Le Langage C

Version 1.2

©2002 – Florence HENRY

Observatoire de Paris – Université de Versailles
florence.henry@obspm.fr

Table des matières

1 Les bases 3

2 Variables et constantes 5

3 Quelques fonctions indispensables 8

4 Les instructions de contrôle 11

5 Les fonctions 15

6 Les tableaux 18

7 Les structures 20

8 Les pointeurs 22

9 Pointeurs et fonctions 25

10 Pointeurs et tableaux 26

11 Allocation dynamique de mémoire 31

Chapitre 1

Les bases

1.1 La structure d'un programme

Un programme simple se compose de plusieurs parties :

- des directives de précompilation
- une ou plusieurs fonctions dont l'une s'appelle obligatoirement `main()`, celle-ci constitue le programme principal et comporte 2 parties :
  - la déclaration de toutes les variables et fonctions utilisées
  - des insctructions

Les commentaires débutent par `/*` et finissent par `*/`, ils peuvent s'étendre sur plusieurs lignes.

1.1.1 Les directives de précompilation

Elles commencent toutes par un `#`.

commande	signification
`#include <stdio.h>`	*permet d'utiliser les fonctions* `printf()` *et* `scanf()`
`#include <math.h>`	*permet d'utiliser les fonctions mathématiques*
`#define PI 3.14159`	*définit la constante* `PI`
`#undef PI`	*à partir de cet endroit, la constante* `PI` *n'est plus définie*

`#ifdef PI`
instructions 1 ...
`#else`
instructions 2 ...
`#endif`
	si la constante `PI` *est définie, on compile les instructions 1, sinon, les instructions 2*

Parmi ces directives, une seule est obligatoire pour le bon fonctionnement d'un programme : `#include <stdio.h>` . En effet, sans elle, on ne peut pas utiliser les fonctions utiles pour l'affichage à l'écran : `printf()` et la lecture des données au clavier : `scanf()`. Nous verrons le fonctionnement de ces fonctions plus tard.

1.1.2 La fonction `main()`

Elle commence par une accolade ouvrante `{` et se termine par une accolade fermante `}`. À l'intérieur, chaque instruction se termine par un point-virgule. Toute variable doit être déclarée.

```
main(){
   int i; /* declaration des variables */    instruction_1;
   instruction_2;
   ...
}
```

Exemple de programme simple :

```
#include <stdio.h>
/* Mon 1er programme en C */
main(){
   printf("Hello world\n");
}
```

1.2 La compilation sous Unix

Une fois le programme écrit, on ne peut pas l'exécuter directement. Pour que l'ordinateur comprenne ce que l'on veut lui faire faire, il faut traduire le programme en **langage machine**. Cette traduction s'appelle la **compilation**.

On compile le programme par la commande `cc prog.c`, où prog.c est le nom du programme. La compilation crée un fichier exécutable : `a.out`. On peut vouloir lui donner un nom plus explicite, et pour cela, à la compilation, on compile avec la commande `cc -o prog prog.c` qui va appeler le programme exécutable `prog` au lieu de `a.out`.

On démarre alors le programme avec la commande `./prog`.

Chapitre 2

Variables et constantes

2.1 Les constantes

Constantes entières `1,2,3`,...

Constantes caractères `'a','A'`,...

Constantes chaînes de caractères `"Bonjour"`

Pas de constantes logiques Pour faire des tests, on utilise un entier. 0 est équivalent a faux et tout ce qui est $\neq 0$ est vrai.

2.2 Les variables

2.2.1 Noms des variables

Le C fait la différence entres les MAJUSCULES et les minuscules. Donc pour éviter les confusions, on écrit les noms des variables en minuscule et on réserve les majuscules pour les constantes symboliques définies par un `#define`. Les noms **doivent** commencer par une **lettre** et ne contenir **aucun blanc**. Le seul caractère spécial admis est le soulignement (_). Il existe un certain nombre de noms réservés (while, if, case, ...), dont on ne doit pas se servir pour nommer les variables. De plus, on ne doit pas utiliser les noms des fonctions pour des variables.

2.2.2 Déclaration des variables

Pour déclarer une variable, on fait précéder son nom par son type.
Il existe 6 types de variables :

type	signification	val. min	val. max
`char`	caractère codé sur 1 octet (8 bits)	-2^7	2^7-1
`short`	entier codé sur 1 octet	-2^7	2^7-1
`int`	entier codé sur 4 octets	-2^{31}	$2^{31}-1$
`long`	entier codé sur 8 octets	-2^{63}	$2^{63}-1$
`float`	réel codé sur 4 octets	$\sim -10^{38}$	$\sim 10^{38}$
`double`	réel codé sur 8 octets	$\sim -10^{308}$	$\sim 10^{308}$

On peut faire précéder chaque type par le préfixe `unsigned`, ce qui force les variables à prendre des valeurs uniquement positives.

Exemples de déclarations :

déclaration	signification
`int a;`	`a` *est entier*
`int z=4;`	`z` *est entier et vaut 4*
`unsigned int x;`	`x` *est un entier positif (non signé)*
`float zx, zy;`	`zx` *et* `zy` *sont de type réel*
`float zx=15.15;`	`zx` *est de type réel et vaut 15.15*
`double z;`	`z` *est un réel en double précision*
`char zz;`	`zz` *est une variable caractère*
`char zz='a';`	`zz` *vaut* `'a'`

Il n'y a pas de type complexe.

2.3 Les opérateurs

Le premier opérateur à connaître est l'**affectation "="**. Exemple : {`a=b+";`} Il sert à mettre dans la variable de **gauche** la valeur de ce qui est à droite. Le membre de droite est d'abord évalué, et ensuite, on affecte cette valeur à la variable de gauche. Ainsi l'instruction `i=i+1` a un sens.

