    ....
    ....
    */

    for (int i = 0 ; i < 10 ; i++) // Déclaration de i
    {

    } // Destruction automatique de i
}
```

Ca n'a l'air de rien, mais je vous assure qu'en pratique quand on programme, ça c'est vraiment génial
Vous me verrez donc le faire la plupart du temps dans la suite du cours.

LES ALLOCATIONS DYNAMIQUES

Si je vous dis "malloc" et "free", ça vous rappelle de joyeux souvenirs non ?

Code : C

```
int main()
{
    int *variable = NULL;

    variable = malloc(sizeof(int)); // Allocation de mémoire

    free(variable); // Libération de mémoire

    return 0;
}
```

L'allocation dynamique est une technique qui permet de gérer vous-même l'allocation de mémoire pour vos variables.
C'est notamment très pratique dans le cas de l'allocation de tableaux dont on ne connaît pas la taille avant compilation (revoyez le **chapitre sur l'allocation dynamique** au besoin !).

En C++, les allocations dynamiques existent toujours et on en fait toujours. D'ailleurs, les fonctions malloc et free sont toujours utilisables. Cependant, le C++ dispose de nouveaux opérateurs spécialisés dans les allocations dynamiques : **new** et **delete**.

new et delete sont des opérateurs, des mots-clé du langage C++. Contrairement à malloc et free, ce ne sont pas des fonctions.
new et delete font en fait eux-mêmes appel aux fonctions malloc et free (on n'a pas réinventé la roue).
Cependant, ils font aussi des tests et des initialisations supplémentaires, ce qui fait qu'on préfèrera toujours utiliser new et delete au lieu de malloc et free. Ils sont plus adaptés en C++.

Allocation dynamique d'une variable

new et delete ne s'utilisent pas exactement de la même manière que malloc et free.
On va dans un premier temps apprendre à s'en servir pour allouer une variable simple, puis on verra ensuite le cas de l'allocation de tableaux.

On souhaite donc allouer dynamiquement une variable (de type int par exemple).
En C++, on va d'abord devoir créer le pointeur et l'initialiser à NULL, ça on n'y coupe pas :

Code : C++

```
int *variable = NULL;
```

Allocation de mémoire

L'allocation de mémoire avec new se fait comme ceci :

Code : C++

```
variable = new int; // Allocation dynamique
```

Comparé à la "version C", il n'y a pas photo
On n'a plus besoin d'utiliser l'opérateur sizeof() du C. Ici, on indique juste le type de variable à créer.

Libération de mémoire

Lorsque vous avez fini d'utiliser votre variable et que vous n'en avez plus besoin, vous devez la libérer avec l'opérateur delete. Ultra-simple :

Code : C++

```
delete variable; // Libération de mémoire
```

new et delete étant des opérateurs, et non des fonctions (désolé d'insister), on ne met pas de parenthèses.

Résumé

En résumé, voici à quoi ressemble un code d'allocation / libération de mémoire en C++ :

Code : C++

```
int main()
{
    int *variable = NULL;

    variable = new int; // Allocation de mémoire

    delete variable; // Libération de mémoire

    return 0;
}
```

Allocation dynamique d'un tableau

Si on veut allouer un tableau, l'opération est là encore très simple. On n'a plus besoin de faire un calcul du type 20 * sizeof(int) comme on devait le faire en C.

On commence par créer le pointeur :

Code : C++

```
int *tableau = NULL;
```

Allocation de mémoire

Ensuite, l'allocation se fait comme ceci :

Code : C++

```
tableau = new int[20]; // Allocation de mémoire (20 cases)
```

Dans ce cas, un tableau de 20 cases sera alloué. Bien entendu, il est aussi possible de remplacer ce nombre par une variable :

Code : C++

```
tableau = new int[taille]; // Allocation de mémoire ("taille" cases)
```

La longueur du tableau sera définie par la valeur de la variable *taille*.

Libération de mémoire

Lorsque vous n'avez plus besoin du tableau, vous devez le libérer... avec cette fois l'opérateur delete[] pour bien préciser qu'il s'agit d'un tableau. Vous n'avez pas besoin de préciser la taille entre crochets, mais n'oubliez pas ces crochets ils sont importants.

Code : C++

```
delete[] tableau; // Libération de mémoire
```

Résumé

Code : C++

```
int main()
{
    int *tableau = NULL;

    tableau = new int[20]; // Allocation de mémoire (tableau)

    delete[] tableau; // Libération de mémoire (tableau)

    return 0;
}
```

Il y a donc en tout 4 opérateurs :

- new s'utilise avec delete pour allouer une variable
- new[] s'utilise avec delete[] pour allouer un tableau

LE TYPEDEF AUTOMATIQUE

Vous souvenez-vous du chapitre sur les **structures et énumérations** ? :)
On y avait appris à créer nos propres types de variables. On avait notamment utilisé l'exemple d'une structure nommée *Coordonnees* :

Code : C

```
struct Coordonnees
{
    int x;
    int y;
};
```

Le problème des structures en C, c'est qu'il fallait placer le mot-clé struct au début de chaque déclaration d'une variable de type personnalisé :

Code : C

```
struct Coordonnees point;
```

Pour éviter d'avoir à répéter ce mot à chaque déclaration, on avait découvert l'instruction **typedef** qu'on utilisait comme ceci avant la définition de notre structure :

Code : C

```
typedef struct Coordonnees Coordonnees; // typedef permet d'éviter d'avoir à taper
"struct" à chaque déclaration

struct Coordonnees
{
    int x;
    int y;
};
```

Du coup, on pouvait déclarer une variable sans avoir à écrire `struct` devant :

Code : C

```
Coordonnees point; // Le mot-clé struct est inutile grâce au typedef
```

La nouveauté

En C++, qu'on se rassure, les structures existent toujours (il y a même encore mieux, mais n'anticipons pas).

La nouveauté du C++, c'est que le typedef est désormais automatique. A chaque fois que l'on déclare une structure (ou une énumération), un typedef est réalisé automatiquement par le compilateur. On peut donc n'écrire que l'instruction de déclaration de la structure :

Code : C++

```
// Le typedef est réalisé automatiquement par le compilateur, pas besoin de l'écrire

struct Coordonnees
{
    int x;
    int y;
};
```

Grâce à cela, le mot-clé `struct` devient totalement inutile lors d'une déclaration de variable :

Code : C++

```
Coordonnees point; // Le mot-clé struct est inutile grâce au typedef automatique
```

LES RÉFÉRENCES

Nous arrivons maintenant au point le plus important (et délicat) de ce chapitre. Ouvrez grandes vos oreilles (ou plutôt vos yeux).

Le C++ introduit un nouveau concept : **les références**. Une référence est un synonyme d'une autre variable. On verra ce que ça veut dire un peu plus loin
Vous allez voir que les références ressemblent beaucoup aux pointeurs. Elles ont en effet été créées pour simplifier l'utilisation des pointeurs. Attention toutefois : je vous préviens qu'au début vous risquez de confondre les références avec les pointeurs (c'est assez perturbant quand on voit ça la première fois j'avoue).

Les références à l'intérieur d'une fonction

Pour créer une référence, on doit utiliser le symbole & dans la déclaration :

Code : C++

```cpp
int &referenceSurAge;
```

Attention à ne pas confondre !
Dans une déclaration, le symbole & signifie "Je veux créer une référence" (c'est ce qu'on découvre maintenant). Partout ailleurs, le symbole & signifie "Je veux obtenir l'adresse de cette variable" (ça on l'avait déjà vu).
On confond facilement quand on débute. Il faut dire que les programmeurs n'ont pas été très malins en réutilisant le symbole & ici, y'a rien de tel pour confondre
Quand vous voyez un & désormais, vérifiez s'il se trouve dans une déclaration : si c'est dans une déclaration, c'est qu'on cherche à créer une référence, sinon c'est qu'on demande à obtenir l'adresse de la variable.

Bon, on a créé une référence. Et alors ?
Et alors si vous compilez le code ci-dessus, le compilateur va vous insulter poliment :

Citation : Compilateur C++

> error: 'referenceSurAge' declared as reference but not initialized

Si vous lisez l'anglais (et si vous ne le lisez pas vous devriez), vous avez compris le problème : le compilateur veut qu'on initialise immédiatement la référence.
Et ça c'est très important : une référence doit être immédiatement initialisée dès le début, contrairement aux pointeurs. Et ce n'est pas tout : une fois initialisée, la référence ne pourra plus changer !

Il y a donc deux règles que j'aimerais que vous reteniez par coeur :

- **Règle 1** : une référence doit être initialisée dès sa déclaration.
- **Règle 2** : une fois initialisée, une référence ne peut plus être modifiée.

Initialisation d'une référence

On va donc initialiser notre référence.
Comme je vous l'ai dit un peu plus tôt, une référence est un synonyme d'une autre variable. Cela veut donc dire qu'il faut créer une autre variable pour y trouver un minimum d'intérêt

Allez hop, il est l'heure de ressortir la bonne vieille variable qui a fait ses preuves : la variable... age !
(le premier qui ose dire que je fais des cours pas originaux il va tâter de mon sabre)

Code : C++

```
int age = 21; // Déclaration de la variable age (rien de nouveau)
int &referenceSurAge = age; // Déclaration et initialisation d'une référence sur la
variable age
```

Pour initialiser une référence, vous avez juste besoin d'écrire le nom de la variable dont elle sera le synonyme. Pas besoin d'écrire &age comme on le faisait avant avec les pointeurs.

Je vous avais prévenu, vous risquez de confondre avec les pointeurs.
Je vous ferai un résumé comparatif pointeurs / références un peu plus loin pour que vous puissiez bien les comparer.

Utilisation de la référence

Bon, maintenant notre référence est créée. On a un synonyme de la variable age.
Comment on s'en sert concrètement ? Exactement comme la variable age ! Pas besoin de mettre une étoile * devant pour dire qu'on veut obtenir la valeur. Les références permettent, vous allez le voir, de simplifier l'écriture de nos programmes pour éviter au maximum les erreurs (un oubli d'une étoile est si vite arrivé !).

Regardez ce petit programme complet qui affiche la variable age, la modifie, et la réaffiche, le tout en passant par une référence :

Code : C++

```
int main()
{
    int age = 21;
    int &referenceSurAge = age;

    cout << referenceSurAge << endl;
    cout << age << endl;

    referenceSurAge = 40;

    cout << referenceSurAge << endl;
    cout << age << endl;


    return 0;
}
```

Résultat :

Code : Console

```
21
21
40
40
```

Comme vous pouvez le voir, une référence s'utilise exactement comme la variable d'origine. C'est le compilateur qui fait la "conversion" et qui sait qu'il doit affecter la variable "age" lorsqu'on travaille avec la référence.

Comparatif pointeur / référence

En C++, les pointeurs existent toujours. Les références sont juste une alternative aux pointeurs. Elles ont surtout l'avantage d'être plus simples à utiliser, mais elles ne peuvent pas les remplacer complètement.
Pourquoi ? On l'a vu : une référence ne peut pas faire référence à une nouvelle variable une fois qu'elle a été

initialisée. Un pointeur, lui, peut toujours pointer vers une nouvelle variable au cours de l'exécution du programme.

Dans certains langages récents, comme le Java, les pointeurs ont complètement disparu. On n'utilise plus que des références, ce qui limite beaucoup les risques d'erreur et simplifie les programmes. La différence, c'est qu'en Java on peut modifier les références en cours de route, contrairement au C++

Il est très courant de confondre les pointeurs et les références lorsqu'on débute (si ça peut vous rassurer, moi aussi j'ai pas mal confondu au début). Je vais donc vous donner 2 codes source : le premier utilise les pointeurs, le second les références. Si à un moment vous avez un doute et que vous vous mettez à confondre pointeurs et références, servez-vous de l'exemple ci-dessous pour vous assurer que vous faites les choses correctement :

-------- Code d'exemple avec un pointeur --------

Code : C++

```
int main()
{
    int age = 21;
    int *pointeurSurAge = &age;

    cout << *pointeurSurAge;

    *pointeurSurAge  = 40;

    cout << *pointeurSurAge;


    return 0;
}
```

-------- Code d'exemple avec une référence --------

Code : C++

```
int main()
{
    int age = 21;
    int &referenceSurAge = age;

    cout << referenceSurAge;

    referenceSurAge  = 40;

    cout << referenceSurAge;


    return 0;
}
```

Voilà, j'espère que ce comparatif vous permettra d'y voir plus clair
Ce qu'il faut retenir dans l'histoire, c'est que les références sont là pour simplifier l'écriture du code source. Comme on n'a plus besoin d'utiliser le symbole * à chaque fois qu'on veut accéder à la variable age, on minimise les risques d'erreur dans nos programmes.

Les références vers des structures

Si vous faites une référence vers une structure, il faudra utiliser le symbole point "." et non le symbole flèche "->" lorsque vous voulez accéder à un élément d'une structure.

Code : C++

```
struct Coordonnees
{
    int x;
    int y;
};

int main()
{
    Coordonnees point;
    Coordonnees &referenceSurPoint = point;

    referenceSurPoint.x = 10;
    referenceSurPoint.y = 5;

    cout << "x : " << referenceSurPoint.x << endl;
    cout << "y : " << referenceSurPoint.y << endl;


    return 0;
}
```

Code : Console

```
x : 10
y : 5
```

Une fois de plus, vous voyez qu'une référence s'utilise *exactement* comme une variable

Les références lors d'un appel de fonction

Les codes qu'on a vus jusqu'ici n'étaient pas très utiles. En pratique, on n'est pas suffisamment maso pour créer des références juste "pour le plaisir" si elles ne sont pas indispensables.

En fait, comme pour les pointeurs, les références révèlent toute leur utilité lorsqu'on appelle une fonction.

Voyons voir ça dans un exemple :

Code : C++

```
struct Coordonnees
{
    int x;
    int y;
};

void remiseAZero(Coordonnees &pointAModifier);

int main()
{
    Coordonnees point;

    remiseAZero(point); // Pas besoin d'indiquer l'adresse de point avec un & lors de
l'appel

    return 0;
}

void remiseAZero(Coordonnees &pointAModifier) // La fonction indique qu'elle récupère une
référence
{
    // La référence s'utilise exactement comme une variable
    // On utilise donc des points "." et non des flèches "->"
    pointAModifier.x = 0;
    pointAModifier.y = 0;
}
```

On transmet la référence à la fonction RemiseAZero le plus simplement du monde, sans avoir à mettre de symbole &.

Code : C++

```
remiseAZero(point);
```

Le but des références est là encore très clair : éviter d'avoir à taper des symboles en plus pour minimiser les erreurs.

La fonction doit bien préciser qu'elle reçoit une référence. On doit donc placer le symbole & dans la déclaration (et dans le prototype) :

Code : C++

```
void remiseAZero(Coordonnees &pointAModifier)
```

Ensuite, à l'intérieur de la fonction, on se sert de la référence comme si c'était une variable (dans le cas présent, on

utilise donc le symbole point et non la flèche ->) :

Code : C++

```
pointAModifier.x = 0;
pointAModifier.y = 0;
```

Comparaison pointeur / référence

Une fois de plus, je crois qu'il est utile que je vous fasse un comparatif du même code utilisant d'un côté un pointeur, de l'autre une référence.

<table>
<tr><th>---------------------- Code d'exemple avec un pointeur -------------------</th><th>---------------- Code d'exemple avec une référence ---------------------</th></tr>
<tr><td>

Code : C++

```
int main()
{
    Coordonnees point;

    remiseAZero(&point);

    return 0;
}

void remiseAZero(Coordonnees
*pointAModifier)
{
    pointAModifier->x = 0;
    pointAModifier->y = 0;
}
```

</td><td>

Code : C++

```
int main()
{
    Coordonnees point;

    remiseAZero(point);

    return 0;
}

void remiseAZero(Coordonnees
&pointAModifier)
{
    pointAModifier.x = 0;
    pointAModifier.y = 0;
}
```

</td></tr>
</table>

Ces codes fonctionnent tous deux très bien en C++. Autant que possible, on utilisera des références en C++, sauf quand l'utilisation d'un pointeur est obligatoire.

Pour ceux qui se posent la question : on aurait tout à fait pu appeler la référence de la fonction "remiseAZero" *point* au lieu de *pointAModifier*. Il n'y a pas de risque de conflit avec la variable *point* du main car elle se trouve dans une autre fonction. J'ai juste changé le nom pour que vous évitiez de les confondre

A retenir : s'il y a un code que vous devez retenir pour les références, c'est celui de l'appel d'une fonction utilisant une référence (celui que nous venons de voir). Dans 99,99% des cas, on utilise les références lorsqu'on fait appel à une fonction. *Que de nouveautés !* C'est le moins qu'on puisse dire
Et encore, vous n'avez pas tout vu ! Dans le prochain chapitre, nous découvrirons les nouveautés du C++ relatives aux fonctions.

Et après... après, je pense qu'on pourra commencer à parler de POO

Tout ce que nous avons découvert dans ce chapitre est utile et sera largement utilisé par la suite. Prenez le temps de vous familiariser avec. Faites en particulier quelques tests et exercices avec les références car c'est un peu délicat au début, vu qu'on les mélange facilement avec les pointeurs. Heureusement, avec un peu d'expérience, on ne se trompe plus

Nouveautés pour les fonctions

Nous avons vu que le C++ proposait de nombreuses nouveautés relatives pour les variables.
Ce chapitre est la suite du précédent, mais est cette fois axé sur les nouveautés relatives aux fonctions.

Ce chapitre sera un peu plus court car il y a assez peu de changements au final. Ne vous endormez pas pour autant parce que vous allez découvrir les valeurs par défaut et les fonctions surchargées, deux éléments très importants que nous réutiliserons largement dans la suite.

Courage, c'est le dernier chapitre avant la POO (*ou plutôt devrais-je dire : "Profitez-en bien mes petits !* *").*

Des valeurs par défaut pour les paramètres

Si je vous dis "paramètre de fonction", vous voyez de quoi je parle n'est-ce pas ?
Je l'espère, parce qu'il serait temps de le savoir à votre niveau maintenant

Bon allez, un petit rappel !

Comme un petit rappel ne fait jamais de mal, voici un exemple de fonction :

Code : C++

```
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}
```

Cette fonction calcule le nombre de secondes en additionnant les heures, minutes et secondes qu'on lui envoie. Rien de bien compliqué

Les variables *heures*, *minutes* et *secondes* sont les **paramètres** de la fonction nombreDeSecondes. Ce sont des valeurs qu'elle reçoit, celles avec lesquelles elle va travailler.
Il est facile de reconnaître les paramètres d'une fonction, car ceux-ci se trouvent toujours écrits entre les parenthèses

Les valeurs par défaut

La nouveauté en C++, c'est qu'on peut donner des valeurs par défaut à certains paramètres de nos fonctions. Ainsi, on ne sera pas obligé d'indiquer à chaque fois tous les paramètres lorsqu'on appelle une fonction !

Pour bien voir comment on doit procéder, on va regarder le code complet. J'aimerais que vous le copiez dans votre IDE pour faire les tests en même temps que moi :

Code : C++

```
#include <iostream>

using namespace std;

// Prototype de la fonction
int nombreDeSecondes(int heures, int minutes, int secondes);

// Main
int main()
{
    cout << nombreDeSecondes(1, 10, 25) << endl;

    return 0;
}

// Définition de la fonction
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}
```

Ce code donne le résultat suivant :

Code : Console

```
4225
```

Sachant qu'1 heure = 3600s, 10 minutes = 600s, 25 secondes =... 25s, le résultat est logique car 3600 + 600 + 25 = 4225
Bref, tout va bien.

Maintenant supposons que l'on veuille rendre certains paramètres facultatifs, par exemple parce qu'on utilise en pratique plus souvent les heures que le reste.
On va devoir modifier le prototype de la fonction (et non sa définition, attention).

Indiquez la valeur par défaut que vous voulez donner aux paramètres si on ne les a pas renseigné lors de l'appel de la fonction :

Code : C++

```
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);
```

Dans cet exemple, seul le paramètre heures sera obligatoire, les deux autres étant désormais facultatifs. Si on ne renseigne pas les minutes et les secondes, les variables vaudront alors 0 dans la fonction.

Voici le code complet que vous devriez avoir sous les yeux :

Code : C++

```
#include <iostream>

using namespace std;

// Prototype avec les valeurs par défaut
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);

// Main
int main()
{
    cout << nombreDeSecondes(1, 10, 25) << endl;

    return 0;
}

// Définition de la fonction, SANS les valeurs par défaut
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}
```

Si vous avez lu attentivement ce code, vous avez dû vous rendre compte de quelque chose : les valeurs par défaut sont spécifiés uniquement dans le prototype, PAS dans la définition de la fonction ! On se fait souvent avoir, je vous préviens
Si vous vous trompez, le compilateur vous indiquera une erreur à la ligne de la définition de la fonction.

Bon, ce code ne change pas beaucoup du précédent. A part les valeurs par défaut dans le prototype, rien n'a été modifié (et le résultat à l'écran sera toujours le même).
La nouveauté maintenant, c'est qu'on peut supprimer des paramètres lors de l'appel de la fonction (ici dans le *main*).
On peut par exemple écrire :

Code : C++

```
cout << nombreDeSecondes(1) << endl;
```

Le compilateur lit les paramètres de gauche à droite. Comme il n'y en a qu'un et que seules les heures sont obligatoires, il devine que la valeur "1" correspond à un nombre d'heures.

Le résultat à l'écran sera le suivant :

Code : Console

```
3600
```

Mieux encore, vous pouvez indiquer juste les heures et les minutes si vous le désirez :

Code : C++

