

Support de cours de Langage C

Version provisoire du 27 septembre 2004

LICENCE EEA-IE

Responsable pédagogique du cours : Eric Magarotto

CM : E. Magarotto

TD – TP : E. Pigeon – P. Dorleans – J.M. Janik - E. Magarotto

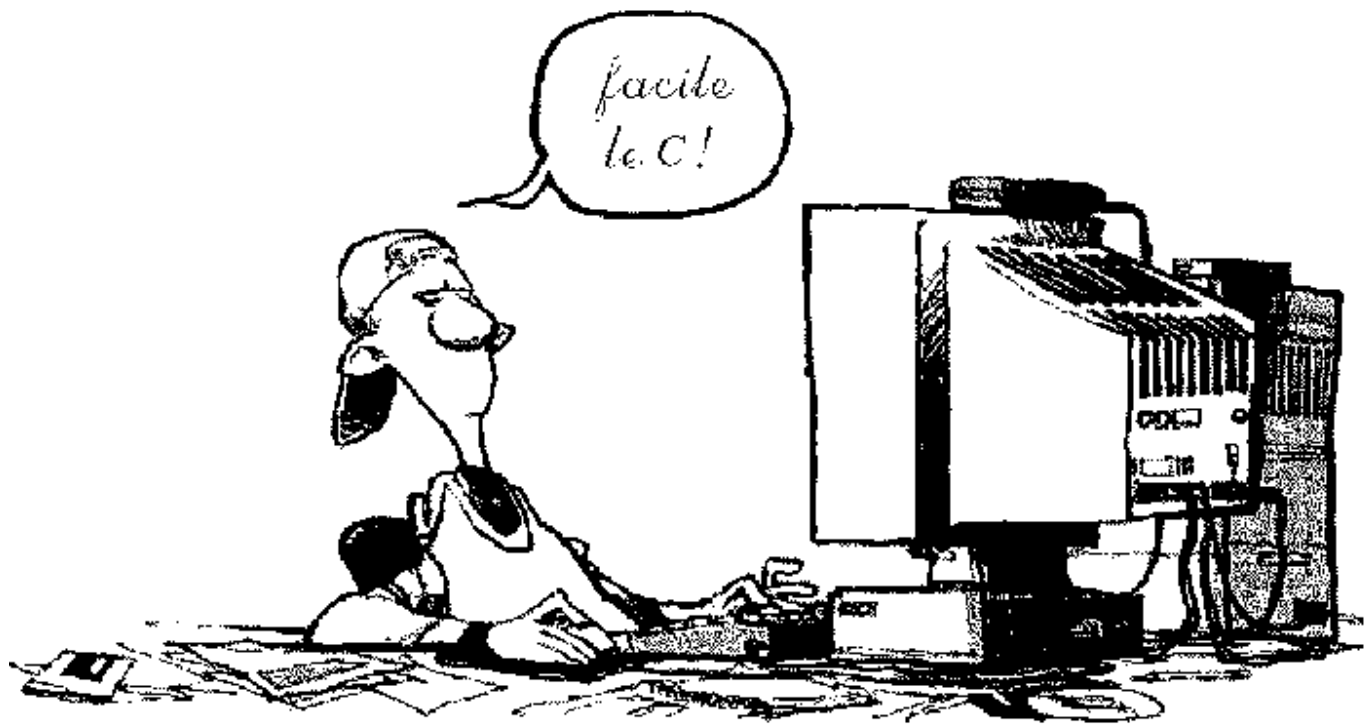
Durée approximative : 10 h de cours

Pour tout renseignements relatifs au cours :

e-mail : eric.magarotto@greyc.ismra.fr

web : www.greyc.ismra.fr/EquipeAuto/EricM/EMWelcome.html

tel : 02-31-45-27-09 (LAP-ISMRA)



I.	ELEMENTS DE BASE.....	2
I.1 - Commentaires.....		2
I.2 - Déclarations & types de variables (C : langage typé).....		2
I.3 - Initialisation de variables		3
I.4 - Assignations		4
I.5 - Entrées / Sorties Conversationnelles : printf et scanf.....		5
I.6 - Opérateurs arithmétiques condensés +=, -=, *=, /=, %=		6
I.7 - Opérateurs séquentiels : « ; » composition : « , ».....		6
I.8 - Opérateurs relationnels		6
I.9 - Opérateurs logiques		6
I.10 - Opérateurs de traitement du bit (uniquement sur <u>int</u> , <u>long</u> et <u>char</u>)		7
I.11 - Priorité des opérateurs		7
I.12 - Opérations d'incrément et de décrémentation		7
II.	STRUCTURES DE CONTROLE (BOUCLES).....	8
II.1 - Condition SI ... ALORS ... SINON ...(IF ... ELSE)		8
II.2 - Sélection (SWITCH).....		10
II.3 - Boucle tant que (WHILE).....		10
II.4 - Boucle faire tant que (DO WHILE).....		11
II.5 - Boucle pour (FOR)		11
II.6 - Ruptures de séquence.....		12
III.	LES TABLEAUX	14
III.1 - Déclaration de tableau (Statique – 1D = vecteur)		14
III.2 - Accès aux éléments du tableau.....		14
III.3 - Tableaux multidimensionnels (tableaux 2d).....		15
III.4 - Chaînes de caractères (string)		15
IV.	LES POINTEURS.....	18
IV.1 - Introduction		18
IV.2 - Opérations arithmétiques sur les pointeurs.....		19
IV.3 - Pointeurs et tableaux.....		20
IV.4 - tableau de pointeurs et pointeur de tableau		23
V.	ALLOCATION DYNAMIQUE	25
VI.	LES FONCTIONS	27
VI.1 - Introduction		27
VI.2 - Passage d'arguments par adresse		29
VI.3 - Evolution de la pile lors de l'appel à une fonction		31
VI.4 - Résumé (types).....		33
VI.5 - Portée, visibilité et classe d'allocation.....		33

VII. PRE-PROCESSEUR.....	36
VIII. LES STRUCTURES	38
VIII.1 - Déclaration d'une structure	38
VIII.2 - Utilisation d'une structure	38
VIII.3 - Utilisation des types synonymes : TYPEDEF	39
VIII.4 - Imbrication de structures.....	40
VIII.5 - Transmission d'une structure en argument d'une fonction.....	42
VIII.6 - Transmission d'une structure en valeur de retour d'une fonction.....	43
VIII.7 - Portée d'un structure.....	44
VIII.8 - Structure et listes chaînées	44
IX. GESTION DE FICHIERS.....	45
IX.1 - Introduction	45
IX.2 - Fichiers niveau haut.....	45
IX.3 - Action sur le pointeur de fichier	47
X. LISTES CHAINEES.....	48
X.1 - Introduction.....	48
X.2 - Création d'une liste chaînée	48
X.3 - Affichage de la liste.....	49
X.4 - Recherche d'un élément	50
X.5 - Rangement par ordre alphabétique	50
XI. POUR ALLER PLUS LOIN	51

GENERALITES

: présentation du C

- *créé en 1972, langage de haut niveau compilé (≠ interprété)*
- *progs C réutilisables, compilateurs respectent norme ANSI (88-90)*

: création d'un programme en C (3 phases)

- *édition d'un fichier source (nom.c) : éditeur de texte*
- *compilation (nom.obj) : traduction du programme source en langage machine, génère fichier objet (obj)*
- *édition de liens (linkage) : liaison entre obj, bibliothèque standard et fonctions. Groupe fichier obj et bin pour création d'un exécutable exe*
- *exécutable dépendant du système d'exploitation (recompiler le fichier source si changement d'OS)*

: un programme C se compose d'un ou plusieurs fichiers de la forme

< directives préprocesseur >

include (préprocesseur : préparation à la compilation), < et > (resp. ‘ ’)
indique de rechercher hors (resp. dans) répertoire actif

< définition de types >

< variables globales >

< fonctions >

structures des fonctions en norme ANSI :

< type > nom_de_fonction (<déclaration des arguments>)

{ <déclaration locales>

<instructions> }

programme principal (main)

ex : main(){Bloc d'instructions}

void main(void) ne renvoie pas de valeur

: principales caractéristiques

- *programmation modulaire multi-fichier*
- *jeu d'opérateurs très riche*
- *faible contrôle des types de données*
- *uniquement des fonctions*

I. ELEMENTS DE BASE

On distingue :

- les variables scalaires (entières, réelles et pointeurs)
- les variables structurées (tableaux, structures et unions)

I.1 - Commentaires

/ ceci est un commentaire */*

tout commentaire comporte une ouverture et une fermeture

I.2 - Déclarations & types de variables (C : langage typé)

Toutes les variables doivent être déclarées avant utilisation (2 paramètres)

<type> <identificateur>

type de base	signification	taille (octets) DOS - WINDOWS	valeurs limites (DOS)
char	caractère	1 – 1	-128 à 127
unsigned char	caractère	1 – 1	0 à 255
int	entier	2 – 4	-32768 à 32768
short	entier court	2 – 2	-32768 à 32768
long	entier long	4 – 4	± 2147483648
unsigned int	entier non signé	2 – 4	0 à 65535
unsigned long	entier long non signé	4 – 4	0 à 4294967295
float	réel	4 – 8	± 10 ⁻³⁷ à ±10 ⁺³⁸
double	réel double précision	8 – 8	± 10 ⁻³⁰⁷ à ±10 ⁺³⁰⁸
long double	réel double long	10 - 8	± 10 ⁻⁴⁹³² à ±10 ⁺⁴⁹³²

 : noms de variables différents des mots réservés (instructions):

break	char	case	continue	default	do
double	else	exit	extern	float	for
goto	if	int	long	register	return
short	sizeof	static	struct	switch	typedef
union	unsigned	while	const	enum	void

identificateur :

- 32 caractères max lettres ou chiffres mais pas de caractères accentués (codes ASCII étendus)
- différence entre majuscules et minuscules
- noms de variables réservés

type de base : - à l'exécution, une zone mémoire est réservée et contient la variable. La taille de la zone mémoire dépend du type

CHAR définit des variables de type caractère et ne stocke qu'un seul caractère à chaque fois. Un caractère peut être une lettre, un chiffre ou tout autre élément du code ASCII et peut être un caractère non imprimable dit de contrôle : $0 < \text{ASCII} < 31$. Un **CHAR** est codé sur 1 octet et ne peut prendre que des valeurs

- positives entre 0 et 255 s'il est non signé
- négatives entre -127 et +127 s'il est signé

INT : Selon la valeur que l'on est amené à affecter à un entier et la présence ou non du signe, on peut opter pour **INT**, **SHORT**, **UNSIGNED INT** ...

FLOAT : Selon la valeur que l'on est amené à affecter à un nombre à virgule flottante simple ou double précision, on peut opter pour **FLOAT**, **DOUBLE**, **LONG DOUBLE**

I.3 - Initialisation de variables

une valeur initiale peut être spécifiée dès la déclaration de la variable

```
main()  
{  
    int i, j=5;           /* seul j est initialisé */  
    int k=2*3;  
    int h=0x0A;  
    float x=1.5, y=2.3e-4; /* représentation scientifique */  
    char c='A';  
    char c='0x41';  
    long z=123L;  
}
```

différentes représentations pour les variables : les littéraux : entiers :

<i>codage</i>	<i>syntaxe</i>
décimal	123
octal	0777
hexadécimal	0xff ou 0Xff
long	65538 L
non signé	255 U

Si un nombre est supérieur à 65535, le programmeur peut demander que ce nombre soit codé sur un entier LONG par "L" (ex : 65538L).