Pour les opérations dites naturelles, on utilise les opérateurs `+`, `-`, `*`, `/`, `%`.
`%` est l'opération modulo : `5%2` est le reste de la division de 5 par 2. `5%2` est donc égal à `1`.
Le résultat d'une opération entre types différents se fait dans le type le plus haut. Les types sont classés ainsi :

```
char < int < float < double
```

Par ailleurs, l'opération `'a'+1` a un sens, elle a pour résultat le caractère suivant à `a` dans le code ASCII.

En C, il existe un certain nombre d'opérateurs spécifiques, qu'il faut utiliser prudemment sous peine d'erreurs.

`++` incrémente la variable d'une unité.

`--` décrémente la variable d'une unité.

Ces 2 opérateurs ne s'utilisent pas avec des réels. Exemples d'utilisation :

```
i++; /* effectue i=i+1 */
i--; /* effectue i=i-1 */
```

Leur utilisation devient délicate quand on les utilise avec d'autres opérateurs. Exemple :

```
int i=1 , j;
j=i++;        /* effectue d'abord j=i et ensuite i=i+1 */
              /* on a alors j=1 et i=2 */
j=++i;        /* effectue d'abord i=i+1 et ensuite j=i */
              /* on a alors j=2 et i=2 */
```

Quand l'opérateur ++ est placé avant une variable, l'incrémentation est effectuée en premier. L'incrémentation est faite en dernier quand ++ est placé aprés la variable. Le comportement est similaire pour `--`.

D’autres opérateurs sont définis dans le tableau qui suit :

```
i+=5; /* i=i+5 */
i-=3; /* i=i-3 */
i*=4; /* i=i*4 */
i/=2; /* i=i/2 */
i%=3; /* i=i%3 */
```

Pour finir, ajoutons que les opérateurs qui servent à comparer 2 variables sont :

`==`	*égal à*	`!=`	*différent de*
`<`	*inférieur*	`<=`	*inférieur ou égal*
`>`	*supérieur*	`>=`	*supérieur ou égal*
`&&`	*’et’ logique*	`||`	*’ou’ logique*

ATTENTION !
Ne pas confondre l’opérateur d’affectation = et l’opérateur de comparaison ==.

Chapitre 3

Quelques fonctions indispensables

3.1 La fonction `printf()`

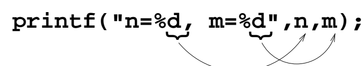
Elle sert à afficher à l'écran la chaîne de caractère donnée en argument, c'est-à-dire entre parenthèses.
`printf("Bonjour\n");` affichera Bonjour à l'écran.
Certains caractères ont un comportement spécial :

`\n`	retour à la ligne
`\b`	n'imprime pas la lettre précédente
`\r`	n'imprime pas tout ce qui est avant
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\"`	"
`\'`	'
`\?`	?
`\!`	!
`\\`	\

Mais `printf()` permet surtout d'afficher à l'écran la valeur d'une variable :

```
main(){
    int n=3, m=4;
    printf("%d",n); /* affiche la valeur de n au format d (decimal) */
    printf("n=%d",n); /* affiche 'n=3' */
    printf("n=%d, m=%d",n,m); /* affiche 'n=3, m=4' */
    printf("n=%5d",n); /* affiche la valeur de n sur 5 caracteres : 'n=     3'
*/
}
```

Le caractère `%` indique le format d'écriture à lécran. Dès qu'un format est rencontré dans la chaîne de caractère entre `" "`, le programme affiche la valeur de l'argument correspondant.



ATTENTION ! le compilateur n'empêche pas d'écrire un `char` sous le format d'un réel $\Rightarrow$ affichage de valeurs délirantes. Et si on écrit un `char` avec un format décimal, on affiche la valeur du code ASCII du caractère.

TAB. 3.1 – Tableau des formats utilisables

`%d`	`integer`	*entier (décimal)*
`%u`	`unsigned`	*entier non signé (positif)*
`%hd`	`short`	*entier court*
`%d`	`long`	*entier long*
`%f`	`float`	*réel, notation avec le point décimal (ex. 123.15)*
`%e`	`float`	*réel, notation exponentielle (ex. 0.12315E+03)*
`%lf`	`double`	*réel en double précision, notation avec le point décimal*
`%le`	`double`	*réel en double précision, notation exponentielle*
`%c`	`char`	*caractère*
`%s`	`char`	*chaîne de caractères*

Remarque : une chaîne de caractères est un tableau de caractères. Elle se déclare de la façon suivante : `char p[10];`. Mais nous reviendrons sur la notion de tableau plus tard.

3.2 La fonction `scanf()`

Dans un programme, on peut vouloir qu'une variable n'ait pas la même valeur à chaque exécution. La fonction `scanf()` est faite pour cela. Elle permet de lire la valeur que l'utilisateur rentre au clavier et de la stocker dans la variable donnée en argument.

Elle s'utilise ainsi :

```
main(){
    int a;
    scanf("%d",&a);
}
```

On retrouve les formats de lecture précisés entre `" "` utilisés pour `printf()`. Pour éviter tout risque d'erreur, on lit et on écrit une même variable avec le même format.

Le **&** est indispensable pour le bon fonctionnement de la fonction. Il indique l'adresse de la variable, mais nous reviendrons sur cette notion d'adresse quand nous aborderons les pointeurs.