```
cout << nombreDeSecondes(1, 10) << endl;
```

Code : Console

```
4200
```

Du temps que vous indiquez au moins les paramètres obligatoires, il n'y a pas de problème

Cas particuliers, attention danger

Bon, mine de rien il y a quand même quelques pièges, ce n'est pas si simple que ça
On va voir ces pièges sous la forme de questions / réponses :

Et si je veux envoyer à la fonction juste les heures et les secondes, mais pas les minutes ?

Tel quel, c'est impossible. En effet, je vous l'ai dit plus haut, le compilateur va analyser les paramètres de gauche à droite. Le premier correspondra forcément aux heures, le second aux minutes et le troisième aux secondes.

Vous ne pouvez PAS écrire :

Code : C++

```cpp
cout << nombreDeSecondes(1,,25) << endl;
```

C'est interdit. Si vous le faites, le compilateur vous fera comprendre qu'il n'apprécie guère vos manoeuvres. C'est comme ça : en C++, on ne peut pas "sauter" des paramètres, même s'ils sont facultatifs. Si vous voulez indiquer le premier et le dernier paramètre, il vous faudra obligatoirement spécifier ceux du milieu. On devra donc écrire :

Code : C++

```cpp
cout << nombreDeSecondes(1, 0, 25) << endl;
```

Est-ce que je peux rendre juste les heures facultatives, et rendre les minutes et secondes obligatoires ?

Si le prototype est défini dans le même ordre que tout à l'heure : non.
Les paramètres facultatifs doivent obligatoirement se trouver à la fin (à droite).

Ce code ne compilera donc pas :

Code : C++

```cpp
int nombreDeSecondes(int heures = 0, int minutes, int secondes);
                                    // Erreur, les paramètres par défaut doivent être à droite
```

La solution, pour régler ce problème, consiste à placer le paramètre *heures* à la fin :

Code : C++

```cpp
int nombreDeSecondes(int secondes, int minutes, int heures = 0);
                                                 // OK
```

Est-ce que je peux rendre tous mes paramètres facultatifs ?

Oui, ça ne pose pas de problème :

Code : C++

```
int nombreDeSecondes(int heures = 0, int minutes = 0, int secondes = 0);
```

Dans ce cas, l'appel de la fonction pourra être fait comme ceci :

Code : C++

```
cout << nombreDeSecondes() << endl;
```

Le résultat retourné sera bien entendu 0 dans notre cas

Règles à retenir

En résumé, il y a 2 règles que vous devez retenir pour les valeurs par défaut :

- Seul le prototype doit contenir les valeurs par défaut (pas la définition de la fonction).
- Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres ("à droite").

LA SURCHARGE DES FONCTIONS

Ca, c'est probablement *la* nouveauté la plus importante des fonctions ! Cela nous aidera énormément lorsque nous ferons de la POO un peu plus loin

De quoi s'agit-il ? D'un nouveau système en C++ qui permet de **surcharger des fonctions**.
En gros, et pour faire simple, c'est une technique qui nous permet de créer plusieurs fonctions ayant le même nom... sans que le compilateur crie au loup

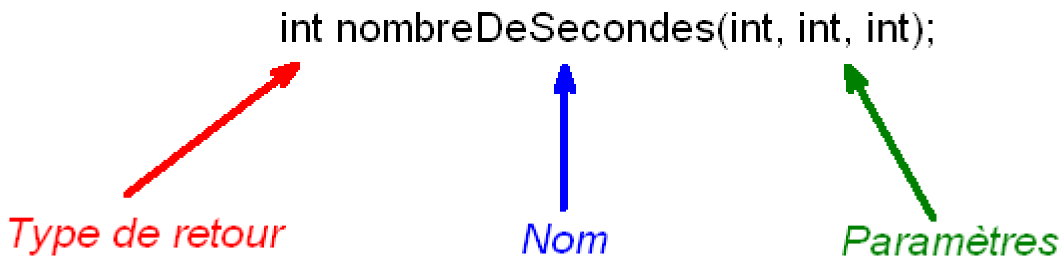
La signature d'une fonction

Avant toute chose, il faut que je vous parle de ce qu'on appelle la **signature d'une fonction**. C'est un peu sa carte d'identité, ce qui permet au compilateur de différencier les fonctions entre elles.

Chaque fonction est constituée de 3 éléments, ni plus ni moins :

- Un type de retour
- Un nom
- Une liste de paramètres

On va représenter ça sur un schéma pour être sûr qu'on voie bien la même chose



Le compilateur se moque complètement des noms des variables passées en paramètre. Ce qui compte pour lui, c'est juste le type de ces paramètres. Je vous l'avais d'ailleurs dit dans le **chapitre sur la compilation modulaire**
Voilà donc pourquoi j'ai marqué (int, int, int) pour les paramètres. C'est ce que le compilateur "voit", le nom des variables est donc ignoré pour l'identification de la fonction.

Bon, qu'est-ce qui permet d'identifier une fonction d'après vous ? Comment le compilateur fait-il pour vérifier si une fonction est bien différente d'une autre ?

En C, le compilateur se basait sur le nom, et uniquement sur le nom. Si 2 fonctions avaient le même nom, la compilation plantait. L'identification était donc faite sur le nom.
En C++, le compilateur se base sur le nom et les paramètres ! On peut avoir du coup 2 fonctions avec le même nom, à condition que celles-ci reçoivent des paramètres différents.

Le nom et les paramètres de la fonction constituent ce qu'on appelle la **signature de la fonction**. C'est ce qui permet au compilateur de l'identifier en C++.

Le type de retour n'est donc pas pris en compte pour identifier la fonction.

La surcharge d'une fonction

La surcharge consiste à créer des fonctions qui ont le même nom, mais qui ont des paramètres différents (donc une signature différente).

Voici ce qui peut varier :

- Le nombre de paramètres
- Le type de chacun de ces paramètres

Encore une fois, je le rappelle, le nom que l'on donne à chacun des paramètres, le compilo il s'en fout complètement

Prenons un exemple pour bien comprendre ce que ça va nous permettre de faire. Imaginez une fonction addition. On peut additionner des entiers (int), mais aussi des décimaux (double).

En C, il aurait fallu nommer les deux fonctions différemment (par exemple sommeEntiers et sommeDecimaux). En C++, on peut leur donner le même nom et ça va grandement simplifier leur utilisation, vous allez voir.

Code : C++

```
int somme(int nb1, int nb2)
{
    return nb1 + nb2;
}

double somme(double nb1, double nb2)
{
    return nb1 + nb2;
}
```

Les prototypes de ces fonctions sont donc :

Code : C++

```
int somme(int nb1, int nb2);
double somme(double nb1, double nb2);
```

Leurs signatures sont :

Code : C++

```
somme(int, int)
somme(double, double)
```

Ces fonctions ont des signatures différentes et portent le même nom. **Ce sont des fonctions surchargées**

Maintenant, dans le main on peut faire appel à la fonction somme pour additionner indifféremment des entiers ou des décimaux. C'est le compilateur qui décide quelle fonction il appelle en fonction du nombre et du type des paramètres.

Voici un code complet que vous pouvez tester :

Code : C++

```
#include <iostream>

using namespace std;


int somme(int nb1, int nb2);
double somme(double nb1, double nb2);

int main()
{
    cout << somme(10, 15) << endl << somme(2.5, 0.3) << endl;

    return 0;
}

int somme(int nb1, int nb2)
{
    return nb1 + nb2;
}

double somme(double nb1, double nb2)
{
    return nb1 + nb2;
}
```

Résultat :

Code : Console

```
25
2.8
```

On a appelé 2 fois la fonction "somme", mais c'est en fait une fonction différente qui a été appelée à chaque fois

Vous pouvez surcharger la fonction autant de fois que vous le désirez. On pourrait donc aussi rajouter par exemple la fonction qui fait la somme d'un entier et d'un décimal :

Code : C++

```
double somme(int nb1, double nb2)
{
    return nb1 + nb2;
}
```

... ou encore celle qui fait la somme de 3 entiers :

Code : C++

```
int somme(int nb1, int nb2, int nb3)
{
    return nb1 + nb2 + nb3;
}
```

Les possibilités sont infinies

Bien entendu, on fait de la surcharge de fonction pour des choses plus "intéressantes" que des sommes, mais ça on le découvrira petit à petit en fonction de nos besoins.

Les fonctions inline

Ce que nous allons voir ici ressemble à beaucoup aux macros (relisez le **chapitre sur le préprocesseur** si vous avez oublié ce que c'est).

Les macros sont un bon moyen, utilisées intelligemment, d'accélérer la vitesse d'exécution du programme si certains bouts de code sont souvent réutilisés.
Toutefois, les macros sont assez délicates à manipuler et impliquent l'utilisation du préprocesseur.

En C++, on a inventé le mot-clé *inline* qui permet de faire, grosso modo, la même chose que les macros sans cette fois passer par le préprocesseur. C'est donc le compilateur qui se charge de faire le "remplacement de code" au moment de la compilation. L'avantage, c'est qu'on peut faire plus de vérifications (notamment sur les types des paramètres).

Exemple d'utilisation d'une fonction inline

Prenons l'exemple suivant (on le discutera ensuite) :

Code : C++

```
inline int carre(int nombre);

int main()
{
    cout << carre(10) << endl;

    return 0;
}

inline int carre(int nombre)
{
    return nombre * nombre;
}
```

Vous voyez que j'ai ajouté le mot-clé *inline* au début du prototype ET au début de la définition de la fonction. Cela signifie pour le compilateur "*A chaque fois qu'on fera appel à la fonction carre, je placerai directement le code de cette fonction à l'endroit de l'appel*".

En clair, après compilation voici ce qu'il restera dans votre exécutable :

Code : C++

```
int main()
{
    cout << 10 * 10 << endl;

    return 0;
}
```

La fonction inline disparaît complètement après compilation. Tout son code se trouve placé à l'endroit de l'appel (la ligne du cout dans notre cas).

L'avantage est que l'exécution du programme sera plus rapide, surtout si la fonction est appelée plusieurs fois. En effet, lors d'un appel "classique" de fonction, le processeur va sauter à l'adresse de la fonction en mémoire, retenir l'adresse où il en était pour revenir à la fonction appelante une fois l'autre fonction terminée... Bref, c'est très rapide, mais si la fonction est amenée à être appelée très souvent, il est préférable d'en faire une inline (on dit l'*inliner*) pour éviter de répéter tout ce processus.
Le défaut, c'est que le programme risque de grossir un peu une fois compilé (le même code étant répété dans l'exécutable). Mais bon, en général cette différence est quand même négligeable

En règle générale, les fonctions inline sont donc des fonctions très courtes que l'on est susceptible de réutiliser souvent, comme c'est le cas de la fonction carre ici.

En pratique, on utilise quand même assez peu les fonctions inline, sachez-le (c'est comme les macros, je ne pense pas que vous vous en soyez beaucoup servis jusqu'ici). Ca reste cependant une des nouveautés du C++ relatives aux fonctions que je devais vous présenter

pssst, puisqu'on y est, serez-vous capables de surcharger la fonction inlinée carre pour qu'elle calcule le carré d'un nombre décimal ? Bon, vous êtes capables de surcharger des fonctions inlinées avec des paramètres par défaut, qu'est-ce qui pourrait bien vous faire peur maintenant ?
Oh, mais ne faites pas les malins, tout ceci n'était qu'une misérable mise en bouche comparé à ce qui vous attend

En effet, dans le prochain chapitre on va rentrer en plein dans le coeur du C++ : on va découvrir la **programmation orientée objet**. Bien sûr, on va y aller pas à pas, en douceur, sinon ça risque d'être un peu... violent

Quand vous êtes prêts, rendez-vous au proch... bon, je suis déjà dans le chapitre suivant moi, qu'est-ce que vous attendez ?

La magie de la POO par l'exemple : string

Nous attaquons maintenant la 2ème moitié de la partie IV sur le C++. Et comme dans la vie rien n'est jamais simple, cette "deuxième moitié" sera la plus grosse et... la plus délicate aussi

Nous allons maintenant, et dans les chapitres suivants, découvrir la notion de **programmation orientée objet** (POO). Comme je vous l'ai dit plus tôt, c'est une nouvelle façon de programmer. Ca ne va pas révolutionner immédiatement vos programmes, ça va vous paraître un peu inutile au début (comme lorsque vous aviez découvert les pointeurs), mais faites-moi confiance : faites l'effort de faire ce que je dis à la lettre, et bientôt vous serez bien plus efficaces lorsque vous programmerez.

Ce chapitre va vous parler des 2 facettes de la POO, le côté *utilisateur* et le côté *créateur*.
Puis, je vais faire carrément l'inverse de ce que tous les cours de programmation font (je sais je suis fou) : au lieu de commencer par vous apprendre à *créer* des objets, je vais d'abord vous montrer comment les *utiliser* avec pour exemple le type **string** fourni par le langage C++.

DES OBJETS... POUR QUOI FAIRE ?

Ils sont beaux, ils sont frais mes objets

S'il y a bien un mot qui doit vous frustrer depuis que vous en entendez parler, c'est celui-ci : **objet**.

> Encore un concept mystique ? Un délire de programmeurs après une soirée trop arrosée ?
> Non parce que franchement, un objet c'est quoi ? Mon écran est un objet, ma voiture est un objet, mon téléphone portable... ce sont tous des objets !

Bien vu, c'est un premier point
En effet, nous sommes entourés d'objets. En fait, tout ce que nous connaissons (ou presque) peut être considéré comme un objet. L'idée de la programmation orientée objet, c'est de manipuler des éléments que l'on appelle des "objets" dans son code source.

> Mais concrètement, c'est quoi ? Une variable ? Une fonction ?

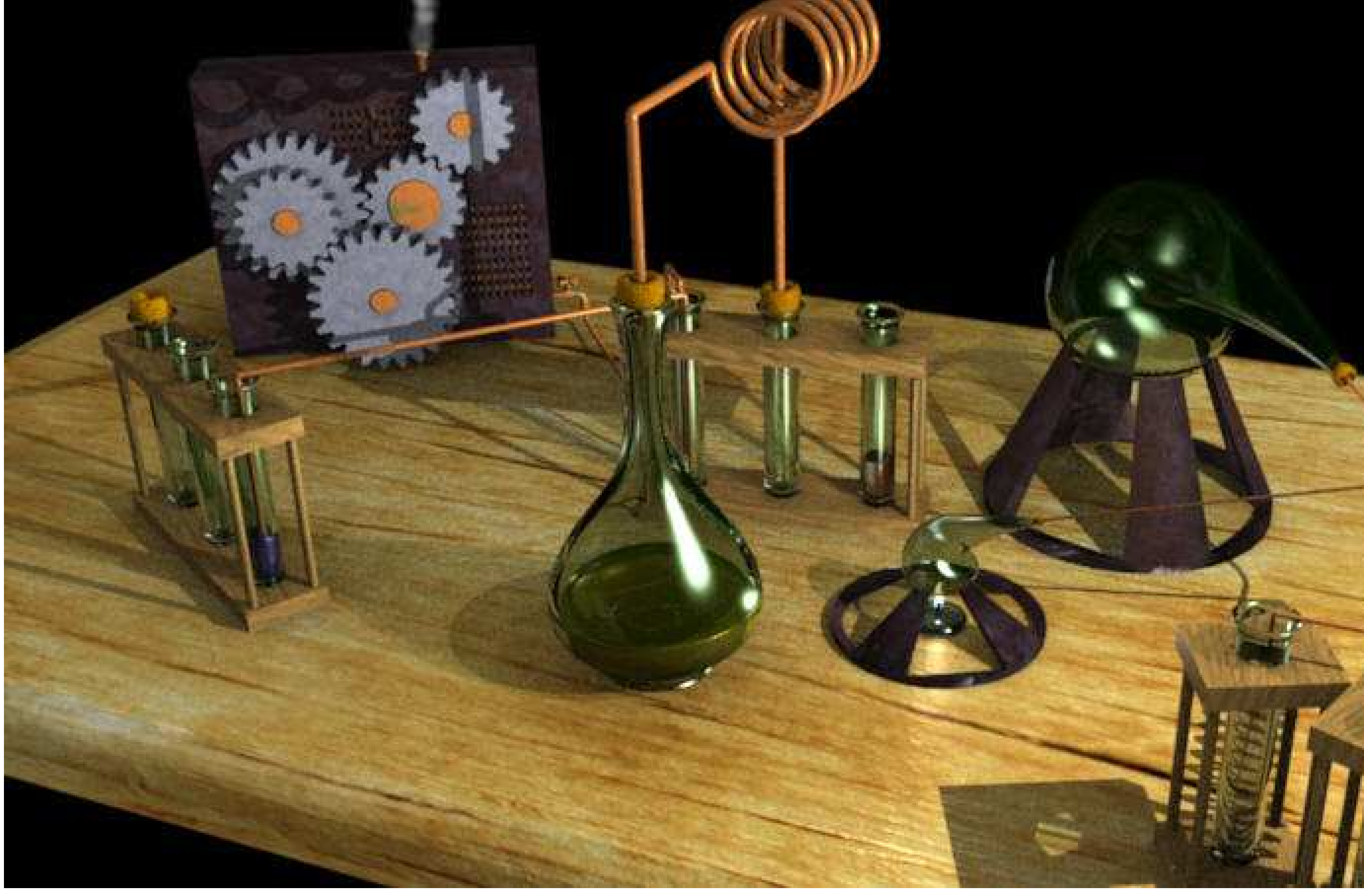
Ni l'un, ni l'autre. C'est un nouvel élément en programmation.
Pour être plus précis, un objet c'est... un mélange de plusieurs variables et fonctions

Ne faites pas cette tête-là, vous allez découvrir tout cela par la suite

Imaginez... un objet

Pour éviter que ce que je vous raconte ressemble à un traité d'art moderne conceptuel, on va imaginer ensemble ce qu'est un objet à l'aide de plusieurs schémas concrets.
Les schémas 3D que vous allez voir par la suite ont été réalisés pour moi par l'ami Nab, que je remercie d'ailleurs vivement au passage.

Imaginez qu'un programmeur décide un jour de créer un programme qui permet d'afficher une fenêtre à l'écran, de la redimensionner, de la déplacer, de la supprimer... Le code est complexe : il va avoir besoin de plusieurs fonctions qui s'appellent entre elles, et de variables pour mémoriser la position, la taille de la fenêtre, etc.
Il met du temps à écrire ce code, c'est un peu compliqué, mais il y arrive. Au final, le code qu'il a écrit est composé de plusieurs fonctions et variables. Quand on regarde ça pour la première fois, ça ressemble à une expérience de savant fou à laquelle on ne comprend rien :



Ce programmeur est content de son code et veut le distribuer sur internet pour que tout le monde puisse créer des fenêtres sans passer du temps à tout réécrire. Seulement voilà, à moins d'être un expert en chimie certifié, vous allez mettre pas mal de temps avant de comprendre comment tout ce bazar fonctionne.

Quelle fonction appeler en premier ? Quelles valeurs envoyer à quelle fonction pour redimensionner la fenêtre ?

C'est là que notre ami programmeur pense à nous. Il conçoit son code de manière orientée objet. Cela signifie qu'il place tout son bazar chimique à l'intérieur d'un simple cube. Ce cube est ce qu'on appelle un objet :

Ici, une partie du cube a été volontairement mise en transparence pour vous montrer que nos fioles chimiques sont bien situées à l'intérieur du cube. Mais en réalité, le cube est complètement opaque, on ne voit **rien** de ce qu'il y a à l'intérieur :

Ce cube contient toutes les fonctions et les variables (nos fioles de chimie), mais il les masque à l'utilisateur.

Au lieu d'avoir des tonnes de tubes et fioles chimiques dont il faut comprendre le fonctionnement, on nous propose juste quelques boutons sur la face avant du cube : un bouton "ouvrir fenêtre", un bouton "redimensionner", etc. L'utilisateur n'a plus qu'à se servir des boutons du cube et n'a plus besoin de se soucier de tout ce qui se passe à l'intérieur. Pour l'utilisateur, c'est donc complètement simplifié.

En clair : programmer de manière orienter objet, c'est *créer* du code source (peut-être complexe), mais que l'on masque en le plaçant à l'intérieur d'un cube (un objet) à travers lequel on ne voit rien. Pour le programmeur qui va l'*utiliser*, travailler avec un objet est donc beaucoup plus simple qu'avant : il a juste à appuyer sur des boutons et n'a pas besoin d'être diplômé en chimie pour s'en servir.

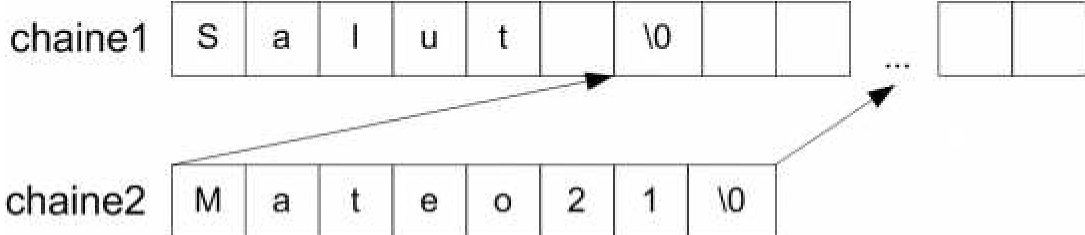
Bien sûr, c'est une image, mais c'est ce qu'il faut comprendre et retenir pour le moment

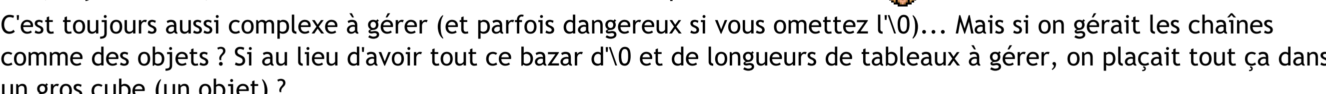
Nous n'allons pas voir tout de suite comment faire pour *créer* des objets. En revanche, nous allons apprendre à en *utiliser* un. Nous allons créer des objets de type **string**. Le type string est fourni par la librairie standard du C++.
Ce qui va suivre devrait vous convaincre une fois pour toutes que travailler avec des objets, bon sang que c'est simple (bon les créer sera une autre paire de manches, mais ne gâchons pas la fête qui va suivre).

LIRE ET ÉCRIRE DANS UNE CHAÎNE VIA STRING

Vous vous souvenez des chaînes de caractères ? Vous vous souvenez de ce chapitre un peu compliqué où je vous avais dit qu'une chaîne était un tableau de char, que toute chaîne devait se terminer par un \0, qu'il fallait bien calculer la longueur de son tableau lorsqu'on le déclarait pour ne pas oublier la place pour l'\0 ?

Ca avait donné lieu à des schémas comme celui-ci, pour expliquer par exemple le concaténation de 2 chaînes :



Bon, soyons clairs, en C++ votre ordinateur ne fonctionne pas différemment

C'est toujours aussi complexe à gérer (et parfois dangereux si vous omettez l'\0)... Mais si on gérait les chaînes comme des objets ? Si au lieu d'avoir tout ce bazar d'\0 et de longueurs de tableaux à gérer, on plaçait tout ça dans un gros cube (un objet) ?

Notre cube proposerait en façade plusieurs boutons pour faire toutes les opérations possibles et imaginables avec des chaînes de caractères, tout en nous évitant d'avoir à savoir comment ça fonctionne à l'intérieur.

Cet type d'objet existe, il est déjà créé et il est livré dans la librairie standard du C++. Il s'appelle **string** ("chaîne" en anglais).

Votre premier objet

Allez n'attendons plus, et voyons comment on crée un objet de type string dans un code source

Code : C++

