CPR Informatique

(poste 3159 ou 3164)

ECOLE NATIONALE
DES SCIENCES
GEOGRAPHIQUES

24 septembre 2000

Programmer en C++

avec le ...

Table des matières

1. Présentation du langage C++

Le C++ est une extrapolation du langage C, lui-même inventé pour écrire le système d'exploitation UNIX et très répandu dans l'industrie. Le C++ est un langage compilé relativement structuré : le C étant très proche du système d'exploitation, on peut accéder directement à la mémoire et c'est même parfois nécessaire.

Le C++ est considéré comme un langage orienté objet car il répond aux trois principes fondamentaux : **encapsulation**, **polymorphisme** et **héritage**. En plus, des types simples et complexes du C, on peut définir des classes qui sont des structures évoluées dont certains **champs** (appelés **membres**) sont des fonctions (appelées **méthodes**) selon le principe d'encapsulation. Le programmeur doit gérer l'accessibilité aux champs des classes qu'il définit : certains membres peuvent être rendus accessibles directement comme un champ d'une structure. En C++, deux fonctions peuvent avoir le même nom (principe de polymorphisme) si leurs paramètres ont des types différents. Elles peuvent également avoir des paramètres facultatifs dont la valeur par défaut est mentionnée dans le prototype (appelé signature). Les classes peuvent être définies à partir d'autres classes selon le principe d'héritage multiple. De même, il est possible de définir des fonctions ou des classes génériques dont le comportement dépendra, lors de l'édition de lien dynamique, d'un type fourni par l'utilisateur : la fonction min(x,y) est générique car elle compare deux valeurs x et y de même type et renvoie la plus petite des 2 valeurs.

Pour montrer comment programmer en C++, ce support de cours utilisera comme fil conducteur la manipulation de tableaux (ensembles de cases contiguës de même type). Le premier programme sera écrit de la manière la plus simple possible jusqu'à l'écriture d'un véritable gestionnaire de tableaux avec sauvegarde sur fichiers. Chaque programme apportera son lot de précisions ou de nouveautés…

2. Historique du C / C++

Le langage C est un langage de type déclaratif comme le Pascal. Il a été créé dans les années 70 par une seule personne Dennis Ritchie (AT & T). Son but était de réécrire le système d'exploitation UNIX en langage évolué car Ken Thompson avait écrit UNIX en B : langage non évolué qui posait des problèmes de portage sur d'autre machine que PDP-11.
Le C n'a pas été créé pour concurrencer Fortran ou Pascal, mais le succès d'UNIX a contribué à sa popularité. La réussite incontestable du C comme langage de programmation système lui a forgé une fausse réputation de langage portable.

Le C diffère du Pascal ou de l'ADA en ce sens qu'il est plus proche de la machine (plus bas niveau) :
- il demande plus au programmeur ;
- il permet de mieux comprendre certains mécanismes systèmes.

En 1978, Brian Kernnighan et D. Ricthie publient la "bible du C", le fameux "K&R" qui fait référence ("norme", 2ème édition en 1987). Parallèlement en 1982, l'American National Standards Institute lance le comité de normalisation du C, ce qui a donné la Norme Ansi en 1989 et les meilleurs compilateurs. L'évolution naturelle est le C++. Les programmes écrits en ANSI-C sont censés pouvoir être recompilés n'importe où …

Le langage C++ a été conçu en 1983 par Rick Mascitti pour être utilisé dans le même environnement que le C. Une version du C avec "classes" existe depuis 1980. Ce langage est considéré comme une extension du langage C, il s'appelle C++ par référence à l'opérateur d'incrémentation du C.

3. Ouvrages de référence

On peut citer comme ouvrages de référence :

- L'incontournable : Le Langage C (2ème édition) de "K&R"
- Pour l'apprentissage : Langage C - Norme ANSI de Ph.Drix
- Pour les problèmes de portage : Le Manuel de Référence de Harbison-Steele.

4. Le langage C : les bases du C++

4.1. Les sources et les commentaires

Un programme écrit en C est un ensemble de fichiers textes documentés (appelés **sources**) respectant une grammaire précise, bien qu'il existe toujours plusieurs façons d'écrire la même chose. Ce document présente la solution préconisée mais aussi les alternatives les plus fiables sous forme de **commentaires** (c'est-à-dire une suite de caractères, de préférence non accentués, qui seront ignorés par le compilateur et ne servent qu'à documenter le programme). On indique le début des commentaires et la fin, éventuellement sur des lignes différentes :

/ ce symbole indique le debut d'un commentaire qui peut etre sur plusieurs lignes et ne se terminera qu'apres le symbole */*

Un source est organisé de la manière suivante :

```
/* Commentaires sur le contenu du fichier                              */

/* Indications des fichiers qui devront etre inclus par le compilateur */

/* Definitions des constantes et des types complexes utilises par la suite */

/* Prototypes des fonctions qui seront decrites plus loin              */

/* Corps des fonctions                                                 */

/* Fonction principale (appelee "main") si ce fichier est celui qui
   contient le "programme principal"                                   */
```

L'affichage et la saisie sont gérés par la bibliothèque **stdio.h** Les fonctions **printf** et **scanf** sont utilisées respectivement pour l'affichage à l'écran et la saisie au clavier.
Dans ce chapitre, nous ne présenterons que la fonction **printf**, car l'utilisation de la fonction **scanf** requiert des notions que nous verrons plus tard (cf. chapitre 4.9 – Les pointeurs)

La fonction **printf** permet de formater l'affichage à l 'écran. Il s'agit d'une fonction à arguments variables (on peut avoir autant de paramètres que l'on veut). Dans le format d'affichage, chaque "%" annonce l'utilisation d'un paramètre fourni par la suite.

Exemples :

`printf ("Bonjour") ;` affiche "Bonjour" à l'écran.
`printf ("Bonjour\n") ;` affiche "Bonjour" à l'écran et effectue un retour à la ligne.
`printf ("la somme de 2 et 3 fait %d\n",n) ;` affiche à l'écran : `La somme de 2 et 3 fait 5,` si la variable n vaut 5.
`printf ("la somme de %d et %d fait %d\n",a,b,n) ;` affiche à l'écran : `La somme de 2 et 3 fait 5,` si les variables a, b et n valent respectivement 2, 3 et 5.

%d représente ici le format d'affichage d'une variable de type entière. Il existe un format d'affichage pour chaque type simple de données (cf. chapitre suivant 4.2).

L'exemple qui suit montre l'utilisation de la fonction **printf** dans un programme simple calculant la somme de deux entiers.

```
/* Ce fichier presente un exemple d'affichage d'une somme de
   2 entiers a l'ecran */


#include <stdio.h>


int main () {

  /* Declaration des variables */
  int a,b,c;

  a=2 ;
  b=3 ;
  c=0 ;

  printf ("a = %d\n",a);
  printf ("b = %d\n",b);

  c = a + b ;

  printf ("La somme %d + %d = %d\n",a,b,c);


  return 0 ;

} /* main */
```

Pour permettre la saisie et l'affichage, il est nécessaire d'inclure le fichier **stdio.h.** Le symbole # indique une directive (un ordre) de précompilation qui est donc effectuée en priorité. Le fichier à inclure étant fourni avec le compilateur, son nom est indiqué entre chevrons car il n'est pas nécessairement présent sur le répertoire courant.

on affecte des valeurs aux variables *a* et *b*

à l'écran

```
a = 2
b = 3
```

à l'écran

```
La somme 2 + 3 = 5
```

on renvoie la valeur 0 au système, ce qui signifie que le programme s'est correctement terminé

4.2. Les types simples de données

4.2.1 - La déclaration des variables

déclaration : <type> <identificateur> ;

<identificateur> :
- lettres non accentuées **(attention minuscules ≠ majuscules)**
- chiffres (sauf le premier caractère)
- '_'

- attention à ne pas utiliser les mots-clefs du langage (**class**, **for**, **while**, **if**, **else**, ...)

<type> : entiers, décimaux.

Attention : les intervalles de valeurs et la précision dépendent de la machine utilisée.

4.2.2 - Les entiers

On distingue quatre types d'entiers : **char, short (ou short int), int et long (ou long int).**

	char	**short**	**int**	**long**
Taille (octet)	1	2	2 ou 4	4
Etendue	0 .. 255	-32768 .. 32767	32768 .. 32767 -2^{31} .. $2^{31}-1$	-2^{31} .. $2^{31}-1$
Format d'affichage	%c	%hd	%d	%ld

Remarques :
- selon la machine les caractères sont signés ou pas ;
- on peut préciser entiers signés (**signed**) ou non signés (**unsigned**) ;
- un entier sur un octet peut être codé en **char** ;
- les **char** sont des entiers donc les calculs sont possibles.

4.2.3 - Les décimaux

On distingue deux type de décimaux : **float** (6 chiffres de précision) **et double** (8 chiffres de précision)**.**

	float	**double**
Taille (octet)	4 ou 8	16
Etendue	-2^{31} .. $2^{31}-1$	-2^{63} .. $2^{63}-1$
Format d'affichage	%f	%lf

Remarques :
- **long float** : simple ou double précision selon le compilateur ;
- L'égalité de deux nombres flottants est une ineptie.

4.2.4 – Les opérations arithmétiques

On distingue cinq opérations arithmétiques fondamentales : **+, -, /, *, %**

Exemples :

```
int i ; // declaration
...
i = i + 1 ;   //  incrementation equivalente a :   i++ ;
i = i - 1 ;   //  decrementation equivalente a :   i-- ;
```

les pièges des raccourcis : i++ et ++i

```
int i, n ;    //  declarations
...
n = i++ ;    //  n prend la valeur i puis i est incremente
n = ++i ;    //  i est incremente puis n prend la valeur de i
```

les raccourcis peu explicites : +=, -=, /=, *=, %=

```
int i, n ;    //   declarations
...
n += i ;    //  incrementation egale a   :   n = n + i ;
```

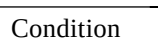
Les **opérations logiques** sur les bits (>>, <<, **&**, | , …) sont très peu utilisés en C. Ils permettent de réaliser des opérations directement sur les bits des nombres entiers. Ce document ne traite pas de ces opérations. Il s'agit d'un sujet « pointu » très bien expliqué dans la plupart des manuels de langage C. (Cf. K&R pages 48 et 49 ou le Drix pages 49 et 50).

4.3. Les instructions : de l'algorithme au programme

L'ADL ("Algorithm Descriptive Language") est un langage de description d'algorithmes très concis (pas de déclarations) proche de la sténographie. Pourquoi faire le parallèle avec l'ADL ?

- ce langage est utilisé à l'IGN et surtout à l'ENSG
- ce langage est concis et facile à comprendre
- cette méthode de structuration des algorithmes est facile à transposer en programmation
- les algorithmes des exercices seront présentés en ADL (universalité des cours)

Signification	**ADL**	**C/C++**
Affectation	X←3	`x = 3 ;`
Test	Condition ? | ¿	`if (condition)` `{` `  instructions ;` `}` `else` `{` `  instructions ;` `} /* if */`
Test à choix multiple	cas cas1 ⇨ ..... cas2 ⇨ ..... = ...	`switch (cas)` `{` `  case cas1 : instructions ;` `  break ;` `  case cas2 : instructions ;` `  break ;` `  default : instructions ;` `} /* switch */`
Boucle "tant que"	/ condition ?	`while (condition)` `{` `  instructions ;` `} /* while */`
Boucle avec compteur	Bf , Pas i : Bi ?i	`for (i=Bi;i<=Bf;i=i+Pas)` `{` `  instructions ;` `} /* for i */`
Boucle infinie	?	`for (; ;)` `{` `  instructions ;` `} /* for */`
Boucle "jusqu'à ce que"	/ condition ?	`do` `{` `  instructions ;` `} while ! ( condition) ;`

Sortie de boucle		**break** ;
Au suivant		**continue** ;
Sortie de procédure		**return** ;
Autres débranchements		**goto** debranchement; instructions debranchement: instructions
vrai		Toute valeur entière non nulle
faux		Toute valeur entière nulle
non	 Condition	! **(**condition**)**
et	∩	**&&**
ou	∪	||
égal	=	**==**
différent	≠	**!=**

Les principales remarques :

- l'affectation "=" ne doit pas être confondue avec le test d'égalité "==" ;
- dans les tests simples, le bloc "**else**" est facultatif et les limites d'un bloc sont facultatives s'il ne contient qu'une instruction (piège pour la maintenance) ;
- dans les tests à choix multiples, le test doit porter sur une valeur d'un type discret (entier ou booléen), le dernier "**break**" est facultatif (pas recommandé) et l'absence de séparation ("**break**") entre 2 choix ("**case**") entraîne l'exécution des instructions du 2ème choix (pas de débranchement implicite) ;
- la boucle "**while**" est la boucle de base du langage (la boucle "**for**" en dérive) ;
- la sortie de programme est "**return**" (sortie de la fonction principale), "**exit**" est un arrêt *violent* ;
- pas de débranchement de niveau supérieur à 1 (utiliser "**goto**" avec parcimonie) ;
- pas de type booléen, les valeurs entières nulles sont fausses, réciproquement les valeurs entières non nulles sont vraies ;
- les tests logiques "**&&**" et "||" ne doivent pas être confondus avec les opérations logiques sur les bits ("**&**" et "|") ;
- dans les tests, en cas d'égalité de priorité, le membre le plus à gauche est évalué en premier (les parenthèses sont fortement conseillées), les autres membres peuvent ne pas être évalués (compilation ANSI ou non).

4.4. Les conversions et les transtypages

Les compilateurs C effectuent des **transtypages** (conversions de type) implicites qui sont dangereux, par défaut les valeurs sont de type "**int**" ou "**float**". Il est recommandé d'expliciter les transtypages ("cast"). Sans transtypage explicite, une opération sur un entier et un flottant peut être effectuée en entier ou en flottant selon le compilateur, le résultat n'est donc pas garanti, par exemple (**int**)(2 ***** 2.5) peut donner 4 ou 5.

Exemples de transtypages explicites :

```
{
int i;
long l;
float f;
double d;

i = 1;          /* les valeurs entieres sont par defaut des "int"       */
l = (long)1;    /* transtypage explicite en entier long                 */
l = 1L;         /* tres utilise pour les constantes mais risque d'oubli */
f = 1.0 ;       /* les valeurs decimales sont par defaut des "float"    */
f = 1. ;        /* possible de ne mettre que le '.' mais risque d'oubli */
d = (double)1;/* transtypage explicite en double precision              */
d = 1.L ;       /* possible, particulièrement dangereux et peu fiable   */

d = (double)2.0e+12;
i = (int)d;     /* partie entiere de d modulo la borne superieure(int)  */
                /* en 32 bits, i prend une valeur inferieure a 2.4e+9   */
                /* si le type "int" etant signe, la valeur est peut     */
                /* etre negative                                        */
}
```

4.5. Tableaux et chaînes de caractères

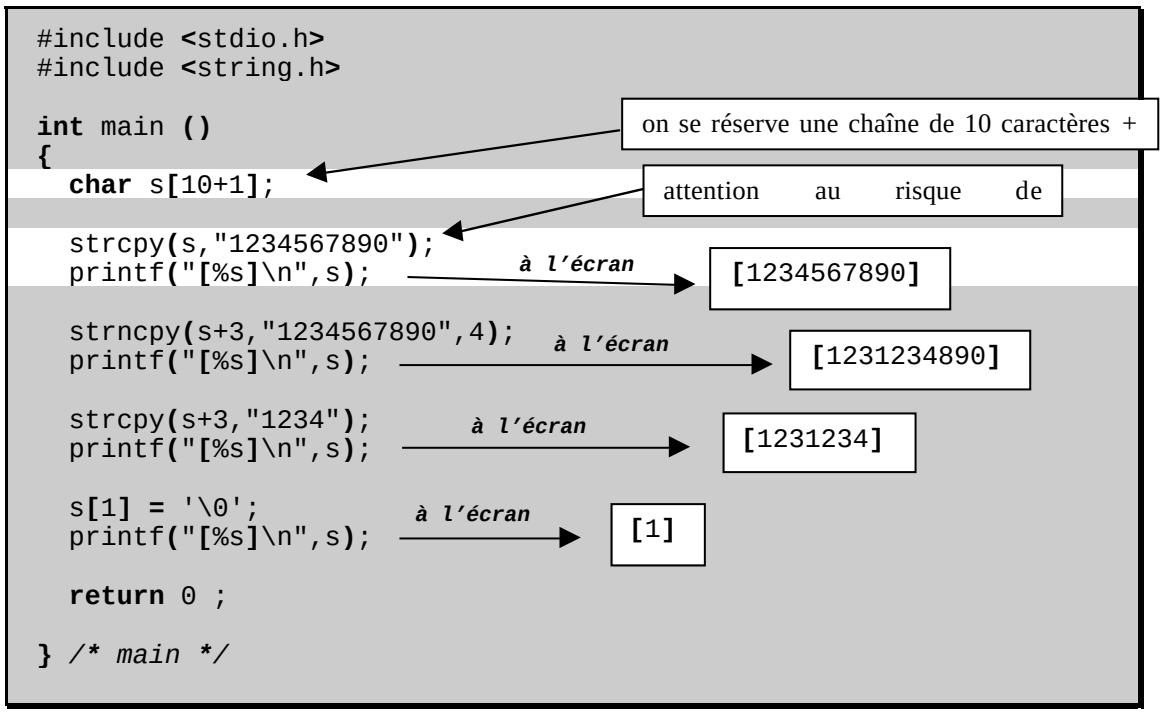
Un tableau est une suite continue (en mémoire) de variables d'un même type. En pratique, le C réserve en mémoire la place nécessaire au tableau mais ne conserve pas la taille de ce tableau. Il y a de ce fait un risque important de dépassement des bornes du tableau. Par exemple, on déclare un tableau de 10 entiers de la manière suivante : **`int TabEntiers [10] ;`** mais rien n'empêche d'écrire : **`TabEntiers[12] = 4 ;`** C'est au programmeur d'être vigilant !