 : réels :

<i>codage</i>	<i>syntaxe</i>
double précision	3.14159
long double (ANSI)	3.14159 L
simple précision (ANSI)	3.14159 F

 : caractères :

caractères imprimables (chiffres - lettres ...)

caractères non imprimables dit de contrôle

<i>caractères de contrôle</i>	<i>symbole</i>	<i>signification</i>
'\n'	LF	saut de ligne (Line Feed)
'\a'	BEL	bip sonore (Bell)
'\t'	HT	tabulation horizontale (Horizontal Tabulation)
'\b'	BS	suppression arrière (Back Space)
'\f'	FF	saut de page (Form Feed)
'\r'	CR	retour chariot (Carriage Return)

I.4 - Assignations

- Le signe = désigne l'opération d'assignation (affectation)
- Possibilité d'en enchaîner plusieurs : `i=j=2;` (2 dans j puis dans i)
- Assignation des constantes : `const int n=20;` empêche toute modification future de la variable n

 : variables liées à un bloc

main()

```

{ int i=3;
  {
    int i=2;          ⇒ affichage de i /* i=2 */
  }                  ⇒ affichage de i /* i=3 */
}

```

👉 : conversions automatiques lors d'assignments

Entre char et int

char → int : le char se retrouve dans l'octet le - significatif (LSB) de int

int → char : perte de l'octet le plus significatif (MSB) de l'entier

Entre int et long

int → long : conservation du bit de signe et donc la valeur de l'entier

long → int : résultat tronqué, perte des 2 octets les plus significatifs

Entre unsigned et long

unsigned → long: les 2 octets les plus significatifs du long passent à 0

Entre int et float

int → float : 10 → 10.0

float → int : perte partie décimale : 3.5 → 3 (idem en négatifs : -3.5 → -3)

Entre float et double

float → double : aucun problème

double → float : perte de précision

Exemple : `int n, long p, float x ;` évaluer `n*p+x` ;

Conversion de n en long, puis multiplication par p, résultat de type long, conversion en float pour être additionné à x → résultat de type float.

👉 : conversions forcées (opérateur cast)

`int n=10, int p=3;`

`(double) (n/p)` : calcul de `n/p` en `int` → conversion en `double` → `10/3=3`

`(double) n/p` : conversion `n` en `double` puis conversion (implicite) `p` en `double` puis division en `double` → `10/3=3.33`

I.5 - Entrées / Sorties Conversationnelles : printf et scanf

printf permet de réaliser un affichage formaté donc il faut spécifier

- le nom de la variable
- le format d'affichage avec caractères de contrôle
- le texte d'accompagnement

syntaxe : `printf("const char %format", nom_de_variable);`

exemple : `printf("Leur somme est : %5d\n", somme) ;`

⇒ 5 espaces réservés, justification à droite

`printf("Leur somme est : %.2f\n", somme) ;`

⇒ 2 chiffres après le point

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

<i>%d : entier en décimal</i>	<i>%x ou %X : hexadécimal</i>	<i>%o : octal</i>
<i>%f : float (réel)</i>	<i>%e ou %E : réel (scientifique)</i>	
<i>%u : entier non signé</i>	<i>%g (réel selon longueur valeur décimale)</i>	
<i>%lf : double</i>	<i>%ld : long</i>	<i>%Lf : long double</i>
<i>%c : caractère</i>	<i>%s : chaîne de caractères</i>	<i>%% affiche %</i>

Rque : Action sur le gabarit d’affichage et sur la précision : **voir TD**

Rque : la macro **putchar(c)** joue le même rôle que **printf("%c",c)**.
 Exécution plus rapide car pas d’appel au mécanisme de format. **puts(string)**

scanf est analogue à printf sauf lecture puis rangement. Pas de conversion automatique puisque l’argument transmis à **scanf** est l’adresse d’un emplacement mémoire (utilisation de &).

syntaxe : `scanf("const char %format", adresse_variable);`

exemple : `scanf("Leur somme est : %d\n", & somme) ;`

Rque : Action sur le gabarit max, espace dans le format, caractère invalide et arrêt prématuré : **voir TD**

Rque : macro **getchar()** joue le même rôle que **scanf("%c",&)**, mais plus rapide. **kbit** utilisé en TD (très utile pour boucle)

I.6 - Opérateurs arithmétiques condensés +=, -=, *=, /=, %=

Au lieu d’écrire : `i = i+20`, on peut écrire `i+=20`

de même : `i = i+k`, on peut écrire `i+=k`

`i = 3*i`, on peut écrire `i*=3`

I.7 - Opérateurs séquentiels : « ; » composition : « , »

« ; » plusieurs instructions à la suite, terminaison d’instruction

exemple : `int x = 5 ; x = x + 1;`

« ; » instruction vide. Attention boucles !!!

« , » exécution à la suite

exemple : `int x = 18, y;`

I.8 - Opérateurs relationnels

`==` et `!=` test d’égalité et d’inégalité

`<` et `>` inférieur à et supérieur à

`<=` et `>=` (inférieur ou égal à) et (supérieur ou égal à)

I.9 - Opérateurs logiques

Le langage C dispose de 3 opérateurs logiques classiques

`&&` et `||` : ET et OU logiques (ex `(x=1)&&(y=10) → vrai` et `vrai=vrai`)

! : non

I.10 - Opérateurs de traitement du bit (uniquement sur int, long et char)

& ET (niveau binaire) (ex (x=1)&(y=10)) → 0001 & 1010=0000)

| OU inclusif (niveau binaire)

^ OU exclusif (niveau binaire)

>> et << décalages à droite/gauche (division/multiplication par 2)

~ complément à un

I.11 - Priorité des opérateurs

CATEGORIE	OPERATEURS	ASSOCIATIVITE
Référence	() [] -> .	→
Unaire	+ - ++ -- ! ~ * & (cast) sizeof	←
Arithmétique	* / % + -	→
Décalage	<< >>	→
Relationnels	< <= > >= == !=	→
Manipulation de bits	&, ^,	→
Logique	&& 	→
Conditionnel	? :	→
Affectation	= += -= *= /= %= &= ^= = <<= >>=	←
séquentiel	,	→

Exemples : $a+b*c \Leftrightarrow a+(b*c)$
 $-a/-b+c \Leftrightarrow ((-a)/(-b))+c$
 $a<b==c<d \Leftrightarrow (a<b)==(c<d)$

I.12 - Opérations d'incrément et de décrémentation $i = i+1$ peut s'écrire $i++$ (postincrément)

incrément après utilisation de la valeur

ou $++i$ (préincrément)

incrément avant utilisation de la valeur

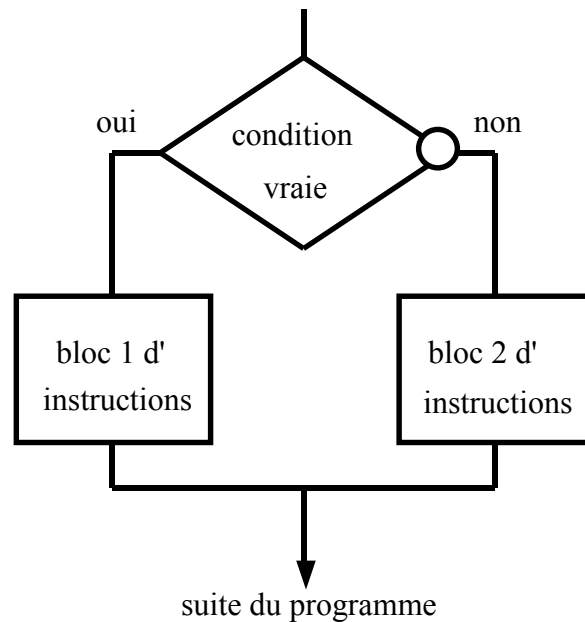
 $i = i-1$ peut s'écrire $i--$ (postdécrément)ou $--i$ (prédécrément)

II. STRUCTURES DE CONTROLE (BOUCLES)

II.1 - Condition SI ... ALORS ... SINON ...(IF ... ELSE)

Instruction: **si** (*expression conditionnelle vraie*)
 alors {BLOC 1 D'INSTRUCTIONS}
 sinon {BLOC 2 D'INSTRUCTIONS}

Organigramme:

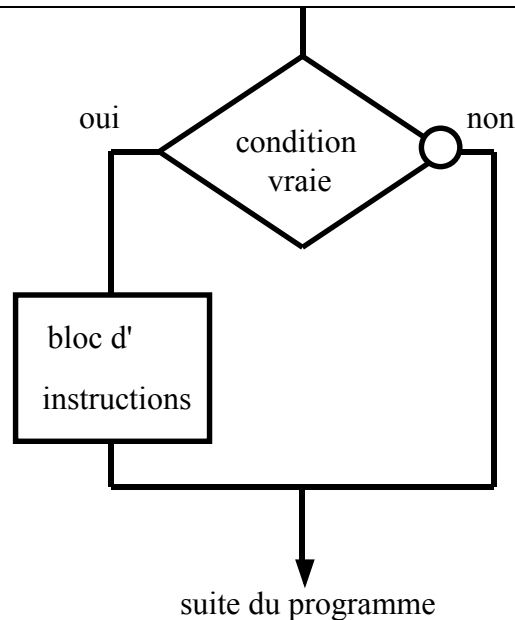


Syntaxe :

```
if (expression)
{
    .....; /* bloc 1 d'instructions */
    .....;
}
else
{
    .....; /* bloc 2 d'instructions */
    .....;
}
```

*Dans le cas d'imbrication de plusieurs **if**, la clause **else** se rapporte au **if** le plus proche.*

Le bloc "sinon" est optionnel: **si** (*expression vraie*)
 alors {BLOC D'INSTRUCTIONS}



Rq: les `{ }` ne sont pas nécessaires lorsque les blocs ne comportent qu'une seule instruction.

👉 : Rappels des opérateurs logiques (pour test de condition)

- *égalité:* `if (a==b)` « si a égal b »
- *non égalité:* `if (a!=b)` « si a différent de b »
- *relation d'ordre:* `if (a<b)` `if (a<=b)` `if (a>b)` `if (a>=b)`
- *ET LOGIQUE:* `if ((expression1) && (expression2))`
« si l'expression1 ET l'expression2 sont vraies »
- *OU LOGIQUE* `if ((expression1) || (expression2))`
« si l'expression1 OU l'expression2 est vraie »
- *NON LOGIQUE* `if (!(expression1))`
« si l'expression1 est fausse »

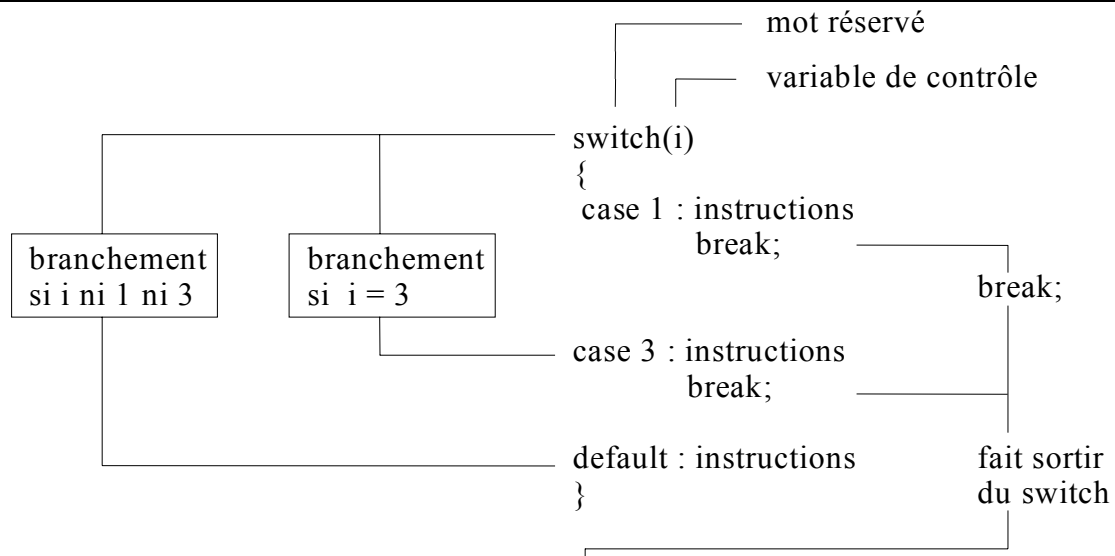
Toutes les combinaisons sont possibles entre ces tests.

👉 : incrémentations et décrémentations dans une condition

```

if ( --i==5) /* i est d'abord décrémenté et le test i==5 est effectué */
i=4;
if (i++==5) printf("EGALITE (%d) ", i);
else printf("INEGALITE (%d) ", i);
/* le test i==5 est d'abord effectué, i est incrémenté avant même d'exécuter
printf("INEGALITE (%d)",i); */
  
```

II.2 - Sélection (SWITCH)

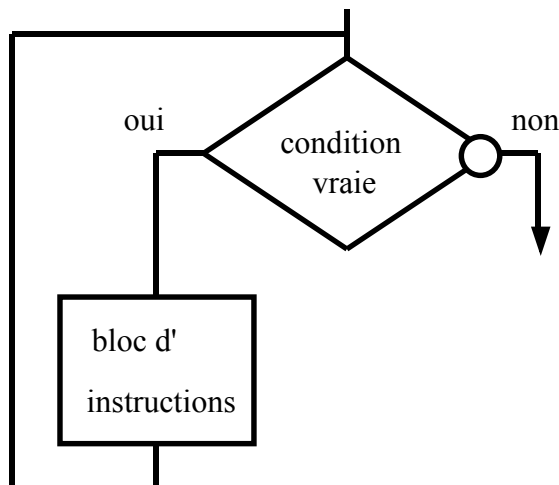


C'est une instruction de choix multiples qui permet d'effectuer un aiguillage direct vers les instructions en fonction d'un cas (branchement conditionnel). Si la comparaison entre la variable de contrôle (de type int, short, char ou long) et la valeur des constante de chaque cas réussit alors l'instruction du case est effectuée jusqu'à la première instruction d'arrêt rencontrée (break). Défaut est une étiquette pour le cas ou aucune valeur n'est satisfaisante.