```
#include <iostream>
#include <string> // Obligatoire pour pouvoir utiliser les objets string

using namespace std;


int main()
{
    string maChaine; // Création d'un objet "maChaine" de type string

    return 0;
}
```

Vous remarquerez pour commencer qu'il est nécessaire d'inclure le header de la librairie *string* pour pouvoir utiliser des objets de type string dans le code C'est ce que j'ai fait à la 2ème ligne.

Intéressons-nous maintenant à la ligne où je crée un objet de type string...

Mais... on crée un objet de la même manière qu'on crée une variable ?

Il y a plusieurs façons de créer un objet, celle que vous venez de voir est la plus simple. Et, oui, c'est exactement comme si on avait créé une variable !

Mais mais... comment on fait pour différencier les objets des variables ?

C'est bien tout le problème. Pour éviter la confusion, il y a des conventions (qu'on n'est pas obligé de suivre). La plus célèbre d'entre elles est la suivante :

- Si c'est une **variable**, la première lettre du type doit être en **minuscule** (ex : int)
- Si c'est un **objet**, la première lettre du type doit être en **majuscule** (ex : Voiture)

Je sais ce que vous allez me dire : "*string ne commence pas par une majuscule !*". Il faut croire que ceux qui ont créé string ne respectaient pas cette convention. Mais rassurez-vous, maintenant la plupart des gens mettent une majuscule au début de leurs objets (dont moi), ça sera pas la foire dans la suite de ce cours

Affecter une valeur à la chaîne lors de la déclaration

Pour affecter une valeur à notre objet au moment de la déclaration, il y a plusieurs possibilités. La plus courante consiste à ouvrir des parenthèses comme si on appelait une fonction :

Code : C++

```
int main()
{
    string maChaine("Bonjour !"); // Création d'un objet "maChaine" de type string et
affectation

    return 0;
}
```

(vous commencez à comprendre ce que je vous racontais tout à l'heure quand je disais que les objets étaient une sorte de mélange de variables et fonctions)

Dans la plupart des cas donc, on ouvrira des parenthèses pour affecter une valeur à l'objet lors de sa création. Pour le type string cependant, il est possible de faire encore plus simple en utilisant le symbole égal :

Code : C++

```
int main()
{
    string maChaine = "Bonjour !"; // Création d'un objet "maChaine" de type string et
affectation

    return 0;
}
```

Bref, voilà qui est fait. On a un objet maChaine qui contient la chaîne "Bonjour !".
On peut l'afficher comme n'importe quelle chaîne de caractères avec un cout :

Code : C++

```
int main()
{
    string maChaine = "Bonjour !";
    cout << maChaine << endl; // Affichage du string comme si c'était une chaîne de
caractères

    return 0;
}
```

Code : Console

```
Bonjour !
```

Affecter une valeur à la chaîne après déclaration

Maintenant que notre objet est créé, ne nous arrêtons pas là. Changeons la chaîne qui se trouve à l'intérieur. Donnez-lui la chaîne que vous voulez, il la stockera :

Code : C++

```
int main()
{
    string maChaine = "Bonjour !";
    cout << maChaine << endl;

    maChaine = "Bien le bonjour !";
    cout << maChaine << endl;

    return 0;
}
```

Code : Console

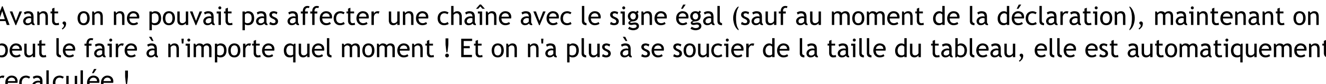
```
Bonjour !
Bien le bonjour !
```

Mais... on n'a pas précisé la longueur de la chaîne au début, et maintenant on stocke dedans une chaîne plus grande qu'avant ! Comment on sait s'il y aura la place de stocker une chaîne aussi longue ?

C'est là que la magie de la POO opère. Vous, l'utilisateur, vous avez appuyé sur un bouton pour dire "Je veux maintenant que la chaîne à l'intérieur change pour *Bien le bonjour !*". A l'intérieur de l'objet, des mécanismes (des fonctions) se sont activées lorsque vous avez fait ça. Ces fonctions ont vérifié entre autres s'il y avait de la place pour stocker la chaîne. Elles ont vu que non. Elles ont alors créé un nouveau tableau de char, suffisamment long cette fois, pour stocker la nouvelle chaîne. Et elles ont détruit l'ancien tableau qui ne servait plus à rien, tant qu'à faire.

Et permettez-moi de vous parler franchement : ce qui s'est passé à l'intérieur de l'objet, on s'en fout royalement
C'est bien là tout l'intérêt de la POO : l'utilisateur n'a pas besoin de comprendre comment ça marche à l'intérieur. L'objet est en quelque sorte intelligent et gère tous les cas. Nous, on ne fait que l'utiliser ici.

Du coup, pour nous c'est simplifié comme vous avez pas idée



Avant, on ne pouvait pas affecter une chaîne avec le signe égal (sauf au moment de la déclaration), maintenant on peut le faire à n'importe quel moment ! Et on n'a plus à se soucier de la taille du tableau, elle est automatiquement recalculée !

Et ça, c'est rien qu'un début

Concaténation de chaînes

Allez, je vais continuer à vous faire baver
On va concaténer (assembler) 2 chaînes. Regardez comment on fait :

Code : C++

```
int main()
{
    string chaine1 = "Bonjour !";
    string chaine2 = "Comment allez-vous ?";
    string chaine3;

    chaine3 = chaine1 + chaine2; // 3... 2... 1... Concaténatioooooon
    cout << chaine3 << endl;

    return 0;
}
```

Code : Console

```
Bonjour !Comment allez-vous ?
```

Ah, allez je reconnais, il manque un espace au milieu. On n'a qu'à changer la ligne de la concaténation :

Code : C++

```
chaine3 = chaine1 + " " + chaine2;
```

Résultat :

Code : Console

```
Bonjour ! Comment allez-vous ?
```

Niveau de simplicité : effarante
Avant, il aurait fallu appeler une fonction pour la concaténation, et faire attention à la longueur de la chaîne qui est modifiée pour être sûr qu'il y ait suffisamment de place dedans. Ici, rien de tout cela

On assemble donc nos chaînes de caractères à l'aide du symbole +. Et avouez franchement, si vous avez un tant soit

peu programmé avant, que ça va vous faire gagner un temps de fou

Comparaison de chaînes

Vous en voulez encore ? Très bien !
Sachez que l'on peut comparer des chaînes entre elles à l'aide des symboles == ou != (que l'on peut donc utiliser dans un if !).

Code : C++

```
int main()
{
    string chaine1 = "Bonjour !";
    string chaine2 = "Comment allez-vous ?";

    if (chaine1 == chaine2) // Faux
    {
        cout << "Les chaines sont identiques" << endl;
    }
    else
    {
        cout << "Les chaines sont differentes" << endl;
    }

    return 0;
}
```

Code : Console

```
Les chaines sont differentes
```

C'est tout bête
Vous pouvez vérifier que ça marche aussi dans le cas contraire :

Code : C++

```
int main()
{
    string chaine1 = "Bonjour !";
    string chaine2 = "Bonjour !";

    if (chaine1 == chaine2) // Vrai
    {
        cout << "Les chaines sont identiques" << endl;
    }
    else
    {
        cout << "Les chaines sont differentes" << endl;
    }

    return 0;
}
```

Code : Console

```
Les chaines sont identiques
```

Si ça vous amuse, testez aussi le symbole "différent de" (**!=**), vous verrez que 2 objets de type string sont capables de se comparer entre eux

Conclusion : plus simple tu meurs

Opérations sur les string

Le type string ne s'arrête pas à ce que nous venons de voir. Comme tout bon objet qui se respecte, il propose un nombre important d'autres fonctionnalités qui permettent de faire tout ce qu'on a besoin avec.

Nous n'allons pas passer toutes les fonctionnalités des string en revue (elles sont pas toutes indispensables et ce serait un peu long). Nous allons voir les principales dont vous pourriez avoir besoin dans la suite du cours

Attributs et méthodes

Je vous avais dit qu'un objet était constitué de variables et de fonctions. En fait, on en reparlera plus tard mais le vocabulaire est un peu différent avec les objets. Les variables contenues à l'intérieur des objets sont appelées **attributs**, et les fonctions sont appelées **méthodes**.

Imaginez que chaque méthode (fonction) que propose un objet correspond à un bouton différent sur la façade avant du cube

On parle aussi de "variables membres" et de "fonctions membres", ce qui est peut-être un peu moins déroutant que "attributs" et "méthodes" qui sont des mots complètement nouveaux et auxquels on a un peu de mal à se faire au début

Appeler une méthode d'un objet se fait de la même manière qu'avec les structures. On utilise le point pour séparer l'objet de sa méthode :

```
objet.methode()
```

En théorie, on peut aussi accéder aux variables membres (les "attributs") de l'objet de la même manière qu'on le faisait avec les structures. Cependant, en POO, il y a une règle super importante qui dit que l'utilisateur ne doit pas pouvoir accéder aux variables membres, mais seulement aux fonctions membres. On en reparlera dans le prochain chapitre plus en détail.

Quelques méthodes utiles du type string

La méthode size()

La méthode size() permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type string. C'est un peu l'équivalent de strlen(), mais pour les string cette fois

Cette méthode ne prend aucun paramètre et renvoie la longueur de la chaîne. Comme vous venez de le découvrir, il va falloir appeler la méthode de la manière suivante :

Code : C++

```cpp
maChaine.size()
```

Essayons ça dans un code complet qui affiche la longueur de la chaîne :

Code : C++

```
int main()
{
    string maChaine = "Bonjour !";

    cout << "Longueur de la chaine : " << maChaine.size();

    return 0;
}
```

Code : Console

```
Longueur de la chaine : 9
```

Bingo !
C'est là toute la subtilité. Avant on aurait dû faire :

Code : C

```
strlen(maChaine)
```

La fonction strlen était valable pour n'importe quelle chaîne, mais il fallait préciser en paramètre à chaque fois quelle était la chaîne sur laquelle la fonction devait travailler.

Maintenant, l'ordre est un peu inversé. La fonction size() est contenue dans l'objet maChaine. Quand on l'appelle comme on vient de le faire, la fonction membre size() sait qu'elle doit calculer la longueur de la chaîne contenue dans l'objet où elle se trouve :

Code : C++

```
maChaine.size()
```

La fonction size() est donc propre à l'objet maChaine. Si vous créez un deuxième objet de type string, il y aura donc une autre fonction size() propre à cet autre objet.

Rassurez-vous, les compilateurs C++ sont suffisamment intelligents pour optimiser l'utilisation de la mémoire. Si vous créez 50 string, il n'y aura pas 50 fois la même fonction en mémoire (une seule suffit). Mais ce que je vous dis là, c'est ce qu'il faut "imaginer". Dans chaque objet (chaque boîte), il y a une fonction size() qui est propre à l'objet. C'est comme cela qu'il faut le voir.

La méthode erase()

Cette méthode très simple supprime tout le contenu de la chaîne :

Code : C++

```
int main()
{
    string chaine = "Bonjour !";

    chaine.erase();
    cout << "La chaine contient : " << chaine << endl;

    return 0;
}
```

Code : Console

```
La chaine contient :
```

Comme on pouvait s'y attendre, la chaîne ne contient plus rien

Notez que c'est équivalent à faire :

Code : C++

```
chaine = "";
```

La méthode substr()

Une autre méthode qui peut s'avérer utile : substr(). Elle permet de ne prendre qu'une partie de la chaîne stockée dans un string.

substr signifie "substring", soit "sous-chaîne" en anglais.

Tenez, on va regarder son prototype, vous allez voir que c'est intéressant :

Code : C++

```
string substr( size_type index, size_type num = npos );
```

Cette méthode retourne donc un objet de type string. Ce sera la sous-chaîne après "découpage".
Elle prend 2 paramètres, ou plus exactement : 1 paramètre obligatoire, 1 paramètre facultatif. En effet, *num* possède une valeur par défaut (npos) ce qui fait que le second paramètre ne doit pas obligatoirement être renseigné.

- **index** permet d'indiquer à partir de quel caractère on doit couper (ce doit être un numéro de caractère)
- **num** permet d'indiquer le nombre de caractères que l'on prend. Par défaut, la valeur est npos, ce qui correspond à prendre tous les caractères qui restent. Si vous indiquez 2, la méthode ne renverra que 2 caractères.

Allez, un exemple sera plus parlant je crois

Code : C++

```
int main()
{
    string chaine = "Bonjour !";

    cout << chaine.substr(3) << endl;

    return 0;
}
```

Code : Console

```
jour !
```

On a demandé à couper à partir du 3ème caractère (soit la lettre "j" vu que la première lettre correspond au caractère n°0).
On a volontairement omis le second paramètre facultatif, ce qui fait que du coup substr() a renvoyé tous les caractères restants avant la fin de la chaîne. Essayons de renseigner le paramètre facultatif pour ne pas prendre le point d'exclamation par exemple :

Code : C++

```
int main()
{
    string chaine = "Bonjour !";

    cout << chaine.substr(3, 4) << endl;

    return 0;
}
```

Code : Console

```
jour
```

Bingo !
On a demandé à prendre 4 caractères en partant du caractère n°3, ce qui fait qu'on a récupéré "jour"

La méthode c_str()

Celle-là est un peu particulière, mais parfois fort utile. Son rôle ? Retourner un pointeur vers le tableau de char que contient l'objet de type string.

Quel intérêt me direz-vous ? En C++, à priori aucun intérêt.
Mais il peut (j'ai bien dit il "peut") arriver que vous deviez envoyer à une fonction un tableau de char classique, façon C. Dans ce cas, la méthode c_str() vous permet de récupérer un bon vieux tableau de char comme on faisait en C.

Code : C++

```
int main()
{
    string chaine = "Bonjour !";
    char* chaineC = NULL;

    chaineC = chaine.c_str(); // On récupère le tableau de char dans chaineC
    cout << "La chaine contient : " << chaineC << endl; // On l'affiche pour vérifier que
ça fonctionne

    return 0;
}
```

Récupérer le tableau de char vous sera utile si vous devez envoyer une chaîne à une fonction à la base prévue pour le C qui ne reconnaît pas les string. C'est rare, mais ça arrive. Je préfère que vous sachiez qu'on a cette possibilité pour pas que vous soyez bêtement bloqué à un moment.

Autant que possible, utilisez des objets de type string plutôt que des tableaux de char : vous avez vu que c'était bien plus facile à utiliser Comme le disait si bien ma prof d'informatique "*C'est plus confortable de travailler avec un string*" (je vous jure que c'est vrai, j'étais là)

Bon plus sérieusement
Vous avez découvert le côté utilisateur de la POO et à quel point ces nouveaux mécanismes pouvaient vous simplifier la vie.

Le côté utilisateur est en fait le côté simple de la POO. Les choses se compliquent lorsqu'on passe du côté créateur.

Nous allons justement apprendre à créer des objets dans le prochain chapitre et tous les suivants. Une longue route pleine de péripéties nous attend

Les classes (Partie 1/2)

Dans le chapitre précédent, vous avez vu que la programmation orientée objet pouvait nous simplifier la vie en "masquant" en quelque sorte le code complexe. Ca c'est un des avantages de la POO, mais ce n'est pas le seul comme vous allez le découvrir petit à petit. Par exemple, un autre gros avantage des objets est qu'ils sont facilement réutilisables et modifiables.

A partir de maintenant, **nous allons apprendre à créer des objets**. Vous allez voir que c'est tout un art et que ça demande de la pratique. Il y a beaucoup de programmeurs qui prétendent faire de la POO et qui le font pourtant très mal (et je ne m'exclue pas forcément du lot). En effet, on peut créer un objet de 100 façons différentes, et c'est à nous de choisir à chaque fois la meilleure, la plus adaptée. Pas évident. Il faudra donc bien réfléchir avant de se lancer dans le code comme des forcenés

Allez, on prend une grande inspiration, et on plonge ensemble dans l'océan de la POO !

Créer une classe

Commençons par la question qui doit vous brûler les lèvres

Je croyais qu'on allait apprendre à créer des objets, pourquoi tu nous parles de créer une classe maintenant ?
Quel est le rapport ?

Eh bien justement, pour créer un objet, il faut d'abord créer une classe !
Je m'explique : pour construire une maison, vous avez besoin d'un plan d'architecte non ? Eh bien imaginez simplement que la classe c'est le plan, et que l'objet c'est la maison.

"Créer une classe", c'est donc dessiner les plans de l'objet.

Une fois que vous avez les plans, vous pouvez faire autant de maisons que vous voulez en vous basant sur les plans. Pour les objets c'est pareil : une fois que vous avez fait la classe (le plan), vous pourrez créer autant d'objets du même type que vous voulez

Vocabulaire : on dit qu'un objet est une **instance** d'une classe. C'est un mot très courant que l'on rencontre souvent en POO. Cela signifie qu'un objet est la matérialisation concrète d'une classe (tout comme la maison est la matérialisation concrète du plan de la maison).
Oui je sais c'est très métaphysique la POO, mais vous allez voir on s'y fait

Créer une classe, oui mais laquelle ?

Avant tout, il va falloir choisir la classe sur laquelle nous allons travailler.

Pour reprendre mon exemple sur l'architecture : allons-nous créer un appartement, une villa avec piscine, un spacieux loft ?
En clair, quel type d'objet voulons-nous être capable de créer ?

Les choix ne manquent pas. Je sais que, quand on débute, on a du mal à imaginer ce qui peut être considéré comme un objet. La réponse est : presque tout !

Vous allez voir, vous allez petit à petit avoir le feeling qu'il faut avec la POO. Puisque vous débutez, c'est moi qui vais choisir (vous avez pas trop le choix de toute façon).
Pour notre exemple, nous allons créer une classe **Personnage** qui va permettre de représenter un personnage de jeu de rôle (RPG).

Si vous n'avez pas l'habitude des jeux de rôle, rassurez-vous, moi non plus. Vous n'avez pas besoin de savoir jouer à des RPG pour suivre ce chapitre. J'ai choisi cet exemple car il me paraît didactique, amusant, et qu'il peut déboucher sur la création d'un jeu à la fin

Bon, on la crée cette classe ?

C'est parti

Pour commencer, je vous rappelle qu'une classe est constituée :

- De variables, ici appelées **attributs** (on parle aussi de *variables membres*)
- De fonctions, ici appelées **méthodes** (on parle aussi de *fonctions membres*)

(n'oubliez pas ce vocabulaire, il est fon-da-men-tal !)

Pour tout vous dire, les classes ressemblent beaucoup aux **structures** qu'on avait étudiées en C, sauf qu'elles contiennent en plus des méthodes (les fonctions).
Vous allez donc voir que cela ressemble pas mal aux structures, du moins au premier abord.

Voici le code minimal pour créer une classe :

Code : C++