3.3 La librairie `string.h`

La librairie `string.h` fournit un certain nombres de fonctions très utiles pour manipuler des chaînes de caractères en C. En effet, le C ne sait faire que des affectations et des comparaisons pour 1 seul caractère.

```
char p,q;                        /* p et q sont des caractères */
char chaine1[10], chaine[10];    /* chaine1 et chaine2 sont des chaînes */
                                 /* de caractères */
p = 'A';                 /* instruction valide */
chaine1 = "Bonjour";     /* instruction NON valide (1) */
chaine2 = "Hello";       /* instruction NON valide (2) */
if (p == q);             /* instruction valide */
if (chaine1 == chaine2)  /* instruction NON valide (3) */
```

Pour faire les affectations (1) et (2), et la comparaison (3), il faudrait donc procéder caractère par caractère. Ces opérations étant longues et sans intérêt, on utilise les fonctions déjà faites dans `string.h`. Pour les opérations d'affectation `(1) et (2)`, il faut utiliser la fonction `strcpy` (string copy), et pour une comparaison `(3)`, la fonction `strcmp` (string compare).

```
#include<stdio.h>
#include<string.h>
main(){
   char chaine1[10], chaine2[10];
   int a;
   strcpy(chaine1,"Bonjour"); /* met "Bonjour" dans chaine1 */
   strcpy(chaine2,"Hello");   /* met "Hello" dans chaine2 */
   a=strcmp(chaine1,chaine2);
   /* a reçoit la différence chaine1 et chaine2 */
   /* si chaine1 est classé alphabétiquement avant chaine2, alors a<0 */
   /* si chaine1 est classé après chaine2, alors a>0 */
   /* si chaine1 = chaine2, alors a = 0 */
   /* Ici, "Bonjour" est alphabétiquement avant "Hello", */
   /* donc chaine1 est plus petite que chaine2, et a < 0 */
}
```

Chapitre 4

Les instructions de contrôle

Ce sont des instructions qui permettent de notamment faire des tests et des boucles dans un programme. Leur rôle est donc essentiel. Pour chaque type d'instruction de contrôle, on trouvera à la fin de la partie 4 les organigrammes correspondant aux exemples donnés.

4.1 Les instructions conditionnelles

4.1.1 Les tests `if`

L'opérateur de test s'utilise comme suit :

```
if (expression) then {instruction;}
/* Si expression est vraie alors instruction est executee */
```

```
if (expression) {
   instruction 1;
   } else {
   instruction 2;
}
/* Si expression est vraie alors l'instruction 1 est executee */
/* Sinon, c'est l'instruction 2 qui est executee */
```

```
if (expression 1) {
   instruction 1;
} else if (expression 2){
   instruction 2;
} else if (expression 3){
   instruction 3;
} else {
   instruction 4; }
/* Souvent, on imbrique les tests les uns dans les autres */
```

Remarque : les instructions à éxécuter peuvent être des instructions simples {`a=b;`} ou un bloc d'instructions `{a=b; c=d; ...}`.

Comme nous l'avons déjà vu, une expression est vraie si la valeur qu'elle renvoie est non nulle.

ATTENTION ! les expressions `(a=b)` et `(a==b)` sont différentes :

`if (a==b)`	*vrai si a et b sont égaux*
`int b=1;`	
`if (a=b)`	*on met la valeur de b dans a. a vaut alors 1. l'expression est donc vraie*

4.1.2 Les tables de branchement : `switch`

Cette instruction s'utilise quand un **entier** ou un caractère prend un nombre fini de valeurs et que chaque valeur implique une instruction différente.

```
switch(i) {
   case 1 : instruction 1; /* si i=1 on exécute l'instruction 1 */
          break; /* et on sort du switch */
   case 2 : instruction 2; /* si i=2 ... */
          break;
   case 10 : instruction 3; /* si i=10 ... */
          break;
   default : instruction 4; /* pour les autres valeurs de i */
          break;
}
```

ATTENTION ! on peut ne pas mettre les `break;` dans les blocs d'instructions. Mais alors on ne sort pas du `switch`, et on exécute toutes les instructions des `case` suivants, jusqu'à rencontrer un `break;`.
Si on reprend l'exemple précédent en enlevant tous les `break`, alors
⋆ si i=1 on exécute les instructions 1, 2, 3 et 4
⋆ si i=2 on exécute les instructions 2, 3 et 4
⋆ si i=10 on exécute les instructions 3 et 4
⋆ pour toutes les autres valeurs de i, on n'exécute que l'instruction 4.

4.2 Les boucles

4.2.1 La boucle `for`

Elle permet d'exécuter des instructions plusieurs fois sans avoir à écrire toutes les itérations. Dans ce genre de boucle, on doit savoir le nombre d'itérations avant d'6etre dans la boucle. Elle s'utilise ainsi :

```
for (i=0; i<N; i++) {
   instructions ...;
}
```

Dans cette boucle, `i` est le **compteur**. Il ne doit pas être modifié dans les instuctions, sous peine de sortie de boucle et donc d'erreur.
⋆ La $1^{ère}$ instruction entre parenthèses est l'initialisation de la boucle.
⋆ La $2^{ème}$ est la condition de sortie de la boucle : tant qu'elle est vraie, on continue la boucle. ⋆ Et la $3^{ème}$ est l'instruction d'itération : sans elle, le compteur reste à la valeur initiale et on ne sort jamais de la boucle.

4.2.2 La boucle `while`

Contrairement à la boucke `for`, on n'est pas obligés ici de connaître le nombre d'itérations. Il n'y a pas de compteur.

```
while (expression) {
    instructions ...;
}
```

L'expression est évaluée à chaque itération. Tant qu'elle est vraie, les instructions sont exécutées. Si dès le début elle est fausse, les instructions ne sont jamais exécutées.