II.3 - Boucle tant que (WHILE)

Instruction: ***tant que (expression vraie)***
faire{BLOC D'INSTRUCTIONS}

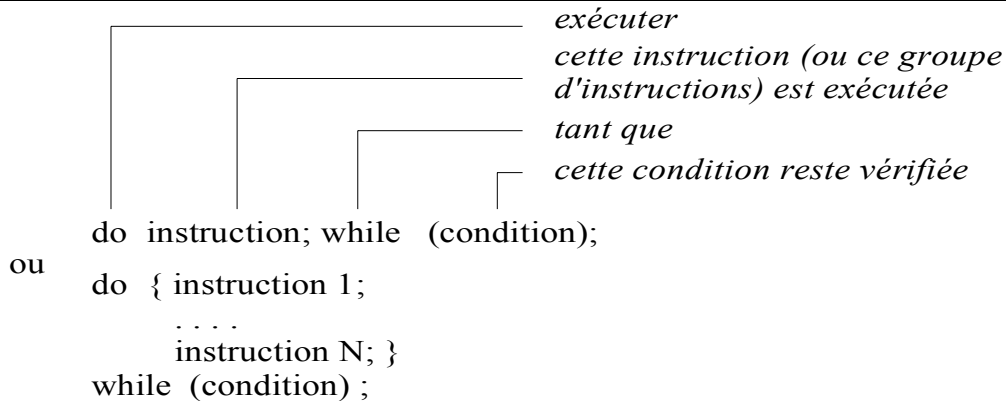
Organigramme:



Syntaxe : `while (expression)`
`{`
`.....;` `/* bloc d'instructions */`
`}`

La condition est examinée avant. La boucle peut ne jamais être exécutée. On peut rencontrer la construction suivante: **while (expression);** terminée par un ; et sans la présence du bloc d'instructions. Cette construction signifie: « **tant que l'expression est vraie attendre** ».

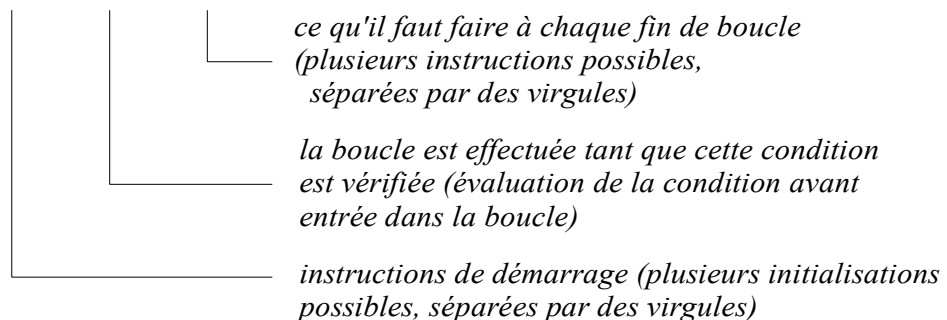
II.4 - Boucle faire tant que (DO WHILE)



La différence avec le **WHILE** réside dans le fait que la boucle **DO WHILE** est exécutée au moins une fois.

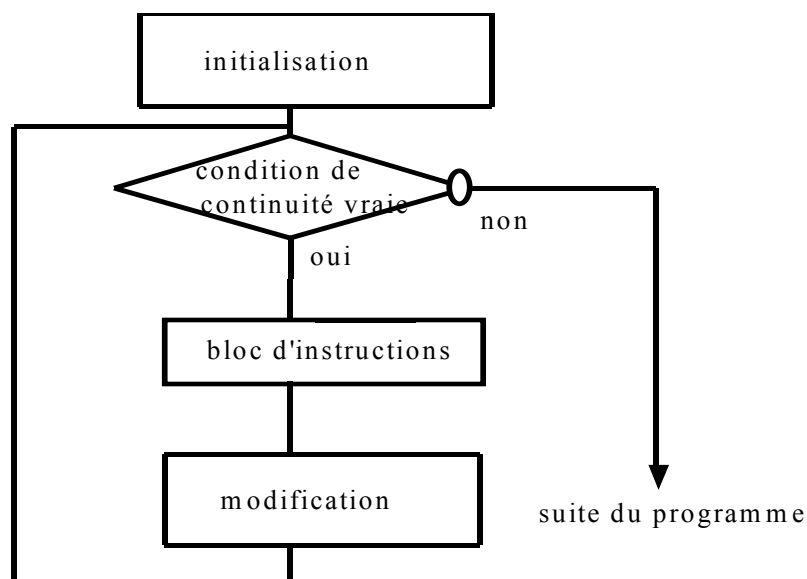
II.5 - Boucle pour (FOR)

for (expr 1; expr 2; expr 3) instructions;



Instruction :

Organigramme:



II.6 - Ruptures de séquence

break : - interruption de l'exécution d'une boucle FOR, WHILE, DO..
 - pour un FOR, la sortie s'effectue sans exécuter les instructions de fin de boucle
 - fait également sortir d'un SWITCH
 - ne fait pas sortir d'une structure conditionnelle IF

continue : - permet de passer à l'itération suivante d'une boucle
break et continue ne s'appliquent qu'à la boucle concernée

exit (n) : sortie d'un programme et retour au système d'exploitation.
La valeur de retour est n.

return (n) : sortie d'un sous-programme avec la valeur n

goto (n) : *branchement en un emplacement quelconque (à proscrire pour une « bonne programmation »).*

Exemple 1 : résoudre $ax^2 + bx + c = 0$.

[illegible]

```

{
    delta = b*b-4*a*c;
    printf("DELTA= %f\n", delta);
    if(delta<0)printf("DELTA NEGATIF PAS DE SOLUTION\n");cc
    else
    {
        if(delta==0)printf("DELTA NUL, UNE SOLUTION X= %f\n", -b/2/a);
    }
    else{
        x1= (-b+sqrt(delta))/2/a;
        x2= (-b-sqrt(delta))/2/a;
        printf("DELTA POSITIF DEUX SOLUTIONS\n");
        printf("X1= %f   X2= %f\n", x1, x2);
    }
}

/* calculs termines */
printf("\n\nPOUR CONTINUER FRAPPER UNE TOUCHE");
getch();
}

```

Exemple 2 : Saisir une suite de caractères, compter et afficher le nombre de lettres e et d'espaces. Utiliser les propriétés du tampon.

```

#include <stdio.h>
#include <conio.h>
void main()
{
    char c, compt_espace= 0, compt_e= 0;
    printf("ENTRER UNE PHRASE:\n");/* l'utilisateur saisit la totalite de sa phrase */

    while((c=getchar())!='\n')      /* lors du 1er passage, getch char ne prend */
                                    /* en compte que le 1er caractere */
    {                                /* les autres sont ranges dans le tampon */
        if(c=='e')compt_e++;        /* et recuperes par getch char lors */
                                    /* des autres passages */
        if(c==' ')compt_espace++;
    }

    printf("NOMBRE DE e: %d\n", compt_e);
    printf("NOMBRE D'ESPACE: %d\n", compt_espace);
    printf("POUR SORTIR FRAPPER UNE TOUCHE ");
    getch();
}

```


III. LES TABLEAUX

III.1 - Déclaration de tableau (Statique – 1D = vecteur)

<Type (int,char,etc...> <Nom> <[nb éléments]>

 : utiliser un mnémonique pour la taille du tableau

```
#define Nbval 20
int tab[Nbval];      /*réservation mémoire pour 20*sizeof(int) = 40 octets */
float c[Nbval];      /*réservation mémoire pour 20*sizeof(float) = 80 octets */
char str[Nbval];     /*réservation mémoire pour 20*sizeof(char) = 20 octets */
```

III.2 - Accès aux éléments du tableau

- Accès à chaque élément du tableau par l'intermédiaire d'un indice
- L'indice 0 donne accès au premier élément
- L'indice **Nbval-1** donne accès au dernier élément
- Il peut être une valeur, une variable ou une expression arithmétique

```
int i = 4;          tab[i] = 2   tab[i*2-1] = 3;
```

 : aucun contrôle de dépassement de bornes : `tab[Nbval]` → ?

 : tous les éléments doivent être de même type

 : aucune opération possible sur la totalité du tableau

`tab1=tab2` ne recopie pas le tableau `tab2` dans le tableau `tab1`
`tab1==tab2` ne compare pas le tableau `tab2` et le tableau `tab1`

 : initialisation des tableaux

déclaration tableau = {v1, v2,} : liste de valeurs constantes qui initialise le tableau

```
int tab[3] = {0, 1, 2}      /* tableau de 3 entiers */
int tab[] = {0, 1, 2}
int tab[5] = {0, 1, 2}      /* tableau de 5 entiers dont les 3
premiers sont initialisés. Les 2 autres sont initialisés à 0 si le tableau est déclaré
en global.*/
```

👉 : taille du tableau

exemple pour obtenir la dimension du tableau en nombre d'éléments :

```
int tab[10] = {1, 2, 3}
for (i=0; i<sizeof tab / sizeof(int); i++) printf("%d\n ", tab[i]);
sizeof tab / sizeof tab[0] donne le même résultat
```

👉 : adresse d'implantation d'un tableau

- `printf("%p\n", tab);` affiche l'adresse d'implantation en mémoire
- `&tab[0]` ; donne l'adresse de départ du tableau
- On peut écrire `&tab(Nbval)` pour des tests de bornes (boucles) mais ne pas accéder à sa zone mémoire.
- On peut additionner (ou soustraire un entier) aux adresses :
Exemple : `&tab[3]+2` équivalent à `&tab[5]`.

III.3 - Tableaux multidimensionnels (tableaux 2d)

- Descriptif : tableau de tableau → Matrice
- Syntaxe : type identificateur [dim1] [dim2] ... [dimn]
- peu usité car préférence pour les tableaux de pointeurs

Exemple : `int tab[2][3] = {{1, 2, 3}, {4, 5, 6}} ;`

*Le compilateur va réserver 2*3 places mémoire pour ce tableau.*

L'implantation en mémoire se fait de la manière suivante :

t[0][0]
t[0][1]
t[0][2]
t[1][0]
t[1][1]
t[1][2]

Les colonnes sont placées les une à la suite des autres, ce qui peut permettre d'identifier les éléments avec un seul indice si l'on utilise les adresses.

III.4 - Chaînes de caractères (string)

chaîne de caractère = tableau de caractères ou d'octets terminé par '\0'.

La taille d'un tableau de caractère doit être suffisante pour contenir la sentinelle (le '\0').

exemple : `ch[10]` ne peut que contenir au plus que 9 caractères.

Saisie complète de la chaîne :

Exemple : `char ch[20] = 'bonjour';` (8 caractères plus les autres à 0)

 : comparaison de deux chaînes de caractères

Les opérateurs = et == ne permettent pas de copier une chaîne dans une autre ni de comparer le contenu de deux chaînes. Il faut impérativement passer par une fonction de traitement de chaîne de caractères.

strcmp(chaîne1, chaîne2)

retourne 0 si les 2 chaînes contiennent les mêmes caractères y compris '\0'

retourne un entier négatif si chaîne1 < chaîne2

retourne un entier positif si chaîne1 > chaîne2

strncmp(chaîne1, chaîne2, lgmax) : limitation de la comparaison sur un nombre « lgmax » de caractères.

stricmp(chaîne1, chaîne2) : pas de différence entre majuscules et minuscules.

strncmp(chaîne1, chaîne2, lgmax) : idem plus contrôle longueur max.

 : copie de chaînes de caractères

strcpy (cible, source)

copie le contenu d'une chaîne (source) dans une autre (cible). Le second argument peut être une variable de type chaîne de caractères ou une constante de ce type.