```
class Personnage
{

}; // N'oubliez pas le point-virgule à la fin !
```

On utilise comme vous le voyez le mot-clé `class`.
Il est suivi du nom de la classe que l'on veut créer. Ici, c'est Personnage.

Souvenez-vous de cette règle très importante : il faut que le nom de vos classes commence toujours par une lettre majuscule ! Bien que ce ne soit pas obligatoire (le compilateur ne gueulera pas si vous commencez par une minuscule), cela vous sera très utile par la suite pour différencier les types de variable classiques (int, double, bool, ...) des classes (Personnage, ...).

C'est entre les accolades que nous allons écrire la définition de la classe. Tout ou presque se passera à l'intérieur de ces accolades.
Et surtout, super important, le truc qu'on oublie au moins une fois dans sa vie : **il y a un point-virgule après l'accolade fermante**, tout comme avec les structures !

Ajout de méthodes et d'attributs

Bon c'est bien beau, mais notre classe Personnage est plutôt... vide.
Que va-t-on mettre dans la classe ? Vous le savez déjà voyons

- Des **attributs**, c'est le nom que l'on donne aux variables contenues dans des classes
- Des **méthodes**, c'est le nom que l'on donne aux fonctions contenues dans des classes

Le but du jeu maintenant, c'est justement d'arriver à faire la liste de tout ce qu'on veut mettre dans notre Personnage. De quels attributs et de quelles méthodes a-t-il besoin ? Ca, c'est justement l'étape de *réflexion*, la plus importante. C'est pour ça que je vous ai dit au début de ce chapitre qu'il fallait surtout pas coder comme des barbares dès le début, mais prendre le temps de *réfléchir*.

Cette étape de réflexion avant le codage est essentielle quand on fait de la POO. Beaucoup de gens, dont moi, ont l'habitude de sortir une feuille de papier et un crayon pour arriver à établir la liste des attributs et méthodes dont ils vont avoir besoin.
On en reparlera plus tard, mais sachez déjà qu'un langage spécial appelé **UML** a été spécialement conçu pour "dessiner" les classes avant de commencer à les coder.

Par quoi commencer : les attributs ou les méthodes ? Il n'y a pas d'ordre en fait, mais je trouve un peu plus logique de commencer par voir les attributs *puis* les méthodes.

Les attributs

C'est ce qui va caractériser votre classe, ici le personnage. Ce sont des variables, elles peuvent donc évoluer au fil du temps. Mais qu'est-ce qui caractérise un personnage de jeu de rôle ? Allons, un petit effort

- Par exemple, tout personnage a un niveau de vie. Hop, ça fait un premier attribut : **vie** ! On dira que ce sera un int, et qu'il sera compris entre 0 et 100 (0 = mort, 100 = toute la vie).
- Dans un jeu de rôle (RPG), il y a le niveau de magie, aussi appelé **mana**. Là encore, on va dire que c'est un int compris entre 0 et 100. Si le personnage a 0 de mana, il ne peut plus lancer de sorts magiques et doit attendre que sa mana se recharge toute seule au fil du temps (ou boire une potion de mana !).
- On pourrait rajouter aussi le nom de l'arme que porte le joueur : **nomArme**. Puisque c'est une chaîne de caractères et qu'on fait du C++, on n'est pas fou, on va utiliser un string
- Enfin, il me semble indispensable d'ajouter un attribut **degatsArme**, un int toujours, qui indiquerait cette fois le nombre de dégâts que fait notre arme à chaque coup

On peut donc déjà commencer à compléter notre classe avec ces premiers attributs :

Code : C++

```cpp
class Personnage
{
    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};
```

Deux trois petites choses à savoir sur ce code :

- Ce n'est pas une obligation, mais une grande partie des programmeurs (dont moi) a l'habitude de faire commencer tous les noms des attributs de classe par m_ (le "m" signifiant "membre", pour indiquer que c'est une variable membre, c'est-à-dire un attribut). Cela permet de bien différencier les attributs des variables

"classiques" (contenues dans des fonctions par exemple).
- Il est impossible d'initialiser les attributs ici. Cela doit être fait via ce qu'on appelle un constructeur, comme on le verra un peu plus loin.
- Comme on utilise un objet string, il faut bien penser à rajouter un `#include <string>` dans votre fichier.

Les méthodes

Les méthodes, elles, sont grosso modo les actions que le personnage peut faire ou qu'on peut lui faire faire. Les méthodes lisent et modifient les attributs.

Voici quelques actions qu'on peut faire avec notre personnage :

- **recevoirDegats** : le personnage prend un certain nombre de dégâts, donc perd de la vie.
- **attaquer** : le personnage attaque un autre personnage avec son arme. Il fait autant de dégâts que son arme lui permet d'en faire (c'est-à-dire degatsArme).
- **boirePotionDeVie** : le personnage boit une potion de vie et regagne un certain nombre de points de vie.
- **changerArme** : le personnage récupère une nouvelle arme plus puissante. On change le nom de l'arme et les dégâts qui vont avec.

Voilà c'est un bon début je trouve

On va rajouter ça dans la classe avant les attributs (on préfère présenter les méthodes *avant* les attributs en POO, bien que ça ne soit pas obligatoire) :

Code : C++

```
class Personnage
{
    // Méthodes
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Attributs
    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};
```

Je n'ai pas écrit le code des méthodes exprès, on le fera après

Ceci dit, vous devriez déjà avoir une petite idée de ce que vous allez mettre dans ces méthodes.

Par exemple, *recevoirDegats* retranchera le nombre de dégâts indiqués en paramètre par nbDegats à la vie du personnage.
Intéressante aussi : la méthode *attaquer*. Elle prend en paramètre... un autre personnage, plus exactement une référence vers le personnage cible que l'on doit attaquer ! Et que fera cette méthode à votre avis ? Eh oui, elle appellera la méthode recevoirDegats de la cible pour lui infliger des dégâts

Vous commencez à comprendre un peu comment tout cela est lié et terriblement logique ?
On met en général un peu de temps avant de "penser objet" correctement. Si vous vous dites que vous n'auriez pas pu inventer un truc comme ça tout seul, rassurez-vous, tous les débutants passent par là. A force de pratiquer, ça va venir

Pour info, toutes les méthodes que l'on pourrait créer ne sont pas là : par exemple, on n'utilise pas de magie (mana) ici. Le personnage attaque seulement avec une arme (une épée par exemple) et n'utilise donc pas de sorts magiques.
Je laisse exprès quelques fonctions manquantes pour vous inciter à compléter la classe avec vos idées

En résumé : comme je vous l'avais dit, un objet est bel et bien un mix de "variables" (les attributs) et de "fonctions" (les méthodes). La plupart du temps, les méthodes lisent et modifient les attributs de l'objet pour le faire évoluer.
Un objet est au final un petit système intelligent et autonome qui est capable de surveiller son bon fonctionnement tout seul.

Droits d'accès et encapsulation

Nous allons maintenant nous intéresser au concept le plus fondamental de la POO : **l'encapsulation**. Ne vous laissez pas effrayer par ce mot, vous allez vite comprendre ce que ça signifie.

Tout d'abord un petit rappel. En POO, il y a 2 parties bien distinctes :

- On **crée** des classes pour définir le fonctionnement des objets. C'est ce qu'on apprend à faire ici.
- On **utilise** des objets. C'est ce qu'on a appris à faire dans le chapitre précédent.

Il faut bien distinguer ces 2 parties, car ça devient ici très important.

Je mets un exemple création / utilisation côte à côte pour que vous puissiez bien les différencier :

<table>
<tr><th>Création de la classe</th><th>Utilisation de l'objet</th></tr>
<tr><td>Code : C++</td><td>Code : C++

```cpp
int main()
{
    Personnage david, goliath;

    goliath.attaquer(david);
    david.boirePotionDeVie(20);
    goliath.attaquer(david);
    david.attaquer(goliath);
    goliath.changerArme("Double hache 
tranchante vénéneuse de la mort", 40);
    goliath.attaquer(david);


    return 0;
}
```
</td></tr>
</table>

<table>
<tr><th>Création de la classe</th><th>Utilisation de l'objet</th></tr>
<tr><td><pre>
class Personnage
{
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int
quantitePotion)
    {

    }

    void changerArme(string
nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;

};
</pre></td><td></td></tr>
</table>

Tenez, pourquoi on n'essaierait pas ce code ?
Allez, on met tout dans un même fichier (en prenant soin de définir la classe *avant* le main), et zou !

Code : C++

```
#include <iostream>
#include <string>

using namespace std;

class Personnage
{
    // Méthodes
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Attributs
    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};

int main()
{
    Personnage david, goliath; // Création de 2 objets de type Personnage : david et
goliath

    goliath.attaquer(david); // goliath attaque david
    david.boirePotionDeVie(20); // david boit une potion de vie qui lui rapporte 20 de
vie
    goliath.attaquer(david); // goliath réattaque david
    david.attaquer(goliath); // david contre-attaque... c'est assez clair non ? ^^
    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
    goliath.attaquer(david);


    return 0;
}
```

Compilez et admirez... la belle erreur de compilation ?!

```
Error : void Personnage::attaquer(Personnage&) is private within this context
```

Une nouvelle insulte ?
Vous allez voir, le compilateur ne manque pas d'insultes en C++, vous allez sûrement en rencontrer pas mal

Les droits d'accès

On en arrive justement au problème qui nous intéresse : celui des droits d'accès (eh ouais j'ai fait exprès de

provoquer cette erreur de compilation, vous aviez quand même pas cru que j'avais pas tout prévu ?).

Ouvrez grand vos oreilles : chaque attribut et chaque méthode d'une classe peut posséder son propre droit d'accès. Il existe grosso modo 2 droits d'accès différents :

- **public** : l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
- **private** : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet. Par défaut, tous les éléments d'un objet sont private.

Il existe d'autres droits d'accès mais ils sont un peu plus complexes. Nous les verrons plus tard.

Concrètement, qu'est-ce que ça signifie ? Qu'est-ce que "l'extérieur" de l'objet ?
Eh bien sur notre exemple, "l'extérieur" c'est le main. En effet, c'est là où on utilise l'objet. On fait appel à des méthodes, mais comme elles sont privées par défaut, on ne peut pas les appeler depuis le main !

Pour modifier les droits d'accès et mettre par exemple public, il faut taper `public` suivi du symbole **:** (deux points). Tout ce qui se trouvera à la suite sera public.

Voici ce que je vous propose de faire : on va mettre en public toutes les méthodes, et en privé tous les attributs.
Ca donne ça :

Code : C++

```
class Personnage
{
    // Tout ce qui suit est public (accessible depuis l'extérieur)
    public:

    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Tout ce qui suit est privé (inaccessible depuis l'extérieur)
    private:

    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};
```

Tout ce qui suit le `public:` est public. Donc toutes nos méthodes sont publiques.
Ensuite vient le mot-clé `private:`. Tout ce qui suit ce mot-clé est privé. Donc tous nos attributs sont privés.

Voilà, vous pouvez maintenant compiler ce code, et vous verrez qu'il n'y a pas de problème (même si le code ne fait rien pour l'instant). On appelle des méthodes depuis le main, mais comme elles sont publiques, on a le droit de le faire.
... par contre, nos attributs sont privés, ce qui veut dire qu'on n'a pas le droit de les modifier depuis le main. En clair, on ne peut pas faire depuis le main :

Code : C++

```
goliath.m_vie = 90;
```

Essayez, vous verrez que le compilateur vous ressort la même erreur que tout à l'heure : "ton bidule est private... bla bla bla... pas le droit d'appeler un élément private depuis l'extérieur de la classe".

Mais alors... ça veut dire qu'on ne peut pas modifier la vie du personnage depuis le main ? Eh oui !
C'est nul ? Non au contraire, c'est très bien pensé, ça s'appelle l'encapsulation

L'encapsulation

Moi j'ai une solution ! Si on mettait tout en public ? Les méthodes ET les attributs en public, comme ça on peut tout modifier depuis le main et plus aucun problème !
... quoi j'ai dit une connerie ?

Oh, trois fois rien, vous venez juste de vous faire autant d'ennemis qu'il n'y a de programmeurs qui font de la POO dans le monde

Il y a une règle d'or en POO, et *tout* découle de là. S'il vous plaît, imprimez ceci en gros sur une feuille, et placardez cette feuille sur un mur de votre chambre :

Encapsulation : tous les attributs d'une classe
doivent toujours être privés

Ca a l'air bête, stupide, irréfléchi, et pourtant tout ce qui fait que la POO est un principe puissant vient de là.
En clair, si j'en vois un à partir de maintenant qui me met ne serait-ce qu'un seul attribut en public, je le brûle, je le torture, je l'écorche vif sur la place publique, compris ?
Et vous, si vous voyez quelqu'un d'autre faire ça un jour, écorchez-le vif en pensant à moi, vous serez sympa

Voilà qui explique pourquoi j'ai fait exprès dès le début de mettre les attributs en privé. Comme ça, on ne peut pas les modifier depuis l'extérieur de la classe, et ça respecte le principe d'encapsulation.

Vous vous souvenez de ce schéma du chapitre précédent ?

Les fioles chimiques, ce sont les **attributs**.
Les boutons sur la façade avant, ce sont les **méthodes**.

Et là, pif paf pouf, vous devriez avoir tout compris d'un coup. En effet, le but du modèle objet c'est justement de masquer les informations complexes à l'utilisateur (les attributs) pour éviter qu'il ne fasse des bêtises avec.

Imaginez par exemple que l'utilisateur puisse modifier la vie... qu'est-ce qui l'empêcherait de mettre 150 de vie alors que la limite maximale est 100 ? C'est pour ça qu'il faut toujours passer par des méthodes (des fonctions) qui vont *d'abord* vérifier qu'on fait les choses correctement avant de modifier les attributs.
Cela permet de faire en sorte que le contenu de l'objet reste une "boîte noire". On ne sait pas comment ça fonctionne à l'intérieur quand on l'utilise, et c'est très bien. C'est une sécurité, ça permet d'éviter de faire péter tout le bazard de fioles chimiques à l'intérieur

SÉPARER PROTOTYPES ET DÉFINITIONS

Bon, on avance mais on n'a pas fini
Voici ce que je voudrais qu'on fasse :

- Séparer les méthodes en prototypes et définitions dans 2 fichiers différents pour avoir un code plus modulaire
- Implémenter les méthodes de notre classe Personnage (c'est-à-dire écrire le code à l'intérieur parce que pour le moment y'a rien)

Pour le moment, on a mis notre classe dans le fichier main.cpp, juste au-dessus du main. Et les méthodes sont directement écrites dans la définition de la classe.
Ca fonctionne, mais c'est un peu bourrin. Tout comme on avait appris en C à faire du code modulaire, on va voir comment on procède en POO pour séparer tout ça proprement dans des fichiers différents.

Tout d'abord, il faut clairement séparer le main (qui se trouve dans main.cpp) des classes.
Pour *chaque* classe, on va créer :

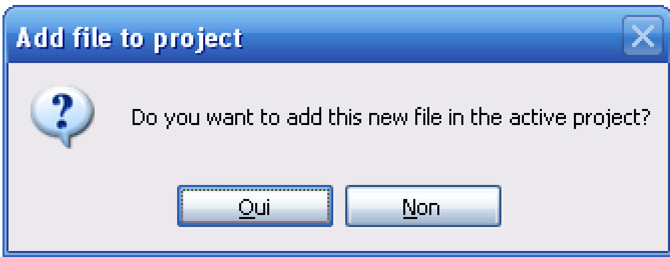
- Un header (*.h) qui contiendra les attributs et les prototypes de la classe
- Un fichier source (*.cpp) qui contiendra la définition des méthodes et leurs implémentations

Je vous propose d'ajouter à votre projet 2 fichiers nommés très exactement :

- Personnage.h
- Personnage.cpp

(vous noterez que je mets aussi une majuscule à la première lettre du nom de fichier, histoire d'être cohérent jusqu'au bout)

Vous devriez être capables de faire ça tous seuls avec votre IDE favori. Sous Code::Blocks, je fais File / New File, je rentre par exemple le nom "Personnage.h" avec l'extension, et je réponds "Oui" quand Code::Blocks me demande si je veux ajouter le nouveau fichier au projet en cours :



Personnage.h

Le fichier .h va donc contenir la déclaration de la classe avec les attributs et les prototypes des méthodes. Dans notre cas, pour la classe Personnage, ça va donner ça :

Code : C++

```
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

class Personnage
{
    public:

    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();


    private:

    int m_vie;
    int m_mana;
    std::string m_nomArme; // Pas de using namespace std, donc il faut mettre std::
devant string.
    int m_degatsArme;
};

#endif
```

Comme vous pouvez le constater, seuls les prototypes des méthodes sont présents dans le .h. C'est déjà beaucoup plus clair

Dans les .h, il est recommandé de ne jamais mettre la directive `using namespace std;` car cela pourrait avoir des effets néfastes lorsque vous utiliserez la classe par la suite.
Par conséquent, il faut rajouter le préfixe "std::" devant chaque string du .h. Sinon, le compilateur vous sortira une erreur du type "`string does not name a type`".

Personnage.cpp

C'est là qu'on va écrire le code de nos méthodes (on dit qu'on **implémente** les méthodes).
Première chose à ne pas oublier, sinon ça va pas bien se passer, c'est d'inclure <string> "Personnage.h".
On peut aussi rajouter ici un `using namespace std;`. On a le droit de le faire car on est dans le .cpp (par contre comme je vous l'ai expliqué plus tôt, il faut éviter de le mettre dans le .h).

Code : C++

```cpp
#include <string>
#include "Personnage.h"

using namespace std;
```

Veillez à inclure <string> AVANT Personnage.h, sinon la déclaration de la classe contenue dans Personnage.h n'aura pas connu au préalable le type string... et donc la compilation plantera.

Maintenant, voilà comment ça se passe : pour chaque méthode, vous devez faire précéder le nom de la méthode par le nom de la classe suivi de deux fois deux points "::". Pour recevoirDegats ça donne ça :

Code : C++

```cpp
void Personnage::recevoirDegats(int nbDegats)
{

}
```

Cela permet au compilateur de savoir que cette méthode se rapporte à la classe Personnage. En effet, comme la méthode est ici écrite en dehors de la définition de la classe, le compilateur n'aurait pas su à quelle classe appartenait cette méthode.

Personnage::recevoirDegats

Maintenant, c'est parti, implémentons la méthode recevoirDegats. Je vous avais expliqué un peu plus haut ce qu'il fallait faire. Vous allez voir, c'est très simple :

Code : C++

```cpp
void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats; // On enlève le nombre de dégâts reçus à la vie du personnage

    if (m_vie < 0) // Pour éviter d'avoir une vie négative
    {
        m_vie = 0; // On met la vie à 0 (ça veut dire mort)
    }
}
```

La méthode modifie donc la valeur de la vie. La méthode a le droit de modifier l'attribut, car elle fait partie de la classe. Ne soyez donc pas surpris, c'est justement l'endroit où on a le droit de toucher aux attributs

La vie est diminuée du nombre de dégâts reçus. En théorie, on aurait pu se contenter de la première instruction, mais on fait une vérification supplémentaire. Si la vie est descendue en-dessous de 0 (parce qu'on a reçu 20 de dégâts alors qu'on n'avait que 10 de vie), on ramène la vie à 0 afin d'éviter d'avoir une vie négative (ça fait pas très pro une vie négative). De toute façon, à 0 de vie, le personnage est considéré comme mort

Et voilà pour la première méthode ! Allez on enchaîne hop hop hop !

Personnage::attaquer

Code : C++

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme); // On inflige à la cible les dégâts que causent notre arme
}
```

Cette méthode est peut-être très courante, elle n'en est pas moins très intéressante !
On reçoit en paramètre une référence vers un objet de type Personnage. On aurait pu recevoir un pointeur aussi, mais comme les références sont plus faciles à manipuler (cf les chapitres précédents) on ne va pas s'en priver.

La référence concerne le personnage cible que l'on doit attaquer. Pour infliger des dégâts à la cible, on appelle sa méthode recevoirDegats en faisant : `cible.recevoirDegats`

> On ne peut pas modifier directement la vie de la cible en faisant `cible.m_vie` car la cible est un AUTRE objet (même s'il est aussi issu de la classe Personnage). On n'a le droit d'accéder qu'aux éléments publics de cet autre objet, donc à ses méthodes.

Quelle quantité de dégâts envoyer à la cible ? Vous avez la réponse sous vos yeux : le nombre de points de dégâts indiqués par l'attribut `m_degatsArme` ! On envoie donc la valeur des m_degatsArme de notre personnage à la cible.

Personnage::boirePotionDeVie

Code : C++

```
void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;

    if (m_vie > 100) // Interdiction de dépasser 100 de vie
    {
        m_vie = 100;
    }
}
```

Le personnage reprend autant de vie que ce que la potion qu'il boit lui permet d'en récupérer. On vérifie au passage qu'il ne dépasse pas les 100 de vie, car comme on l'a dit plus tôt, il est interdit d'avoir plus de 100 de vie.

Personnage::changerArme

Code : C++

```
void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_nomArme = nomNouvelleArme;
    m_degatsArme = degatsNouvelleArme;
}
```

Pour changer d'arme, on stocke dans nos attributs le nom de la nouvelle arme ainsi que ses nouveaux dégâts. Les instructions sont très simples : on fait juste passer ce qu'on a reçu en paramètres dans nos attributs.
Grâce à l'objet string d'ailleurs, il suffit de faire un simple "=" pour affecter la chaîne, et on n'a plus à se préoccuper de la taille du tableau car l'objet string se débrouille tout seul pour ça (à chaque fois que j'y pense je trouve ça génial).

Personnage::estVivant

Code : C++

```
bool Personnage::estVivant()
{
    if (m_vie > 0) // Plus de 0 de vie ?
    {
        return true; // VRAI, il est vivant !
    }
    else
    {
        return false; // FAUX, il n'est plus vivant !
    }
}
```

Cette méthode permet de vérifier si le personnage est toujours vivant. Elle renvoie vrai (true) s'il a plus de 0 de vie, et faux (false) sinon.

Code complet de Personnage.cpp

En résumé, le code complet de notre Personnage.cpp est le suivant :

Code : C++

```
#include "Personnage.h"

using namespace std;

void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats; // On enlève le nombre de dégâts reçus à la vie du personnage

    if (m_vie < 0) // Pour éviter d'avoir une vie négative
    {
        m_vie = 0; // On met la vie à 0 (ça veut dire mort)
    }
}

void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme); // On inflige à la cible les dégâts que causent
notre arme
}

void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;

    if (m_vie > 100) // Interdiction de dépasser 100 de vie
    {
        m_vie = 100;
    }
}

void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_nomArme = nomNouvelleArme;
    m_degatsArme = degatsNouvelleArme;
}

bool Personnage::estVivant()
{
    if (m_vie > 0) // Plus de 0 de vie ?
    {
        return true; // VRAI, il est vivant !
    }
    else
    {
        return false; // FAUX, il n'est plus vivant !
    }
}
```

main.cpp

Retour au main. Première chose à ne pas oublier : inclure Personnage.h pour pouvoir créer des objets de type Personnage.