> **Attention** : en C, les indices des tableaux commencent à 0.

Le cas des chaînes de caractères

Les chaînes de caractères sont des tableaux de caractères ("**char**"). Tous les caractères présents dans le tableau ne sont pas nécessairement significatifs. Pour gérer la longueur variable des chaînes, celles-ci sont postfixées : les chaînes sont terminées par le caractère **'\0'** (de code ASCII : 0) qui indique la fin des caractères utiles (autrement dit à afficher grâce au format "%s" dans les fonctions **printf**).Une chaîne ***s*** devant contenir au plus 10 caractères sera donc déclarée : "**char** s**[**10+1**]** ;". Attention, les chaînes constantes sont indiquées par des guillemets qui contiennent implicitement le caractère de fin '\0'.

Une bibliothèque de fonctions de manipulation de chaînes est fournie en standard, il s'agit de la bibliothèque **string.h**. Exemple de manipulations de chaînes de caractères.



A noter que la fonction **`strcpy`** affecte dans le premier paramètre, la valeur du second paramètre.
La fonction **`strcnpy`** affecte dans le premier paramètre, **n** caractères de la valeur du second paramètre, **n** étant le troisième paramètre de la fonction.

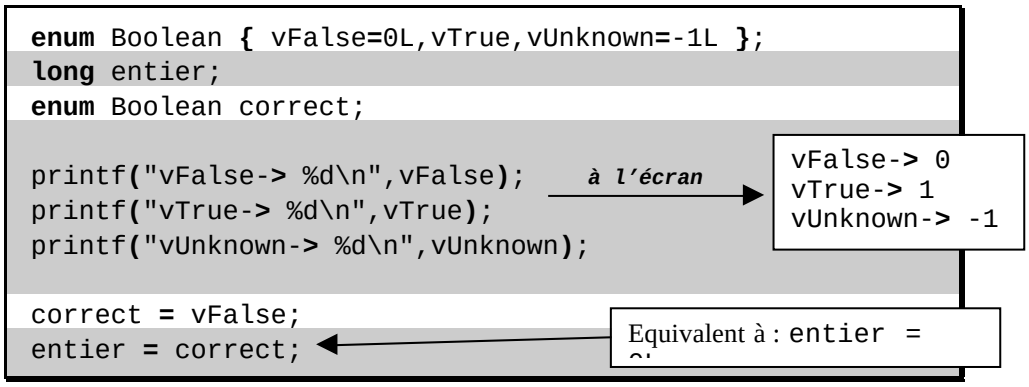
4.6. Les types "complexes" et prédéfinis

Il existe 3 types complexes : **énumérés**, **structures** et **unions**.

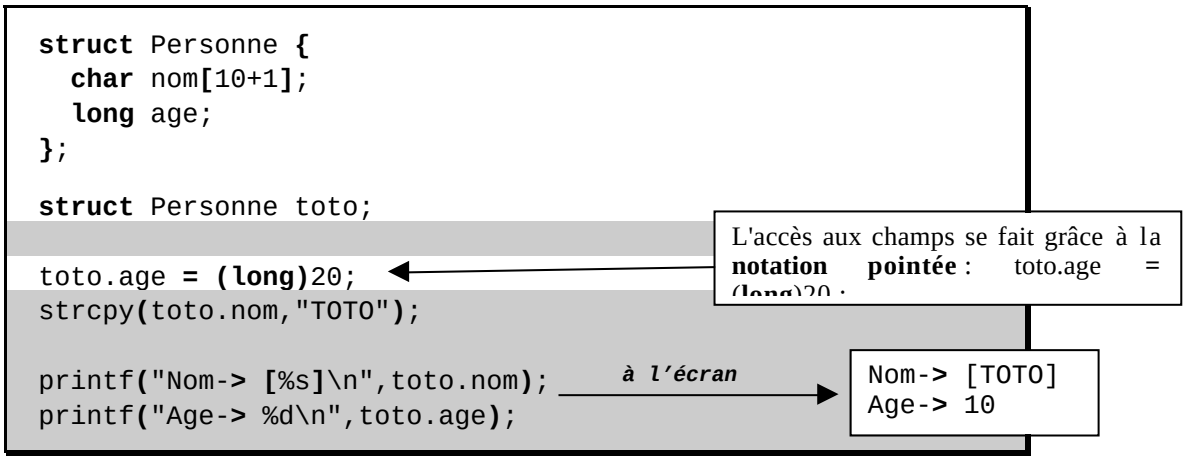
Le type énuméré est une liste de constantes. Exemple :

```
enum Jour { lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche };
enum Jour today;
today = mardi;
```

Les opérations sont impossibles sur les énumérés. On peut cependant affecter une valeur aux valeurs des énumérés, pour pouvoir utiliser ces valeurs comme des constantes numériques.



Le type "structure" est un regroupement de variables de types (qui peuvent être différents) appelées "membres de la structure". L'accès aux champs se fait grâce à la **notation pointée** : *toto.age* (où age est un champ de la structure toto).



Le type "**union**" est un regroupement de plusieurs types sur une même zone mémoire.

```
union Personne {
  char nom[10+1];
  long age;
};
union Personne toto;
```

```
toto.age = (long)20;
strcpy(toto.nom,"TOTO");

printf("Nom-> [%s]\n",toto.nom);
printf("Age-> %d\n",toto.age);
```

à l'écran

```
Nom-> [TOTO]
Age-> 1414485071
```

En fait, `1414485071` n'a aucune signification, il s'agit de la forme numérique de la chaine TOTO.

4.7. La redéfinition d'un type

Pour pouvoir facilement gérer des tableaux d'entiers ou des tableaux de nombres flottants, il est préférable de définir un type d'éléments. Cette opération se fait grâce au mot-clef "**typedef**". L'instruction "**typedef long** tELT;" fait de tELT un nouveau type synonyme de "**long**". Un tableau tab de 10 cases contenant chacune un tELT se déclare évidemment par : "tELT tab**[**10**]**;".

De même, pour simplifier les déclarations et surtout le passage des paramètres, il est recommandé de prédéfinir les types complexes.

```
#include <stdio.h>

typedef enum Boolean { vFalse=0L,vTrue,vUnknown=-1L } tBoolean;


typedef struct Personne {
  char nom[10+1];
  long age;

} tPersonne;

int main() {

tBoolean correct;
tPersonne mum,dad,toto;

correct = vFalse;
printf("correct-> %d\n",correct);

strcpy(mum.nom,"MAMAN");
mum.age = (long)45;

strcpy(dada.nom,"PAPA");
mum.age = (long)54;

strcpy(toto.nom,"TOTO");
toto.age = (long)20;

printf("Nom-> [%s]\n",toto.nom);
printf("Age-> %d\n",toto.age);

return 0;
} /* main */
```

tBoolean est maintenant équivalent à **enum**

tPersonne est maintenant équivalent à **struct**

Déclarations simplifiées de *correct* et de *toto :*
- *correct* est un énuméré
- *toto* et ses parents sont des structures

à l'écran

```
correct-> 0
```

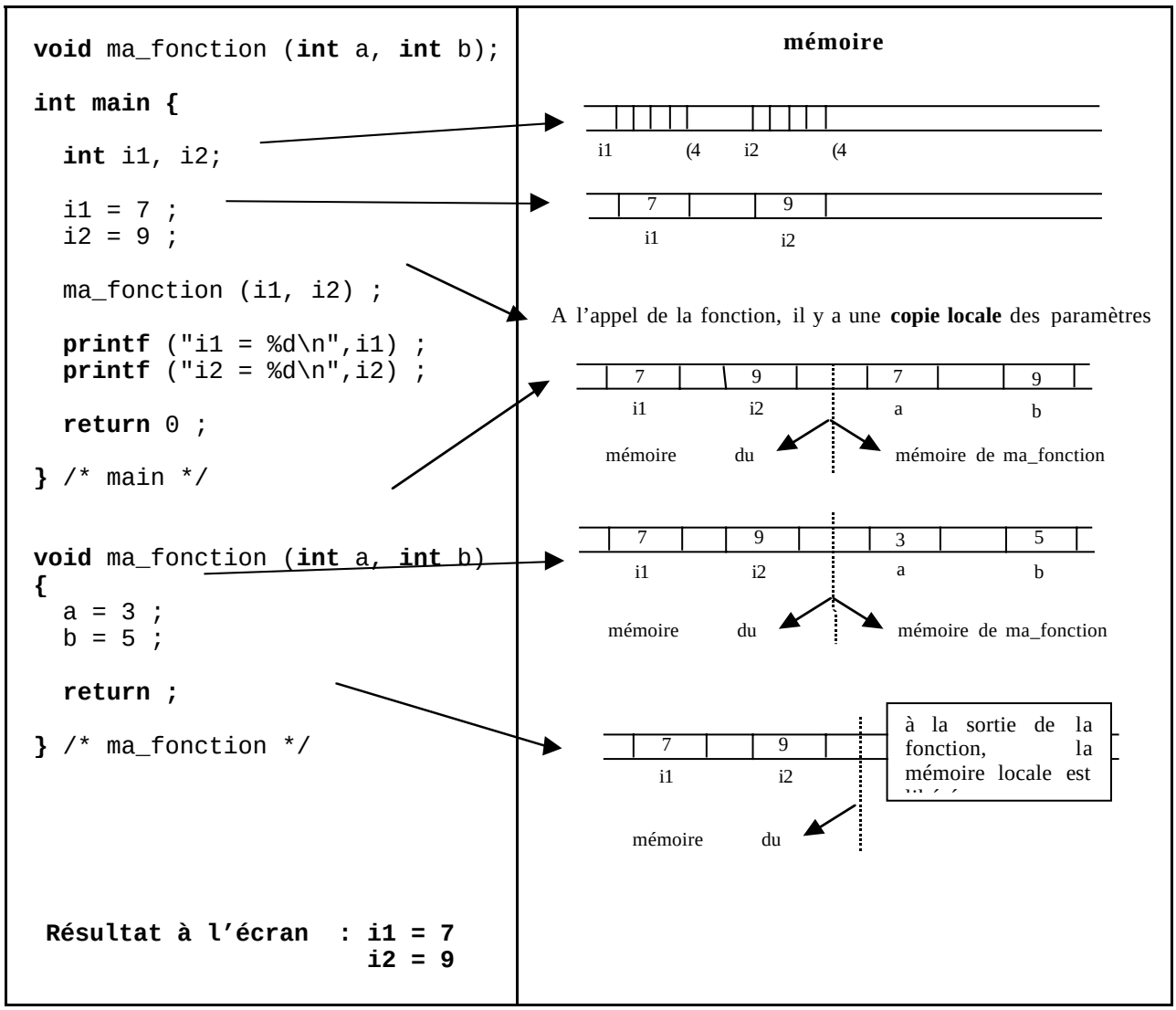
à l'écran

```
Nom-> [TOTO]
Age-> 20
```

4.8. Les fonctions et le passage des paramètres

Il est toujours préférable de découper l'ensemble du programme en modules réutilisables c'est-à-dire en **fonctions**. En C, il n'y a pas de procédure sans type, comme en ADL. On ne manipule que des fonctions. Le seul mécanisme **de passage de paramètres** est le passage par valeur, aussi il faut faire appel à la notion d'adresse pour simuler les modes de passages sortie ("out") et entrée/sortie ("inout") grâce aux opérateurs ***** sur les paramètres et **&** sur les variables.

Dessin explicatif du passage par valeur (sans adresse – mode "in")

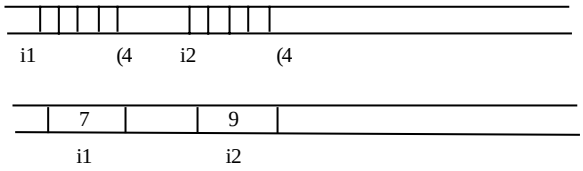


Dessin explicatif du passage par valeur (avec adresse – mode "out" ou "inout").