 : détermination de la taille d'une chaîne de caractères

strlen

Cette fonction retourne le nombre de caractères présents dans une chaîne sans la sentinelle.

 : lecture d'une chaîne de caractères

gets(buf)

*Tous les caractères introduits au clavier sont copiés dans **buf** jusqu'au retour chariot. Le caractère de retour chariot est remplacé par la sentinelle. (préférable à **scanf("%s",&buf)** pour qui l'espace est un caractère invalide)*

 : affichage d'une chaîne de caractères

puts(buf)

Cette fonction non formatée **puts** ajoute automatiquement le caractère **\n** de saut de ligne en fin d'affichage.

C'est équivalent à **printf("%s\n",buf)**.

 : concaténation de 2 chaînes de caractères

strcat(destination,source)

Cette fonction recopie la seconde chaîne (source) à la suite de la première (destination) après avoir effacé le caractère **'\0'**.

Il est donc nécessaire que l'emplacement réservé pour la première chaîne soit suffisant pour y recevoir la partie à lui concaténer.

strcat() est intéressante pour les noms de fichier avec des extensions différentes.

Strncat(destination,source,long) permet de contrôler la longueur du nombre de caractère de la chaîne d'arrivée

 : duplication de 2 chaînes de caractères

char strdup(source)

La duplication crée la chaîne qui va recevoir la copie et va prévoir la place mémoire nécessaire.

 : recherche d'une chaîne

strstr(chaîne1,chaîne2)

Cette fonction entraîne la recherche de la chaîne chaîne2 dans la chaîne chaîne1.

IV. LES POINTEURS

IV.1 - Introduction

Pointeur = variable qui contient l'adresse d'un autre objet (variable ou fonction). Il est possible d'accéder à une variable par :

- son nom ou identificateur
- son adresse en utilisant un pointeur

Il est conseillé mais pas obligatoire de faire commencer chaque identificateur de pointeur par « ptr_ » pour les distinguer des autres variables.

: déclaration d'un pointeur

ex 1 : `int a; a=98;` *initialisation de la variable a*

adresse : `&a=0x50`

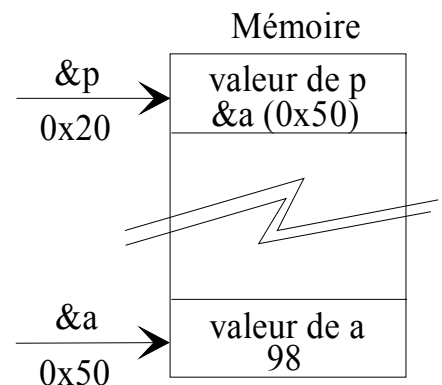
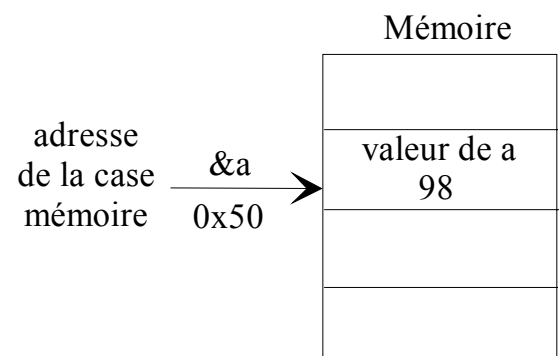
valeur : 98

ex 2 : `int a, *ptr_p;`

`a=98;` a : variable allouée par le système dont l'adresse est 0x50 par exemple

`ptr_p=&a;` p : pointeur sur un entier dont l'adresse est 0x20

- a désigne le contenu de a
- &a désigne l'adresse de a
- ptr_p désigne l'adresse de a
- *ptr_p désigne le contenu de a
- &ptr_p désigne l'adresse du pointeur ptr_p
- *a est illégal (a n'est pas un pointeur)



: affichage du contenu d'un pointeur

affichage en hexadécimal par le format %p ou %x de printf.

Exemple :

```
printf("VOICI p: %d\n",i);
```

```
printf("VOICI SON ADRESSE EN HEXADECIMAL: %p\n",&ptr_p);
```

 : initialisation d'un pointeur

à la définition du pointeur
à l'exécution par une affectation

IV.2 - Opérations arithmétiques sur les pointeurs : incrémentation/décrémentation d'un pointeur

ptr_p++ : adresse avant l'incrément (valeur actuelle de p)
 $++ptr_p$: adresse après l'incrément
 ptr_p-- : adresse avant la décrémentation (valeur actuelle de p)
 $--ptr_p$: adresse après la décrémentation

L'unité d'incrément et de décrémentation d'un pointeur est toujours la taille de la variable pointée (intérêt pour les tableaux). Si ptr_p est un pointeur sur un entier, ptr_p++ entraîne l'incrément de p de la taille d'un entier soit 4 octets.

 : comparaison de 2 pointeurs

un pointeur est défini à la fois par une adresse mémoire et un type. On ne peut donc comparer que des pointeurs de même type.

 : soustraction de 2 pointeurs

La différence de 2 pointeurs (de même type) fournit le nombre d'éléments du type en question situés entre les 2 adresses correspondantes.

 : pointeur nul

Pour certains besoins, il peut être utile de vouloir comparer un pointeur avec la valeur nulle. On utilisera alors la constante `NULL` prédéfinie dans `<stdio.h>`.

Exemple : `int*ptr_n ; if (ptr_n==NULL)...`

 : conversion de pointeurs

Il n'existe aucune conversion implicite d'un type dans un autre pour les pointeurs → opérateur `cast` mais danger !

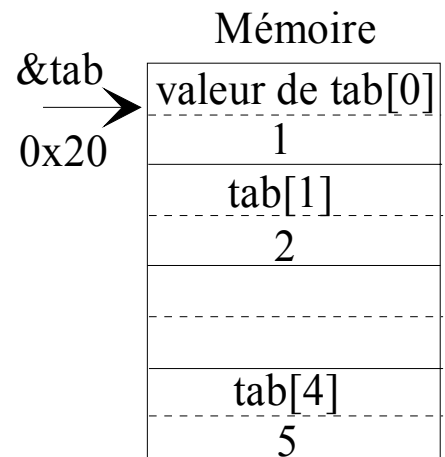
Attention aux contraintes d'alignement : un entier (4 octets) sera toujours placé à une adresse paire tandis qu'un `char` (1 octet) pourra être placé à n'importe quelle adresse → PB à la conversion d'un `char*` en `int*` !!!

IV.3 - Pointeurs et tableaux

Les pointeurs sont particulièrement utilisés lors d'accès aux tableaux.

adresse d'un tableau = nom du tableau = adresse du 1er élément

```
main()
{
  int tab[10]={1,2,3,4,5}, *p, x;
  p=&tab[0]; équivalent à p=tab :
  x=*tab;    valeur du 1er élément, x=1
  x=*(tab+0); équivalent à tab[0] càd x=1
  x=*(tab+i); équivalent à tab[i]
               tab+i équivalent à &tab[i]
}
```



 : un tableau peut être déclaré de 2 façons

```
type identificateur [dimension]    /* vrai tableau */
type *identificateur                /* pointeur seul */
```

Remarque : seule, la 1ère méthode assure l'allocation mémoire du tableau

exemple : affichage d'un tableau de réels

- indices entiers

```
#define dim 15
main()
{ int y; double x[dim]={1,2,3,4,5,6,7,8,9,10};
  Cls(); for (y=0; y<dim; y++)
    { printf("%lf \n", x[y]);
      /* ou bien */
      printf("%lf \n", *(x+y)); }
}
```

- indices pointeurs

```
#define dim 15
main()
{ double x[dim]={1,2,3,4,5,6,7,8,9,10}, *ptr_p=NULL;
  Cls();
  for (ptr_p=x; ptr_p<x+dim; ptr_p++)
    printf("%lf \n", *ptr_p);
  for (ptr_p=x; ptr_p<x+dim; ptr_p++)
    printf("%lf \n", *ptr_p++);
}
```

Step0 : init**Step1 : 4^{ème} boucle**

Variables		
Globals		
main		
x	0069F8C4	double [15]
x[0]	1.0000000000000000	double
x[1]	2.0000000000000000	double
x[2]	3.0000000000000000	double
x[3]	4.0000000000000000	double
x[4]	5.0000000000000000	double
x[5]	6.0000000000000000	double
x[6]	7.0000000000000000	double
x[7]	8.0000000000000000	double
x[8]	9.0000000000000000	double
x[9]	10.0000000000000000	double
x[10]	0.0000000000000000	double
x[11]	0.0000000000000000	double
x[12]	0.0000000000000000	double
x[13]	0.0000000000000000	double
x[14]	0.0000000000000000	double
ptr_p	NULL	double *

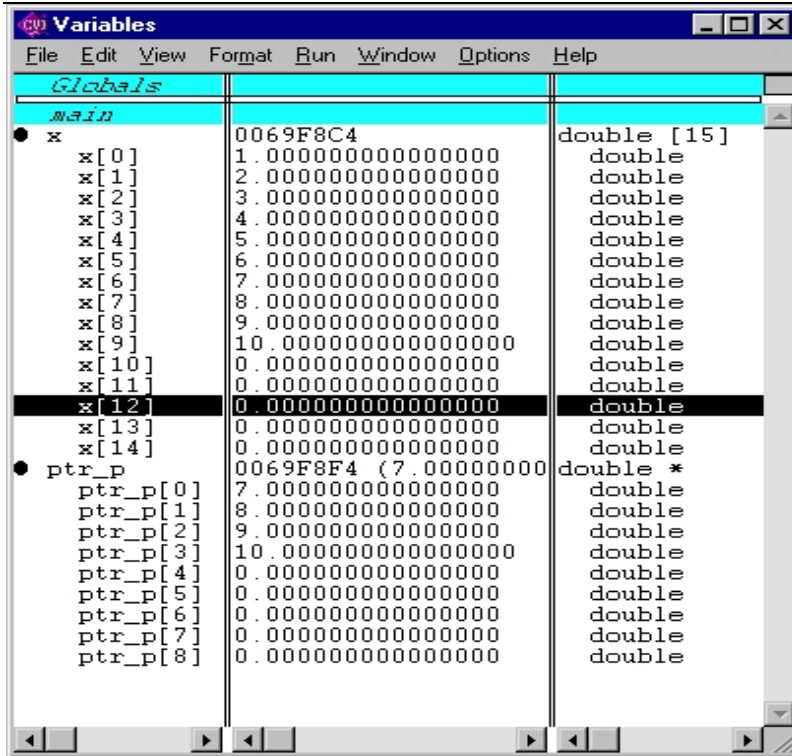
Variables		
Globals		
main		
x	0069F8C4	double [15]
x[0]	1.0000000000000000	double
x[1]	2.0000000000000000	double
x[2]	3.0000000000000000	double
x[3]	4.0000000000000000	double
x[4]	5.0000000000000000	double
x[5]	6.0000000000000000	double
x[6]	7.0000000000000000	double
x[7]	8.0000000000000000	double
x[8]	9.0000000000000000	double
x[9]	10.0000000000000000	double
x[10]	0.0000000000000000	double
x[11]	0.0000000000000000	double
x[12]	0.0000000000000000	double
x[13]	0.0000000000000000	double
x[14]	0.0000000000000000	double
ptr_p	0069F8DC (4.0000000000000000)	double *

Step2 : 12^{ème} boucle**Step3 : init version2**

Variables		
Globals		
main		
x	0069F8C4	double [15]
x[0]	1.0000000000000000	double
x[1]	2.0000000000000000	double
x[2]	3.0000000000000000	double
x[3]	4.0000000000000000	double
x[4]	5.0000000000000000	double
x[5]	6.0000000000000000	double
x[6]	7.0000000000000000	double
x[7]	8.0000000000000000	double
x[8]	9.0000000000000000	double
x[9]	10.0000000000000000	double
x[10]	0.0000000000000000	double
x[11]	0.0000000000000000	double
x[12]	0.0000000000000000	double
x[13]	0.0000000000000000	double
x[14]	0.0000000000000000	double
ptr_p	0069F924 (0.0000000000000000)	double *
ptr_p[0]	0.0000000000000000	double
ptr_p[1]	0.0000000000000000	double
ptr_p[2]	0.0000000000000000	double