Code : C++

```
#include "Personnage.h" // Ne pas oublier
```

Après, le main reste le même que tout à l'heure, on n'a pas besoin de le changer. Au final, le code du main est donc très court, et le fichier main.cpp ne fait qu'**utiliser** les objets :

Code : C++

```
#include <iostream>
#include <string>
#include "Personnage.h" // Ne pas oublier

using namespace std;


int main()
{
    Personnage david, goliath; // Création de 2 objets de type Personnage : david et goliath

    goliath.attaquer(david); // goliath attaque david
    david.boirePotionDeVie(20); // david boit une potion de vie qui lui rapporte 20 de vie
    goliath.attaquer(david); // goliath réattaque david
    david.attaquer(goliath); // david contre-attaque... c'est assez clair non ? ^^
    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
    goliath.attaquer(david);


    return 0;
}
```

N'exécutez pas le programme pour le moment. En effet, nous n'avons toujours pas vu comment faire pour initialiser les attributs, ce qui fait que notre programme n'est pas encore utilisable.
Nous verrons comment le rendre pleinement fonctionnel dans le chapitre suivant, et vous pourrez alors (enfin) l'exécuter

Il faudra donc pour le moment vous contenter de votre imagination. Essayez d'imaginer que David et Goliath sont bien en train de combattre ! (et je veux pas faire mon gros spoiler, mais normalement c'est David qui gagne à la fin). Là, on peut dire qu'on est rentré en plein dans la POO
Pourtant, ce n'est encore qu'un début ! De nombreuses nouvelles choses complètement dingues vous attendent dans les chapitres qui suivent (et elles vont vous rendre dingues ça c'est sûr)

Un conseil si je puis me permettre : assurez-vous d'avoir bien compris qu'il y avait deux faces dans la POO, la création de la classe, et l'utilisation des objets. Il faut être à l'aise avec ce concept.
Mais tout n'est pas si simple. Comme vous le verrez, ce que font les objets la plupart du temps c'est... utiliser d'autres objets ! Et c'est en combinant plusieurs objets entre eux que l'on découvrira le vrai pouvoir de la POO

Les classes (Partie 2/2)

Allez, hop hop hop, on enchaîne ! Pas question de s'endormir, on est en plein dans la POO là

Dans le chapitre précédent, nous avons appris à créer une classe basique, à rendre le code modulaire en POO, et surtout nous avons découvert le principe d'encapsulation (suuuper important l'encapsulation, c'est la base de tout je le rappelle).

Dans cette seconde partie du chapitre, nous allons découvrir comment initialiser nos attributs à l'aide d'un constructeur, un élément indispensable à toute classe qui se respecte. Puisqu'on parlera de constructeur, on parlera aussi de destructeur, ça va de paire vous verrez.
Nous complèterons notre classe Personnage et nous l'associerons avec une nouvelle classe Arme que nous allons créer. Nous découvrirons alors tout le pouvoir qu'il y a de combiner des classes entre elles, et vous devriez normalement commencer à imaginer pas mal de possibilités à partir de là

CONSTRUCTEUR ET DESTRUCTEUR

Reprenons. Nous avons maintenant 3 fichiers :

- **main.cpp** : il contient le main, dans lequel on a créé 2 objets de type Personnage : david et goliath.
- **Personnage.h** : c'est le header de la classe Personnage. On y liste les prototypes des méthodes et les attributs. On y définit la portée (public / private) de chacun des éléments. Pour respecter le principe d'encapsulation, tous nos attributs sont privés, c'est-à-dire non accessibles de l'extérieur.
- **Personnage.cpp** : c'est le fichier dans lequel on implémente nos méthodes, c'est-à-dire qu'on écrit le code source des méthodes.

Pour l'instant, nous avons défini et implémenté pas mal de méthodes. Je voudrais vous parler ici de 2 méthodes particulières que l'on retrouve dans la plupart des classes : le constructeur et le destructeur.

- **Le constructeur** : c'est une méthode qui est appelée automatiquement à chaque fois que l'on crée un objet basé sur cette classe.
- **Le destructeur** : c'est une méthode qui est automatiquement appelée lorsqu'un objet est détruit, par exemple à la fin de la fonction dans laquelle il a été déclaré ou lors d'un delete si l'objet a été alloué dynamiquement avec new.

Voyons voir plus en détail comment fonctionnent ces méthodes un peu particulières...

Le constructeur

Comme son nom l'indique, c'est une méthode qui sert à *construire* l'objet. Dès qu'on crée un objet, le constructeur est automatiquement appelé s'il existe.

Par exemple, lorsqu'on fait dans notre main :

Code : C++

```cpp
Personnage david, goliath;
```

S'il existe, le constructeur de l'objet david est appelé, et de même pour le constructeur de l'objet goliath.
Mais... comme nous n'avons pas encore défini de constructeur dans la classe Personnage, rien de particulier ne s'est passé. Le constructeur n'est pas obligatoire, mais on a presque toujours besoin d'en créer un, vous allez vite comprendre pourquoi.

Le rôle du constructeur

Si le constructeur est appelé lors de la création de l'objet, ce n'est pas pour faire joli. En fait, le rôle principal du constructeur est d'*initialiser* les attributs.
En effet, souvenez-vous : nos attributs sont déclarés dans Personnage.h, mais pas initialisés !

Revoici Personnage.h :

Code : C++

```
#include <string>

class Personnage
{
    public:

    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();


    private:

    int m_vie;
    int m_mana;
    std::string m_nomArme;
    int m_degatsArme;
};
```

Nos attributs m_vie, m_mana, et m_degatsArmes ne sont pas initialisés ! Pourquoi ? Parce qu'on n'a pas le droit d'initialiser les attributs ici. C'est justement dans le constructeur qu'il faut le faire.

En fait, le constructeur est indispensable pour initialiser les attributs qui ne sont pas des objets (type classique : int, double, char...). En effet, ceux-ci ont une valeur inconnue en mémoire (ça peut être 0 comme -3451).
En revanche, les attributs qui sont des objets, comme c'est le cas de m_nomArme ici qui est un string, sont automatiquement initialisés par le langage C++ avec une valeur par défaut.

Créer un constructeur

Le constructeur est une méthode, mais une méthode un peu particulière.
En effet, pour créer un constructeur, il y a 2 règles à respecter :

- Il faut que la méthode aie le même nom que la classe. Dans notre cas, la méthode devra s'appeler "Personnage".
- La méthode ne doit RIEN renvoyer, pas même void ! C'est une méthode sans aucun type de retour.

Si on déclare son prototype dans Personnage.h, ça donne ça :

Code : C++

```
#include <string>

class Personnage
{
    public:

    Personnage(); // Constructeur
    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();


    private:

    int m_vie;
    int m_mana;
    std::string m_nomArme;
    int m_degatsArme;
};
```

Le constructeur se voit du premier coup d'oeil : déjà parce qu'il n'a aucun type de retour (pas de void ni rien), et ensuite parce qu'il a le même nom que la classe

Et si on en profitait pour implémenter ce constructeur dans Personnage.cpp maintenant ?
Voici à quoi pourrait ressembler son implémentation :

Code : C++

```
Personnage::Personnage()
{
    m_vie = 100;
    m_mana = 100;
    m_nomArme = "Epée rouillée";
    m_degatsArme = 10;
}
```

Vous noterez une fois de plus qu'il n'y a pas de type de retour, pas même void (très important, c'est une erreur que l'on fait souvent).
J'ai choisi de mettre la vie et la mana à 100, le maximum, ce qui est logique. J'ai mis par défaut une arme appelée "Epée rouillée" qui fait 10 de dégâts à chaque coup.

Et voilà ! Notre classe Personnage a un constructeur qui initialise les attributs, elle est désormais pleinement utilisable
Maintenant, à chaque fois que l'on crée un objet de type Personnage, celui-ci est initialisé à 100 points de vie et de mana, avec l'arme "Epée rouillée". Nos deux compères david et goliath commencent donc à égalité lorsqu'ils sont créés dans le main :

Code : C++

```
Personnage david, goliath; // Les constructeurs de david et goliath sont appelés.
```

Autre façon d'initialiser avec un constructeur : la liste d'initialisation

Le C++ permet d'initialiser les attributs de la classe d'une autre manière (un peu déroutante) appelée **liste d'initialisation**.
Reprenons le constructeur qu'on vient de créer :

Code : C++

```
Personnage::Personnage()
{
    m_vie = 100;
    m_mana = 100;
    m_nomArme = "Epée rouillée";
    m_degatsArme = 10;
}
```

Le code que vous allez voir ci-dessous produit le même effet :

Code : C++

```
Personnage::Personnage() : m_vie(100), m_mana(100), m_nomArme("Epée rouillée"),
m_degatsArme(10)
{
    // Rien à mettre dans le corps du constructeur, tout a déjà été fait !
}
```

La nouveauté, c'est qu'on rajoute un symbole deux-points (:) suivi de la liste des attributs que l'on veut initialiser avec la valeur entre parenthèses. Avec ce code, on initialise la vie à 100, la mana à 100, l'attribut m_nomArme à "Epée rouillée", etc.

Cette technique est un peu surprenante, surtout que du coup on n'a plus rien à mettre dans le corps du constructeur entre les accolades, vu que tout a déjà été fait avant ! Elle a toutefois l'avantage d'être "plus propre" et se révèlera pratique dans la suite du chapitre.
On va donc utiliser autant que possible les listes d'initialisation avec les constructeurs, c'est une bonne habitude à prendre.

Le prototype du constructeur (dans le .h) ne change pas. Toute la partie après les deux-points n'apparaît pas dans le prototype.

Surcharger le constructeur

Vous savez qu'en C++ on a le droit de surcharger les fonctions, donc de surcharger les méthodes. Et comme le constructeur est une méthode, on a le droit de le surcharger lui aussi.
Pourquoi je vous en parle ? Ce n'est pas par hasard : en fait, le constructeur est une méthode que l'on a tendance à beaucoup surcharger. Cela permet de créer un objet de plusieurs façons différentes.

Pour l'instant, on a créé un constructeur sans paramètres :

Code : C++

```
Personnage();
```

On appelle ça : **le constructeur par défaut** (il fallait bien lui donner un nom le pauvre).

Supposons que l'on souhaite créer un personnage qui ait dès le départ une meilleure arme... comment diable faire ? C'est là que la surcharge devient utile. On va créer un 2ème constructeur qui prendra en paramètre le nom de l'arme et ses dégâts.

Dans Personnage.h, on va donc rajouter ce prototype :

Code : C++

```
Personnage(std::string nomArme, int degatsArme);
```

Le préfixe std:: est obligatoire ici comme je vous l'ai dit plus tôt car on n'utilise pas la directive `using namespace std;` dans le .h (cf chapitre précédent).

L'implémentation dans Personnage.cpp sera la suivante :

Code : C++

```
Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100),
m_nomArme(nomArme), m_degatsArme(degatsArme)
{

}
```

Vous noterez ici tout l'intérêt de mettre le préfixe m_ au début des attributs : comme ça on peut faire la différence dans notre code entre *m_nomArme*, qui est un attribut, et *nomArme*, qui est le paramètre envoyé au constructeur. Ce qu'on fait ici, c'est juste placer dans l'attribut de l'objet le nom de l'arme envoyé en paramètre. On recopie juste la valeur. C'est tout bête, mais il faut le faire, sinon l'objet ne se "souviendra pas" du nom de l'arme qu'il possède.

La vie et la mana, eux, sont toujours fixés à 100 (il faut bien les initialiser), mais l'arme, elle, peut maintenant être indiquée par l'utilisateur lorsqu'il crée l'objet.

Quel utilisateur ?

Souvenez-vous, l'utilisateur c'est celui qui crée et utilise les objets. Le concepteur c'est celui qui crée les classes. Dans notre cas, la création des objets est faite dans le main. Pour le moment, la création de nos objets ressemble à ça :

Code : C++

```
Personnage david, goliath;
```

Comme on n'a spécifié aucun paramètre, c'est le constructeur par défaut (celui sans paramètres) qui sera appelé. Maintenant supposons que l'on veuille donner dès le départ une meilleure arme à Goliath (c'est lui le plus fort après tout). On va indiquer entre parenthèses le nom et la puissance de cette arme :

Code : C++

```
Personnage david, goliath("Epée aiguisée", 20);
```

Goliath est équipé de l'épée aiguisée dès sa création. David est équipé de l'arme par défaut, l'épée rouillée. Comme on n'a spécifié aucun paramètre lors de la création de david, c'est le constructeur par défaut qui sera appelé pour lui. Pour goliath, comme on a spécifié des paramètres, c'est le constructeur correspondant à la signature (string, int) qui sera appelé.

Si vous avez oublié ce qu'est une signature de fonction (ou de méthode, c'est pareil), je vous invite très fortement à relire ce passage du cours, que vous avez normalement dû lire quelques chapitres plus tôt

Exercice : on aurait aussi pu permettre à l'utilisateur de modifier la vie et la mana de départ, mais je ne l'ai pas fait ici. Ce n'est pas compliqué, vous pouvez le faire pour vous entraîner. Ca vous fera un troisième constructeur surchargé

Le destructeur

Le destructeur est une méthode appelée lorsque l'objet est supprimé de la mémoire. Son principal rôle est de désallouer la mémoire (via des delete) qui a été allouée dynamiquement.

Dans le cas de notre classe Personnage, on n'a fait aucune allocation dynamique (il n'y a aucun new). Le destructeur est donc inutile. Cependant, vous en aurez certainement besoin un jour où l'autre, car on est souvent amené à faire des allocations dynamiques.
Tenez, l'objet string par exemple, vous croyez qu'il fonctionne comment ? Il a un destructeur qui lui permet, juste avant la destruction de l'objet, de supprimer le tableau de char qu'il a alloué dynamiquement en mémoire. Il fait donc un delete sur le tableau de char, ce qui permet de garder une mémoire propre et d'éviter les fameuses "fuites de mémoire"

Créer un destructeur

Bien que ce soit inutile dans notre cas (je n'ai pas mis d'allocations dynamiques pour ne pas trop compliquer de suite), je vais vous montrer comment on crée un destructeur. Voici les règles à suivre :

- Un destructeur est une méthode qui commence par un tilde ~ suivi du nom de la classe
- Un destructeur ne renvoie aucune valeur, pas même void (comme le constructeur)
- Et, nouveauté : le destructeur ne peut prendre aucun paramètre. Il y a donc toujours un seul destructeur, il ne peut pas être surchargé.

Dans Personnage.h, le prototype du destructeur sera donc :

Code : C++

```
~Personnage();
```

Dans Personnage.cpp, l'implémentation sera :

Code : C++

```
Personnage::~Personnage()
{
    /* Rien à mettre ici car on ne fait pas d'allocation dynamique
    dans la classe Personnage. Le destructeur est donc inutile mais
    je le mets pour montrer à quoi ça ressemble ^^
    En temps normal, un destructeur fait souvent des delete et quelques
    autres vérifications si nécessaire avant la destruction de l'objet */
}
```

Bon vous l'aurez compris, mon destructeur ne fait rien. C'était même pas le peine de le créer (il n'est pas obligatoire après tout).
Cela vous montre néanmoins la procédure à suivre. Soyez rassurés, nous ferons des allocations dynamiques plus tôt que vous ne le pensez (je sais je suis diabolique), et nous aurons alors grand besoin du destructeur pour désallouer la mémoire !

ASSOCIER DES CLASSES ENTRE ELLES

La programmation orientée objet devient vraiment intéressante et puissante lorsqu'on se met à combiner plusieurs

objets entre eux. Pour l'instant, nous n'avons créé qu'une seule classe : Personnage.
Or en pratique, un programme objet est un programme constitué d'une multitude d'objets différents !

Il n'y a pas de secret, c'est en pratiquant que l'on apprend petit à petit à penser objet.
Ce que nous allons voir par la suite ne sera pas nouveau : vous allez réutiliser tout ce que vous savez déjà sur la création de classes, de manière à améliorer notre petit RPG et à vous entraîner encore plus à manipuler des objets

La classe Arme

Ce que je vous propose dans un premier temps, c'est de créer une nouvelle classe Arme. Plutôt que de mettre les informations de l'arme (m_nomArme, m_degatsArme) directement dans le Personnage, **nous allons l'équiper d'un objet de type Arme**. Le découpage de notre programme sera alors un peu plus dans la logique d'un programme orienté objet.

Souvenez-vous ce que je vous ai dit au début : il y a 100 façons différentes de concevoir un même programme en POO. Tout est dans l'organisation des classes entre elles, comment elles communiquent, etc.
Ce que nous avons fait jusqu'ici était pas mal, mais je veux vous montrer ici qu'on peut faire *autrement*, un peu plus dans l'esprit objet, donc... mieux

Qui dit nouvelle classe dit 2 nouveaux fichiers :

- Arme.h : contient la définition de la classe
- Arme.cpp : contient l'implémentation des méthodes de la classe

On n'est pas obligé de procéder ainsi. On pourrait tout mettre dans un seul fichier. On pourrait même mettre plusieurs classes par fichier, rien ne l'interdit en C++. Cependant, pour des raisons d'organisation, je vous recommande de faire comme moi.

Arme.h

Voici ce que je propose de mettre dans Arme.h :

Code : C++

```
#ifndef DEF_ARME
#define DEF_ARME

class Arme
{
    public:

    Arme();
    Arme(std::string nom, int degats);
    void changer(std::string nom, int degats);
    void afficher();

    private:

    std::string m_nom;
    int m_degats;
};

#endif
```

Mis à part les includes qu'il ne faut pas oublier, le reste de la classe est très simple.

On met le nom de l'arme et ses dégâts dans des attributs, et comme ce sont des attributs, on vérifie qu'ils soient bien privés (encapsulation). Vous remarquerez qu'au lieu de m_nomArme et m_degatsArme, j'ai choisi de nommer mes attributs m_nom et m_degats tout simplement. C'est plus logique en effet : vu qu'on est *déjà* dans l'Arme, ce n'est pas la peine de repréciser dans les attributs qu'il s'agit de l'arme, on le sait déjà, on est dedans

Ensuite, on ajoute un ou deux constructeurs, une méthode pour changer d'arme à tout moment, et une autre allez, soyons fous , pour afficher le contenu de l'arme.

Reste à implémenter toutes ces méthodes dans Arme.cpp. Pfeuh, fastoche !

Arme.cpp

Entraînez-vous à écrire Arme.cpp, c'est tout bête, les méthodes font maxi 2 lignes, bref c'est à la portée de tout le monde

Voici mon Arme.cpp pour comparer :

Code : C++

```
#include <iostream>
#include <string>
#include "Arme.h"

using namespace std;

Arme::Arme() : m_nom("Epée rouillée"), m_degats(10)
{

}

Arme::Arme(string nom, int degats) : m_nom(nom), m_degats(degats)
{

}

void Arme::changer(string nom, int degats)
{
    m_nom = nom;
    m_degats = degats;
}

void Arme::afficher()
{
    cout << m_nom << " (Dégâts : " << m_degats << ")" << endl;
}
```

Bon là je n'ai rien à ajouter vraiment, c'est beaucoup trop simple
N'oubliez quand même pas d'inclure "Arme.h" si vous voulez que ça marche

Et ensuite ?

Bon, notre classe Arme est créée, c'est bon pour ça. Mais maintenant, il va falloir adapter la classe Personnage pour qu'elle utilise non pas m_nomArme et m_degatsArme, mais un objet de type Arme.
Et là... c'est là que ça se complique

Adapter la classe Personnage pour utiliser une Arme

La classe Personnage va subir quelques modifications pour utiliser la classe Arme. Restez attentifs, car utiliser un

objet DANS un objet, c'est un peu particulier.

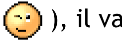
Personnage.h

Zou, direction le .h. On commence par virer nos 2 attributs m_nomArme et m_degatsArme qui ne servent plus à rien.

Les méthodes n'ont pas besoin d'être changées. En fait, il ne vaut mieux pas les changer. Pourquoi ? Parce que les méthodes sont déjà potentiellement utilisées par quelqu'un (par exemple dans notre main). Si on les renomme ou si on en supprime, notre programme ne fonctionnera plus.

Ce n'est peut-être pas grave pour un si petit programme, mais dans le cas d'un gros programme si on supprime une méthode, c'est la cata assurée dans le reste du programme. Et je vous parle même pas de ceux qui écrivent des librairies C++ : si d'une version à l'autre des méthodes disparaissent, tous les programmes qui utilisent la librairie ne fonctionneront plus !

Je vais peut-être vous surprendre en vous disant ça, mais c'est là tout l'intérêt de la programmation orientée objet, et plus particulièrement de l'**encapsulation**. On peut changer nos attributs comme on veut, vu qu'ils ne sont pas accessibles de l'extérieur, on ne prend pas le *risque* que quelqu'un les utilise déjà dans le programme.
En revanche, pour les méthodes, faites plus attention. Vous pouvez ajouter de nouvelles méthodes, modifier l'implémentation des méthodes existantes, mais PAS en supprimer ou en renommer, sinon l'utilisateur risque d'avoir des problèmes.

Cette petite réflexion sur l'encapsulation étant faite (vous en comprendrez tout le sens avec la pratique ), il va falloir ajouter un objet de type Arme à notre Personnage.

Il faut penser à ajouter un include de "Arme.h" si on veut pouvoir utiliser un objet de type Arme.