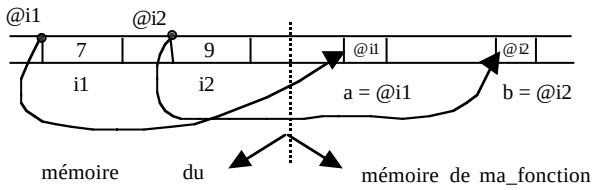
```
void ma_fonction (int *a, int *b);

int main {

  int i1, i2;

  i1 = 7 ;
  i2 = 9 ;

  ma_fonction (&i1, &i2) ;

  printf ("i1 = %d\n",i1) ;
  printf ("i2 = %d\n",i2) ;

  return 0 ;

} /* main */


void ma_fonction (int *a, int *b)
{
  *a = 3 ;
  *b = 5 ;

  return ;

} /* ma_fonction */
```

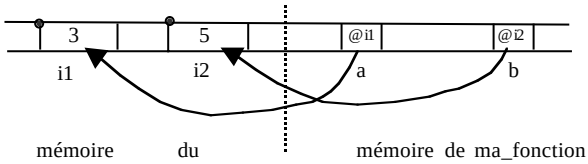
Résultat à l'écran : i1 = 3
i2 = 5

mémoire

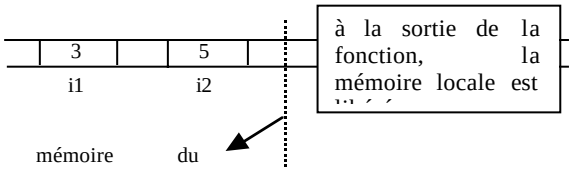


A l'appel de la fonction, il y a une **copie locale** des **adresses**

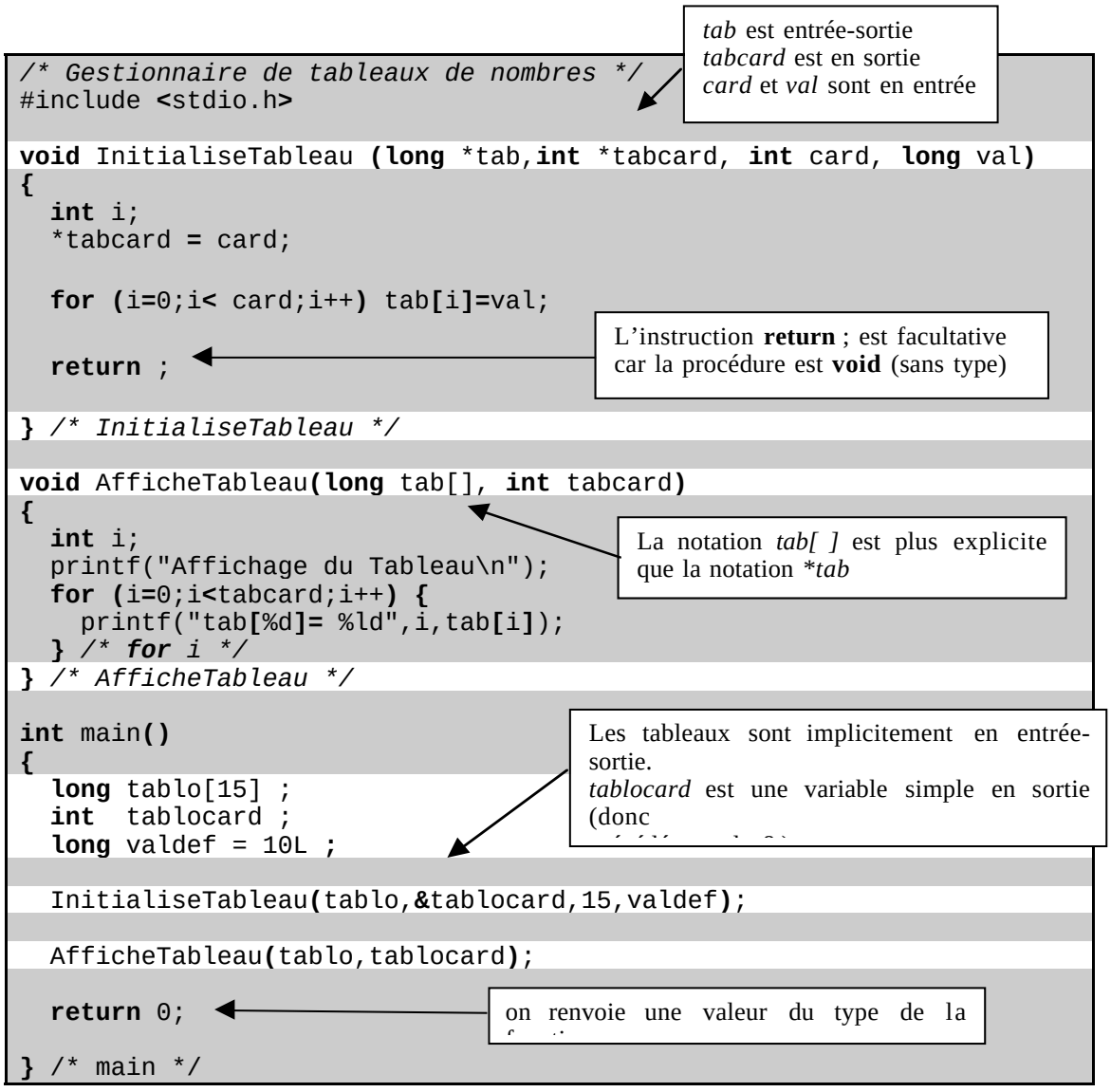




Ici, l'opérateur ***** signifie la valeur de la variable pointée. Ainsi,
L'instruction **a = 5 ;* modifie la valeur de la variable pointée par *a*, donc *i1*.



Le mécanisme du passage des paramètres expliqué précédemment est appliqué à la gestion d'un tableau.




```
/* Gestionnaire de tableaux de nombres */
#include <stdio.h>

void InitialiseTableau (long *tab,int *tabcard, int card, long val)
{
  int i;
  *tabcard = card;

  for (i=0;i< card;i++) tab[i]=val;

  return ;
} /* InitialiseTableau */

void AfficheTableau(long tab[], int tabcard)
{
  int i;
  printf("Affichage du Tableau\n");
  for (i=0;i<tabcard;i++) {
    printf("tab[%d]= %ld",i,tab[i]);
  } /* for i */
} /* AfficheTableau */

int main()
{
  long tablo[15] ;
  int  tablocard ;
  long valdef = 10L ;

  InitialiseTableau(tablo,&tablocard,15,valdef);

  AfficheTableau(tablo,tablocard);

  return 0;
} /* main */
```

Le cas particulier du passage des structures

Il existe 3 méthodes de passage d'une structure en paramètre d'une fonction :

- chaque membre de la structure séparément ;
- la structure entière par valeur : copie ou adresse selon le compilateur ;
- **par adresse** : solution rapide à l'exécution mais on risque de modifier le contenu.

Dans chaque fonction, on est obligé de passer en paramètre le cardinal (le nombre d'éléments présents) de chaque tableau. Il est plus commode de définir une **structure** regroupant le tableau et son cardinal pour n'avoir qu'un paramètre à passer et éviter des erreurs.

Si on définit une structure comme suit :

```
typedef struct _tTableau {
  long tab[gkNBMAX];
  int  card;
} tTableau;
```

Le gestionnaire de tableau décrit précédemment devient alors :

```
/* Gestionnaire de tableaux de nombres */

#include <stdio.h>

typedef struct _tTableau {
  long tab[15];
  int  card;
} tTableau;


void InitialiseTableau(tTableau *t,int card,long val)
{
  int i;

  t->card = card;

  for (i=0;i< t->card;i++) t->tab[i]=val;

  return ;

} /* InitialiseTableau */

void AfficheTableau(tTableau *t)
{
  int i;

  printf("Affichage du Tableau\n");
  for (i=0;i<t->card;i++) {
    printf("tab[%d]= %ld",i,t->tab[i]);
  } /* for i */

  return ;

} /* AfficheTableau */

int main()
{
  tTableau tablo;

  InitialiseTableau(&tablo,15,(long)10);

  AfficheTableau(tablo);

  return 0;
} /* main */
```

définition d'une structure *tTableau* pour réduire le nombre de paramètres

t.card* est dangereux car **(t.card)* est incohérent. On lui préfère **la notation fléchée

tablo est une variable locale de type *tTableau*

La portée des variables :

- les variables locales ne sont connues que dans la fonction (ou le bloc) où elles sont déclarées ;
- les variables globales ne sont connues que dans les fonctions qui suivent les déclarations (au sein du même fichier source) ;
- les variables globales exportées sont connues dans tous les fichiers (car la compilation est séparée). Il existe de multiples solutions, la plus propre étant la suivante :

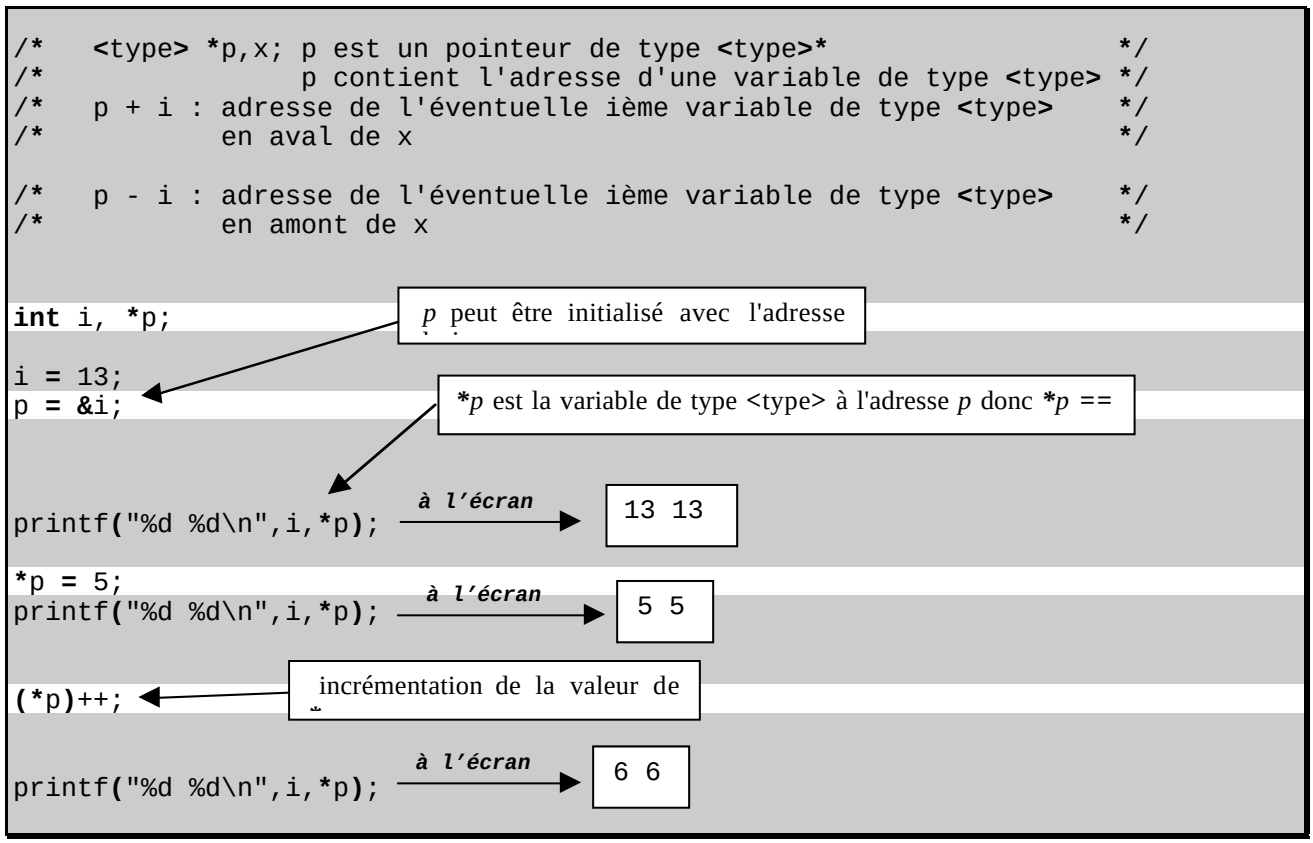
```
/* FICHIER 1 */

/* definition ANSI (declaration et initialisation) avec allocation */
int x = 0;

int main ()
{
  ....
} /* main */
```

```
/* FICHIER 2 */

/* seulement une déclaration sans allocation */
extern int x;

int ma_fonction(int x)
{
  ....
}/* ma_fonction */
```

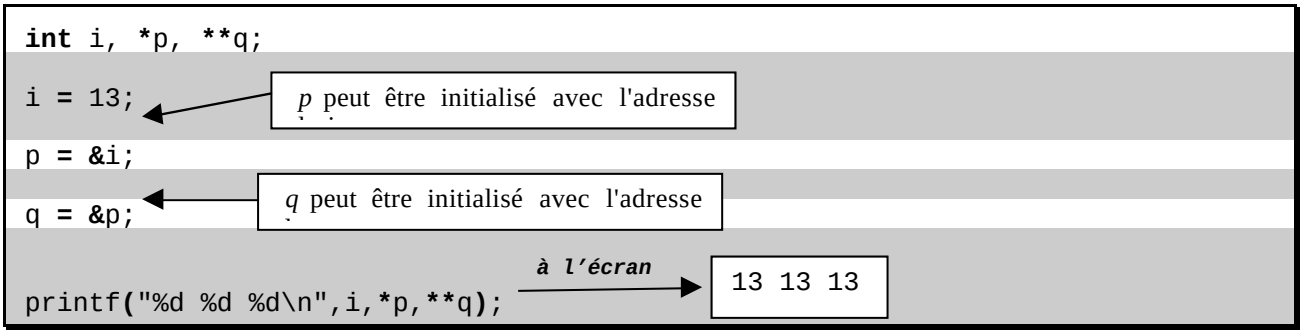
4.9. Les pointeurs

Un **pointeur** est une variable contenant l'**adresse d'une variable** typée : par extension les pointeurs sont typés. Le typage des pointeurs permet de définir une "arithmétique" réduite sur les pointeurs et une simplification pour les types structurés : ("(*p).a ⇔ p->a").

Exemple :

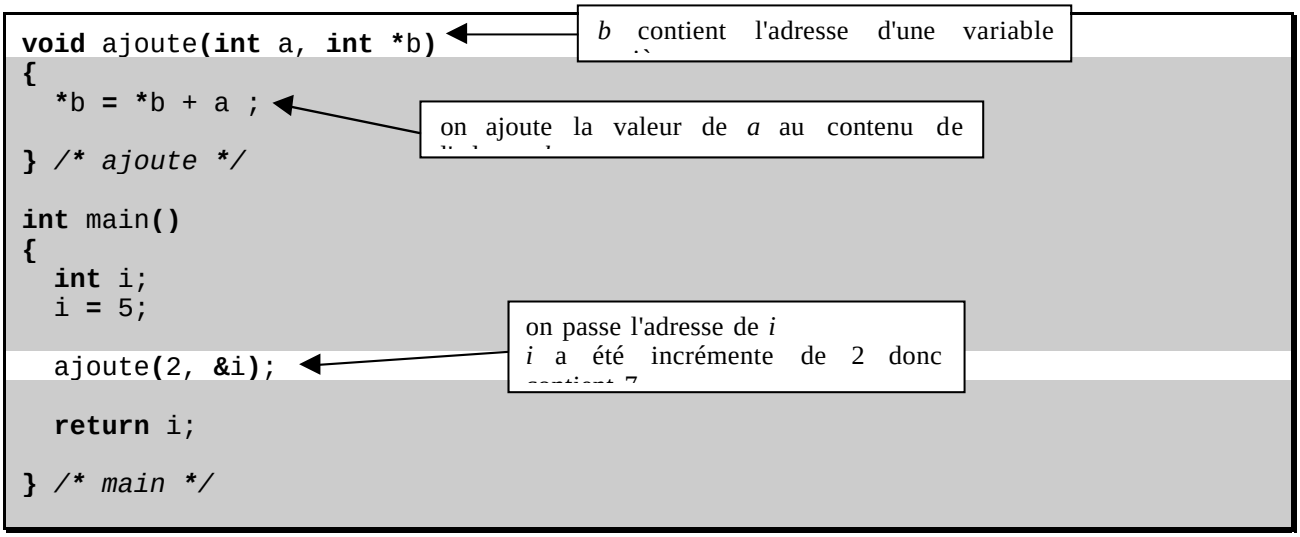


Attention, une adresse non réservée a peut-être été réservée par ailleurs et donc contenir une donnée…
Un **pointeur de pointeur** est un pointeur sur une variable de type pointeur !



Quel que soit le type de pointeur, l'affectation avec NULL est autorisée (initialisation recommandée pour indiquer que la mémoire n'a pas été réservée).

Le passage par adresse consiste à passer par valeur l'adresse de la variable (cf. page 16 - schéma du mécanisme) :




```
void ajoute(int a, int *b)
{
  *b = *b + a ;

} /* ajoute */

int main()
{
  int i;
  i = 5;

  ajoute(2, &i);

  return i;

} /* main */
```

La fonction *scanf*, qui permet de saisir des données au clavier, utilise le mécanisme du passage par adresse. Comme la fonction *printf* (cf. chapitre 4.1.), il s'agit d'une fonction à arguments variables. Dans le format de saisie, chaque "%" annonce l'utilisation d'un paramètre.

Exemples :

`char ch[10] ;`
`scanf ("%s",ch) ;` attend que l'utilisateur entre une chaîne de caractère au clavier. Le résultat de la saisie est stocké dans la variable *ch*. Attention, la chaîne saisie ne doit pas dépasser 9 caractères (*scanf* ajoute le caractère de fin de chaîne '\0').

int age ;
`scanf ("%d",age) ;` attend que l'utilisateur entre une chaîne de caractère au clavier. Le résultat de la saisie est converti en un entier et stocké dans la variable *age* de type int.
Attention : le format de saisie doit être précis. Par exemple, pour les variables de type *short*, le format "%hd" est obligatoire (contrairement à printf qui peut se contenter de %d). A noter également que le transtypage est interdit dans la fonction scanf (ne pas utiliser %d pour lire un flottant par exemple…)

Les structures récursives sont des structures dont un membre pointe sur une structure du même type. C'est possible car la déclaration d'un pointeur ne réserve pas la mémoire de la variable pointée (ce qui tombe bien puisque le compilateur ne connaît pas la taille de la structure tant que sa déclaration n'est pas finie). Ceci est particulièrement utile pour gérer le chaînage des éléments d'une liste.

```
struct Personne {
  char nom[10+1];
  long age;
  struct Personne *mere,*pere;
};
```

4.10. Les fichiers

Les **fichiers** sont des collections (ensembles) d'octets, donc tous binaires. La manipulation des fichiers est gérée par la bibliothèque **stdio.h**. Seuls leur contenu et surtout le moyen d'y accéder différencient fichiers textes et fichiers (purement) binaires. Après l'ouverture lors de laquelle on indique le mode d'accès, on manipule une **adresse** de structure "**FILE**".