Variables		
Globals		
main		
x	0069F8C4	double [15]
x[0]	1.0000000000000000	double
x[1]	2.0000000000000000	double
x[2]	3.0000000000000000	double
x[3]	4.0000000000000000	double
x[4]	5.0000000000000000	double
x[5]	6.0000000000000000	double
x[6]	7.0000000000000000	double
x[7]	8.0000000000000000	double
x[8]	9.0000000000000000	double
x[9]	10.0000000000000000	double
x[10]	0.0000000000000000	double
x[11]	0.0000000000000000	double
x[12]	0.0000000000000000	double
x[13]	0.0000000000000000	double
x[14]	0.0000000000000000	double
ptr_p	0069F8C4 (1.0000000000000000)	double *

Step4 : 7^{ème} boucle version2



 : pointeur, adresse de pointeur et variable pointée

p désigne le contenu du pointeur *p*

&p désigne l'adresse du pointeur *p*

**p* désigne le contenu de la variable pointée par *p*

 : tableau multidimensionnel

int tab [dim1],[dim2];

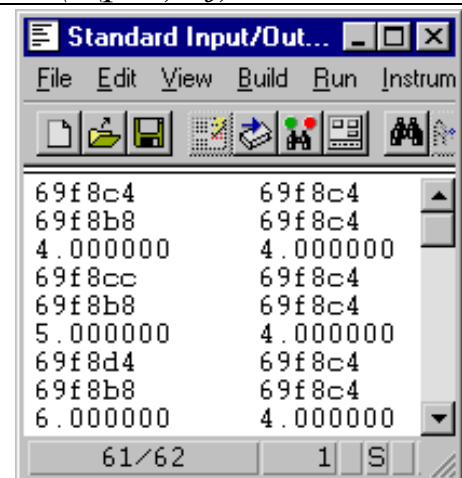
tab : adresse du tableau $\Rightarrow p$

tab[i] : adresse de la (i+1)ème ligne $\Rightarrow *(p+i)$

attention : $\neq$ de $(*p)+i$ qui rajoute *i* au contenu pointé par *p*

tab[i][j] : élément *i,j* $\Rightarrow *((*(p+i)+j)$

```
double x[dim]={1,2,3,4,5,6,7,8,9,10};
double *ptr_p=NULL;
for (ptr_p=x;ptr_p<x+dim;ptr_p++)
{
    printf("%x \t\t %x\n", ptr_p,x);
    printf("%x \t\t %x\n", &ptr_p,&x);
    printf("%lf \t %lf\n", *ptr_p+3,x[3]);
}
```



IV.4 - tableau de pointeurs et pointeur de tableau

IV.4.1 tableau de pointeurs :

*type *nom[taille] :*

exemple :

```
int tab[3]={1,5};
int *ptr_p[4] = {tab, tab+1, tab+2, tab+3};
```

Globals		
z	0	int
● tab	00DE0DF0	int [3]
tab[0]	1	int
tab[1]	5	int
tab[2]	0	int
● ptr_p	00DE0DFC	int *[4]
● ptr_p[0]	00DE0DF0 (1)	int *
ptr_p[0][0]	1	int
ptr_p[0][1]	5	int
ptr_p[0][2]	0	int
● ptr_p[1]	00DE0DF4 (5)	int *
ptr_p[1][0]	5	int
ptr_p[1][1]	0	int
● ptr_p[2]	00DE0DF8 (0)	int *
ptr_p[2][0]	0	int
ptr_p[3]	Over Array Bound	int *
main		

size of tab : 12 octets

size of tab[0] : 4 octets

size of ptr_p : 16 octets

size of ptr_p[0] : 4 octets, pointe sur 12 octets (3 éléments)

size of ptr_p[0][0] : 4 octets

IV.4.2 **pointeur de tableau :**

*type (*nom)[taille] :*

exemple :

```
int tab[1][2]={5,2} ;
int (*ptr_p[3]) [4] = {tab, tab+1, tab+2} ;
```

Variable	Address	Type
z	0	int
tab	00DE0180	int [1][2]
tab[0]	00DE0180	int [2]
tab[0][0]	5	int
tab[0][1]	2	int
ptr_p	00DE0188	int (*[3])[4]
ptr_p[0]	00DE0180	int (*)[4]
(*ptr_p[0])[0]	5	int
(*ptr_p[0])[1]	2	int
(*ptr_p[0])[2]	14549376	int
(*ptr_p[0])[3]	14549384	int
ptr_p[1]	00DE0188	int (*)[4]
(*ptr_p[1])[0]	14549376	int
(*ptr_p[1])[1]	14549384	int
(*ptr_p[1])[2]	14549392	int
(*ptr_p[1])[3]	0	int
ptr_p[2]	00DE0190	int (*)[4]
(*ptr_p[2])[0]	14549392	int
(*ptr_p[2])[1]	0	int
(*ptr_p[2])[2]	1090519040	int
(*ptr_p[2])[3]	1986201677	int

size of tab : 8 octets

size of tab[0] : 8 octets

size of tab[0][0] : 4 octets

size of ptr_p : 12 octets

size of ptr_p[0] : 4 octets, pointe sur 16 octets (4 éléments)

*size of *ptr_p[0]) [0]: 4 octets*

V. ALLOCATION DYNAMIQUE

La création de données dynamiques ou leur libération s'effectuent lors de l'exécution d'un programme. Intérêt : allocation ou libération en fonction des besoins.

: allocation d'un élément de taille définie

*Malloc réserve une zone mémoire et renvoie un pointeur de type indéfini (pointeur générique ou universel **void***).*

- Mettre un « cast » si on veut forcer le type.
- Retourne pointeur NULL si l'allocation n'a pas eu lieu.
- Syntaxe : **(void*)malloc(size)** alloue "size" octets et retourne le pointeur (adresse) du 1er octet alloué.

: allocation de plusieurs éléments consécutifs

***(void*)calloc (nb, size)** réalise l'allocation de "nb" éléments de taille "size"*

- Retourne l'adresse du 1er octet alloué
- Initialise la mémoire à 0..
- Retourne un pointeur NULL si l'allocation n'a pas eu lieu.

remarque : $p=calloc(152, sizeof(int))$ équivaut à $p=malloc(152 sizeof(int))$*

: changement de taille d'un tableau alloué dynamiquement

***realloc (pointeur sur le bloc mémoire à modifier, size)** permet de réallouer de la mémoire en conservant les valeurs initiales.*

Le nouveau bloc peut être plus grand ou plus petit (attention à la perte de données) que l'original.

: libération de l'espace mémoire

L'un des intérêts essentiels de la gestion dynamique est de pouvoir récupérer des emplacements dont on n'a plus besoin.

***free(pointeur sur le 1^{er} emplacement)** libère une zone mémoire préalablement alloué. Attention : vous devez posséder une trace du pointeur !*

: copie de blocs de mémoire

***memcpy(pointeur cible, pointeur source, nb octets)** recopie les nb premiers octets de pointeur source sur les nb premiers octets de pointeur cible.*

Possible uniquement si les deux blocs ne se recoupent pas !

Exemple (allocation dynamique)

```

#include <ansi_c.h> #include <utility.h>
#include <stdio.h> #include <stdlib.h>
/* Déclaration d'un pointeur d'entier */
int *ptr_p1, *ptr_p2;
int taille = 3;
main()
{
  Cls();
  /* Réservation mémoire pour un entier */
  ptr_p1 = (int *) malloc(taille*sizeof(int));
  ptr_p2 = (int *) calloc(taille,sizeof(int));
  /* Test si l'allocation a eu lieu */
  if (ptr_p1==NULL || ptr_p2==NULL) printf("Pas assez de
  mémoire\n");
  else
  {
    /* Initialisation de valeur pointé par ptr_int */
    *ptr_p1=6;
    *(ptr_p1+1)=15;
    /* *ptr_p2=*ptr_p1; */
    memcpy(ptr_p2,ptr_p1,taille*sizeof(int));
    /* Affichage de la valeur pointé par ptr_int */
    printf("La valeur pointé par ptr_p2 est %d\n",*ptr_p2);
    /* Libération de la zone mémoire pointé par ptr_int */
    free(ptr_p1);
    free(ptr_p2);
  }
}

```

①
②
③
④

VI. LES FONCTIONS

VI.1 - Introduction

Les fonctions permettent de décomposer un programme en entités plus limitées et donc d'en simplifier à la fois la réalisation et la mise au point.

Une fonction peut :

- *se trouver dans le même fichier que main() ou dans un autre*
- *être appelée à partir de main(), d'une autre fonction ou d'elle même*
- *admettre ou non des arguments (paramètres d'entrée)*
- *retourner une valeur ou non (paramètres de sortie)*
- *posséder ses propres variables*

Mais on ne peut pas définir une fonction à l'intérieur d'une autre fonction (toutes les fonctions sont "au même niveau").

*Chaque fonction doit avoir un **prototype** et une **déclaration***

Remarque : main() n'est autre qu'une fonction qui joue un rôle particulier (rigoureusement : `void main(void)`). Les variables déclarées au début de la fonction principale main sont locales à main !!!

Exemple de programme faisant appel à une fonction :

```
int plus2 (int ); /* prototype (présence du point virgule !)*/  
main( )          /* début de la fonction main */  
{ int i,j;       /* variables appartenant uniquement à main() */  
  i=2;  
  j=plus2(i);     /* appel de la fonction appelée plus2 */  
  j=plus2(4);     /* nouvel appel de la fonction */  
}  
int plus2(int nb) /* déclaration de la fonction et type d'argument */  
{return(nb+2);}  /* valeur entière retournée */
```

: prototypes de fonctions

Un prototype donne la règle d'usage de la fonction : nombre et type d'arguments ainsi que le type de la valeur retournée. En norme ANSI, un prototype s'écrit :

type identificateur (déclaration des paramètres)

Un prototype permet au compilateur de vérifier le type et le nombre des arguments de la fonction. Les prototypes sont placés en début de programme et la portée du prototype s'étend à tout le programme.

autres exemples de prototypes :

- `void f(int, float);` *f admet 2 arguments et ne retourne aucune valeur*
- `float f(int, int);` *f admet 2 arguments entiers et retourne un réel*
- `char *f(void);` *f n'admet aucun argument et retourne un pointeur sur un char (l'adresse d'un caractère ou d'une chaîne de caractères).*

**: appel de fonction**

- *appel de fonction par son identificateur et paramètres effectifs*
- attention : *identificateur = adresse de la fonction*

**: corps de la fonction**

- *variables locales à la fonction*
- *instructions de la fonction*
- **return** *provoque :*
 - *évaluation de l'expression*
 - *conversion automatique du résultat de l'expression dans le type de la fonction*
 - *renvoi du résultat*
 - *terminaison de la fonction*

Remarque : un oubli de **return** provoque un résultat indéfini. Comment faire si une erreur se produit dans la fonction avant le return ?

Remède : utiliser **exit** (biblio `<stdlib>`). Elle interrompt l'exécution du programme en renvoyant un code d'erreur à l'environnement :

Exemple :

```
Double tangent(double x)
{if (cos(x) !=0) return sin(x)/cos(x)
else {printf("fonction tangent c : erreur division par 0 ! " \n)
      exit(-1) ; /* code erreur -1 */}}
```

**: déclaration de la fonction.**

Sa place : La tendance naturelle est de placer sa déclaration à l'intérieur des déclarations de fonctions l'utilisant → *déclaration locale* (portée limitée à la fonction qui l'utilise).

La déclaration globale est plus intéressante (compilation séparée, portée maximale, etc...) : à l'extérieur (même avant) du main ou des autres fonctions qui l'utilise.

**: les arguments d'une fonction**

Les arguments (qui sont considérés comme des variables de la fonction), sont automatiquement initialisés aux valeurs passées en argument. Seul, l'ordre des arguments intervient, le nom des arguments ne jouant aucun rôle.

Les arguments sont toujours transmis par valeur : les valeurs des paramètres ne sont pas modifiés par la fonction. Il y a promotion de constante à variable lors du passage des paramètres par valeur. Il est en effet possible d'appeler une fonction avec des constantes et d'utiliser en interne de cette même fonction des variables.

Problème : Comment passer des structures et des tableaux en arguments ?

Il est possible de passer un pointeur donc passage par leur adresse !!!

VI.2 - Passage d'arguments par adresse

Le passage d'arguments par adresse permet à une fonction de modifier une ou des variables de la fonction appelante. Ce qui devient possible car nous allons passer en argument non pas la valeur d'une variable mais bien son adresse. Les adresses sont considérées comme des pointeurs dans la fonction appelée.