Voici mon nouveau Personnage.h :

Code : C++

```
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

#include "Arme.h" // Ne PAS oublier d'inclure Arme.h pour en avoir la définition

class Personnage
{
    public:

    Personnage();
    ~Personnage();
    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();


    private:

    int m_vie;
    int m_mana;
    Arme m_arme; // Notre arme est "contenue" dans le Personnage
};

#endif
```

Personnage.cpp

Nous n'avons besoin de changer que les méthodes qui utilisent l'arme pour les adapter.
On commence par les constructeurs :

Code : C++

```cpp
Personnage::Personnage() : m_vie(100), m_mana(100)
{

}

Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100),
m_arme(nomArme, degatsArme)
{

}
```

Notre objet m_arme est ici initialisé avec les valeurs reçues en paramètre par Personnage (nomArme, degatsArme). C'est là que la liste d'initialisation devient utile. En effet, on n'aurait pas pu initialiser m_arme sans une liste d'initialisation !

Peut-être ne voyez-vous pas bien pourquoi. Conseil perso : ne vous prenez pas la tête à essayer de comprendre le pourquoi du comment ici, et contentez-vous de toujours utiliser les listes d'initialisation avec vos constructeurs, ça vous évitera bien des problèmes.

Revenons au code.
Dans le premier constructeur, c'est le constructeur par défaut de la classe Arme qui est appelé, tandis que dans le second c'est celui ayant la signature (string, int) qui est appelé.

La méthode recevoirDegats n'a pas besoin de changer.
En revanche, la méthode attaquer est délicate. En effet, on ne peut pas faire :

Code : C++

```cpp
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme.m_degats);
}
```

Pourquoi est-ce interdit ? Parce que m_degats est un attribut, et que comme tout bon attribut qui se respecte, il est *privé* ! Diantre... On est en train d'utiliser la classe Arme au sein de la classe Personnage, et comme on est utilisateurs, on ne peut pas accéder aux éléments privés

(La POO, ça peut parfois donner mal à la tête j'avais oublié de vous prévenir)

Bon, comment résoudre le problème ? Il n'y a pas 36 solutions. Ca va peut-être vous surprendre, mais on doit créer une méthode pour récupérer la valeur de cet attribut. Cette méthode est appelée accesseur et commence généralement par le préfixe get (*récupérer*, en anglais). Dans notre cas, notre méthode s'appellerait getDegats.

On conseille généralement de rajouter le mot-clé const aux accesseurs.

Une méthode... constante ? Qu'est-ce que ça signifie ?

Une méthode constante est une méthode qui ne peut pas modifier les attributs de la classe. Cela garantit que la méthode ne fait que "lire" les attributs et qu'elle ne modifie donc pas l'objet. C'est une bonne habitude de

programmation de créer des accesseur const, bien que là encore ça ne soit pas obligatoire.

Voici à quoi ressemble la méthode, avec le mot-clé const :

Code : C++

```
int Arme::getDegats() const
{
    return m_degats;
}
```

Oubliez pas de mettre à jour Arme.h avec le prototype aussi, qui sera le suivant :

Code : C++

```
int Arme::getDegats() const;
```

Voilà, c'est con comme bonjour, ça peut paraître lourd, et pourtant c'est une sécurité nécessaire. On est parfois obligé de créer une méthode qui fait juste un return pour accéder indirectement à un attribut.

De même, on crée parfois des accesseurs permettant de modifier des attributs. Ces accesseurs sont généralement précédés du préfixe *set* (*mettre*, en anglais).
Vous avez peut-être l'impression qu'on viole la règle d'encapsulation ? Eh bien non. Car la méthode nous permet de faire des tests pour vérifier qu'on ne met pas n'importe quoi dans l'attribut, donc ça reste une façon sécurisée de modifier un attribut.

Vous pouvez maintenant retourner dans Personnage.cpp et écrire :

Code : C++

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme.getDegats());
}
```

getDegats renvoie le nombre de dégâts, qu'on envoie à la méthode recevoirDegats de la cible. Pfiou !

Le reste des méthodes n'a pas besoin de changer, à part changerArme de la classe Personnage :

Code : C++

```

void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_arme.changer(nomNouvelleArme, degatsNouvelleArme);
}
```

On appelle la méthode changer de m_arme.
Le Personnage répercute donc la demande de changement d'arme à la méthode changer de son objet m_arme.

Comme vous pouvez le voir, on peut faire communiquer des objets entre eux, à condition d'être bien organisé et de se demander à chaque instant "est-ce que j'ai le droit d'accéder à cet élément ou pas ?".
N'hésitez pas à créer des accesseurs si besoin est, même si ça peut paraître lourd c'est la bonne méthode. En aucun cas vous ne devez mettre un attribut public pour simplifier un problème. Vous perdriez tous les avantages et la sécurité de la POO (et vous n'auriez aucun intérêt à continuer le C++ dans ce cas).

ACTION !

Nos personnages combattent dans le main, mais... on ne voit rien de ce qui se passe. Il serait bien d'afficher l'état de chacun des personnages pour savoir où ils en sont.

Je vous propose de créer une méthode afficherEtat dans Personnage. Cette méthode sera chargée de faire des *cout* pour afficher dans la console la vie, la mana et l'arme du personnage.

Prototype et include

On va rajouter le prototype, tout bête, dans le .h :

Code : C++

```cpp
void afficherEtat();
```

Implémentation

Implémentons ensuite la méthode. C'est simple, on a juste des cout à faire. Grâce aux attributs, on peut indiquer toutes les infos sur le personnage :

Code : C++

```cpp
void Personnage::afficherEtat()
{
    cout << "Vie : " << m_vie << endl;
    cout << "Mana : " << m_mana << endl;
    m_arme.afficher();
}
```

Comme vous pouvez le voir, les informations sur l'arme sont demandées à l'objet m_arme via sa méthode afficher(). Encore une fois, les objets communiquent entre eux pour récupérer les informations dont ils ont besoin.

Appel de afficherEtat dans le main

Bien, tout ça c'est bien beau, mais tant qu'on n'appelle pas la méthode, elle ne sert à rien
Je vous propose donc de compléter le main et de rajouter à la fin les appels de méthode :

Code : C++

```cpp
int main()
{
    // Création des personnages
    Personnage david, goliath("Epée aiguisée", 20);

    // Au combat !
    goliath.attaquer(david);
    david.boirePotionDeVie(20);
    goliath.attaquer(david);
    david.attaquer(goliath);
    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
    goliath.attaquer(david);

    // Temps mort ! Voyons voir la vie de chacun...
    cout << "David" << endl;
    david.afficherEtat();
    cout << endl << "Goliath" << endl;
    goliath.afficherEtat();

    return 0;
}
```

On peut *enfin* exécuter le programme et voir quelque chose dans la console

Code : Console

```
David
Vie : 40
Mana : 100
Arme : Epée rouillée (Degats : 10)