ATTENTION ! Si rien ne vient modifier l'expression dans les instructions, on a alors fait une boucle infinie : `while (1) { instructions }` en est un exmple.
Exemple d'utilisation :

```
#include <stdio.h>
main(){
    float x,R;
    x=1.0;
    R=1.e5;
    while (x < R) {
      x = x+0.1*x;
      printf("x=%f",x);
    }
}
```

4.2.3 La boucle `do ... while`

À la différence d'une boucle `while`, les instructions sont exécutées au moins une fois : l'expression est évaluée en fin d'itération.

```
do {
    instructions ...;
} while (expression)
```

Les risques des faire une boucle infinie sont les mêmes que pour une boucle `while`.

4.2.4 Les instructions `break` et `continue`

`break` fait sortir de la boucle.

`continue` fait passer la boucle à l'itération suivante.

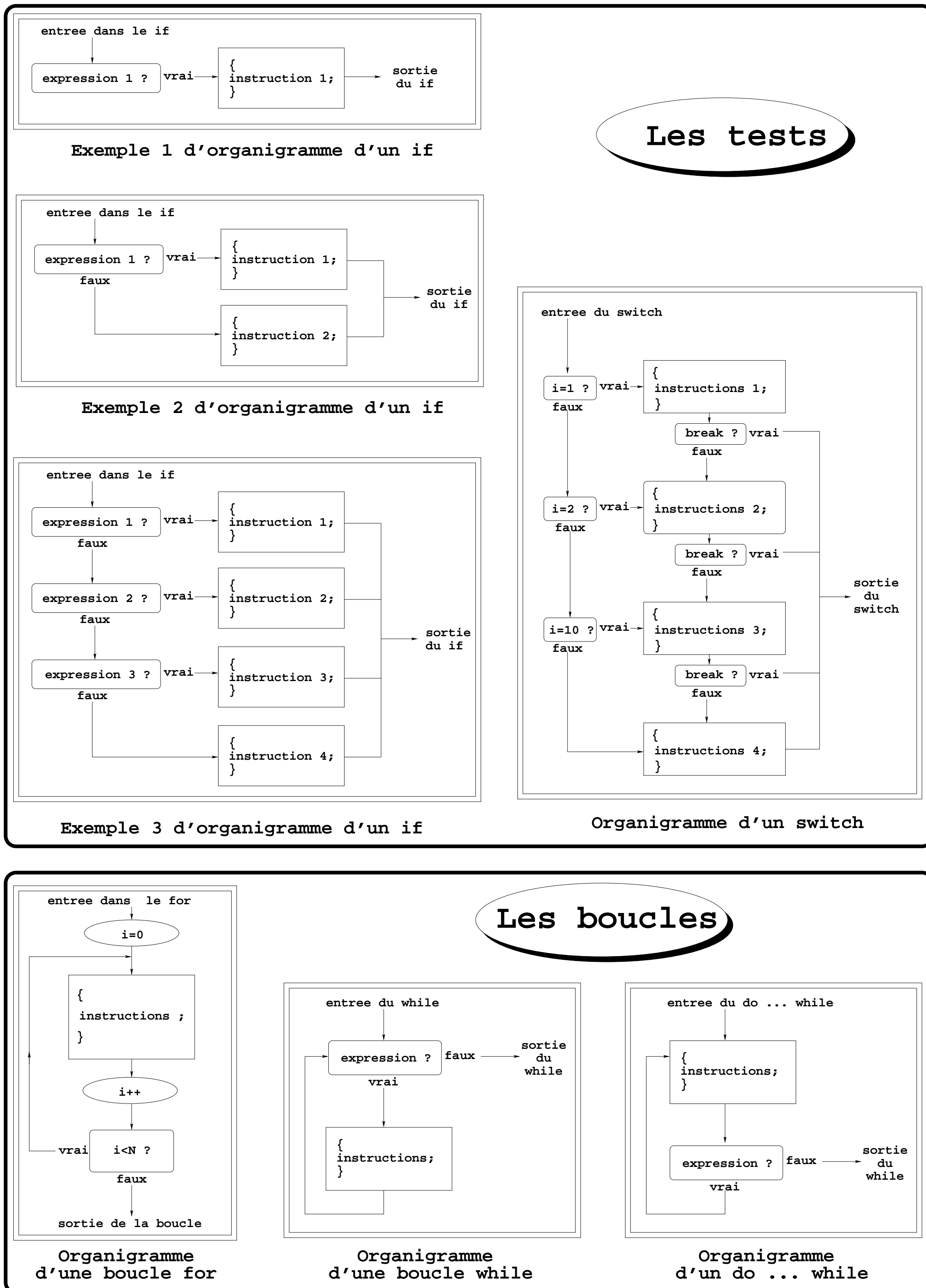


FIG. 4.1 – Organigramme récapitulatif des structures de contrôle.

Chapitre 5

Les fonctions

Créer une fonction est utile quand vous avez à faire le même type de calcul plusieurs fois dans le programme, mais avec des valeurs différentes. Typiquement, pour le calcul de l'intégrale d'une fonction mathématique f. Comme en mathématique, une fonction prend un ou plusieurs arguments entre parenthèses et renvoie une valeur.

5.1 Déclaration

On déclare une fonction ainsi :

```
type nom( type1 arg1 , type2 arg2, ... , typen argn) { */ prototype */
   déclaration variables locales;
   instructions;
   return (expression);
}
```

`type` est le type de la valeur renvoyée par la fonction. `type1` est le type du 1er argument `arg1` ... . Les variables locales sont des variables qui ne sont connues qu'à l'intérieur de la fonction. `expression` est évaluée lors de l'instruction `return (expression);`, c'est la valeur que renvoie la fonction quand elle est appelée depuis `main()`.