Exemple :

```
int plus2_val (int ); /* prototype */
int plus2_adr (int * ); /* prototype */

main()
{ int i=2, j;
  j = plus2_val(i); /* passage par valeur */
  printf ("après passage d'arguments par valeur : %d \n", i);
  j = plus2_adr(&i); /* passage par adresse */
  printf ("après passage d'arguments par adresse : %d\n", i);
}

int plus2_val(int nb) /*déclaration, argument : valeur d'un entier */
{return nb=nb+2;}

int plus2_adr(int *nb)/*déclaration, argument : adresse d'un int */
{return *nb=(*nb)+2;}
```

Résultat :

- après passage d'argument par valeur : $i=2, j=4$
- après passage d'argument par adresse : $i=4, j=4$

Analyse :

- **nb de `plus2_val` n'a aucun rapport avec nb de `plus2_adr`. `plus2_val` travaille sur nb qui a été automatiquement initialisé au contenu de i lors**

de l'appel de la fonction. **plus2_adr** travaille sur la variable **i** de **main()** par l'intermédiaire du pointeur.

- En argument de **plus2_adr**, on trouve **int *nb** ce qui signifie que **nb** est de type pointeur sur un entier. En d'autres termes, **nb** est une variable locale à **plus2_adr** contenant l'adresse de **i**. Une variable qui contient l'adresse d'une variable de type **int** est un pointeur sur un entier.
- Lors de l'appel de la fonction **plus2_adr**, **nb** de **plus2_adr** est automatiquement initialisé c'est à dire à l'adresse de **i**. **nb** de **plus2_adr** pointe sur **i** de **main()**.
- L'instruction de la fonction est : ***nb = *nb + 2** ce qui signifie ajouter 2 à l'entier dont l'adresse se trouve en **nb** et placer le résultat dans l'entier dont l'adresse se trouve en **nb**.

: passage d'un tableau en argument

Puisque l'identificateur d'un tableau correspond à l'adresse du 1er élément du tableau, les tableaux sont toujours passés par adresse. La principale raison est qu'un passage d'argument implique une opération de copie sur la pile. Recopier tout un tableau aurait un effet désastreux sur les performances et provoquerait (peut-être ?) une saturation de la pile.

Exemple :

```
void affiche (int *);           /* prototype */
void affiche (int []);

main()
{
    int tab[10];
    affiche (tab);              /* appel de la fonction */
}
```

: passage d'une structure en argument (prochain cours)

Il est possible de passer un autre type plus évolué (une structure) par valeur mais il est préférable d'effectuer un passage par adresse s'il y a beaucoup de données dans la structure. Dans ce cas, seule l'adresse de début de la structure est recopiée sur la pile. Voir prochain cours sur les structures.

: récursivité

Attention, la récursivité est gourmande en temps et mémoire, il ne faut l'utiliser que si l'on ne sait pas facilement faire autrement

31

Pour gérer efficacement vos bibliothèques de fonctions, il est conseillé de maintenir des fichiers pas trop volumineux où les fonctions sont regroupées par thème. Pour construire une bibliothèque, vous devrez créer deux fichiers:

- Un fichier d'en-tête (avec l'extension ".h") qui sera utilisé pour stocker les prototypes des fonctions.
- Un fichier de définition (ayant le même nom mais utilisant l'extension ".c") qui contiendra les déclarations de ces fonctions.

Il suffira ensuite de faire un `#include` du fichier ".h" dans votre programme principal.

: utilisation de pointeurs sur des fonctions

déclaration : exemple : `int (*test)(double, int)`

*(*test)* est une fonction qui a 2 arguments et retournant un résultat entier

test est un pointeur sur une fonction

Intérêt → programmer un appel de fonction variable (fonction appelée varie au fil de l'exécution du programme)

Exemple :

- Soient 2 fonctions différentes possédant les prototypes suivants :
`int fonct1(double, int);`
`int fonct2(double, int);`
- Affectations possibles lors d'un bloc d'instructions
`test=fonct1; test=fonct2;`
- Appel de la fonction *test*
`(*test)(5.35,4);` /* va chercher l'adresse contenue dans *test* */
/* cette adresse est celle de *fonct1* (ou bien de *fonct2*) */
/* *test* va transmettre les valeurs à *fonct1* (ou *fonct2*) */
/* c'est équivalent à exécuter *fonct1(5.35,4)* (ou *fonct2(5.35,4)*) */

: fonctions transmises en arguments

exemple : fonction qui calcule " tang^{-1} " d'une fonction.

`float myarctan(float(*fonct)(float), int v2, float v3, ...)`

- *(*fonct)* est une fonction
- *fonct* est un pointeur sur une fonction recevant un argument de type float et fournissant un résultat de type float.
- à l'intérieur de *myarctan*, on pourra appeler cette fonction dont on aura reçu l'adresse ainsi : *(*fonct)(x)*
- attention : *fonct(x)* désigne un pointeur contenant l'adresse de la fonction et non pas directement l'adresse de la fonction !!!

VI.4 - Résumé (types)

exemple avec le type « int »

Déclaration	Description du type de « nom »
<code>int nom</code>	<code>int</code>
<code>int *nom</code>	pointeur sur <code>int</code>
<code>int nom()</code>	fonction <code>nom</code> retournant un résultat <code>int</code>
<code>int nom[n]</code>	tableau de <code>n</code> éléments <code>int</code>
<code>int *nom()</code>	fonction <code>nom</code> retournant un pointeur sur un <code>int</code>
<code>int (*nom) ()</code>	pointeur sur des fonctions retournant un <code>int</code>
<code>int *nom[n]</code>	tableau <code>nom</code> de <code>n</code> pointeurs sur <code>int</code>
<code>int (*nom) [n]</code>	pointeur sur un tableau de <code>n</code> éléments de type <code>int</code>
<code>int (*nom[n]) ()</code>	tableau de <code>n</code> pointeurs sur des f° retournant un <code>int</code>
<code>int (*(nom[n]) ()</code>	tableau de <code>n</code> pointeurs sur des f° retournant un pointeur sur un <code>int</code>

VI.5 - Portée, visibilité et classe d'allocation

: Portée des variables

- Variables **locales** : les variables déclarées à l'intérieur d'un bloc d'instructions (entre `{}`) sont uniquement visibles à l'intérieur de ce bloc (variables **locales** à ce bloc). Elles sont créées lors de l'entrée dans la fonction puis détruites lors de la sortie de la fonction (sur instruction `return` ou après exécution de la dernière instruction). Elles ne sont pas initialisées (sauf si on le fait dans le programme) et elles perdent leur valeur à chaque appel à la fonction. On peut allonger la durée de vie d'une variable locale en la déclarant **static**. Lors d'un nouvel appel à la fonction, la variable garde la valeur obtenue à la fin de l'exécution précédente. Une variable **static** est initialisée à 0 lors du premier appel à la fonction :

Exemple: `int i;` devient `static int i;`

- Variables **globales** : les variables déclarées au début du fichier, à l'extérieur de **toutes** les fonctions sont disponibles à toutes les fonctions du programme. Ce sont les variables globales. En général, déclarées immédiatement après les `#include` et **initialisées à 0 au début de l'exécution du programme, sauf si on les initialise à une autre valeur**

: Classe d'allocation

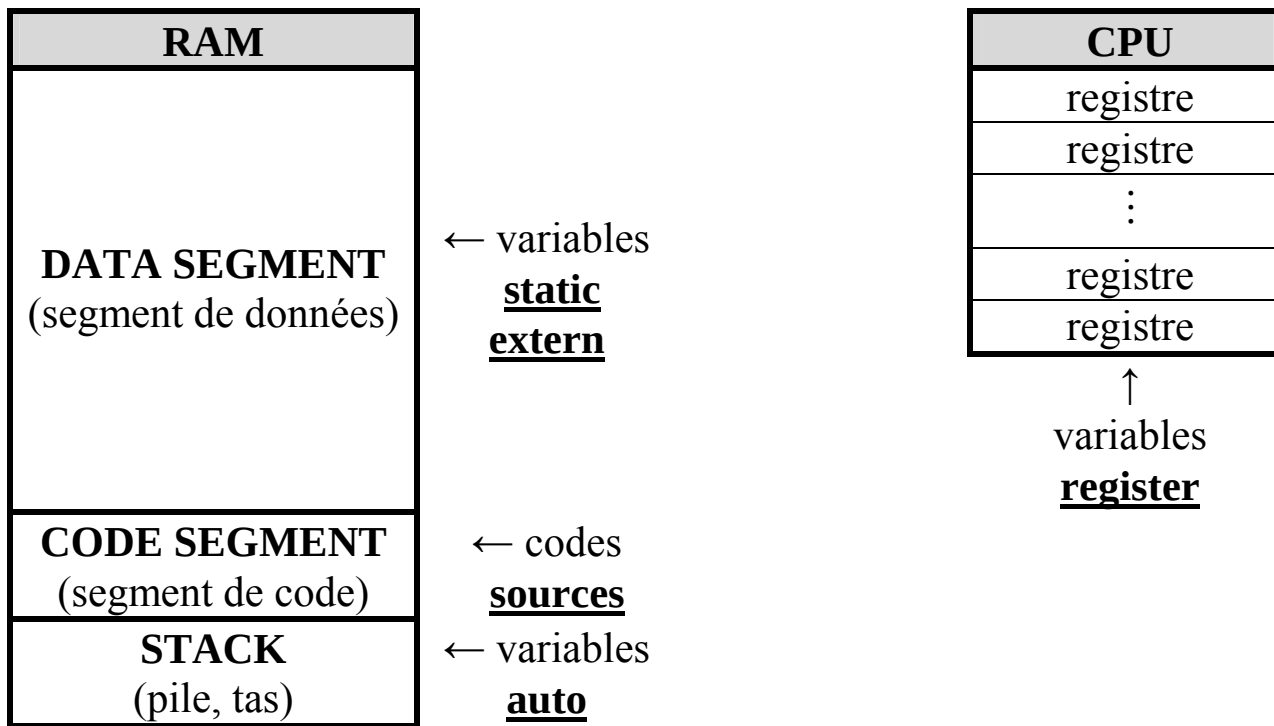
La classe d'allocation d'une donnée indique comment celle-ci est rangée en mémoire. En C, le programmeur peut spécifier au compilateur de quelle manière il souhaite voir manipuler une donnée. On dispose alors de plusieurs spécificateurs :

static : Allocation mémoire (adresse) fixe pendant toute l'exécution du programme. On peut initialiser une variable statique (par défaut, elle est initialisée à 0). Les données globales sont forcément statiques (elles existent au démarrage). Déclarer une variable locale comme statique permet de retrouver sa valeur antérieure. Si une variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens (pas d'interférence avec la variable de même nom du premier fichier source). Très utile pour développer un ensemble de fonctions générales qui doivent partager des variables globales!

- **auto** : Variable dynamique, créée lors de sa déclaration et détruite à la sortie de sa zone de portée. Les variables locales sont auto par défaut. On utilise donc très peu ce spécificateur
- **const** : la donnée est constante. Elle ne peut pas être modifiée. Elle peut être initialisée lors de sa déclaration.
- **register** : S'applique uniquement aux variables dynamiques. Demande au compilateur d'optimiser le traitement sur une variable en utilisant (éventuellement) un registre du processeur. On ne peut plus accéder à son adresse (par &).
- **volatile** : Le contenu de la variable est susceptible d'être modifié tout seul, indépendamment du programme (interruption, périphérique en entrée...). Le compilateur devra évaluer sa valeur à chaque accès.
- **extern** : aucune réservation mémoire, mais on précise seulement que la variable globale est définie ailleurs et on en précise le type.

Supposons que nous avons 2 fichiers sources. Dans le premier est déclaré la variable `int x` et dans le second `extern int x`. Après compilation, création de 2 modules objets. L'éditeur de liens a pour but de rassembler les modules objet. Il va faire correspondre au symbole `x` du second fichier source l'adresse effective de la variable `x` définie dans le premier fichier source.

Après compilation du fichier source, on trouve une indication qui associe le symbole `x` et son adresse dans le module objet correspondant. Contrairement aux variables locales (pour lesquelles il ne reste aucune trace du nom après compilation), le nom des variables globales `extern` continue à exister au niveau des modules objet (mécanisme identique à celui des fonctions pour lesquelles leur nom subsiste afin que l'éditeur de liens puisse retrouver les modules objet correspondants).