Goliath
Vie : 90
Mana : 100
Arme : Double hache tranchante vénéneuse de la mort (Degats : 40)
```

Si vous êtes sous Windows, vous aurez probablement un bug avec les accents dans la console. Ignorez-le, ne vous en préoccupez pas, ce qui nous intéresse c'est le fonctionnement de la POO ici. Et puis de toute manière, dans la prochaine partie du cours on travaillera avec de vraies fenêtres, donc la console c'est temporaire pour nous

Pour que vous puissiez vous faire une bonne idée du projet dans son ensemble, je vous propose de télécharger un fichier zip contenant :

- main.cpp
- Personnage.cpp
- Personnage.h

... bref, c'est-à-dire tout le projet tel qu'il est sur mon ordinateur à l'heure actuelle.

Télécharger le projet RPG (3 Ko)

Je vous invite à faire des tests pour vous entraîner. Par exemple :

- Continuez à faire combattre david et goliath dans le main en affichant leur état de temps en temps.
- Introduisez un troisième personnage dans l'arène pour rendre le combat plus ~~brutal~~ intéressant
- Rajoutez un attribut m_nom pour stocker le nom du personnage dans l'objet. Pour le moment, nos personnages ne savent même pas comment ils s'appellent, c'est un peu bête
  Du coup, je pense qu'il faudrait modifier les constructeurs et obliger l'utilisateur à indiquer un nom pour le personnage lors de sa création... à moins que vous ne donniez un nom par défaut si rien n'est précisé ? A vous de choisir !
- Rajoutez des cout dans les autres méthodes de Personnage pour indiquer à chaque fois ce qui est en train de se passer ("machin boit une potion qui lui redonne 20 points de vie")
- Rajoutez d'autres méthodes au gré de votre imagination... et pourquoi pas des attaques magiques qui utilisent de la mana ?
- Enfin, pour l'instant le combat est tout écrit dans le main, mais vous pourriez laisser le joueur choisir les attaques dans la console. Vous savez le faire, allez allez !

Prenez cet exercice très au sérieux, ceci est peut-être la base de votre futur MMORPG révolutionnaire !

Précision utile : la phrase ci-dessus était une boutade
Ce cours ne vous apprendra pas à créer un MMORPG, vu le travail phénoménal que cela représente. Mieux vaut commencer par se concentrer sur de plus petits projets réalistes, et notre RPG en est un. Ce qui est intéressant ici, c'est de voir comment est conçu un jeu orienté objet (comme c'est le cas de la plupart des jeux aujourd'hui). Si vous avez bien compris le principe, vous devriez commencer à voir des objets dans tous les jeux que vous connaissez ! Par exemple, un bâtiment dans Age of Empires est un objet qui a un niveau de vie, un nom, il peut produire des unités (via une méthode), etc.

Si vous commencez à voir des objets partout, c'est bon signe ! C'est ce que l'on appelle "penser objet" Si vous avez dû retenir une bonne chose de ce second chapitre, c'est cet échange, cette communication constante entre les objets. Et encore ! On n'avait ici que 2 classes, Personnage et Arme. Je vous laisse imaginer dans un vrai projet ce que ça donne
L'intérêt de la POO est là : une organisation précise, chaque objet fait ce qu'il a à faire et délègue certaines parties de son travail à d'autres objets (ici, Personnage déléguait la gestion de l'arme à un objet de type Arme).

On ne peut pas dire "Je fais de la POO" du jour au lendemain, c'est clair. C'est un travail qui demande de l'organisation, de la méthode. Il faut toujours bien réfléchir avant de se lancer dans un projet, si simple soit-il. Mais réfléchir un peu avant de programmer, est-ce un mal ? Je ne crois pas

Concentrez-vous sur le fichier zip que je vous ai donné et essayez de vous familiariser avec, en faisant par exemple les améliorations proposées. Il ne faut surtout pas que vous soyez perdus.

Dans le chapitre suivant, nous découvrirons la surcharge d'opérateur. Ce sera un chapitre relativement cool pour souffler, avant d'attaquer le gros du morceau : l'héritage. Du très lourd en perspective

PARTIE 5 : ANNEXES

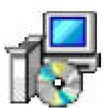
Dans cette partie, vous trouverez des chapitres annexes au cours.
Ils ne sont pas à lire à la fin : vous pouvez les lire n'importe quand. Si certains demandent d'avoir lu au moins quelques chapitres du cours, cela sera indiqué dans l'introduction.

Ne négligez pas les annexes, vous y trouverez sûrement de nouvelles informations intéressantes !

Créer une installation

Lorsque vous commencerez à faire des programmes assez gros, vous aurez sûrement envie de créer **un programme d'installation**.

Jusqu'ici, vous donniez votre programme dans un fichier .zip qu'il fallait décompresser. Ok, ça va un peu, mais quand on veut faire un programme sérieux à distribuer, on aimerait bien pouvoir créer une installation professionnelle. C'est justement l'objet de cette annexe



Cette annexe vous montrera comment je crée un programme d'installation pour le jeu Mario Sokoban réalisé avec la librairie SDL.
Ce jeu a fait l'objet d'un TP dans la partie III sur la librairie SDL.
Bien entendu, ce que je vous montre là sera adaptable pour n'importe quel type de programme, qu'il soit réalisé en C, en C++, en Java, en Python ou que sais-je encore

Cette annexe concerne la création d'installations pour Windows uniquement. Sous les autres systèmes d'exploitation, le concept d'*Assistant d'installation* est moins courant (on fait télécharger des .zip ou des .tar.gz).
C'est sous Windows que l'on trouve le plus de choix au niveau des programmes de création d'installation.

Ne créez pas des installations à tout-va ! Une installation est utile lorsque vous avez terminé un programme sérieux et que vous voulez le diffuser en version finale. Pour toutes les versions intermédiaires de test que vous voudriez transmettre à vos amis, utilisez plutôt un fichier ZIP : ça reste le plus pratique.

TÉLÉCHARGER INNO SETUP

En général, on ne code pas nous-mêmes le programme d'installation. Ce serait bien trop long, une vraie perte de temps.
En plus, c'est assez compliqué car le programme d'installation doit combiner tous les fichiers du programme dans un seul gros .exe, et il doit les compresser aussi !

Cela fait que les programmes d'installation sont vraiment adaptés à une distribution sur Internet. Ils prennent le moins de place possible, et tous les fichiers du programme sont empaquetés dans le .exe de l'installation

Les outils de création d'installation

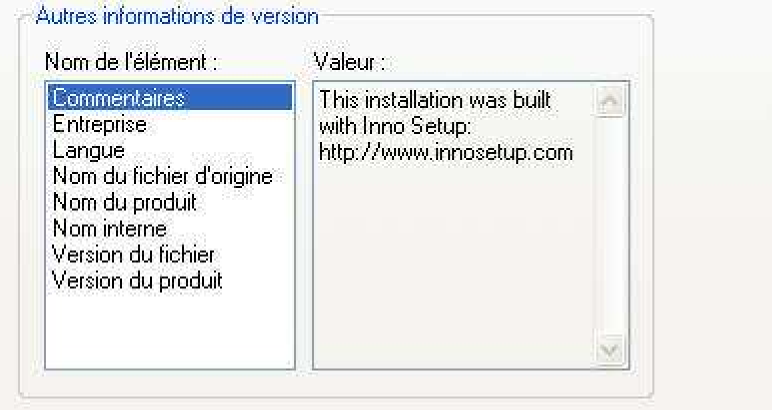
Il existe de nombreux outils permettant de créer une installation.
Je ne vais pas vous faire une liste ici, mais citons quand même **InstallShield** (ce nom doit vous dire quelque chose). C'est un créateur d'installation payant très souvent utilisé.

Il existe aussi de nombreux outils gratuits. Il y a par exemple **NullSoft Install System (NSYS)** créé au départ pour le logiciel Winamp puis rendu gratuit à la disposition de tout le monde.

L'outil que je vais vous présenter ici est très connu et réputé. Son nom est **Inno Setup**. Il possède les avantages suivants :

- Gratuit
- Open Source
- Très discret et professionnel : il n'affiche pas de message "Installation créée avec Bidule Truc".
- Multilingue : il gère les installations dans différentes langues à la fois au besoin.
- Très facile à utiliser : il y a un assistant.
- Très personnalisable et puissant : on peut choisir de nombreuses options, de l'image affichée pendant l'installation aux clés de la base de registre à modifier, en passant par les raccourcis du menu démarrer.

En fait, ce qui est vraiment bien c'est que le programme n'affiche aucun message indiquant que l'installation a été créée avec Inno Setup. Il y a juste un petit commentaire (mais il faut aller le chercher !).
Si vous faites un clic droit sur le .exe d'un programme d'installation, puis propriétés, onglet "Version", vous verrez le petit commentaire suivant :



C'est donc ultra-discret (vos utilisateurs ne le verront probablement jamais).
D'ailleurs j'ai un petit jeu à vous proposer : faites le test sur tous les programmes d'installation que vous avez sur votre disque dur. Comptez le nombre d'installations que vous avez qui ont utilisé Inno Setup : il y en a plein !

Cela devrait vous rassurer, car c'est un programme très utilisé qui ne manque pas de qualités

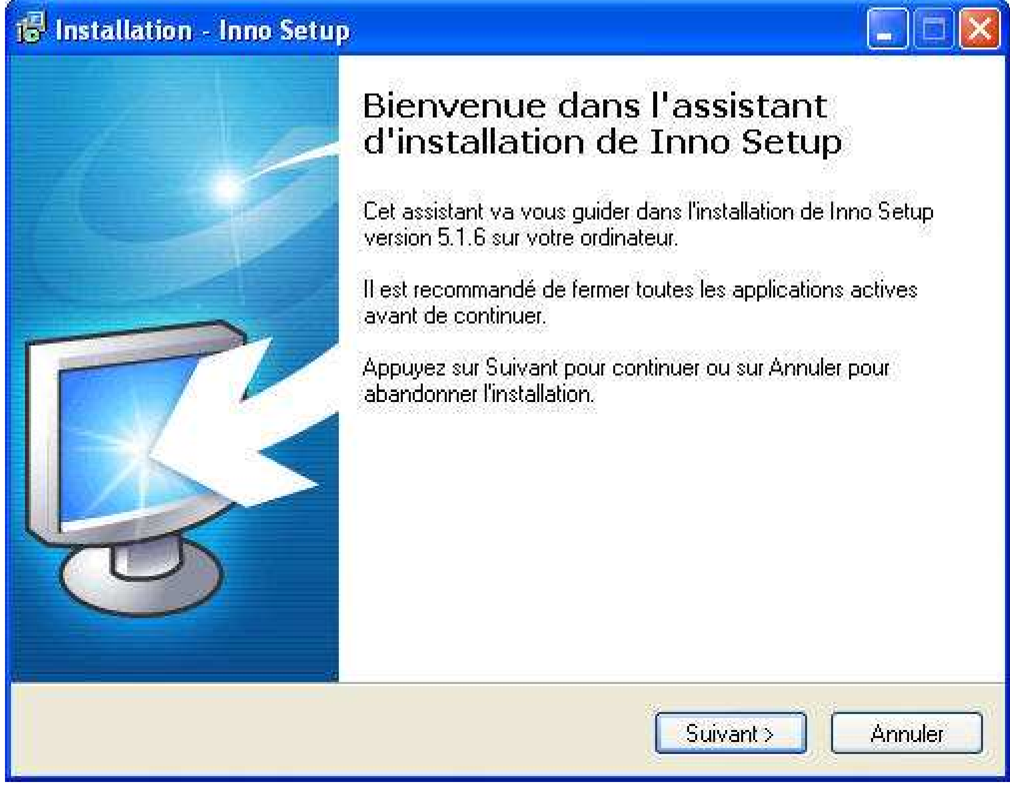
Télécharger Inno Setup

Rendez-vous sur le **site officiel** du logiciel.
Cliquez sur le lien Download et récupérez le programme d'installation.

Petit détail amusant : si vous regardez les commentaires du programme d'installation d'Inno Setup, vous verrez qu'il a été créé avec... Inno Setup

On vous demande en premier lieu votre langue. Normalement, la langue est automatiquement détectée en fonction de la langue utilisée sur votre ordinateur.

Vous voyez ensuite la première fenêtre de l'assistant d'installation :



Sympathique n'est-ce pas ?

Bon je ne vous fais pas une capture d'écran de chacune des étapes de l'installation, je pense que vous êtes assez grands pour savoir cliquer sur Suivant - Suivant - Suivant - Terminer

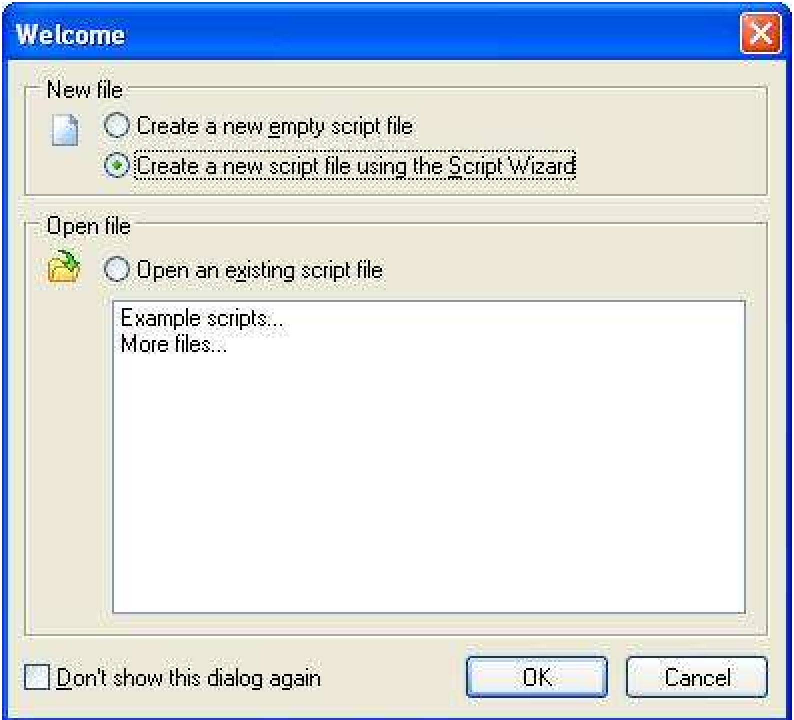
A la fin, on vous demande si vous voulez exécuter Inno Setup. Bonne idée ça, on est justement là pour ça !

Créer une nouvelle installation

Lors du lancement d'Inno Setup, une fenêtre de bienvenue vous demande si vous voulez créer une nouvelle installation ou en ouvrir une déjà existante.

En fait, les installations d'Inno Setup se créent à partir d'un petit langage de script (très facile à utiliser je vous rassure).
Comme l'auteur est sympa, il a pensé aux débutants qui veulent aller vite (comme nous). Il a donc inclus un assistant de création de scripts. **Cet assistant génèrera le script de création de l'installation pour nous**. On ne demandait pas mieux

Cochez donc "*Create a new script file using the Script Wizard*" :

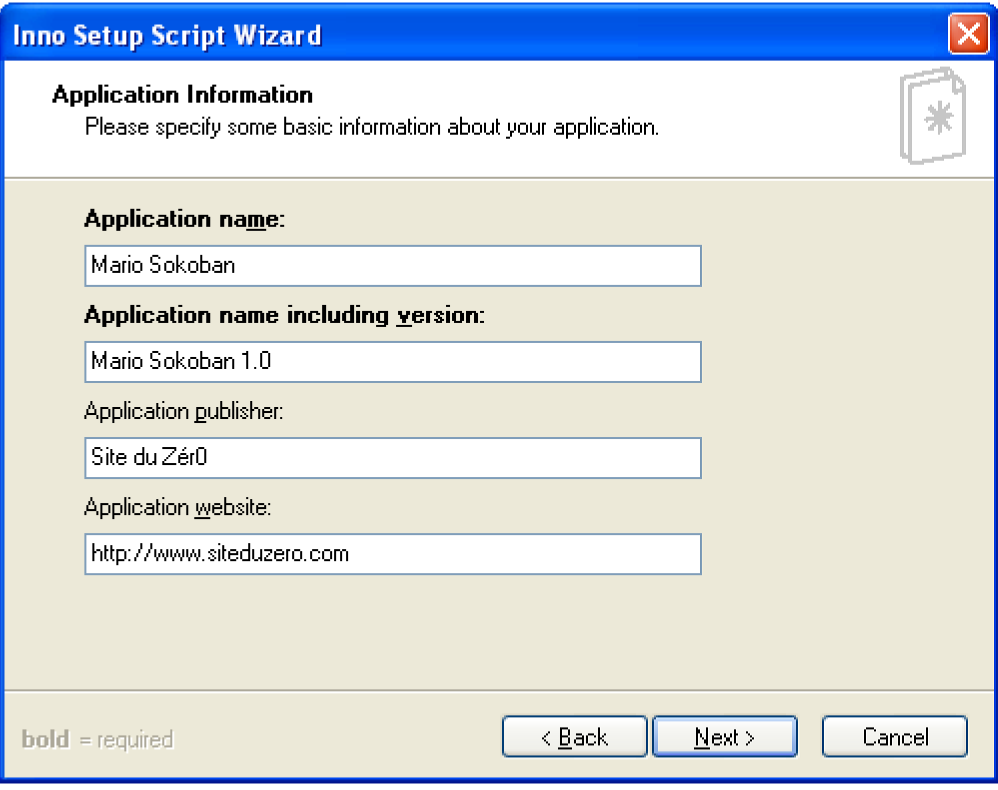


Au fait, je signale au passage que le logiciel Inno Setup est en anglais, mais les installations qu'il génère seront entièrement en français. Don't panic.

Cliquez sur OK.
La première fenêtre d'assistant s'ouvre :

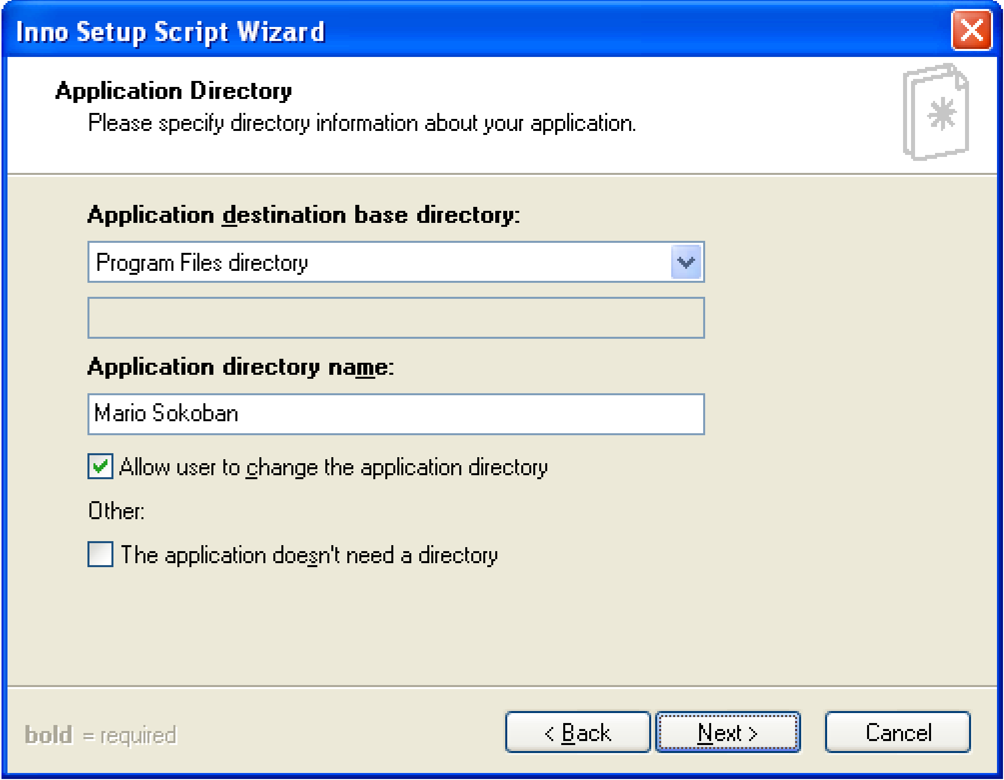
Bla bla bla.
Ne cochez pas la case, cliquez sur Next, c'est tout ce que je vous demande

La fenêtre suivante est déjà plus intéressante :



Vous devez rentrer le nom de votre programme, le nom de votre programme avec le numéro de version, le nom du créateur ainsi que le site web du programme.
Dans mon exemple, je m'apprête à créer une installation pour le jeu **Mario Sokoban**.

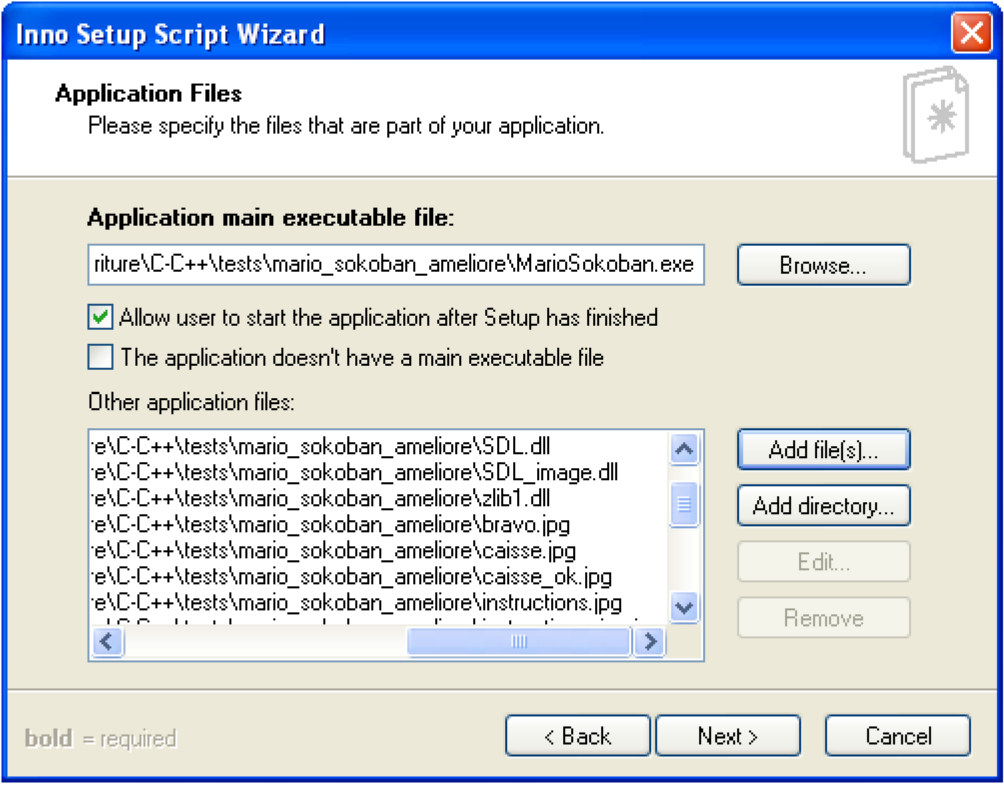
Etape suivante :



On vous demande le dossier d'installation du programme. Vous pouvez choisir entre le mettre dans Program Files ou dans un dossier personnalisé (custom). On va rester classiques, on va mettre le programme dans Program Files

Je vous conseille de laisser cocher la case "Allow user to change the application directory". Cela permettra à l'utilisateur de changer le chemin d'installation s'il le désire.
L'autre case "The application doesn't need a directory" est un peu spéciale. Elle ne sert que pour de rares programmes qui n'ont pas besoin d'un dossier spécial pour être installés. Ca ne nous concerne pas ici.

Ensuite :



Cette fenêtre vous demande les fichiers à empaqueter.
On vous demande tout en haut où se trouve l'exécutable (le .exe du programme). Indiquez donc où se trouve le fichier sur votre disque dur. Dans mon cas, il s'appelle `MarioSokoban.exe`

En-dessous, je vous conseille de laisser cochée la case comme moi : cette case permet de laisser la possibilité à l'utilisateur de démarrer le programme automatiquement à la fin de l'installation.
La case "The application doesn't have a main executable file" ne sera généralement pas cochée. Elle ne sert que pour les programmes ne possédant pas de .exe principal. C'est assez rare, mais ça arrive

Ensuite, et c'est très important là aussi, on vous demande les "Other application files". Vous devez indiquer là-dedans **tous les fichiers dont a besoin votre programme pour fonctionner**. Je vous conseille vivement de n'en oublier aucun, ou votre programme ne marchera pas

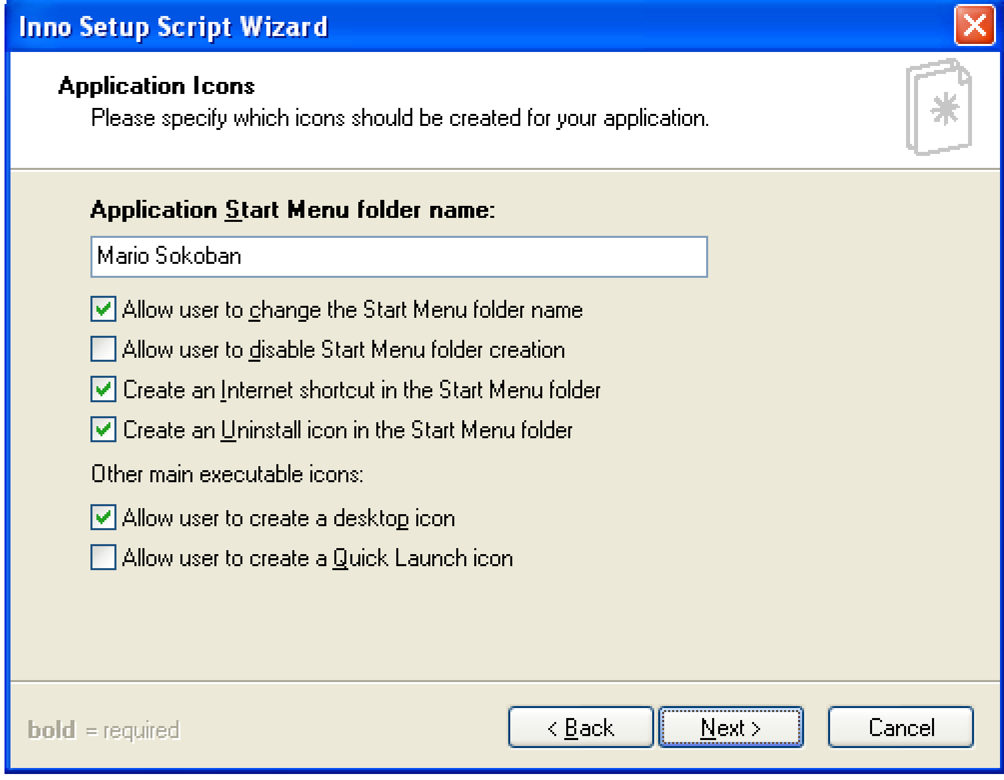
N'indiquez pas à nouveau le .exe. Vous l'avez déjà donné tout à l'heure.
Indiquez en revanche les DLL dont a besoin le programme, les images, les sons etc...

Je ne vous fais pas la liste, mais dans le cas du Mario Sokoban ça fait déjà pas mal de fichiers ! Entre les DLL de la SDL et de SDL_Image, les images du jeu, le fichier niveaux.lvl etc... Ca en fait du monde !

- Si vous voulez ajouter des fichiers qui seront installés dans le même dossier que l'exécutable, cliquez sur **Add Files**
- Si vous voulez ajouter tout un répertoire pour qu'il soit recréé dans le dossier de l'exécutable, cliquez sur **Add Directory**

Dans mon cas, je n'ai pas eu besoin d'ajouter de répertoire, tous les fichiers se trouvent dans le même dossier que l'exécutable.
Dans le cas de très gros programmes, vous aurez sûrement besoin de créer des dossiers (un pour les images, un pour les sons, un pour les niveaux...). Vous cliquerez alors sur Add Directory.

Fenêtre suivante :

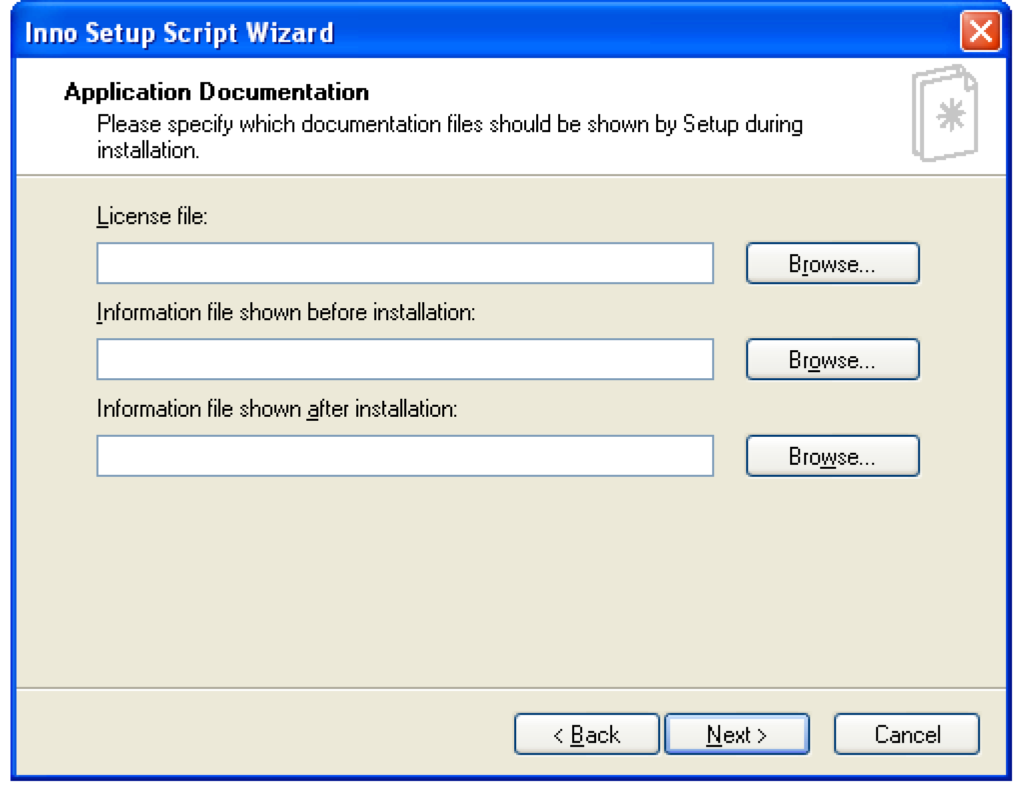


On vous demande quels raccourcis vous voulez créer. En premier lieu, on vous demande le nom du dossier dans le menu démarrer. Personnellement, je laisse la valeur par défaut.

Les cases à cocher sont intéressantes, je traduis pour les non-anglophones :

- **Allow user to change Start Menu folder name** : laisse la possibilité à l'utilisateur de changer le nom du dossier du menu démarrer.
- **Allow user to disable Start Menu folder creation** : laisse la possibilité à l'utilisateur de désactiver la création des raccourcis dans le menu démarrer.
- **Create an Internet Shortcut in the Start Menu folder** : un lien vers votre site web sera ajouté au Menu Démarrer (chic chic)
- **Create an Uninstall icon in the Start Menu folder** : ajoute une icône de désinstallation dans le menu démarrer.
- **Allow user to create a desktop icon** : laisse la possibilité à l'utilisateur de créer un raccourci sur le bureau.
- **Allow user to create a Quick Launch icon** : laisse la possibilité à l'utilisateur de créer un raccourci dans la barre Quick Launch. C'est une barre de raccourcis située juste à droite du menu Démarrer.
  Vous pouvez voir la zone en question sur ma capture d'écran :

Fenêtre suivante (allez c'est presque fini !) :



On vous y demande des fichiers texte à afficher avant et après l'installation (ainsi que la license du programme). Vous pouvez indiquer n'importe quel fichier .txt (ou .rtf si vous voulez faire un peu de mise en forme comme mettre de la couleur, du gras...).

Personnellement, je ne mets rien ici pour mon programme, mais vous aurez sûrement envie d'afficher des informations à vos utilisateurs. Par exemple, vous pourriez indiquer les bugs connus de votre programme, les améliorations apportées par la nouvelle version etc.

La partie "License File" sera utile notamment si vous distribuez votre programme sous license libre (GNU / GPL) comme ça se fait le plus souvent pour les programmes Open Source (c'est-à-dire les programmes dont on peut obtenir le code source).

Ici, on vous demande les langues disponibles dans le programme d'installation. Si vous cochez plusieurs langues, on demandera la langue désirée au début de l'installation.
Dans mon cas, je vais cocher seulement French (na !).

Inno Setup Script Wizard

Compiler Settings

Please specify some basic compiler settings.

Custom compiler output directory:

C:\Documents and Settings\Mateo21\Mes documents\Sites\S

Browse...

Compiler output base file name:

mario_sokoban_setup

Custom Setup icon file:

Browse...

Setup password:

< Back Next > Cancel

Le premier champ permet d'indiquer dans quel dossier devra être créé le programme d'installation. Personnellement, j'ai choisi de le mettre dans le dossier de mon projet pour l'avoir faiclement sous la main.

Ensuite, on vous demande le nom du programme d'installation. Je recommande de changer le "setup" par défaut par quelque chose de plus clair, comme ici : "mario_sokoban_setup".

Le troisième champ permet de choisir un fichier d'icône (.ico) personnalisé pour l'installation. Je vais laisser l'icône par défaut, elle est très bien

Enfin, le 4ème champ permet de protéger l'installation par mot de passe. Seuls ceux qui connaissent le mot de passe pourront installer votre programme.

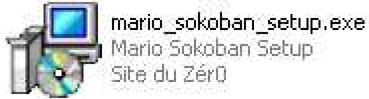
La fenêtre suivante est la dernière : vous n'avez plus qu'à cliquer sur Finish !

Compiler l'installation

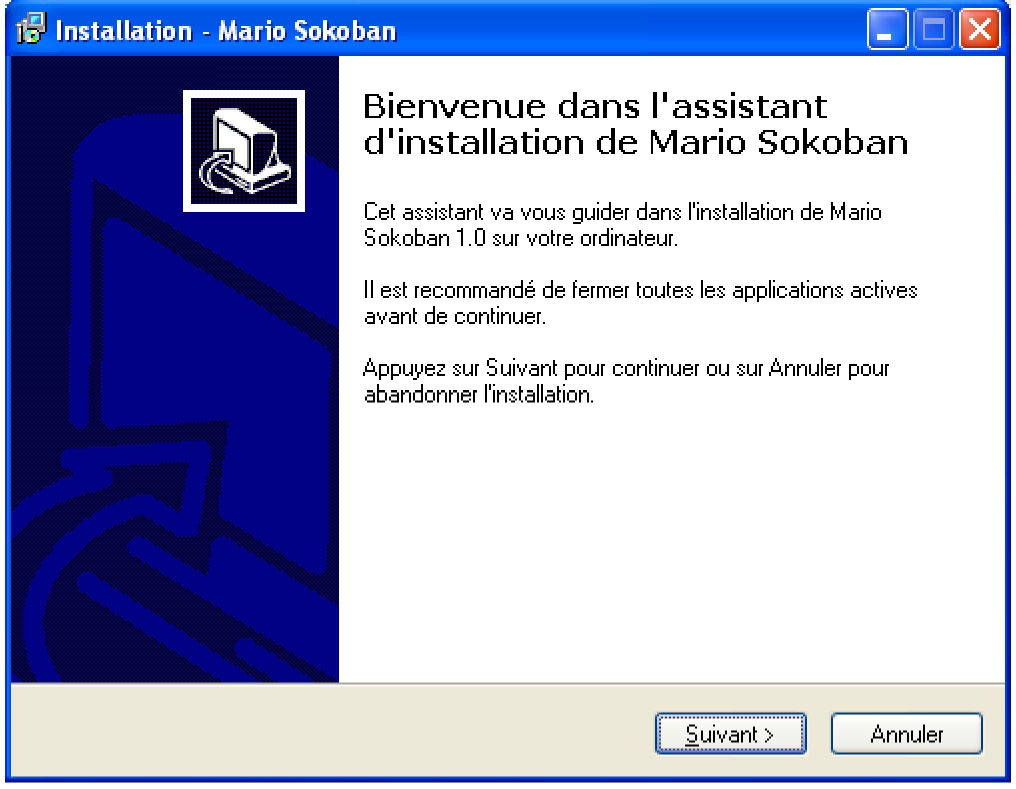
Vous pouvez voir que le script de configuration de l'exécutable a été automatiquement généré par l'assistant en fond.

On vous demande si vous voulez compiler l'installation maintenant. Si vous ne voulez pas personnaliser encore un peu le script à la main, cliquez sur Oui :

Au bout de quelques secondes, le programme d'installation a été généré !



Et voilà un beau programme d'installation tout neuf !



Modifier le script de configuration

Si vous voulez modifier le script de configuration, libre à vous. Vous trouverez de la documentation dans l'aide d'Inno Setup. C'est vraiment simple à utiliser, vous aurez vite fait d'apprendre

En modifiant le script de configuration, vous pourrez faire des choses plus avancées, comme afficher une image personnalisée pendant l'installation du programme, redémarrer l'ordinateur à la fin de l'installation ou encore modifier des clés de la base de registre.
Pour compiler à nouveau l'installation, vous irez dans le menu Build / Compile (Ctrl + F9).

Pour information, j'ai eu besoin de modifier un tout petit peu le script de configuration pour mon jeu Mario Sokoban. En effet, il faut préciser le répertoire de travail (WorkingDir) dans la ligne commandant la création du raccourci dans le menu démarrer :

Code : Autre

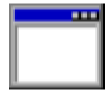
```
[Icons]
Name: "{group}\Mario Sokoban"; Filename: "{app}\MarioSokoban.exe"; WorkingDir: "{app}"
```

J'ai juste rajouté `WorkingDir: "{app}"` pour indiquer que le "répertoire de travail" du programme était celui de l'application {app}. Si je ne l'avais pas fait, le programme n'aurait pas su où aller chercher les images par exemple.
Créer un programme d'installation professionnel est donc un jeu d'enfant avec **Inno Setup** grâce à l'assistant.
Cet assistant est vraiment pratique car il suffit la plupart du temps pour créer une installation rapidement.

Toutefois, il ne vous montre même pas le quart des possibilités d'Inno Setup ! Si vous voulez aller plus loin, il faudra éditer vous-même le fichier de configuration de l'installation.
N'hésitez pas à consulter l'aide, car **l'installation est très personnalisable** et vous pouvez faire de nombreuses choses en éditant le fichier de configuration !

Créer une icône pour son programme

Lorsque vous créez un programme, Windows attribue une icône par défaut à l'exécutable :



La question que vous devez vous poser est :

La réponse se trouve... dans cette annexe
Vous apprendrez à :

- Extraire des icônes d'autres programmes ou de DLL (si vous voulez les récupérer)
- Dessiner et enregistrer vos propres icônes
- Associer une icône à votre programme lors de la compilation

LES LOGICIELS D'ÉDITION D'ICÔNES

Les logiciels d'édition d'icônes sont nombreux et, allez savoir pourquoi, ils sont pratiquement tous payants.

La bonne nouvelle, c'est que je viens de dire pratiquement
J'ai quand même réussi à dénicher pour vous 2 très bons logiciels. Ces logiciels ne sont pas concurrents, ils ne servent pas exactement à faire la même chose. Ils sont au contraire plutôt complémentaires :

- **SnIco Edit** : c'est un éditeur d'icônes, grâce auquel vous pouvez dessiner vos propres icônes. Une sorte de Paint (amélioré !) pour icônes en somme
- **Icon Sushi** : c'est un extracteur d'icônes. Vous pouvez récupérer les icônes situées au sein des programmes .exe et des DLL. Il est vraiment très complet à ce niveau et vous permet d'exporter les icônes au format .ico, .png, .bmp etc... Bref, un outil indispensable

Précision importante avant de commencer : un fichier d'icône (.ico) peut contenir plusieurs versions de la même icône. En effet, une icône peut être enregistrée sous différentes tailles : 16x16, 32x32, 48x48 etc.
En plus de ça, on peut l'enregistrer avec un nombre différent de couleurs : 2 couleurs, 16 couleurs, 256 couleurs, millions de couleurs etc.

Depuis Windows XP, les icônes peuvent être enregistrées en milliards de couleurs (32 bits), être antialiasées et elles supportent la transparence sur plusieurs niveaux (aussi appelée "transparence alpha").

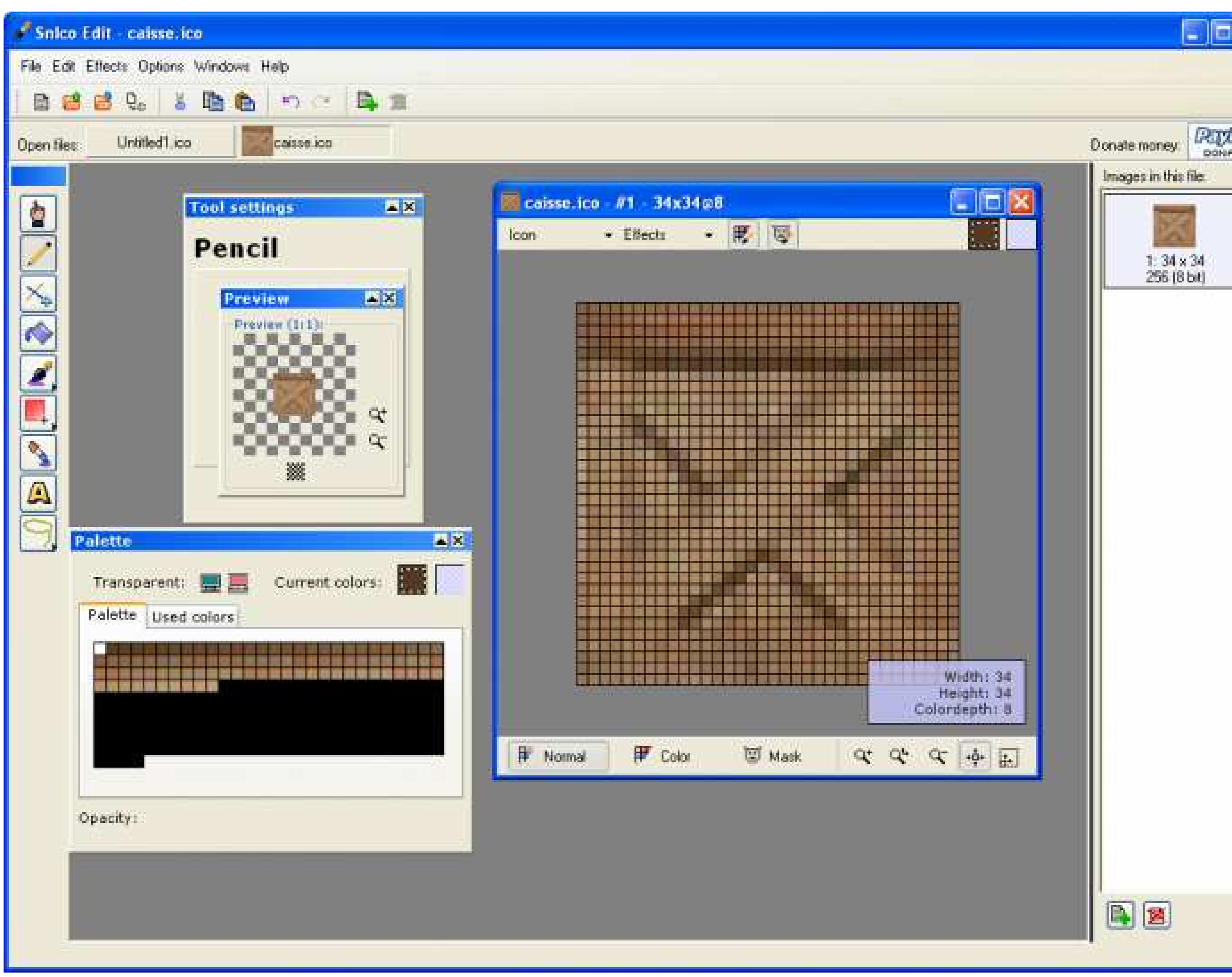
Bref, dans un seul fichier .ico, on peut donc trouver une dizaine de versions différentes de la même icône !
Lorsque vous créez une icône, il est recommandé de créer plusieurs versions (bien que ça ne soit pas obligatoire). Votre icône aura ainsi une meilleure apparence selon la taille dans laquelle elle est affichée et selon le nombre de couleurs qu'affiche le moniteur de l'utilisateur.

SnIco Edit : le Paint des icônes

Il y a une rumeur qui court sur le Net comme quoi il suffirait de renommer un .bmp en .ico pour le transformer en icône. C'est tout à fait faux.
Une icône est codée différemment d'un BMP, on ne peut pas se contenter de renommer le fichier.

Manque de bol, Paint ne permet pas d'enregistrer des icônes. De nombreux éditeurs de logiciels en ont tiré parti et il existe du coup une pléthore d'éditeurs d'icônes, tous payants.
Tous ? Non, car l'un d'entre eux résiste à l'envahisseur (comprenez : il est gratuit), il s'agit de **SnIco Edit**.
Et comme une bonne nouvelle ne vient jamais seule, sachez que ce programme est disponible **en français**

Voici à quoi ressemble le logiciel :



Télécharger SnIco Edit (1,4 Mo)

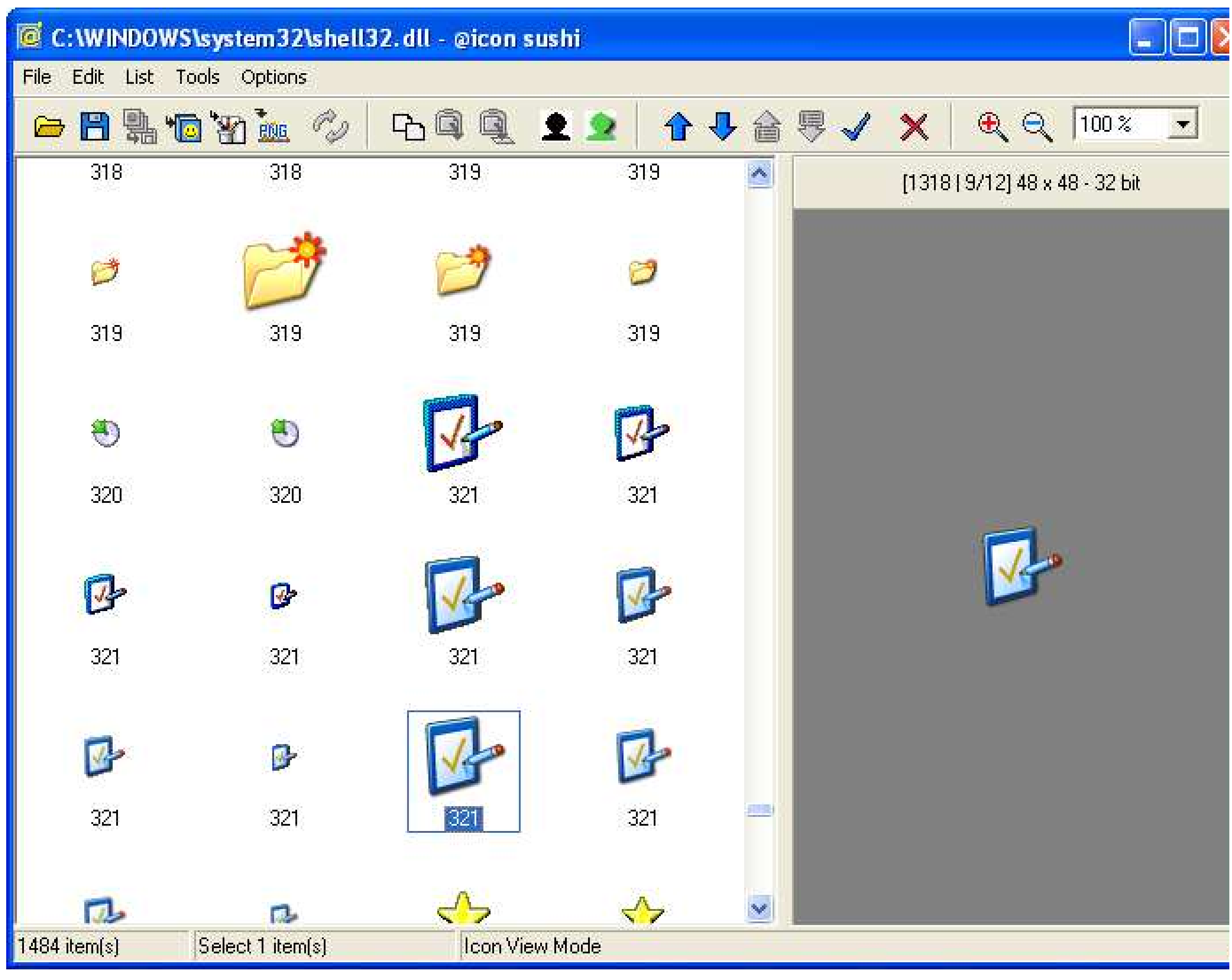
L'installation est en anglais et le programme démarrera d'abord en anglais.
Vous pouvez changer la langue dans le menu Options / Languages / Français. Il vous faudra ensuite redémarrer le programme pour que la langue française soit activée.

Vous êtes des grands, donc vous n'avez pas besoin d'un tuto pour que je vous explique comment vous servir du logiciel (en plus il est en français 😛). C'est une sorte de Paint amélioré qui peut enregistrer des icônes c'est tout 😊

Icon Sushi : l'extracteur d'icônes

Ce programme m'est pratiquement indispensable. Il est capable d'importer et d'exporter des icônes sous de nombreux formats différents.
Il n'est pas vraiment fait pour dessiner des icônes, mais en revanche vous pouvez grâce à lui voir les icônes contenues dans les .exe et les .dll.

Voici un aperçu de ce programme :



Télécharger Icon Sushi (830 Ko)

Le programme est en anglais, mais franchement ça ne devrait vous poser aucun problème
Comme vous le voyez sur cette capture d'écran, j'ai ouvert le fichier shell32.dll.

La DLL *shell32.dll* située dans Windows\System32\shell32.dll contient un très grand nombre d'icônes par défaut de Windows. N'hésitez pas à aller voir tout ce qu'elle contient !

Comme vous pouvez le constater, la DLL contient plusieurs icônes différentes, et chaque icône se trouve dans plusieurs résolutions différentes !

Si vous voulez afficher les icônes de la même manière que moi, je vous recommande d'aller dans le menu List / Icon View (Ctrl + 2).

Vous avez plein de boutons dans la barre d'outils pour exporter l'image au format BMP, ICO, PNG etc...
Si vous voulez utiliser une de ces icônes pour votre programme, sélectionnez celle qui vous intéresse et enregistrez-la en .ico.

Associer une icône à son programme

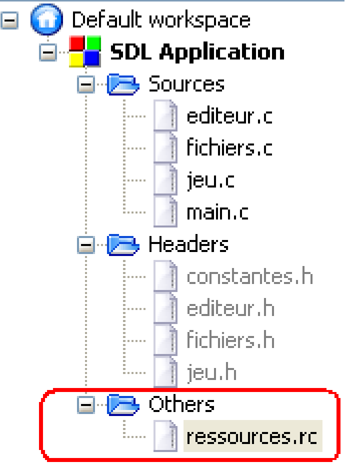
A ce stade, je considère que vous avez votre fichier .ico.
Vous l'avez soit créée (avec SnIco Edit) ou extraite d'un autre programme (avec Icon Sushi).

Maintenant, vous vous demandez probablement comment on fait pour changer l'icône de notre programme.
Il va falloir utiliser **un fichier de ressources**.

Les fichiers de ressources sont propres à Windows (vous n'en trouverez pas sous Linux et Mac OS par exemple).
Leur extension est **.rc**.

Dans votre IDE, demandez à ajouter un fichier à votre projet. Au lieu de donner l'extension .c, .cpp ou .h à ce fichier, donnez-lui l'extension .rc. Par exemple : `ressources.rc`.
Ce fichier doit être ajouté à la liste des fichiers à compiler dans votre IDE.

Sous Code::Blocks par exemple, il apparaîtra comme ceci :



Il se situe juste dans une section "Others" ("Autres").

Que doit-on mettre dans ce fichier .rc ?

Vous devez indiquer les fichiers qui seront enregistrés dans l'exécutable.
En effet, c'est le principe des fichiers de ressources : **ils servent à demander au compilateur d'enregistrer des fichiers dans un .exe**. Vous pouvez y mettre des bitmaps, des icônes, des curseurs de souris etc...

Les fichiers indiqués seront inclus dans l'exécutable.
On peut les extraire lors de l'exécution en faisant appel à certaines fonctions de l'API Windows que je n'ai pas utilisées et qui seraient de toute manière hors de propos ici.

Ce qu'il faut savoir, **c'est que la première icône que vous indiquez dans un fichier de ressources deviendra l'icône de votre programme** (elle apparaîtra dans l'explorateur Windows).

Mettez le code suivant dans votre fichier ressources.rc :

Code : Autre