On verra au chapitre suivant que l'ouverture est donc assimilable à une véritable construction (allocation de mémoire et initialisation). De même, la fermeture est une destruction (libération du fichier après une éventuelle sauvegarde et libération de la mémoire occupée par le descripteur). L'allocation et la désallocation d'un descripteur sont donc "transparentes" à l'ouverture et à la fermeture du fichier.

Les fichiers se manipulent via les fonctions "**fopen**", "**fprintf**", "**fscanf**", "**fwrite**", "**fread**", "**fseek**", "**feof**" et "**fclose**", grâce à des pointeurs sur descripteur de type FILE.

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

typedef enum Boolean { vFalse=0L,vTrue } tBoolean;

typedef struct Personne {
  char nom[10+1];
  long age;
} tPersonne;

int main() {
  FILE *fic;
  tPersonne personne;
  char chaine[255+1];


/* creation d'un fichier personne.txt en acces texte */
  fic = fopen("personne.txt","wt");
  if (fic==NULL) {
    printf("Impossible de creer le fichier texte\n");
    return 1;
  } /* if */


  strcpy(personne.nom,"TOTO");
  personne.age = (long)20;
  fprintf(fic,"%10s %d\n",personne.nom,personne.age);

  strcpy(personne.nom,"TITI");
  personne.age = (long)32;
  fprintf(fic,"%10s %d\n",personne.nom,personne.age);

  fclose(fic);


/* Relecture securisee du fichier "texte" */
  fic = fopen("personne.txt","rt");
  if (fic==NULL) {
    printf("Impossible d'ouvrir le fichier texte\n");
    return 1;
  } /* if */
```

fic est un pointeur sur descripteur de

tentative de **création** en mode

Il y a eu un problème à l'ouverture du fichier, on sort

Dans fichier → TOTO 20

Dans fichier → TITI 32

fermeture du fichier qui peut être signalée par *fic* =

tentative **d'ouverture** en mode

```
  while (NULL != fgets(chaine,255,fic)) {

    strncpy(personne.nom,chaine,10);
    personne.nom[10]='\0';

    sscanf(chaine+10,"%ld",&(personne.age));

    printf("Nom: [%s] Age : %d\n",personne.nom,personne.age);

  } /* while */


  fclose(fic);


/* creation d'un fichier personne.bin en acces binaire */
  fic = fopen("personne.bin","wb");

  if (fic==NULL) {
    printf("Impossible de creer le fichier binaire\n");
    return 1;
  } /* if */

  strcpy(personne.nom,"TOTO");
  personne.age = (long)20;
  fwrite(&personne,sizeof(tPersonne),1,fic);

  strcpy(personne.nom,"TITI");
  personne.age = (long)32;
  fwrite(&personne,sizeof(tPersonne),1,fic);

  fclose(fic);


/* Relecture securisee du fichier "binaire" */
  fic = fopen("personne.bin","rb");

  if (fic==NULL) {
    printf("Impossible d'ouvrir le fichier binaire\n");
    return 1;
  } /* if */

  while (! feof(fic)) {

    if (fread(&personne,sizeof(tPersonne),1,fic)<1) {

      break;

    } /* if */

    printf("Nom : [%s] Age : %d\n",personne.nom,personne.age);

  } /* while */


  fclose(fic);

  return 0;
```

décodage du

décodage de

à l'écran

```
Nom : [TOTO] Age : 20
Nom : [TITI] Age : 32
```

tentative **de création** en mode

écriture d'une variable de type

tentative **d'ouverture** en mode

tant qu'on n'a pas lu la fin de

impossible de lire un *tPersonne*, car on a atteint la

sortie de la

à l'écran

```
Nom : [TOTO] Age : 20
Nom : [TITI] Age : 32
```

```
} /* main */
```

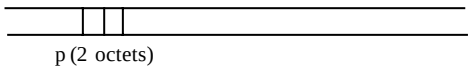
4.11. L'allocation programmée en C

En C, l'allocation de mémoire se fait grâce à la fonction "**malloc**" qui prend en paramètre la taille en octets de la zone à allouer et renvoie l'adresse de la zone allouée (NULL si l'allocation est impossible). La mémoire doit être désallouée grâce à la fonction "**free**" qui prend en paramètre (en entrée uniquement) la variable contenant l'adresse de la zone allouée.

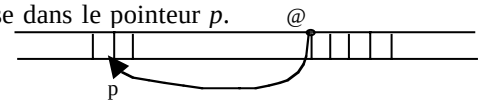
```
 int main {

  long *p;



  p = (long *)malloc(sizeof(long));


  if (p==NULL)
  {
    printf("Pb d'Allocation\n") ;
    return 1;
  } /* if */


  *p = 4 ;

  free (p) ;

  return 0 ;

} /* main */
```

mémoire

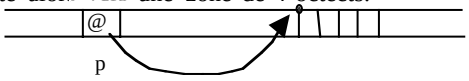
Le compilateur réserve en mémoire la place d'une adresse (2 octets)



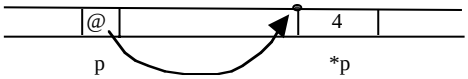
A l'exécution, **malloc** réserve la place nécessaire (4 octets) à une adresse donnée (@) et recopie cette adresse dans le pointeur *p*.



p pointe alors vers une zone de 4 octects.



**p* est le contenu de la variable pointée



Ci-dessous un exemple de manipulation de tableau dynamique :

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

int main()
{
  int  *tab;

  tab=(int*)malloc(10*sizeof(int));
  if (tab==NULL)
  {
    fprintf(stderr,"Allocation impossible\n");
    return 1;
  } /* if */

  tab[0] = 18;


  free(tab);

  return 0;

} /* main */
```

on déclare un pointeur sur

on réserve la place de 10

on manipule *tab* comme un tableau normal

on libère la place occupée par le tableau
attention : tab contient toujours la même

5. Programmer en C++

5.1. Les sources et les commentaires

À l'instar d'un programme C, un programme écrit en C++ est un ensemble de fichiers textes documentés (appelés **sources**) respectant une grammaire précise bien qu'il existe toujours plusieurs façons d'écrire la même chose. Ce document présente la solution préconisée mais aussi les alternatives les plus fiables sous forme de **commentaires** (c'est-à-dire une suite de caractères, de préférence non accentués, qui seront ignorés par le compilateur et ne servent qu'à documenter le programme). On peut indiquer le début des commentaires et préciser qu'ils se terminent à la fin de la ligne courante ou bien (comme en C) indiquer le début et la fin des commentaires éventuellement sur des lignes différentes :

// ce symbole indique le debut d'un commentaire qui se terminera a la fin de la ligne

ou bien

/ ce symbole indique le debut d'un commentaire qui peut être sur plusieurs lignes et ne se terminera qu'apres le symbole */*

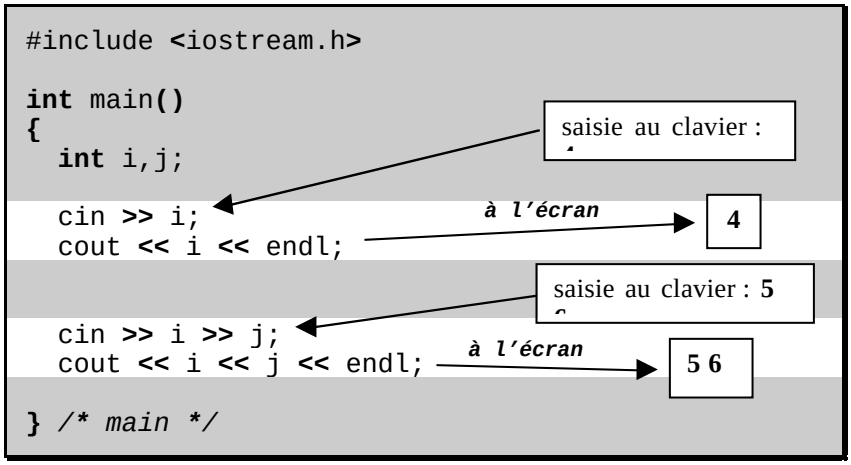
Un source est organisé de la manière suivante :

```
// Commentaires sur le contenu du fichier

// Indications des fichiers qui devront etre inclus par le compilateur

// Definitions des constantes et des types complexes utilises par la suite

// Prototypes des fonctions qui seront decrites plus loin

// Corps des fonctions

/* Fonction principale (appelée "main") si ce fichier est celui qui contient le
"programme principal" */
```

Pour l'affichage et la saisie, les fonctions standards sont toujours connues mais on dispose également de fonctions de gestion des flux gérées par la bibliothèque **iostream.h** :

cout est le flux de sortie standard (écran)
cin est le flux d'entrée standard (clavier)
cerr est le flux de sortie standard d'erreur (écran)
clog est le flux de sortie standard d'erreur tamponné (moins portable que cerr)

L'opérateur "<<" envoie des valeurs dans "**cout**", réciproquement ">>" récupère des valeurs de "**cin**". Le saut de ligne est spécifié par "**endl**", par exemple : "cout << endl;"



Les **manipulateurs** C++ de la bibliothèque **iomanip.h** permettent de formater les entrées/sorties :

dec	Lecture/écriture d'un entier en décimal
oct	Lecture/écriture d'un entier en octal
hex	Lecture/écriture d'un entier en hexadécimal
endl	Insère un saut de ligne et vide les tampons
setw(**int** n)	Affichage de n caractères
setfill(**char** c)	Caractère de remplissage (pour remplacer les blancs)
setprecision(**int** n)	Affichage de la valeur avec n chiffres
fflush	Vide les tampons après écriture

Exemple :

```
cout << setw(4) << setfill('*') << 56 << endl ; // affichage de **56
```

5.2. Les tableaux

Les tableaux se déclarent et se manipulent comme en C. Exemple :

```
/* Ce fichier presente l'utilisation simple d'un tableau d'entiers
de type "long" (c'est-a-dire geres sur 4 octets en compilation
16 ou 32 bits) */

#include <iostream.h>

#define gkNBTAB 15

int main () {

  // Declaration des variables
  long t[gkNBTAB];
  int card=0;
  int i;


  cout << "Combien d'entiers voulez-vous entrer ? ";
  cin >> card;

  while ((card<0)||(card>=gkNBTAB)) {
    cout << "Ce doit etre compris entre 0 et " << gkNBTAB << endl;
    cout << "Combien d'entiers voulez-vous entrer ? ";
    cin >> card;
  } // while


  // On fait saisir une par une les card valeurs entieres
  for (i=0;i<card;i++) {
    cout << "Quelle est la valeur de tab[" << i << "] ? " ;
    cin >> t[i];
  } // for i


  // Affichage des valeurs entieres presentes dans le tableau
  cout << "\nContenu du tableau : "<<endl;
  for (i=0;i<card;i++) {
    cout << "tab[" << i << "] = " << t[i] <<endl ;
  } // for i

  return 0;

} /* main */
```

Pour permettre la saisie et l'affichage, il est nécessaire d'inclure le fichier **iostream.h.** Le symbole # indique une directive (un ordre) de précompilation qui est donc effectuée en priorité. Le fichier à inclure étant fourni avec le compilateur, son nom est indiqué entre chevrons car il n'est pas nécessairement présent sur le répertoire courant

La directive de précompilation **#define** est en fait une sorte de "chercher/remplacer" qui aura lieu avant la compilation. Il est également possible de définir une constante grace au mot clé "**const**" suivi de la declaration d'une variable avec initialisation "const **int** NBTAB=15;" mais il s'agit alors d'une "variable constante" qui (comme cette antinomie l'indique) peut être modifiée ultérieurement !!!

Pour simplifier à l'extrême, on écrit toutes les instructions dans le "**main**". Respectons son prototype de base qui est **int** main () ;

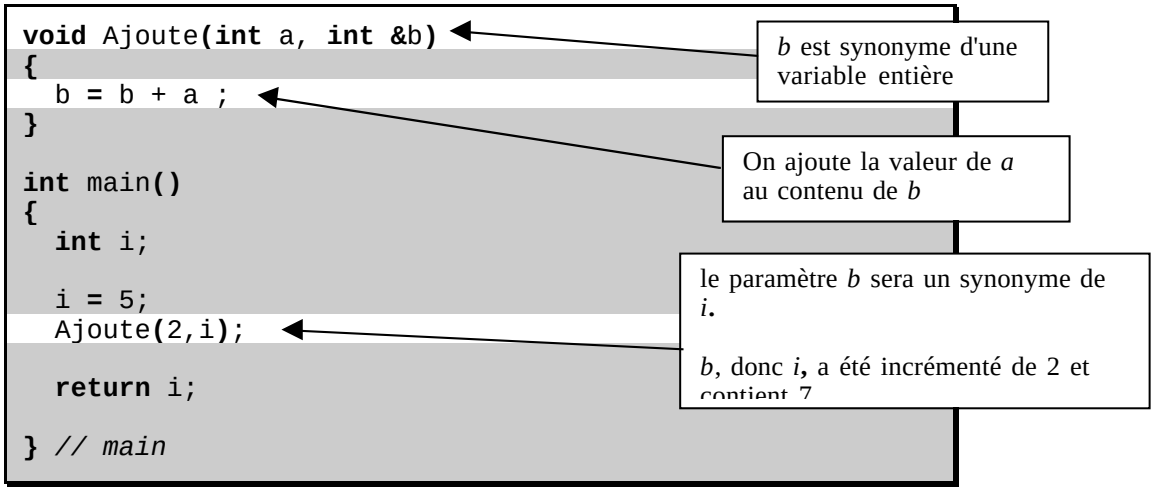
t est un tableau de *gkNBTAB* cases de type "**long**"
card est le nombre d'entiers présents dans le tableau (0 au départ)
i est l'indice de boucle pour saisir ou afficher le contenu du

on récupère dans card le nombre d'entiers à

5.3. Les fonctions

La syntaxe des fonctions C et le passage des paramètres par valeur uniquement (avec son cortège d'adresses) est valable en C++. Mais cette évolution du langage permet également d'utiliser le mécanisme de **référence** qui facilite la gestion des paramètres en entrée/sortie.

En C++, les **références** sont des "synonymes" de variables existantes. Elles doivent obligatoirement être initialisées, qui plus est, par une variable du même type. Elles simplifient le passage des paramètres qui ressemble alors à un passage par variable (et non plus par valeur). Pour éviter la modification d'un paramètre transmis par référence on peut utiliser le mot clé "const".

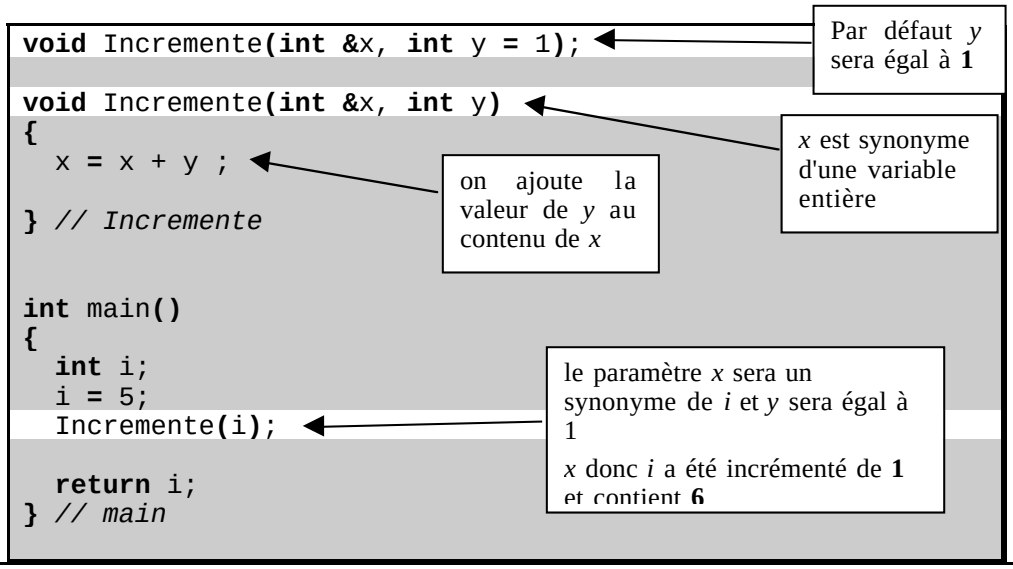


La surcharge des fonctions : un aspect de la notion de **polymorphisme** (inexistante en C).