: Résumé portée & allocation

variables globales (avant le main, dans un fichier) :

classe	extern (défaut)	static	extern (importée)
validité	fichier	fichier	fichier
allocation	à la déclaration	à la déclaration	non
localisation	segment data	segment data	importée
durée de vie	tâche	tâche	importée
initialisation	autorisée	autorisée	interdite

variables locales (à un bloc ou dans une fonction) :

classe	auto (défaut)	static	register	extern (importée)
validité	bloc / fonction	bloc / fonction	bloc / fonction	bloc / fonction
allocation	à l'entrée	à la déclaration	à l'entrée	non
localisation	pile	segment data	pile	importée
durée de vie	bloc / fonction	tâche	bloc / fonction	importée
initialisation	autorisée	autorisée	autorisée	interdite

VII. PRE-PROCESSEUR

Le préprocesseur :

- *traite le fichier source avant le compilateur.*
- *ne manipule que des chaînes de caractères.*
- *retire les parties commentaires (entre /* et */).*
- *prend en compte les lignes commençant par un # pour créer le code que le compilateur analysera.*

Ses possibilités sont de 4 ordres :

- **inclusion de fichier** : *#include nom de fichier. Par convention, les fichiers à inclure ont des noms terminés par .h (header). Il existe 3 façons de nommer un fichier à inclure. Ces façons déterminent l'endroit où le préprocesseur va chercher le fichier.*
 - *par son chemin absolu : #include "/users/chris/essai/header.h"*
 - *à partir du répertoire courant si le nom de fichier est entouré par des guillemets : #include "header.h"*
 - *à partir d'un répertoire prédéfini correspondant à l'installation du compilateur, si le nom de fichier est entouré par un inférieur et un supérieur. #include <stdio.h>. Il est possible de demander au préprocesseur d'ajouter d'autres répertoires à sa recherche en utilisant une option de compilation à l'appel de l'enchaîneur de passes (UNIX).*
- **définition de variables de preprocessing** :
 - *#define NOM valeur*
 - *#undef NOM*
- **définition de macrofonction ou macro-expression** :
 - #define m(x) (128*(x)+342*(x)*(x))*
 - *Une macro est une définition de symbole avec paramètres et ressemble à une fonction :*
 - #define max(x,y) (x<y ? y : x) /* maximum de x et y */*
 - #define min(x,y) (x>y ? y : x) /* minimum de x et y */*
 - #define carre(x) x*x /* carré de x */*
 - *Une fois définie, une macro est appelée comme une fonction, en mentionnant son nom et la liste des paramètres effectifs. Cependant, l'effet n'est pas le même : un appel de macro est remplacé par le texte de sa définition où chaque paramètre formel est remplacé par le paramètre effectif correspondant.*

*Appel de macro**Texte généré par le préprocesseur**carre(z)**z*z**max(t,12)**(t<12 ? 12 : t)**max(8.65,y+9.1)**(8.65<y+9.1 ? y+9.1 : 8.65)**min(carre(a),carre(b))**(a*a>b*b ? b*b : a*a)*

- *On insiste sur la spécificité du pré-traitement des programmes C : il n'est réalisé aucune interprétation du texte (programme source), mais seulement du remplacement/recopie/concaténation. En particulier, il y a 2 différences importantes entre fonction et macro : une macro évite le coût de l'appel de fonction à l'exécution ; une macro est polymorphe, c'est-à-dire qu'elle fonctionne pour tout type de données. Dans l'exemple précédent, les macros min et max sont adaptées pour tous les types de nombres du C.*

- **sélection du code** *en fonction des variables du préprocesseur :*

- *#if*
- *#ifdef*
- *#ifndef*
- *#else*
- *#endif*

La sélection de code permet de générer à partir d'un même fichier source des fichiers exécutables pour des machines et des systèmes différents. Le principe de base consiste à passer ou à supprimer des parties de code suivant des conditions fixées à partir des variables de précompilation.

Ceci permet d'avoir un même fichier source destiné à être compilé sur plusieurs machines, ou plusieurs systèmes, en tenant compte des différences entre ces machines.

La sélection est aussi utilisée pour avoir un source avec des instructions qui donnent des informations sur les variables (traces), et pouvoir générer le fichier exécutable avec ou sans ces traces.

VIII. LES STRUCTURES

VIII.1 - Déclaration d'une structure

Une structure est un ensemble d'éléments de types différents réunis sous un même nom. La syntaxe est :

```
struct caract
{
    int numero; /* membre */
    int qte;
    float prix;
};
struct caract art1, art2 ;
```

*Ceci définit un **modèle** de structure dont le nom est **caract** et précise le nom et le type de chacun des **membres** (ou champs) constituant la structure (numero, qte et prix). **art1** et **art2** sont des variables structurées de type **caract**.*

Ecriture compacte : (attention à définir toutes les variables d'un coup)

```
struct caract
{
    int numero;
    int qte;
    float prix;
} art1, art2;
```

VIII.2 - Utilisation d'une structure

Utilisation d'une structure de 2 manières différentes :

- *en travaillant individuellement sur chacun des champs*
- *en travaillant de manière globale sur l'ensemble de la structure*



: utilisation des champs d'une structure

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. On peut avoir accès à un membre d'une structure par :

nom_variable_structurée.nom_membre

Exemples :

art1.numero = 15; *affecte la valeur 15 au champ numero de la structure art1*

printf("%d", art1.prix);

scanf("%d", &art2.prix); *lit une valeur affectée au champ prix de struct art2*

art1.numero++ *incrémente de 1 la valeur du champ numero de art1*

 : utilisation globale d'une structure

- Affectation d'une structure à une autre structure définie avec le même modèle. Par exemple, si les structures `art1` et `art2` ont été déclarées suivant le modèle **caract** défini précédemment, on peut écrire : **art1=art2**. (réfléchir sur la possibilité d'affecter un tableau à un autre tableau si ils sont contenus dans une structure !!!)
- Opérateur & (adresse)

 : initialisations d'une structure

On retrouve pour les structures les règles d'initialisation qui sont en vigueur pour tous les types de variables, à savoir :

- en l'absence d'initialisation explicite, les structures "statique" sont par défaut initialisées à zéro, les automatiques sont indéfinies.
- initialisation possible lors de sa déclaration mais on ne peut utiliser que des constantes ou des expressions constantes

struct caract art1 = {100, 285, 2000};

Attention :

- pour afficher une variable struct, il faut afficher chacun des champs simples, un à la fois.
- La taille d'une variable struct n'est pas forcément égale à la somme des tailles de ses champs (réalignement de la mémoire : espaces vide entre les champs de types, donc de longueur, différents. Eviter `memcmp` pour comparer 2 struct : comparaison champ par champ

VIII.3 - Utilisation des types synonymes : TYPEDEF

Pour simplifier la déclaration de types, on peut utiliser la déclaration **typedef**. Elle s'applique à tous les types et pas seulement aux structures.

Exemples d'utilisation :

```
typedef int toto;
toto n,p;      /* équivalent à int n,p*/

typedef int * ptr; /* signifie ptr est synonyme de int* */
ptr p1,p2;      /* équivalent à int *p1, *p2 */

typedef int vecteur[3];
vecteur v, w;    /* équivalent à int v[3], w[3] */
```

Exemples d'utilisation et des structures :

```
struct caract
{ int numero;
  int qte;
  float prix;
};
typedef struct caract titi
titi art1,art2;
```

équivalent à :

```
typedef struct caract
{ int numero;
  int qte;
  float prix;
} titi ;
titi art1,art2;
```

VIII.4 - Imbrication de structures

Chaque champ d'une structure peut être d'un type absolument quelconque : pointeur, structure, tableau

: structure comportant des tableaux

Soit la déclaration suivante :

```
struct personne {    char nom[30];
                    char prenom[20];
                    float heures [31];
                    } employe1, employe2;
```

Celle-ci réserve les emplacements pour 2 structures nommées employe1 et employe2 qui comportent 3 champs :

- **nom** qui est un tableau de 30 caractères
- **prenom** qui est un tableau de 20 caractères
- **heures** qui est un tableau de 31 réels

*La notation **employe1.heures[4]** désigne le 5^{ième} élément du tableau **heures** de la structure **employe1**. De même, **employe2.nom[0]** représente le 1er caractère du champ **nom** de la structure **employe2**. **&employe2.heures[4]** désigne l'adresse du 5^{ième} élément du tableau **heures** de la structure **employe2**. **employe2.nom** désigne le champ **nom** de la structure **employe2**, c'est à dire l'adresse du tableau **nom**.*

Exemple d'initialisation :

Struct personne employe1 = {"dupont","jean",{8,7,8,6,0,0,1}};

: tableaux de structures

Soit la déclaration suivante :

```
struct point {    char nom;  
                int  x;  
                int  y;  
                };  
struct point courbe [50];
```

point est un nom de modèle de structure et **courbe** représente un objet de type "tableau de 50 éléments du type point".

La structure **point** pourrait servir par exemple à représenter un point d'un plan, point qui serait défini par son nom (caractère) et ses coordonnées.

Elle pourrait servir à représenter un ensemble de 50 points du type ainsi défini.

Soit *i* un entier. La notation **courbe[i].x** désigne la valeur du champ *x* de l'élément de rang *i* du tableau **courbe**. **courbe[i].nom** représente le nom (du type char) du point de rang *i* du tableau **courbe**

Exemple d'initialisation de la variable courbe :

struct point courbe[50] = { {'A', 10, 25}, {'M', 12, 28} }

: structure comportant d'autres structures

Reprenons l'exemple du modèle de structure personne.

```
struct personne {    char nom[30];  
                    char prenom[20];  
                    float heures [31];  
                    } employe1, employe2 ;
```

Cette structure doit servir à la gestion du personnel d'une entreprise. On doit alors l'étendre en précisant les dates d'embauche et d'entrée dans le dernier poste occupé. Ces dates peuvent alors être représentées par des structures comportant 3 champs (jour, mois, année) :

```
struct date    {    int jour ;  
                  int  mois ;  
                  int  annee;  
                  };
```

```
struct personne {          char nom[30];
                          char prenom[20];
                          float heures [31];
                          struct date date_embauche;
                          struct date date_poste;
                          } employe1, employe2;
```

VIII.5 - Transmission d'une structure en argument d'une fonction

 : transmission de la valeur d'une structure

```
#include <stdio.h>
struct caract {   int a;
                  float b;
                  };

void fct (struct caract y);
main()
{struct caract x;
  x.a=1; x.b=12.5;
  printf("\n avant appel fct: %d %e", x.a, x.b);
  fct(x);
  printf("\n au retour dans main: %d %e", x.a, x.b);
}

void fct (struct caract &tata)
{
  s.a=0; s.b=1;
  printf("\n dans fct: %d %e", s.a, s.b);
}
```

avant appel fct : 1 1.2500e+01

dans fct : 0 1.0000e+00

au retour dans main : 1 1.2500e+01

Les valeurs de la structure x sont copiées localement dans la fonction fct.

 : transmission de l'adresse d'une structure : opérateur ->

*On modifie le programme précédent pour que la fonction **fct** reçoive l'adresse d'une structure et non plus sa valeur d'où la déclaration **fct(&x)**.*

*Cela signifie que son en-tête sera de la forme **void fct (struct caract * tata)**.*

*Le problème se pose alors d'accéder à chacun des champs de la structure d'adresse **tata** (le point "." suppose un nom de structure pour premier opérande et pas une adresse). On fait alors appel à un nouvel opérateur "->" lequel permet d'accéder aux différents champs d'une structure à partir de son adresse de début (en toute rigueur, on pourrait aussi y accéder par (*tata).a et (*tata).b). Voyons sur un exemple l'utilisation de ce nouvel opérateur :*

```
#include <stdio.h>
struct caract { int a;
                float b;
            };

void fct (struct caract *);
main()
{struct caract x;
  x.a=1; x.b=12.5;
  printf("\n avant appel fct : %d %e", x.a, x.b);
  fct(&x);
  printf("\n au retour dans main : %d %e", x.a, x.b);
}

void fct (struct caract s)
{
  tata->a=0; tata->b=1;
  printf("\n dans fct : %d %e", tata->a, tata->b);
}
```

<i>avant appel fct :</i>	1	1.2500e+01
<i>dans fct :</i>	0	1.0000e+00
<i>au retour dans main :</i>	0	1.0000e+00

VIII.6 - Transmission d'une structure en valeur de retour d'une fonction

```
struct caract fct (...)  
{struct caract employez;  
  ...  
  return employez;  
}
```

La fonction renvoie la valeur de employez, mais employez est de type structure locale à la fonction fct.. employez n'est pas obligatoirement créée à l'intérieur de la fonction. Elle aurait pu être reçue en argument.