La 1ère ligne de la déclaration est appelée le **prototype** de la fonction.
Exemple de fonction :

```
float affine( float x ) { /* la fonction 'affine' prend 1 argument réel */
                          /* et renvoie un argument réel */
   int a,b;
   a=3;
   b=5;
   return (a*x+b); /* valeur renvoyee par la fonction */
}
```

On n'est pas obligés de déclarer des variables locales :

```
float norme( int x, int y ){ /* la fonction 'norme' prend 2 arguments */
                              /* entiers et renvoie un résultat réel */
   return (sqrt(x*x+y*y)); /* sqrt est la fonction racine carree */
}
```

On peut mettre plusieurs instructions `return` dans la fonction :

```
float val_absolue( float x ) {
   if (x < 0) {
        return (-x);
   } else {
        return (x);
   }
}
```

Une fonction peut ne pas prendre d'argument, dans ce cas-là, on met à la place de la déclaration des arguments, le mot-clé `void` :

```
double pi( void ) { /* pas d'arguments */
   return(3.14159);
}
```

Une fonction peut aussi ne pas renvoyer de valeur, son type sera alors `void` :

```
void mess_err( void ) {
   printf("Vous n\'avez fait aucune erreur\n");
   return; /* pas d'expression après le return */
}
```

5.2 Appel de la fonction

Une fonction `f()` peut être appelée depuis le programme principal `main()` ou bien depuis une autre fonction `g()` à la condition de rappeler le prototype de `f()` après la déclaration des variables de `main()` ou `g()` :

```
#include <stdio.h>
main(){
    int x,y,r;
    int plus( int x, int y );
    x=5;
    y=235;
    r=plus(x,y); /* appel d'une fonction avec arguments */
}

int plus( int x, int y ){
    void mess( void );
    mess_err(); /* appel d'une fonction sans arguments */
    return (x+y));
}

void mess( void ) {
    printf("Vous n\'avez fait aucune erreur\n");
    return;
}
```

Quand le programme rencontre l'instruction `return`, l'appel de la fonction est terminé. Toute instruction située après lui sera ignorée.

Chapitre 6

Les tableaux

6.1 Déclaration

Comme une variable, on déclare son type, puis son nom. On rajoute le nombre d'éléments du tableau entre crochets `[ ]` :
`float tab[5] ;` est un tableau de 5 flottants.
`int tablo[8] ;` est un tableau de 8 entiers.

ATTENTION !

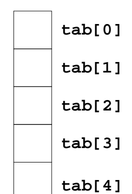
⋆ Les numéros des éléments d'un tableau de n éléments vont de 0 a $n-1$.
⋆ La taille du tableau doit être une constante (par opposition à variable), donc `int t1[n] ;` où `n` serait une variable déjà déclarée est une mauvaise déclaration. Par contre si on a défini `#define N 100` en directive, on peut déclarer `int t1[N] ;` car `N` est alors une constante.

On peut initialiser un tableau lors de sa déclaration :

```
float tab[5] = { 1, 2, 3, 4, 5}; /* init. de tous les éléments de tab */
float toto[10] = {2, 4, 6}; /* equ. à toto[0]=2; toto[1]=4; toto[2]=6; */
                            /* les autres éléments de toto sont mis à 0. */
```

6.2 Utilisation

Comme shématisé ci-contre, on accède à l'élément i d'un tableau en faisant suivre le nom du tableau par le numéro i entre crochets. Un élément de tableau s'utilise comme une variable quelconque. On peut donc faire référence à un élément pour une affectation : `x=tab[2]`, `tab[3]=3`, ou dans une expression : `if (tab[i] < 0)`.



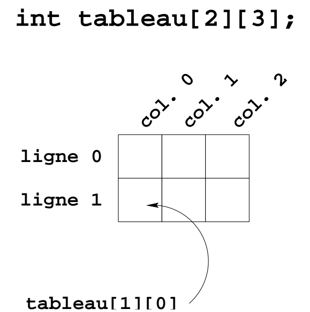
6.3 Cas d'un tableau de caractères

Un tableau de caractères est en fait une **chaîne de caractères**. Son initialisation peut se faire de plusieurs façons :

```
char p1[10]='B','o','n','j','o','u','r' ;
char p2[10]="Bonjour"; /* init. par une chaîne litterale */
char p3[ ]="Bonjour"; /* p3 aura alors 8 éléments */
```

ATTENTION ! Le compilateur rajoute toujours un caractère *null* à la fin d'une chaîne de caractères. Il faut donc que le tableau ait au moins un élément de plus que la chaîne litterale.

6.4 Tableau à 2 dimensions



Un tableau à 2 dimensions est similaire à une matrice. C'est en fait un tableau de tableau à 1 dimension, il se déclare donc de la façon suivante :
`int tableau[2][3]; /* tableau de 2 lignes et 3 colonnes */`
Comme il est shématisé ci-contre, `tableau[i][j]` fait référence à l'élément de la ligne *i* et de la colonne *i* de `tableau`. Tout comme un élément d'un tableau à 1 dimension, `tableau[i][j]` se manipule comme n'importe quelle variable.

Chapitre 7

Les structures

Les structures permettent de rassembler des valeurs de type différent. Par exemple, pour une adresse, on a besoin du numéro (`int`) et du nom de la rue (`char`).

7.1 Déclaration

On déclare une structure ainsi :

```
struct adresse {
   int numero;
   char rue[50];
};
```

Chaque élément déclaré à l'intérieur de la structure est appelé un **champ**. Le nom donné à la structure est appelé **étiquette de structure**. Ici, on a en fait déclaré un type de structure, pas une variable.