En C++, les fonctions peuvent être surchargées : deux fonctions peuvent avoir le même nom si leurs types de paramètres ou leurs nombres de paramètres diffèrent.

```
void affiche(int);          // affichera un entier
void affiche(const char *); // affichera une chaine de caracteres
```

En C++, grâce à la notion de surcharge, les prototypes de fonctions peuvent contenir des paramètres par défaut. Attention, ces paramètres sont forcément les derniers ...



L'opérateur de portée : une conséquence de la notion de **polymorphisme** (inexistante en C).

En C++, apparaît l'**opérateur de portée** :: qui permet d'accéder à des variables définies au niveau externe :

```
/* definition ANSI (declaration et initialisation)
avec allocation */

int x = 0;

int main ()
{
      int x = 0;

      ::x = 115;

      x = 56;
      ....

} /* main */
```

la variable externe prend la valeur 115

la variable locale prend la valeur 56

L'application de ces notions à l'exemple du gestionnaire de tableaux…

Pour illustrer ce chapitre, nous allons enrichir l'exemple précédent sur les tableaux en définissant une fonction de saisie et une fonction d'affichage. La fonction d'affichage ne prendra que des paramètres en entrées (le tableau et le nombre d'entiers présents) et ne renverra rien c'est donc une "procédure sans type" autrement dit "**void**". Par contre, la fonction de saisie doit retourner le nombre d'entiers saisis et mettre à jour le tableau. On peut soit écrire une fonction qui retourne le nombre d'entiers soit écrire une fonction ayant le nombre d'entiers en sortie. Comme la surcharge le permet, on va écrire les deux.

```
// Fonction de saisie du tableau : saisie du nombre d'entiers
// puis des valeurs
int SaisieTableau(long t[]) {

  int card=0,i;

  cout << "Combien d'entiers voulez-vous entrer ? ";
  cin >> card;

  while ((card<0)||(card>=gkNBTAB)) {
    cout << "Ce doit etre compris entre 0 et " << gkNBTAB << endl;
    cout << "Combien d'entiers voulez-vous entrer ? ";
    cin >> card;
  } // while

  // On saisit une par une les card valeurs entieres
  for (i=0;i<card;i++) {
    cout << "Quelle est la valeur de tab["<< i <<"] ?" ;
    cin >> t[i];
  } // for i

  return card;

} // SaisieTableau
```

card est un paramètre local **initialisé** du même type que le

l'indice du tableau doit

on renvoie une valeur du type du retour de la

```
// Procedure de saisie du tableau : saisie du nombre d'entiers
// puis des valeurs
void SaisieTableau(long t[],int &card) {

  int i;

  cout << "Combien d'entiers voulez-vous entrer ? ";
  cin >> card;

  while ((card<0)||(card>=gkNBTAB)) {
    cout << "Ce doit etre compris entre 0 et " << gkNBTAB << endl;
    cout << "Combien d'entiers voulez-vous entrer ? ";
    cin >> card;
  } // while

  // saisie des valeurs
  for (i=0;i<card;i++) {
    cout << "Quelle est la valeur de tab[" << i << "] ? " ;
    cin >> t[i];
  } // for i

  return ;

} // SaisieTableau

// Fonction principale
int main () {

  // Declaration des variables
  long tab[gkNBMAX];
  int ntab=0;

  ntab = SaisieTableau (tab);

  AfficheTableau(tab,ntab);

  return 0;

} // main
```

le paramètre formel s'appelle

Saisie du tableau par appel à la **fonction**.
On aurait pu utiliser la **procédure** de saisie

Affichage des valeurs entières présentes dans le

on renvoie 0 au système car le programme a été jusqu'au

5.4. L'allocation programmée en C++

La définition d'un type structuré simplifie les paramètres mais on ne peut pas modifier le nombre maximum d'éléments dans le tableau sans modifier le source ("#define gkNBMAX 15") et recompiler. En outre, les tableaux ont tous la même taille maximum ; ce qui est certes homogène mais la nécessité d'un grand nombre d'éléments dans un tableau limitera le nombre de tableaux car la mémoire n'est pas infinie. On a donc recourt à l'**allocation** programmée, c'est-à-dire à la possibilité de définir pour chaque tableau la taille maximum occupée en mémoire. Son corollaire est la **désallocation** programmée qui va nous permettre de libérer la place mémoire occupée inutilement. Attention, allocation et désallocation sont complètement à la charge du programmeur qui risque d'oublier de désallouer des zones mémoires voire de désallouer des zones mémoires occupées par d'autres données que celles dont il peut se passer !!!

L'allocation de mémoire se fait grâce au mot-clef "**new**" et la désallocation grâce au mot-clef "**delete**". On indique par une étoile que l'on manipule une variable qui contient, non pas une valeur, mais une adresse en mémoire celle de la zone allouée. Pour indiquer que l'allocation n'a pas encore été faite ou que la désallocation a déjà eu lieu, on utilise la valeur **NULL** car la zone mémoire d'adresse NULL ne peut pas être allouée. D'ailleurs, la tentative de désallocation de la zone d'adresse NULL provoque une erreur du système …

```
#include <stdio.h>

typedef long tELT;

int main () {

  // variables locales
  tELT *tab=NULL;


  tab = new tELT[10];


  if (tab == NULL) {

    cout << "L'allocation a echouee !!!" << endl;
    return 1;

  } // if


  tab[2] = 3;


  if (tab != NULL) {

    delete[] tab ;

    tab = NULL;

  } // if

} // main
```

après allocation *tab* contiendra une adresse

Allocation de 10 cases de type *tELT*

le système renvoie **NULL** si l'allocation est

sortie avec code

tab contient maintenant l'adresse d'une zone mémoire de 10 *tELT* contigus et se manipule comme un tableau de 10 *tELT* (comme si on avait écrit *tELT tab[10] ;*)
Attention : ne pas dépasser les bornes (ici de 0 à 9)

Il faut penser à désallouer toutes les zones réservées !!!
Si on oublie de désallouer la zone dont *tab* contient l'adresse, elle restera marquée comme occupée et le système ne pourra pas l'utiliser librement
Les **[]** indiquent que toutes les cases contigues sont

par précaution, on mentionne que *tab* n'est plus alloué.
Attention : si on inverse les deux lignes, la zone ne

Si on applique ce mécanisme au gestionnaire de tableaux, il faut ajouter un champ *nbmax* dans la structure pour stocker le nombre maximal d'éléments dans le buffer alloué dynamiquement. La fonction *InitialiseTableau* doit être remplacée par une fonction *AlloueTableau* qui fait l'allocation dynamique et l'initialisation. Enfin, il faut écrire une fonction *DesallouеTableau* pour libérer la mémoire et penser à l'appeler avant la fin du programme (cette contrainte sera levée plus tard…) !

```
// Gestionnaire de tableaux de nombres (entiers ou flottants)

#include <stdio.h>
#include <string.h>
#include <iostream.h>

typedef long tELT;

typedef struct _tTableau {
  tELT *tab;
  int  card,nbmax;
} tTableau;

void Affiche(tELT x);

void Affiche(tELT x) {
  cout << (tELT)x << endl;
}

bool CopieTableau(tTableau &t,tTableau &t2)
{
  int i;
  if ((t.tab==NULL)||(t2.tab==NULL)) {
    cout << "Copie impossible car un des tableaux est NULL" << endl;
    return false;
  }

  if (t.nbmax<t2.card) {
    cout << "Copie impossible car " << t.nbmax << "<" << t2.card << endl;
    return false;
  }

  t.card = t2.card;
  for (i=0;i<t.card;i++) t.tab[i]=t2.tab[i];

  return true;

} // CopieTableau

bool AjouteTableau(tTableau &t,tTableau &t2) // mise a jour
{
  int i;

  if ((t.tab==NULL)||(t2.tab==NULL)) {
    cout << "Somme impossible car un des tableaux est NULL" << endl;
    return false;
  }

  if (t.card!=t2.card) {
    cout << "Somme impossible car " << t.card << "!=" << t2.card << endl;
    return false;
  }

  for (i=0;i<t.card;i++) t.tab[i]+=t2.tab[i];

  return true;
```

allocation dynamique du tableau lors de

nbmax est le nombre maximum

```
} // AjouteTableau
```

```
bool AdditionneTableau(tTableau &t,tTableau &t2,tTableau &tr)
{
  int i;

  if ((t.tab==NULL)||(t2.tab==NULL)||(tr.tab==NULL)) {
    cout << "Addition impossible car un des tableaux est NULL" << endl;
    return false;
  } // if

  if (tr.nbmax<t.card) {
    cout << "Addition impossible car " << tr.nbmax << "<" << t.card << endl;
    return false;
  } // if

  if (t.card!=t2.card) {
    cout << "Somme Impossible car " << t.card << "!=" << t2.card << endl;
    return false;
  } // if

  tr.card = t.card;

  for (i=0;i<t.card;i++) tr.tab[i]=t.tab[i]+t2.tab[i];

  return true;

} // AdditionneTableau
```

```
bool AlloueTableau(tTableau &t,int card,tELT val)
{
  int i;

  t.nbmax = 0;
  t.card = 0;
  t.tab = new tELT[card];

  if (t.tab==NULL) {
    cout << "Allocation impossible ...";
    return false;
  } // if

  t.nbmax = card;
  t.card = card;

  for (i=0;i<t.card;i++) t.tab[i]=val;

  return true;

} // AlloueTableau
```

```
void AfficheTableau(tTableau &t)
{
  int i;

  if (t.tab==NULL) {
    cout << "Affichage impossible car le tableau est NULL" << endl;
    return;
  } // if

  cout << "Affichage du Tableau\n";
```

```
  for (i=0;i<t.card;i++) {
    cout << "tab[" << i << "]= ";
    Affiche(t.tab[i]);
  } // for i

} // AfficheTableau
```

```
bool DesalloueTableau(tTableau &t)
{

  if (t.tab==NULL) {
    cout << "Desallocation impossible ...";
    return false;
  } // if

  delete[] t.tab;
  t.nbmax = 0;
  t.card = 0;

  return true;

} // DesallouеTableau


// fonction principale
int main()
{
  int retour=0;
  tTableau t,w;

  AllouеTableau(t,15,(tELT)10);
  cout << "Le tableau t contient :" << endl;
  AfficheTableau(t);

  AllouеTableau(w,10,(tELT)5);
  cout << "Le tableau w contient :" << endl;
  AfficheTableau(w);

  CopieTableau(t,w);
  cout << "Le tableau t=w contient :" << endl;
  AfficheTableau(t);

  if (AjouteTableau(t,t)==true) {
     cout << "Le tableau t=t+t contient :" << endl;
     AfficheTableau(t);
  } // if

  if (AdditionneTableau(t,t,w)==true) {
     cout << "Le tableau w=t+t contient :" << endl;
     AfficheTableau(w);
  } // if

  if (DesallouеTableau(t)==false) {
     cerr << "Le tableau t n'a pas pu etre desalloue" << endl;
     retour=1;
  } // if


  if (DesallouеTableau(w)==false) {
     cerr << "Le tableau w n'a pas pu etre desalloue" << endl;
     retour=1;
```

le programme se déroule normalement

le programme a eu un problème

```
  } // if

  return retour;

} // main
```

5.5. L'encapsulation et les fonctions amies

Comme on l'a vu, les structures permettent de regrouper plusieurs champs et ainsi de réduire le nombre de paramètres des fonctions. Cependant, les fonctions doivent toujours être écrites séparément, avoir un nom explicite (pour éviter les confusion) et au moins un paramètre (une **instanciation** de la structure). Le langage C++ est "orienté objet", il offre donc la possibilité d'**encapsuler**, c'est-à-dire de définir des **classes** qui contiennent des **membres**. Les membres sont soit des **champs** comme les structures soit des **méthodes**. Les méthodes ne sont ni plus ni moins que des fonctions propres à la classe. Pour appeler une méthode, la notation pointée est utilisée comme pour les champs d'une structure. L'**instance** de la classe (appelé **objet**) est alors implicitement passer en paramètre à la méthode. On économise ainsi un paramètre !

Pour le gestionnaire de tableaux, on passe de la notion de structure à la notion de classe selon le schéma suivant :

```
typedef struct {

  tELT *tab;
  int  card,nbmax;
} tTableau ;
// Prototypes des fonctions
AllouеTableau (tTableau &t,
               int card,tELT val);
DesallouеTableau (tTableau &t);


// Corps des fonctions

bool AllouеTableau(tTableau &t,
                   int card,tELT val)
{
  ...
}

bool DesallouеTableau(tTableau &t)
{
  ...
   delete[] t.tab ;
  ...
}
```

```
class tTableau {
  public : //autorise l'acces
  tELT *tab;
  int  card,nbmax;

  // Prototype des méthodes
  Alloue (int card,tELT val);
  Desalloue ();
} ;


// Corps des méthodes

bool tTableau::Alloue (int card,
                       tELT val)
{
   ...
}

bool tTableau::Desalloue ()
{
 ...
  delete[] tab ;
 ...
}
```

Le mot réservé "**class**" remplace les mots "**typedef struct**". On autorise l'accès aux champs grâce à "**public** :" (Cf. § suivant). Les fonctions manipulant une instance de ce type sont définies dans la classe. Elles ont implicitement comme paramètre l'objet qui va servir à les appeler. On écrit le corps

de la fonction à l'extérieur de la classe. Pour éviter toute confusion et préciser qu'il y a un paramètre implicite (l'objet lui-même), la fonction est préfixée par le nom de la classe suivie de "**::**", par exemple : "**bool** tTableau**::**Desalloue() ;".

Dans le corps des méthodes, on peut manipuler les membres de l'objet implicitement passé en paramètre. Pour ce faire, il suffit de nommer le champ ou la méthode sans le (ou la) faire précéder d'un "**.**" (c'est-à-dire omettre la notation pointée), par exemple :

```
delete[] t.tab;    =>   delete[] tab;
```

S'il y a une ambiguïté avec un paramètre ou si on veut être explicite, on peut préfixer chaque membre par "this->", par exemple :

```
delete[] t.tab;   =>    delete[] this->tab;
```

Contrairement aux champs des structures, les membres d'une classe ne sont pas accessibles directement sauf par les méthodes. On autorise l'accès à certains membres en les faisant précéder par "**public :**". En fait, il est possible de restreindre l'accès autrement mais nous le verrons lors la présentation de la notion d'héritage. Cette possibilité d'interdire l'accès à certains membres permet au programmeur d'éviter des manipulations hasardeuses comme la désallocation d'une zone mémoire qui devra servir plus tard.