VIII.7 - Portée d'un structure

*La portée d'une structure peut être locale (à une fonction) ou globale (accessible par toutes les fonctions d'un même fichier source) selon l'endroit de sa déclaration. Elle ne peut pas en revanche être déclarée **extern** (accessible par d'autre fichier .c). La déclaration **extern** s'applique à des noms susceptibles d'être remplacés par une adresse au niveau de l'édition de liens. Un modèle de structure n'a de sens qu'au moment de la compilation du fichier source où il se trouve. On peut toutefois placer les déclaration de modèles dans un fichier que l'on incorpore par des #include.*

Rappel de l'ordre de fonctionnement :

- préprocesseur (macros : directives#, C « pur »)
- compilation (obj, langage machine)
- édition de liens(exe, fonctions standards)

VIII.8 - Structure et listes chaînées

Voir prochain cours sur les listes chaînées...

remarque : Matlab 6.0 (et au delà) utilise les structures et listes chaînées pour la représentation de ses objets systèmes !!!.

IX. GESTION DE FICHIERS

IX.1 - Introduction

L'accès à des fichiers, pour lire ou modifier des fiches, peut se faire grâce à des fonctions standard que l'on rencontre avec tous les compilateurs. Ces fonctions se situent à différents niveaux :

- **au niveau bas**, les fonctions d'accès sont directement calquées sur les possibilités du système d'exploitation
- **au niveau haut** où la notion de mémoire tampon est introduite

IX.2 - Fichiers niveau haut

Au niveau haut, les caractères ne sont pas transmis directement dans le fichier, ils sont stockés temporairement dans un tampon et ce n'est que lorsqu'il est rempli que la transmission a lieu.

: déclaration de fichier

*Au niveau haut, les fichiers sont désignés par un pointeur sur une structure baptisée **FILE** (majuscule obligatoire) définie dans le fichier **stdio.h**. Dans cette structure **FILE**, on trouve des informations telles que l'adresse du tampon, la position actuelle dans le tampon, le nombre de caractères qui y ont déjà été écrits ou qui restent à lire.*

*La déclaration s'effectue par : **FILE *fp**, **fp** est l'identificateur du pointeur (pointeur sur une objet (structure) de type **FILE**. Réservation mémoire uniquement pour un pointeur.*

: ouverture de fichier

*La fonction **fopen** ouvre un fichier spécifié par un nom soit en lecture, soit en écriture et de n'importe quel type. La structure de cette fonction s'écrit :*

FILE *fopen(char *fp, char *mode)

*Le paramètre ***fp** est le pointeur sur la chaîne de caractères qui contient le nom du fichier (il est possible de spécifier un chemin d'accès). Fournit en retour un "flux" (pointeur sur une structure de type prédéfini **FILE**) ou un pointeur **NULL** si l'ouverture a échoué (problème d'existence ou bien de droit d'accès : chemin erroné, saturation de l'espace mémoire, etc...) et un code d'erreur est stocké dans la variable entière globale **errno**. Le paramètre ***mode** pointe sur une chaîne de caractères qui spécifie comment le programme accédera au fichier. Les principaux modes disponibles sont :*

<i>mode</i>	<i>signification</i>
"r"	ouverture en mode lecture seulement (pas de création de nouveau fichier))
"w"	ouverture en mode écriture seulement. Si le fichier existe déjà, sa longueur est ramenée à 0 c'est à dire que toutes les données précédentes du fichier sont écrasées. Par contre s'il n'existe pas, il sera créé
"a"	ouverture en mode ajout seulement (création si besoin est). les données supplémentaires seront ajoutées à la fin du fichier sans écraser les données précédentes.
"r+"	ouverture en mode lecture plus écriture (pas de création). autorise à effectuer des mises à jour dans un fichier existant (lecture, modification, agrandissement)
"w+"	ouverture en mode écriture plus lecture (création si besoin est). si le fichier existe, la fonction le vide
"a+"	ouverture en mode ajout plus lecture (création si besoin est). les ajouts sont réalisés à la fin du fichier. Le fichier peut être lu.

remarque : idem pour les fichiers binaires en ajoutant "b". Il peut contenir des données qui sont transférées sans interprétation par les fonctions de la bibliothèque.

: fermeture de fichier

int fclose(FILE *fp) ferme le fichier (de n'importe quel type) préalablement ouvert par **fopen** (vide le tampon associé au flux concerné, désalloue l'espace mémoire attribué à ce tampon et ferme le fichier correspondant). **fclose** renvoie la valeur 0 si l'opération est un succès et dans le cas contraire renvoie la valeur -1 indiquant que le numéro de l'erreur a été stockée dans la variable **errno**.

: destruction de fichier

int remove(char *fp) détruit le fichier et retourne 0 en cas de succès.

: renommer un fichier

int rename(char *oldname, char *newname) renomme le fichier en "newname" et retourne 0 en cas de succès.

: lecture de fichier binaire

int fread(void *p, int size, int n, FILE *fp) effectue une opération de lecture de **n** objets ayant chacun une longueur de **size** octets sur des fichiers binaires.

**p correspond à une zone mémoire où l'on veut stocker les données. Retourne la valeur correspondant au nombre d'éléments lus ou écrits en cas de succès et retourne 0 en cas d'erreur, ou si la fin du fichier est atteinte.*

: écriture d'un fichier binaire

*int fwrite(p, size, n, fp) effectue une opération d'écriture de **n** objets ayant chacun une longueur de **size** octets dans le fichier **fp**. Les données à écrire sont présentes dans une zone mémoire repérée par **p**. Retourne le nombre de blocs écrits.*

: lecture de fichier texte

*int fscanf(FILE *fp, format, adresse de la variable) lit des caractères depuis le flux spécifié (fp) et les fait correspondre au format voulu. Retourne le nombre de champs d'entrée correctement lus, convertis et mémorisés.*

: écriture de fichier texte

*int fprintf(fp, format, variable) écrit **nb** paramètres selon le format spécifié sur le fichier texte préalablement ouvert par **fopen**. Cette fonction utilise les mêmes spécifications de format que **printf**. Mais **fprintf** écrit les sorties dans le flux spécifié par le pointeur de fichier. Elle retourne le nombre d'octets écrits. Si ce nombre est inférieur **nb**, il y a erreur.*

: détection fin de fichier

*int feof(FILE *fp) (End Of File en anglais) retourne 0 si la fin de fichier n'a pas été détectée et une valeur différente de 0 si la fin de fichier a été détectée.*

IX.3 - Action sur le pointeur de fichier

: FSEEK

*int fseek(FILE *fp, long nbo, int org) place le pointeur du flux indiqué (fp) à un endroit défini comme étant situé à **nbo** octets de "l'origine" spécifiée par **org**. **org=SEEK_SET** correspond au début du fichier, **SEEK_CUR** à la position actuelle du pointeur et **SEEK_END** à la fin du fichier.*

: FTELL

*long ftell(FILE *fp) retourne la position courante du pointeur du flux indiqué (exprimée en octets par rapport au début du fichier).*

*remarque : pour les fichiers textes, FSEEK n'autorise que 0 ou la valeur retournée par FTELL pour l'argument **nbo**.*

X. LISTES CHAÎNÉES

X.1 - Introduction

Le concept de liste chaînée utilise les notions de pointeurs, d'allocation dynamique et de structures !!

Au départ, aucune variable n'est allouée. Dès qu'il est nécessaire d'enregistrer un élément, une allocation de mémoire est faite. Après apparition d'un nouveau besoin, une nouvelle demande est faite. toutefois l'adresse sera conservée dans l'élément précédent (plutôt que de conserver cette adresse dans un pointeur particulier). on crée ainsi une "chaîne". Chaque élément contient un pointeur sur un élément de même type (récursivité).

Exemple : répertoire téléphonique

```
struct Repertoire {    char nom[MAX];  
                      char numeros[16];  
                      struct Repertoire *suivant;  
};
```

- *Il faut conserver un pointeur struct Repertoire * adresse, celui qui conserve l'adresse du début de la liste.*
- *Réservation mémoire d'1 pointeur ≠ espace pour structure.*
- *Parcours séquentiel : pour trouver un élément, il faut avoir lu tous les précédents !*
- *test d'arrêt : par convention, le dernier élément de la liste ne pointe aucun élément : il doit contenir NULL.*

X.2 - Création d'une liste chaînée

2 méthodes :

- *nouvel élément ajouté à la fin de la liste (parcours ordre chronologique)*
- *Introduction en tête de liste (+rapide) : lifo*

les étapes :

- *définir un type pointeur de répertoire (Adresse)*
- *le programme principal réserve un pointeur (Premier de type Adresse) qui conserve l'adresse de début*
- *appel de la fonction de Création avec le passage du paramètre "l'adresse du pointeur Premier" : la fonction écrit l'adresse du début de la liste.*

- la fonction *Création* déclare un autre pointeur *Nouveau* sur un élément de type *article* (variable temporaire)
- initialiser le contenu de **Debut* à *NULL* pour éviter tout parcours intempestif de la mémoire.
- création de la liste élément par élément dans une boucle *for* (par exemple). Il y a une réservation pour chaque élément. Les adresses des zones allouées sont stockées successivement dans *Nouveau*.
- remplissage des différents champs.
- déplacement du pointeur de champs : "*Nouveau->Suivant=*Debut*" écrit dans le champ *Suivant* de la variable pointée, l'adresse de l'élément "accroché" à sa suite. Il s'agit en fait de l'élément réservé et initialisé au tour précédent de la boucle *for*.
- Préciser que le nouvel élément devient nouveau point de départ de la liste.

exemple : création d'une liste de 6 éléments de type répertoire :

```
void Creation (Adresse * Debut)
{
    int i;
    Adresse Nouveau;
    *Debut=NULL;
    for (i=0;i<6;i++)
        {Nouveau=(Adresse)malloc(sizeof(*Nouveau));
        gets (Nouveau->nom);
        gets (Nouveau->numeros);
        Nouveau->suivant=*Debut;
        *Debut=Nouveau;
        }
}
void main(void)
{
    Adresse Premier;
    Creation(&Premier);
}
```

X.3 - Affichage de la liste

```
void Affiche (Adresse Debut)
{while(Debut)
    { printf("%s -> %s | /n",Debut->nom,Debut->numeros);
      Debut=Debut->Suivant;}}
void main(void)
{
    Adresse Premier;
    Creation(&Premier);
    Affiche(Premier);}
```

Affiche possède comme argument d'entrée un pointeur sur une variable de type Repertoire. Elle affiche les éléments les uns après les autres tant qu'elle n'est pas à la fin de la liste (fin=valeur du pointeur nulle). L'instruction Debut=Debut->Suivant permet de pointer (donc de passer) à l'élément suivant de la liste (on affecte, dans le pointeur Debut la valeur qui est écrite dans le champ Suivant de l'objet pointé actuel).

X.4 - Recherche d'un élément

exemple :

```
void Detection(Adresse Selection, int Choix, Adresse* Debut)
{
    char Test[MAX];
    Adresse Precedent;
    printf("\n donnez le nom a rechercher: ");
    gets(Test);
    while((strcmp(Selection->nom, Test))&&(Selection->Suivant))
        {Precedent=Selection;
         Selection=Selection->Suivant;}
}
void main(void)
{
    Adresse Premier;
    Creation(&Premier);

    Detection(premier, choix, &Premier);
}
```

X.5 - Rangement par ordre alphabétique

exemple : Programmer un algorithme de tri sur les éléments de la liste...

XI. POUR ALLER PLUS LOIN ...

- *Manipulations et opérations de bits*
- *Ensembles (partie1)*
 - *tableaux, listes : réunion, intersection, complexité*
- *Ensembles (partie2)*
 - *pires et files : hachage, tri par distribution*
- *Arbres et ensembles ordonnés :*
 - *représentation, définition, opérations*
 - *insertion, suppression, parcours*
- *Graphes : orientés, non orientés*
 - *chemin, chaînes, circuits, cycles*
- *Tri : relations d'ordre, algorithmes, complexité*
 - *selection*
 - *par tas (ou tri maximier sur arbre binaire)*
 - *à bulles*
 - *par insertion*
 - *shellsort (par insertion spécifique)*
 - *par fusion*
 - *rapide (quicksort)*
- *Calcul numérique et programmation de fonctions mathématiques*
- *Vers la programmation orientée objet...*