```
1 ICON "caisse.ico"
```

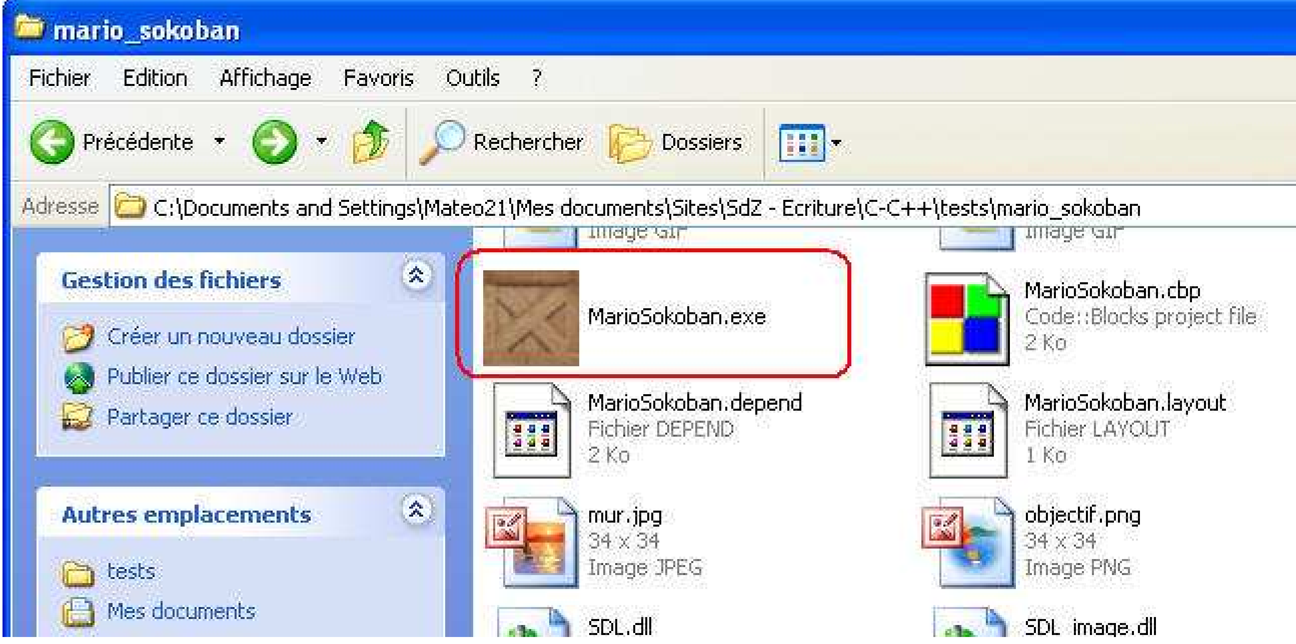
Ce code est composé de 3 parties :

- Un numéro d'identification : chaque ressource doit avoir un numéro d'identification unique (ça peut aussi être un texte d'identification).

- Le type de ressource (ici c'est ICON pour une icône)
- Le nom du fichier à inclure. Ici, mon icône s'appelle caisse.ico. Elle doit se trouver dans le même répertoire que l'exécutable au moment de la compilation.

Voilà, c'est tout ce que vous avez besoin de faire !
Vous pouvez ensuite compiler, et vous verrez que votre exécutable aura pris la forme de votre icône dans l'explorateur Windows !

Voici une preuve (on sait jamais, y'en a peut-être qui me croient pas) :



Vous n'avez pas besoin de livrer le fichier .ico avec votre programme. En effet, le fichier a été inclus dans le .exe au moment de la compilation (c'est le principe même des ressources).

L'IDE Dev-C++ permet d'associer une icône à son programme de manière un peu plus facile (enfin, c'est quand même pas bien difficile les fichiers de ressources).
Rendez-vous dans le menu Projet / Options du projet. Vous verrez dans la fenêtre une section "Icône" et un bouton "Parcourir" qui vous demandera quelle icône associer à l'exécutable.
Dev-C++ ne fait rien de magique : il ne fait que créer un fichier de ressources, exactement comme on l'a fait manuellement nous-mêmes quelques instants plus tôt.

L'ajout d'une icône à un programme est vraiment simple, du moins une fois qu'on a appris la technique

J'insiste sur le fait que tout cela ne fonctionne que sous Windows.
Sous les autres OS, c'est différent. Par exemple, sous Linux il n'y a pas d'icône associée aux exécutables. Ca peut se faire si vous êtes sous KDE, mais du coup ça ne concerne pas vraiment tout le monde. Bref, ne vous prenez pas trop la tête pour associer une icône à votre programme sous les autres OS.

Si vous voulez télécharger des icônes sur Internet, les sites ne manquent pas. Mais attention : certains prétendent offrir des icônes en téléchargement alors que bien souvent c'est payant.
Vous pouvez faire une recherche **Google "Free Icons"**, mais gardez bien en tête que tous les sites que vous visiterez n'offrent pas forcément leurs icônes gratuitement. Si vous êtes obligé de dégainer la carte bancaire, c'est généralement mauvais signe

---

Le cours n'est pas terminé.

D'autres chapitres arriveront, dès que j'aurai le temps de les écrire

Revenez régulièrement vérifier s'il n'y a pas du nouveau (et ne m'envoyez pas de MP pour me demander quand ça arrive svp, je le fais quand j'ai du temps de libre !)
Pour ceux qui souhaitent savoir ce qui les attend dans la suite des cours, je vous invite à lire **ce post-it** que j'ai posté sur le forum de programmation. J'y dévoile mes plans pour la suite du cours et tente de répondre à la plupart des questions que vous vous posez à ce sujet