Si on veut préserver la notion de fonction écrite à l'extérieur de la classe, on peut définir des fonctions dites **amies** qui auront le droit de manipuler les membres y compris ceux qui sont privés. Pour ce faire, il suffit de recopier le prototype de la fonction dans la classe en le faisant précéder de la mention "**friend**" signalant l'autorisation de manipuler tous les membres.

```
// Gestionnaire de tableaux de nombres (entiers ou flottants)

#include <stdio.h>
#include <string.h>
#include <except.h>
#include <iostream.h>

typedef long tELT;

// Definition d'une classe tableau et encapsulation
class tTableau {
   private :
       tELT *tab;
       int  card,nbmax;

   public :

       bool Copie(tTableau &t2);
       bool Ajoute(tTableau &t2);

       friend bool AdditionneTableau(tTableau &t,tTableau &t2,tTableau &tr) ;

       bool Alloue(int card,tELT val);
       bool Desalloue();
};

void Affiche(tELT x);

void Affiche(tELT x) {
  cout << x << endl;
} // Affiche
```

les membres de la classe sont inaccessibles en dehors des méthodes ou fonctions amies. Le mot-clé **private** est facultatif (valeur par

Les membres de la classe sont accessibles partout. Le mot-clé **public** est obligatoire

La fonction *AdditionneTableau* n'est pas une méthode mais une fonction amie

Attention : l'oubli de ce **;** entraine des messages d'erreurs peu explicites à la

```
bool tTableau::Copie(tTableau &t2)
{
  int i;

  if ((tab==NULL)||(t2.tab==NULL)) {
    cout << "Copie impossible car un des tableaux est NULL" << endl;
    return false;
  } // if

  if (nbmax<t2.card) {
    cout << "Copie impossible car " << nbmax << "<" << t2.card << endl;
    return false;
  } // if

  card = t2.card;
  for (i=0;i<card;i++) tab[i]=t2.tab[i];
  return true;

} // tTableau::Copie


bool tTableau::Ajoute(tTableau &t2)
{
  int i;

  if ((tab==NULL)||(t2.tab==NULL)) {
    cout << "Somme impossible car un des tableaux est NULL" << endl;
    return false;
  } // if

  if (card!=t2.card) {
    cout << "Somme impossible car " << card << "!=" << t2.card << endl;
    return false;
  } // if

  for (i=0;i<card;i++) tab[i]+=t2.tab[i];

  return true;

} // tTableau::Ajoute


bool AdditionneTableau(tTableau &t,tTableau &t2,tTableau &tr)
{
  int i;

  if ((t.tab==NULL)||(t2.tab==NULL)||(tr.tab==NULL)) {
     cout << "Addition impossible car un des tableaux est NULL" << endl;
     return false;
  } // if

  if (tr.nbmax<t.card) {
     cout << "Addition impossible car " << tr.nbmax << "<" << t.card << endl;
     return false;
  } // if

  if (t.card!=t2.card) {
     cout << "Somme impossible car " << t.card << "!=" << t2.card << endl;
     return false;
  } // if

  tr.card = t.card;
```

```
  for (i=0;i<t.card;i++) tr.tab[i]=t.tab[i]+t2.tab[i];

  return true;

} // AdditionneTableau

bool tTableau::Alloue (int card,tELT val)
{
  int i;

  nbmax = 0;
  card = 0;
  tab = new tELT[card];

  if (tab==NULL) {
     cout << "Allocation impossible ...";
     return false;
  } // if
```

```
  nbmax = card;
  this->card = card;

  for (i=0;i<card;i++) tab[i]=val;

  return true;

} // tTableau::Alloue


void tTableau::Affiche ()
{
  int i;

  if (tab==NULL) {
    cout << "Affichage impossible car le tableau est NULL" << endl;
    return;
  } // if

  cout << "Affichage du Tableau\n";

  for (i=0;i<card;i++) {
     cout << "tab[" << i << "]= ";
     Affiche(tab[i]);
  } // for i

} // tTableau::Affiche


bool tTableau::Desalloue ()
{
  if (tab==NULL) {
     cout << "Desallocation impossible ...";
     return false;
  } // if

  delete[] tab;
  nbmax = 0;
  card = 0;

  return true;

} // tTableau::Desalloue
```

Pas vraiment nécessaire mais homogène avec *Alloue*

```
// fonction principale
int main()
{
  int retour=0;
  tTableau t,w;

  t.Alloue(15,(tELT)10);
  cout << "Le tableau t contient :" << endl;

  t.Affiche() ;

  w.Alloue(10,(tELT)5);
  cout << "Le tableau w contient :" << endl;
  w.Affiche();

  t.Copie(w);
  cout << "Le tableau t=w contient :" << endl;
  t.Affiche();
```

```
  if (t.Ajoute(t)==true) {
     cout << "Le tableau t=t+t contient :" << endl;
     t.Affiche();
  } // if

  // AdditionneTableau est une fonction "friend" ...
  if (AdditionneTableau(t,t,w)==true) {
     cout << "Le tableau w=t+t contient :" << endl;
     Affiche(w);
  } // if

  if (t.Desalloue()==false) {
     cerr << "Le tableau t n'a pas pu etre desalloue" << endl;
     retour=1;
  } // if

  if (w.Desalloue()==false) {
     cerr << "Le tableau w n'a pas pu etre desalloue" << endl;
     retour=1;
  } // if

  return retour;

} // main
```

5.6. Les constructeurs et le destructeur (d'une classe)

L'encapsulation ayant été présentée, on constate qu'il est encore primordiale de faire appel à la méthode Alloue(…) pour "créer" un tableau et à la méthode "Desalloue()" pour libérer la mémoire. Cette obligation peut être confiée au compilateur grâce aux notions de **constructeur** et de **destructeur**. Un constructeur (respectivement le destructeur) d'une classe est appelé automatiquement à chaque déclaration (respectivement libération) d'une variable. Comme la déclaration d'une variable est explicite, on peut définir plusieurs constructeurs (polymorphisme) dont le nom est celui de la classe. Par contre, la libération d'une variable est implicite donc le destructeur est unique (pas de polymorphisme) et son prototype est le nom de la classe précédé de "~" et il n'a évidemment pas de paramètre (sinon comment les déterminer implicitement ?). Attention, les constructeurs et le destructeur d'une classe n'ont pas de retour car ils sont appelés automatiquement !!!

Pour le gestionnaire de tableaux, on peut considérer que la méthode Alloue(…) est un constructeur et que la méthode Desalloue() est le destructeur :

```
bool Alloue (...)      =>    tTableau(...);
bool Desalloue ()      =>    ~tTableau();
```

Les constructeurs et le destructeur d'une classe sont facultatifs. Lors de la déclaration d'une variable objet, le compilateur recherche des parenthèses derrière le nom de la variable pour identifier le constructeur à appeler. S'il n'y a pas de parenthèses, le compilateur considère que le constructeur sans paramètre est appelé et s'il ne trouve pas de constructeur sans paramètre, il construit la variable comme il construit habituellement une instance de structure. Lors de la libération d'une variable objet, le compilateur recherche le destructeur de la classe. S'il n'y a pas de destructeur, il libère la mémoire comme si c'était une instance de structure.

```
// Gestionnaire de tableaux de nombres (entiers ou flottants)

#include <stdio.h>
#include <string.h>
#include <iostream.h>

typedef long tELT;

// Definition d'un constructeur et du destructeur
class tTableau {

  private :
     tELT *tab;
     int  card,nbmax;

  public :
     bool Copie(tTableau &t2);
     bool Ajoute(tTableau &t2);
     friend bool AdditionneTableau(tTableau &t,tTableau &t2,tTableau &tr);

     tTableau(int card,tELT val);

     ~tTableau();

};

void Affiche(tELT x);

void Affiche(tELT x) {
  cout << x << endl;
} // Affiche


bool tTableau::Copie(tTableau &t2)
{
  int i;

  if ((tab==NULL)||(t2.tab==NULL)) {
    cout << "Copie impossible car un des tableaux est NULL" << endl;
    return false;
  } // if

  if (nbmax<t2.card) {
    cout << "Copie impossible car " << nbmax << "<" << t2.card << endl;
    return false;
  } // if

  card = t2.card;

  for (i=0;i<card;i++) tab[i]=t2.tab[i];

  return true;

} // tTableau::Copie


bool tTableau::Ajoute(tTableau &t2) // mise a jour
{
  int i;

  if ((tab==NULL)||(t2.tab==NULL)) {
    cout << "Somme impossible car un des tableaux est NULL" << endl;
    return false;
```

un constructeur <= **bool** Alloue ( ).

Le destructeur <= **bool** Desalloue ( ).

```
  } // if
```

```
  if (card!=t2.card) {
    cout << "Somme impossible car " << card << "!=" << t2.card << endl;
    return false;
  } // if

  for (i=0;i<card;i++) tab[i]+=t2.tab[i];

  return true;

} // tTableau::Ajoute



bool AdditionneTableau(tTableau &t,tTableau &t2,tTableau &tr)
{
  int i;

  if ((t.tab==NULL)||(t2.tab==NULL)||(tr.tab==NULL)) {
    cout << "Addition impossible car un des tableaux est NULL" << endl;
    return false;
  } // if

  if (tr.nbmax<t.card) {
    cout << "Addition impossible car " << tr.nbmax << "<" << t.card << endl;
    return false;
  } // if

  if (t.card!=t2.card) {
    cout << "Somme Impossible car " << t.card << "!=" << t2.card << endl;
    return false;
  } // if

  tr.card = t.card;

  for (i=0;i<t.card;i++) tr.tab[i]=t.tab[i]+t2.tab[i];

  return true;

} // AdditionneTableau


// UN CONSTRUCTEUR
tTableau::tTableau(int card,tELT val)
{
  int i;

  nbmax = 0;
  card = 0;
  tab = new tELT[card];

  if (tab==NULL) {
    cout << "Allocation impossible ...";
    return ;
  } // if

  nbmax = card;

  this->card = card;
```

```
  for (i=0;i<card;i++) tab[i]=val;
}  // tTableau::tTableau
```

Attention : il n'y a pas de retour possible pour un constructeur, donc pas d'instruction retour ;

```
void tTableau::Affiche ()
{
  int i;

  if (tab==NULL) {
    cout << "Affichage impossible car le tableau est NULL" << endl;
    return;
  } // if

  cout << "Affichage du Tableau\n";
  for (i=0;i<card;i++) {
    cout << "tab[" << i << "]= ";
    Affiche(tab[i]);
  } // for i

} // tTableau::Affiche


// LE DESTRUCTEUR
tTableau::~tTableau()
{
  if (tab==NULL) {
    cout << "Desallocation impossible ...";
  } // if

  delete[] tab;

  //nbmax = 0;
  //card = 0;

} // tTableau::~tTableau


int main()
{
  int retour=0;
  tTableau t(15,(tELT)10),w(10,(tELT)5);

  cout << "Le tableau t contient :" << endl;
  t.Affiche();
  cout << "Le tableau w contient :" << endl;
  w.Affiche();

  t.Copie(w);
  cout << "Le tableau t=w contient :" << endl;
  t.Affiche();

  if (t.Ajoute(t)==true) {
     cout << "Le tableau t=t+t contient :" << endl;
     t.Affiche();
  } // if

  // AdditionneTableau est une fonction "friend" ...
  if (AdditionneTableau(t,t,w)==true) {
     cout << "Le tableau w=t+t contient :" << endl;
```

Ces deux lignes sont maintenant inutiles

Appel au constructeur pour *t* et

```
    Affiche(w);
  } // if

  return retour;

} // main
```

Le destructeur est automatiquement appelé pour t et pour w à la sortie du main

On verra deux chapitres plus loin, un constructeur très particulier : **le constructeur de copie …**

5.7. La surcharge d'opérateur et les fonctions "inline"

Le polymorphisme qui permet de définir des fonctions différentes mais ayant le même nom est étendu à la **surcharge des opérateurs**. On peut effectivement simplifier l'écriture de certaines fonctions en leur donnant le nom d'un opérateur. Par exemple, avec le gestionnaire de tableau que nous manipulons, l'accès direct à un des éléments d'un tableau est difficile (car si t est un tTableau alors t.tab[i] est le ième élément) et même impossible en dehors d'une méthode ou d'une fonction amie (car le membre tab est privé).

Pour permettre et simplifier l'accès direct aux éléments, on peut définir une méthode Valeur(…) comme suit : tELT**&** tTableau::Valeur(**int** i) { **return** tab[i]; } mais l'appel à cette méthode "t.Valeur(2)" est très différent de la notation classique t[2] à laquelle on est accoutumé. Il est alors possible d'appeler cette méthode non pas Valeur mais operator[ ], l'appel devient donc "t.operator[ ] (2)" ce qui est déjà plus explicite. En fait, le mot réservé "**operator**" signifie que l'on surcharge l'opérateur qui suit et on peut donc écrire librement t[2] pour faire appel à la méthode "operator[ ] (**int** i)". Notez l'utilisation de la référence (&), indispensable si on désire modifier le contenu du tableau (ex : t[2] = 4 ;) . Dans le cas contraire (**int** a = t[2] ;), la référence n'est pas obligatoire.

La méthode "operator[ ] (**int** i)" ayant un corps très court et très simple (pas de boucle, pas de test imbriqué), il est dommage de devoir écrire son prototype dans la classe et son corps à l'extérieur, sachant qu'un appel à cette méthode équivaut à un vrai appel de fonction (empilement des variables contextuelles, association des paramètres, exécution de la fonction, identification des paramètres en sortie et désempilement des variables contextuelles). On préféra alors soit faire précéder le corps de la fonction du mot réservé "**inline**", indiquant qu'un appel à cette méthode équivaut à un "chercher/remplacer" avant la compilation, soit carrément écrire le corps de la méthode dans la classe pour la rendre "inline" et ne pas avoir à l'écrire à l'extérieur…

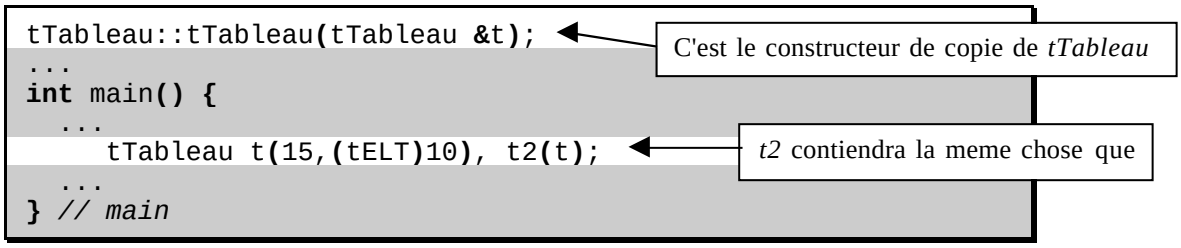
```
class tTableau {
  private :
      tELT *tab;
      int  card,nbmax;

  public :
      bool Copie(tTableau &t2);
      bool Ajoute(tTableau &t2);
      friend bool AdditionneTableau(tTableau &t,tTableau &t2,
                                    tTableau &tr);
      tTableau(int card,tELT val);
      ~tTableau();

      tELT& operator[](int i) { return tab[i];}

};
```

Surcharge de l'opérateur **[ ]** en méthode "inline"

On peut évidemment surcharger tous les opérateurs arithmétiques standards ainsi que "[ ]" (un seul paramètre autorisé) et "( )" (autant de paramètres que l'on veut)… Mais il faut alors définir un constructeur de copie dans certains cas…

5.8. Le constructeur de copie

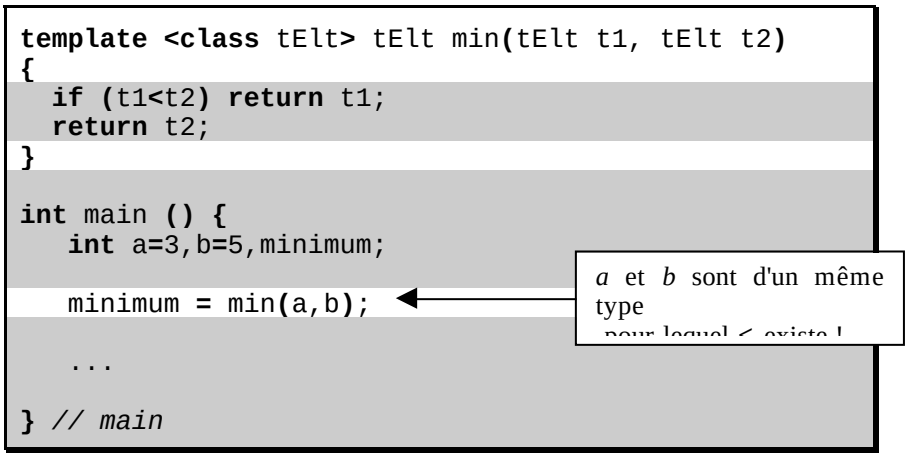
On vient de voir qu'on pouvait définir des constructeurs pour une classe. L'un d'entre eux est très particulier : il s'agit du "**constructeur de copie**" qui prend en paramètre une référence sur un objet de la classe. Il sert, comme son nom l'indique, à instancier un objet de la classe et à l'initialiser en copiant un autre objet (qui sert de modèle), par exemple :