On déclare les variables associées à une structure de cette manière :

```
struct adresse chez_pierre , chez_julie;
```

Car si la structure d'une adresse est toujours la même (numéro et nom de la rue), `chez_pierre` et `chez_julie`, qui sont des `struct adresse` différentes, n'habitent pas au même endroit.

On peut initialiser une structure lors de sa déclaration :

```
struct adresse chez_pierre={ 15 , "rue_Dugommier" };
```

7.2 Manipulation

On accède aux données contenues dans les champs d'une structure et faisant suivre le nom de la strucure par un point "." et le nom du champ voulu :

```
chez_julie.numero=19;
strcpy(chez_julie.rue,"avenue Pasteur");
```

Si 2 structures ont le même type, on peut effectuer :

```
chez_pierre=chez_julie; /* Pierre a emménagé chez Julie! */
```

Mais on ne peut pas comparer 2 structures (avec == ou !=).

7.3 Tableau de structure

On déclare un tableau de structure de la même façon qu'un tableau de variables simples : le nombre d'éléments est précisé entre crochets.

```
struct adresse pers[100];
```

Cette déclaration nécessite que la structure `adresse` ait déjà été déclarée avant. `pers` est alors un tableau dont chaque élément est une structure de type `adresse`. Et `pers[i].rue` fait reférence au champ "rue" de la $i^{ème}$ personne de la structure `pers` .

7.4 Structure de structure

On peut utiliser une structure comme champ d'une autre structure. Dans la lignée des exemples précédents, on peut définir une structure `adresse` qui pourra être utilisée dans la structure `repertoire`. Elle peut également être utilisée dans une autre structure.

```
struct adresse {
    int numero;
    char rue[50]; };
struct repertoire {
    char nom[20];
    char prenom[20];
    struct adresse maison; }; /* déclaration d'un champ structure */
                              /* de type 'adresse' appelé 'maison' */
struct repertoire monrepertoire[100];

strcpy(monrepertoire[0].nom,"Cordier");
strcpy(monrepertoire[0].prenom,"Julie");
monrepertoire[0].maison.numero = 19;
strcpy(monrepertoire[0].maison.rue,"avenue_Pasteur");

strcpy(monrepertoire[1].nom,"Durand");
strcpy(monrepertoire[1].prenom,"Pierre");
monrepertoire[1].maison.numero = 15;
strcpy(monrepertoire[1].maison.rue,"rue_Dugommier");
```

Lorsqu'un tableau fait partie des champs d'une structure, on peut accéder aux valeurs de ce tableau par :

```
char initiale;
initiale=monrepertoire[1].prenom[0]; /* initiale de Pierre */
```

Chapitre 8

Les pointeurs

8.1 Stockage des variables en mémoire

Lors de la compiltaion d'un programme, l'orinateur réserve dans sa mémoire une place pour chaque variable déclarée. C'est à cet endroit que la valeur de la variable est stockée. Il associe alors au nom de la variable l'adresse de stockage. Ainsi, pendant le déroulement du programme, quand il rencontre un nom de variable, il va chercher à l'adresse correspondante la valeur en mémoire.

Pour les déclaractions de variables suivantes :

```
int a=0xa; /* 'a' est un entier codé sur 4 octets */
short b=0x0; /* 'b' est un entier codé sur 2 octets */
int c=0x14; /* 'c' est un entier codé sur 4 octets */
```

ceci est stocké en mémoire :

nom	*adresse en hexa*	*valeur codée en hexadécimal*			
a	(bfbff000)	00	00	00	0a
b	(bfbff004)	00	00		
c	(bfbff006)	00	00	00	14

Ici, on suppose que l'espace mémoire servant à stocker les données commence à l'adresse `(bfbff000)`. Chaque case représente 1 octet.

Explication du shéma :

⋆ `a` est codé sur 4 octets et son adresse est `(bfbff000)`, donc l'adresse de `b` sera
`(bfbff000)+(4)=(bfbff004)`.

⋆ `b` est codé sur 8 octets et son adresse est `(20004)`, donc l'adresse de `c` sera
`(bfbff004)+(2)=(bfbff006)`.

8.2 Définition et déclaration d'un pointeur

Un pointeur est une variable qui a pour valeur l'adresse d'une autre variable : celle sur laquelle elle pointe ! Un pointeur est toujours associé à un type de variable et un seul. Au moment de la déclaration, on détermine le type de variable pointé par le pointeur, en écrivant le type concerné, puis le nom du pointeur avec une `*` devant :

```
int *pta; /* la variable pta est un pointeur sur un entier*/
int a; /* la variable a est un entier*/
```

8.3 Opérateur d'adresse : &

Pour affecter l'adresse de la variable `a` au pointeur `pta`, on écrit l'instruction :

```
pta=&a;
```

Cet opérateur signifie donc **adresse de**.

8.4 Opérateur d'indirection : *

Cet opérateur, mis en préfixe d'un nom de pointeur signifie **valeur de la variable pointée** ou, plus simplement, valeur pointée.

```
int a=1;
int *ptint; /* declaration d'un pointeur sur un entier */
ptint=&a; /* ptint pointe sur a */
*ptint=12; /* la variable pointée par ptint reçoit 12*/
printf("a=%d \n",a); /* affiche "a=12" */
```

En fait, manipuler `ptint` revient à manipuler `a`.

8.5 Mémoire et pointeurs

On reprend l'exemple de la partie 8.1 en ajoutant un pointeur d'entier :

```
int a=0xa;
short b=0x0;
int c=0x14;
int *ptint;
ptint=&a;
```

Etat de la mémoire :

nom	*adresse*	*valeur codée en hexadécimal*			
a	(bfbff000)	00	00	00	0a
b	(bfbff004)	00	00		
c	(bfbff006)	00	00	00	14
ptint	(bfbff010)	bf	bf	f0	00

Explication : Les cases mémoire des variables a, b et c contiennent leur valeur. Celles de la variable ptint contiennent l'adresse de la valeur pointée. En effet, la valeur stockée est (bfbff000), ce qui est bien l'adresse de a.