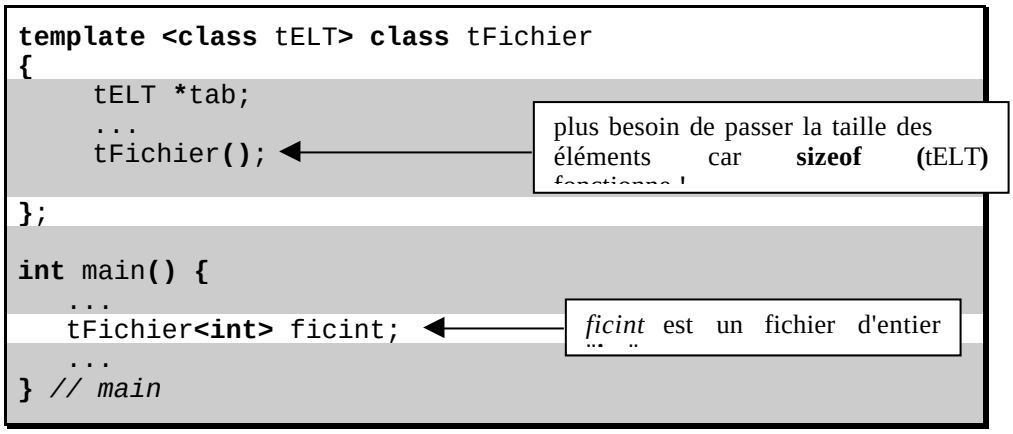
Au delà de cette utilisation, le constructeur de copie est en fait parfois nécessaire car il est appelé implicitement dès que l'on passe un objet en paramètre en entrée (c'est-à-dire sans référence). En effet, un paramètre formel (dans l'entête) en entrée est une copie du paramètre actuel (lors de l'appel). Par défaut, le constructeur de copie est une simple copie bit à bit comme pour une structure. Le constructeur de copie devient nécessaire dès que la construction d'un objet de la classe implique une allocation dynamique, sinon les deux objets (le paramètre formel et l'actuel) partage la même zone mémoire allouée dynamiquement au risque de la modifier (cf. chapitre suivant) !!!

5.9. La généricité

Plutôt que de devoir écrire n versions de la même fonction pour comparer deux variables d'un même type (par exemple trouver le minimum de deux entiers ou de deux nombres flottants), on peut faire appel au principe de **généricité** et définir un **modèle** de fonction "min(tElt t1,tElt t2)" renvoyant la plus petite des valeurs t1 et t2 (ou le plus petit des objets t1 et t2 sous réserve d'avoir surchargé l'opérateur <). Pour ce faire, on définit un modèle de fonction, c'est-à-dire une fonction générique dépendant d'une classe tElt qui sera automatiquement identifiée lors de la compilation et de l'édition de lien :



On peut sur le même principe définir une classe générique dépendant d'une classe tELT qui ne sera connue que lors de l'instanciation d'un objet de la classe sous réserve que tous les opérateurs nécessaires soient surchargés si le type tELT choisi n'est pas un type simple.



5.10. L'héritage

On peut vouloir réutiliser les membres du gestionnaire de fichiers binaires bufferisés et génériques précédent pour définir un gestionnaire de listes génériques ou un gestionnaire de tableaux (voire de matrices) génériques. Plutôt que de recopier inlassablement les mêmes membres dans chaque classe, on utilise le principe d'**héritage** qui permet de définir une classe à partir d'un ou plusieurs autres classes : "**class tListe : public tFichier**". Dorénavant un objet de la classe tListe contient un objet de la classe tFichier.

Le mot réservé "**public :**" permet de transmettre les droits d'accès aux membres de la classe tFichier : en l'absence de ce mot, l'héritage est privé donc les membres de tFichier deviennent tous privés… Réciproquement dans la classe tFichier, on peut étendre l'accès de certains membres privés aux méthodes et fonctions amies des **classes dérivées** en faisant précéder ces membres du mot réservé "**protected** :" : encore une fois, les membres qui suivent "**protected** :" sont privés sauf pour les méthodes et fonctions amies des classes dérivées.

Lorsqu'on décrit un constructeur d'objet de la classe dérivée (par exemple tListe), il faut qu'il fasse appel de manière explicite à un constructeur de la classe de base (ici tFichier). Cet appel au constructeur de la classe de base se fait généralement sous la forme d'une **préinitialisation**, c'est-à-dire juste derrière l'entête du constructeur de la classe dérivée entre "**:**" et "**;**" ou "**{**". On trouvera ci-dessous un gestionnaire de listes génériques et un gestionnaire de tableaux ou de matrices génériques qui héritent du gestionnaire de fichiers génériques.

Lorsqu'il y a héritage, les méthodes des classes dérivées **surchargent** celles des classes de base : si la fonction *affiche* est redéfinie dans une classe dérivée, celle de la classe de base est rendue inaccessible sauf recours à la notation "::" (classe de base::méthode – figures 1 et 2)

Lorsqu'une classe dérive de plusieurs classes, il y a un risque que la même méthode (non surchargée) soit héritée plusieurs fois (une fois par **classe de base**) . Dans ce cas le compilateur indique qu'il y a ambiguïté. Il existe alors deux solutions :

- appeler explicitement la méthode de la classe grâce à la notation "::" ;
- utiliser le mécanisme des classes virtuelles (figure 3).

La figure 1 (schéma ci-dessous) montre comment est déterminé l'endroit où se trouve une méthode. La figure 2 explique le mécanisme de la redéfinition de méthodes.

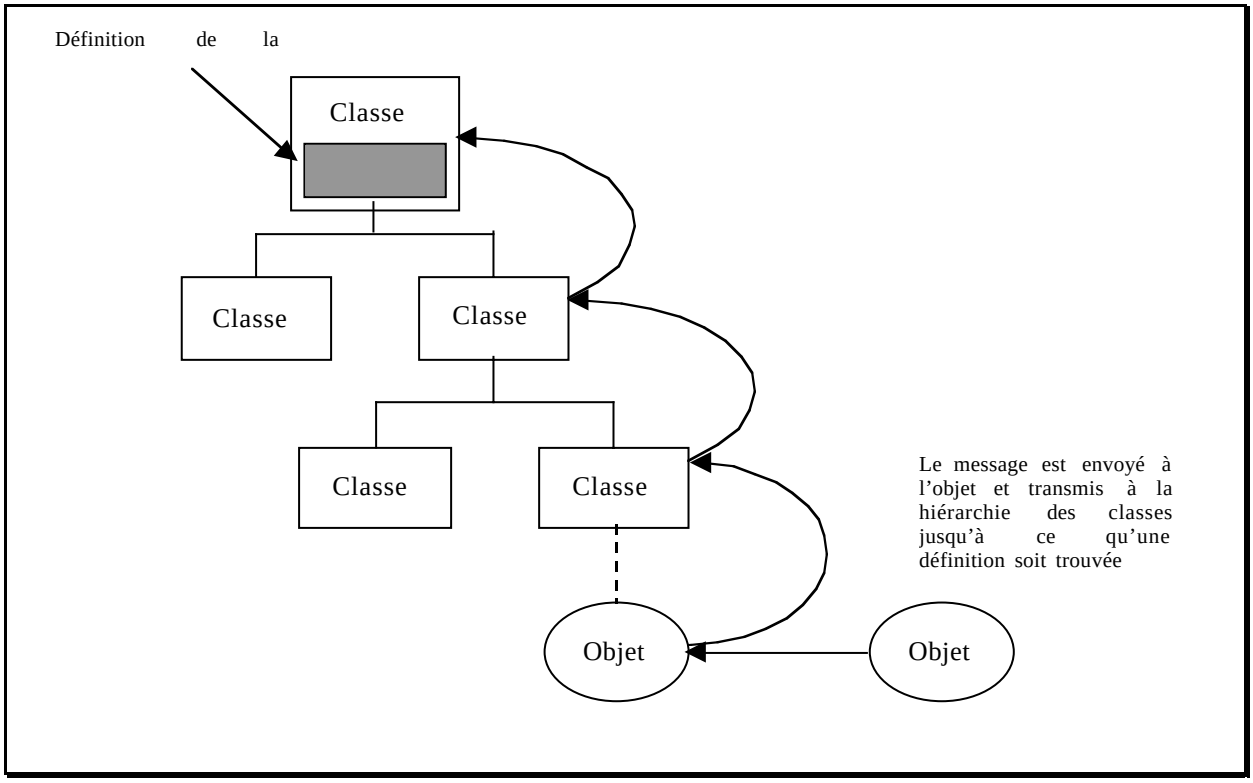


figure 1

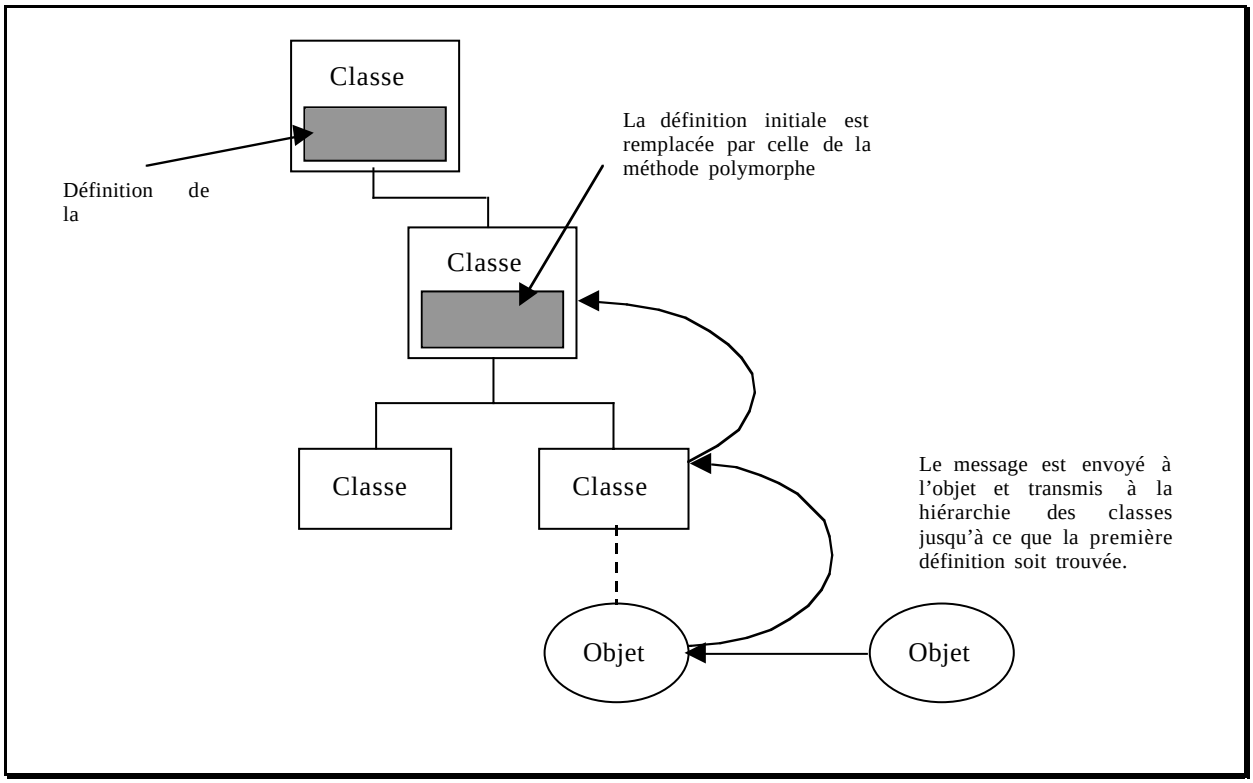


figure 2

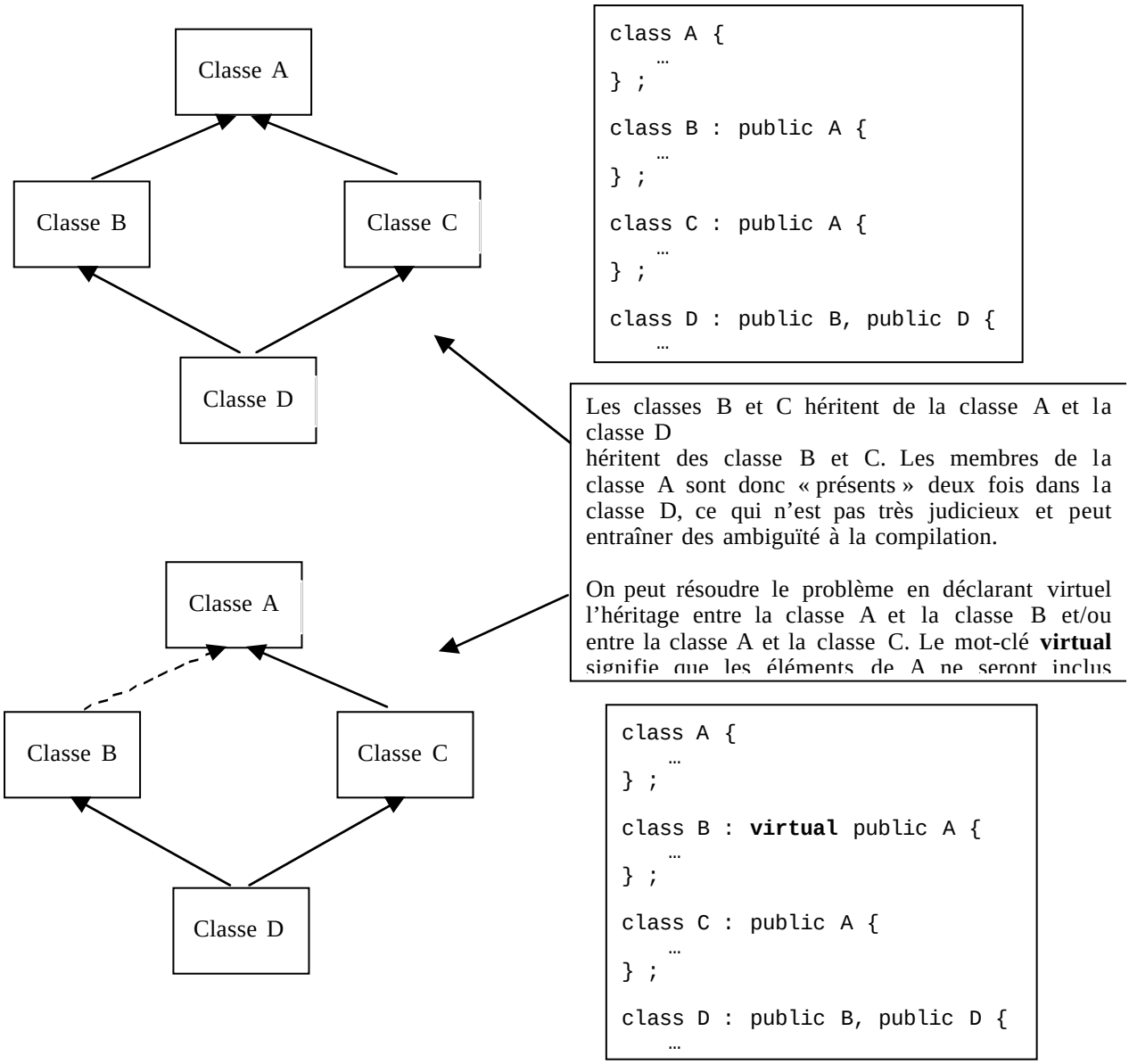


figure 3 – Le mécanisme des classes virtuelles

Un mot sur les méthodes virtuelles

Le mot réservé "**virtual**" est également utilisé pour les méthodes mais signifie tout autre chose. La définition dans une classe de base d'une méthode virtuelle possédant une autre version dans une classe dérivée indique au compilateur qu'il ne doit pas associer de « typage **statique** » à cette méthode. Le choix de la méthode à appeler (à l'exécution – de façon **dynamique**) dépend du type de l'objet pour lequel elle est appelée (figure 4 – exemple).