8.6 Exemple

Voici un petit exemple d'illustration :

```
#include <stdio.h>
main(){

float *px;
float x=3.5;

px=&x;
printf ("adresse de x : 0x%lx \n",&x);
printf ("valeur de px : 0x%lx \n",px);
printf ("valeur de x : %3.1f \n",x);
printf ("valeur pointee par px : %3.1f \n",*px);
return 0;
}
```

Le programme précédent affichera ceci à l'écran :

```
$ adresse de x : 0xbfbffa3c
$ valeur de px : 0xbfbffa3c
$ valeur de x : 3.5
$ valeur pointee par px : 3.5
```

Chapitre 9

Pointeurs et fonctions

Un variable globale est une variable connue dans toutes les fonctions d'un programme (main comme les autres).

Une variable locale n'est connue qu'à l'intérieur d'une fonction (main ou une autre).

Les variables locales d'une fonction sont regroupées dans une partie de la mémoire, et celles d'une autre fonction, dans un autre endroit. Ainsi, il peut exister un `float a;` dans une fonction et un `int a` dans une autre sans qu'il y ait de conflit.

Une conscéquence de cette propriété est que la fonction suivante ne marchera pas :

```
void permute (int a , int b){
int buf;
buf = a;
a = b;
b = buf;
return;
}
```

car lors de l'appel de cette fonction depuis `main`, les valeurs des arguments vont être copiés dans les variables de `permute` et ce sont ces variables locales qui vont être modifiées, pas celles de `main`. Ainsi, dans `main`, un appel du type `permute(i,j)` laissera `i` et `j` inchangés. On dit que le C passe ses arguments **par valeur**.

Pour que `permute` fonctionne, il faut que ses arguments soient les adresses des variables `a` et `b` et utiliser des pointeurs.

```
void permute (int *a , int *b){
int buf;
buf = *a;
*a = *b;
*b = buf;
return;
}
```

Lors de l'appel de la fonction, les pointeurs locaux vont recevoir l'adresse des variables `a` et `b`. Donc travailler sur ces pointeurs revient à travailler sur les variables `a` et `b` de la fonction `main`.

L'appel d'une telle fonction se fait ainsi : `permute (&a ,&b)`

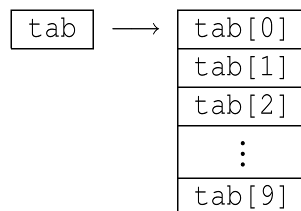
De façon générale, on utilise des pointeurs avec les fonctions quand on veut qu'une fonction modifie des variables du programme principal.

Chapitre 10

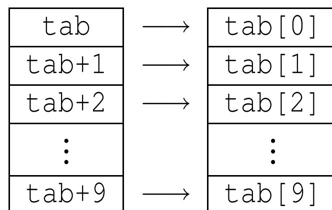
Pointeurs et tableaux

10.1 Pointeurs et tableaux à 1 dimension

Vous avez déja manipulé des pointeurs quand vous avez manipulé les tableaux. En fait, le nom seul du tableau est une constante qui contient l'adresse du premier élément du tableau comme le shématise la figure suivante :



Ainsi, `tab` est égal à `&tab[0]` et donc `*tab` est égal à `tab[0]`.
L'élément `tab[i]` est équivalent à `*(tab+i)`. On a donc les correspondances suivantes :



Au niveau de la mémoire, pour les déclarations suivantes,

```
int tab[3] = { 0xa , 0x3 , 0xd };
int *pta;
int *ptb;
pta = tab;
ptb = pta+2;
```

voici l'état de la mémoire :

nom	*adresse*	*valeur codée en hexadécimal*			
tab	(bfbff000)	00	00	00	0a
	(bfbff004)	00	00	00	03
	(bfbff008)	00	00	00	0d
pta	(bfbff010)	bf	bf	f0	00
ptb	(bfbff014)	bf	bf	f0	08

Explications : L'ordinateur a réservé en mémoire 3 fois 4 octets pour le tableau de 3 entiers `tab`. Le pointeur `pta` contient l'adresse du 1er élément de `tab` : `(bfbff000)`. L'opération `ptb=pta+2` n'ajoute pas `2` à la valeur de `pta`, mais ajoute 2 fois le nombre d'octets correspondant à un `int`, puisque `ptb` est un pointeur d'entiers. Donc `ptb` vaut `(bfbff000)+(2*4)=(bfbff008)`. Et `ptb` contient alors l'adresse de `tab[2]`.

Ainsi, si on déclare

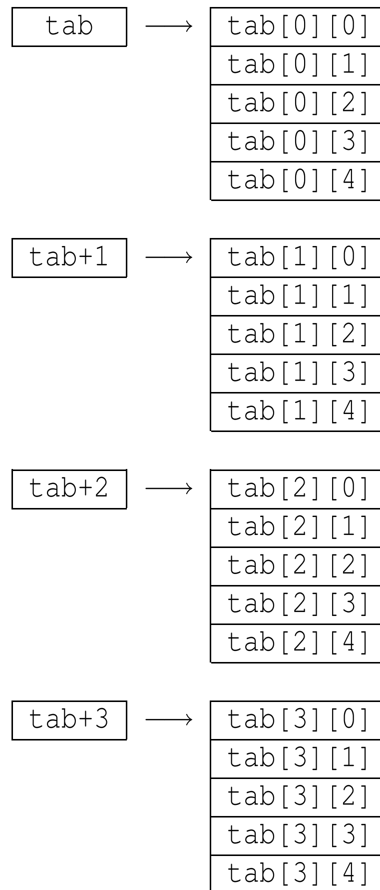
```
int tab[10];
int *pt;
pt = tab;
```

on aura les équivalences suivantes :