Les méthodes virtuelles pures (indiquées par un "=0") sont des méthodes définies dans une classe de base et non implémentées. Ces méthodes doivent **obligatoirement** être surchargées dans toutes les classes qui dérivent de cette classe de base (mécanisme utile pour définir des spécifications).
Exemple : `virtual void Affiche () = 0 ;`

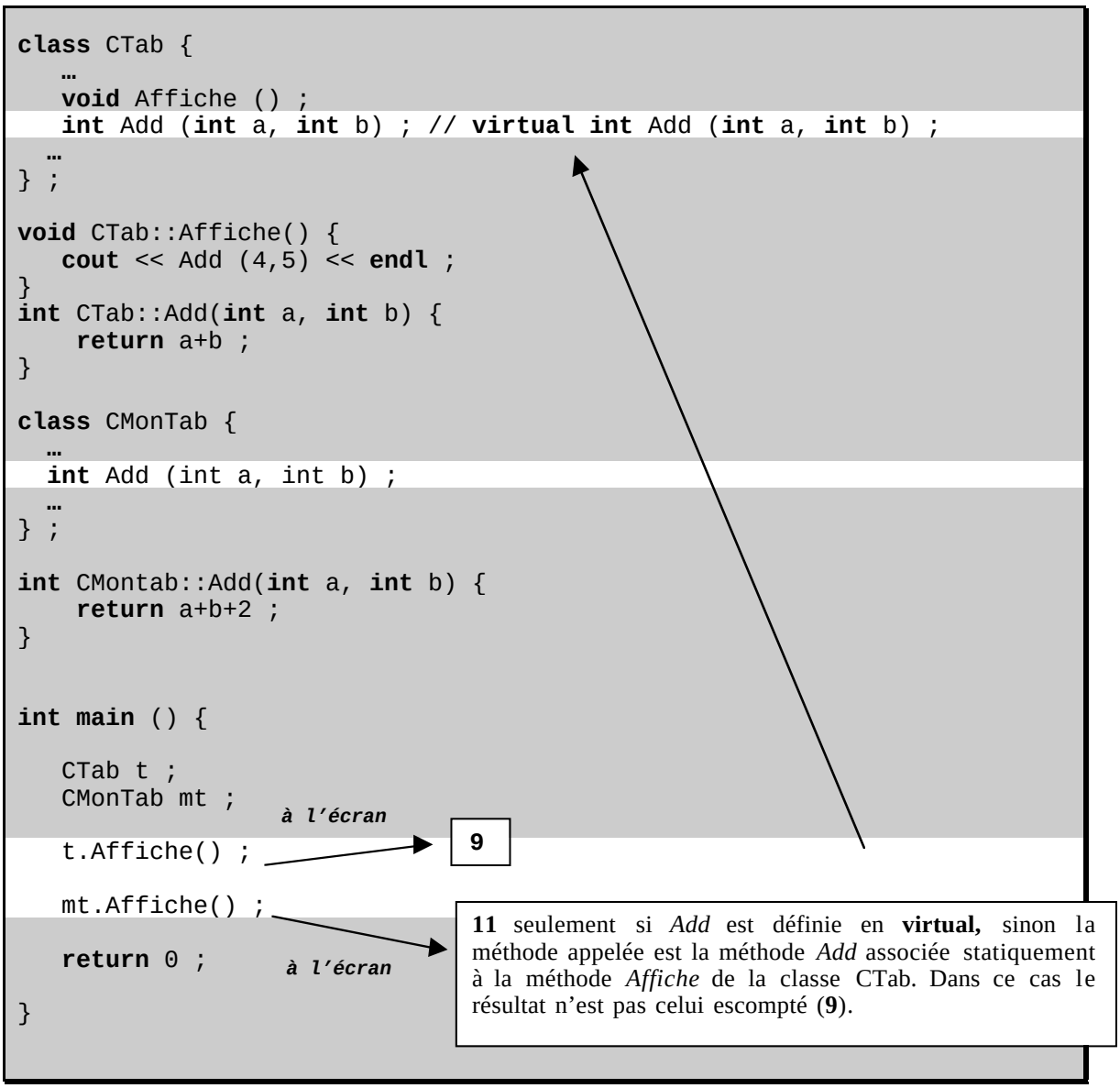
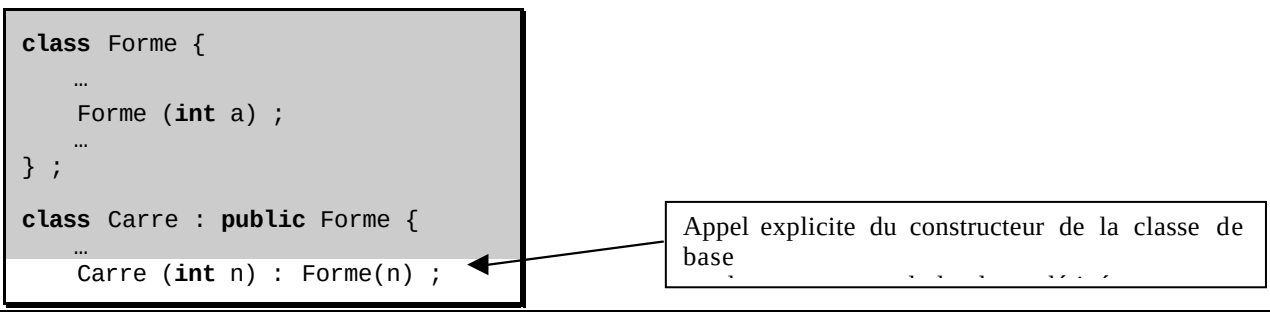


figure 4 – Exemple de méthode virtuelle

L'héritage et l'appel des constructeurs

Certaines classes dérivées ont besoin de constructeurs. Si la classe de base a un constructeur, il doit être appelé, s'il a besoin d'arguments, ils doivent être donnés. Les arguments du constructeur de la classe de base sont précisés dans la définition du constructeur de la classe dérivée. La classe de base agit ainsi comme un membre de la classe dérivée. Exemple :



```
    …
} ;
```

5.11. Les flux

Comme nous l'avons déjà précisé dans le chapitre 5.1, l'affichage et la saisie sont gérés par la bibliothèque **iostream.h** et son cortège de flux *cout, cerr, clog* et *cin*. En C++, les flux peuvent également être associés à des fichiers. La classe *fstreambase* est la classe de base pour les flux associés aux fichiers. Les classes *ifstream* et *ofstream* dérivent de *fstreambase* et gèrent respectivement les flux en entrée (lecture - in) et les flux en sortie (écriture - out). Il existe également la classe *fstream* gérant les flux à la fois en entrée et en sortie (lecture/écriture).

Les méthodes de lecture (**get**, **read**, …) et d'écriture (**put**, **write**, …) de ces classes ne peuvent gérer qu'une suite d'octets contrairement aux fonctions utilisées pour la gestion des fichiers en C (cf. chapitre 4.10). Cette philosophie rend la manipulation des fichiers en C++ plus complexe et beaucoup de programmeurs préfèrent utiliser la gestion des fichiers en C même dans un programme écrit en C++.

Exemple : copie d'un fichier dans un autre (sans écraser le fichier destination s'il existe déjà).

```
/* Copie un fichier dans un autre */

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

int main () {

   char nomficin[20], nomficout[20] ;

   ifstream f_in ;
   ofstream f_out ;


   // saisie des noms des fichiers
   cout << "Entrez le nom du fichier en lecture : " ;
   cin >> nomficin ;

   cout << "Entrez le nom du fichier en ecriture : " ;
   cin >> nomficout ;


   // ouverture des fichiers
   f_in.open (nomficin) ;
   if (!f_in) {
      cerr << "Probleme à l'ouverture du fichier " ;
      cerr << nomficin << endl;
      return 1 ;
   }

   f_out.open (nomficout,ios::noreplace) ;
   if (!f_out) {
      cerr << "Probleme à l'ouverture du fichier" ;
      cerr << nomficout << endl;
      return 1 ;
   }
```

Ouverture du fichier *f_in* en mode lecture (par défaut puisque c'est un ifstream)

Ouverture, seulement s'il existe, du fichier *f_out* en mode écriture (par défaut puisque c'est un ofstream)

```
   // Copie d'un fichier dans l'autre
   unsigned char byte ;

   byte = f_in.get() ;
   while (!f_in.eof()) {
     f_out.put(byte) ;
     byte = f_in.get() ;
   }

   f_in.close() ;
   f_out.close() ;

   return 0 ;

} // main
```

Lecture et écriture des

Fermeture des deux

Il est cependant possible (et même souhaitable) de créer ses propres classes de manipulation de fichiers en utilisant le mécanisme de l'héritage. On peut ainsi définir une classe dérivant de la classe *fstream* et écrire ses propres méthodes de lecture et d'écriture. Un tel procédé facilite grandement la manipulation des fichiers en C++. Reste que la mise en œuvre est plus longue et plus difficile qu'en utilisant le mécanisme des fichiers en C. De plus, cette méthode est intéressante pour la gestion de flux de chaînes de caractères mais elle est très difficile à appliquer aux flux binaires. Il faut en effet écrire toutes les méthodes permettant de lire et d'écrire en binaire chaque type simple (**int, float, double** …) ainsi que la méthode équivalente à la fonction **sizeof ()** du C. Pour la lecture et l'écriture de structures, on risque en plus de se heurter à des problèmes d'alignements mémoire (« padding ») difficile à gérer.

Ci-dessous un exemple d'utilisation de l'héritage pour écrire un gestionnaire de fichiers manipulant des chaînes de caractères.

```
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>

#define NOMFIC "fichier.txt"

    class T {
       public :
       float x,y,z;
       bool b;

       T() {x=0.0;y=0.0;z=0.0;b=true;}
    } ;

    class mon_fichier : public fstream  {
       public :
       mon_fichier (char nomfic[], int mode) { this->open(nomfic,mode); }
       ~mon_fichier () { this->close(); }

       void mon_put (T t)  {
         *this << t.x << " " << t.y << " " << t.z << " " << t.b << endl;
       }

       int mon_get (T &t)  {
          *this >> t.x >> t.y >> t.z >> t.b ;
```

Ma classe *mon_fichier* hérite de la classe

Les " " sont obligatoires pour permettre à l'opérateur >> (méthode *mon_get*) de relire correctement les champs écrits par *mon_put* (opérateur <<).

A travers le pointeur *this*, c'est un objet *fstream* complet qui est

```
        return !this->eof();
      }
   };
```

```
int main() {

   mon_fichier ficout("fout.txt",ios::out);

   T t1, t2;

   if (!ficout)
   {
      cout << "Probleme a la creation du fichier " << NOMFIC << endl;
      return 1;
   }

   t1.x=3.456;
   t1.y=34.56575;
   t1.z=0.0;
   t1.b=true;

   for (int i=0;i<3;i++)
      ficout.mon_put(t1);

   ficout.close() ;

   mon_fichier ficin("fin.txt",ios::in);

   if (!ficin)
   {
      cout << "Probleme a l'ouverture du fichier " << NOMFIC << endl;
      return 2;
   }

   while (ficin.mon_get(t2))
      cout << t2.x <<"  "<< t2.y << "  "<< t2.z <<"  "<< t2.b << endl;

   ficin.close() ;

   return 0;
}
```

Ouverture du fichier en mode

Il est nécessaire de fermer le fichier

Ouverture du fichier en mode

à l'écran

```
3.456 34.5658 0 1
3.456 34.5658 0 1
3.456 34.5658 0 1
```

Cette ligne n'est pas obligatoire
car le destructeur de l'objet

5.12. Les exceptions

L'auteur d'une bibliothèque peut détecter les endroits de son code où des erreurs sont susceptibles de se produire à l'exécution, mais il n'a souvent aucune idée de ce qu'il faut faire ensuite. L'utilisateur d'une bibliothèque peut au contraire savoir comment traiter ces erreurs, mais ne peut pas les détecter (sinon elles auraient été traitées dans le code de l'utilisateur et non dans la bibliothèque). La notion ***d'exception*** est offerte aux programmeurs C++ pour résoudre une partie de ces problèmes. L'idée fondamentale est qu'une fonction qui rencontre un problème impossible à traiter immédiatement ***lève*** (instruction ***throw***) une exception en espérant que le programme appelant pourra la traiter. Une fonction qui désire gérer ce genre de problème peut indiquer qu'elle est disposée à ***intercepter*** l'exception (instruction **try**, **catch**).

Reprenons l'exemple du chapitre 5.7 (la surcharge des opérateurs) en lui ajoutant la gestion des index de tableau hors des limites.

```
#include <assert.h>
#include <except.h>
#include <iostream.h>

typedef long tELT;

class Error {};
Error error;

class Erreur {
  public :
  int ligne;

  Erreur(int ligne) { this->ligne=ligne; }
  void Affiche() {
    cout << "Erreur dans le fichier "<<__FILE__<<" en ligne:"<<ligne<<endl;
  }
};

class tTableau {
  private :
      tELT *tab;
      int  card,nbmax;

  public :
     tTableau(int nbmax) { card=0;tab=new tELT[nbmax];this->nbmax=nbmax; }
     ~tTableau() { card=0; delete[] tab; nbmax=0; }
     tELT& operator[](int i) ;
};

tELT& tTableau::operator[](int i) {
   if (i>=0 && i<nbmax)
      return tab[i];
   throw (Erreur(__LINE__)) ;
}

int main() {

  tTableau t(5);

  try {
    t[0]=12;
    cout << "On utilise un indice incorrect..."<< endl;
    t[7]=23;
    return 0;
```

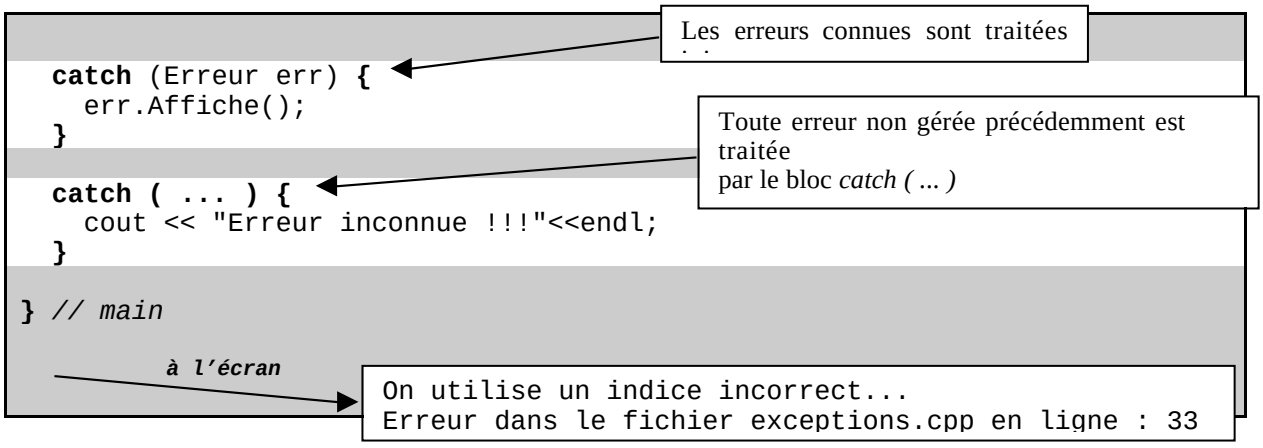
Contient __FILE__ et __LINE__

Classes pour gérer les

Tout le code source initial est inclus dans le bloc

Erreur levée car l'index est supérieur

```
}
```




```
  catch (Erreur err) {
    err.Affiche();
  }

  catch ( ... ) {
    cout << "Erreur inconnue !!!"<<endl;
  }

} // main
```

```
On utilise un indice incorrect...
Erreur dans le fichier exceptions.cpp en ligne : 33
```

INDEX