`*tab`	⟷	`tab[0]`	⟷	`*pt`	⟷	`pt[0]`
`*(tab+1)`	⟷	`tab[1]`	⟷	`*(pt+1)`	⟷	`pt[1]`
et de façon plus générale, pour i de 0 à 9 :						
`*(tab+i)`	⟷	`tab[i]`	⟷	`*(pt+i)`	⟷	`pt[i]`

10.2 Pointeurs et tableaux à plusieurs dimensions

Un tableau à plusieurs dimensions est un tableau dont les éléments sont eux-mêmes des tableaux. Ainsi, le tableau défini par `int tab[4][5];` contient 4 tableaux de 5 entiers chacuns. `tab` donne l'adresse du 1er sous-tableau, `tab+1` celle du 2ème sous-tableau et ainsi de suite :



Ici, l'opération `tab+2` n'ajoute pas `2` à la valeur de `tab` mais ajoute 2 fois le nombre d'octets correspondant à un tableau de 5 entiers, à savoir `5*4 = 20` octets.

10.3 Tableaux de pointeurs

La déclaration d'un tableau de pointeurs se fait comme pour un tableau de variables quelconques : le type puis le nom du tableau avec

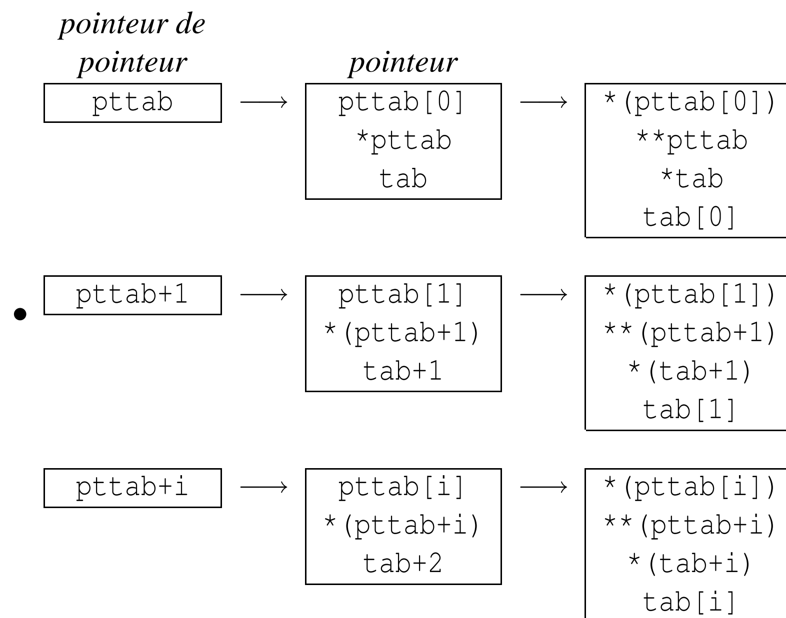
- le nombre d'éléments entre crochets derrière le nom
  et une `*` devant.

```
int *pttab[4]; /* pttab est un tableau de 4 pointeurs d'entiers */
```

1er exemple d'utilisation d'un tableau de pointeurs

```
int tab[4]; /* tab est un tableau de 4 entiers */
int *pttab[4] = { tab , tab+1 , tab+2 , tab+3 };
```

Les valeurs du tableau `pttab` sont des adresses de données. On dit alors que `pttab` est un **pointeur de pointeurs** car `(pttab)` pointe sur l'adresse de son 1er élément, qui est lui-même une adresse. Voici les relations qu'il y a entre `pttab` et `tab` :

- 

Note : Les expressions se situant dans une même case sont équivalentes.
Au niveau de la mémoire, les éléments d'une colonne contiennent les adresses des éléments de la colonne qui est juste à sa droite.

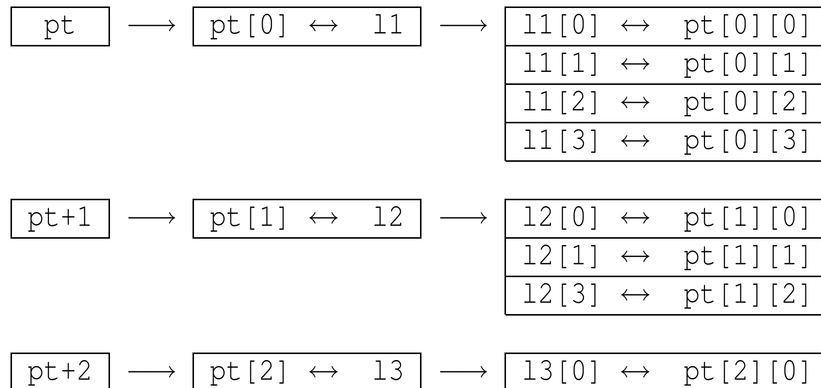
Voici l'état de la mémoire pour un tel exemple :

nom	*adresse*	*valeur codée en hexadécimal*			
tab	(bfbff000)	00	00	00	0a
	(bfbff004)	00	00	00	03
	(bfbff008)	00	00	00	0d
	(bfbff00c)	00	00	00	6c
pttab	(bfbff010)	bf	bf	f0	00
	(bfbff014)	bf	bf	f0	04
	(bfbff018)	bf	bf	f0	08
	(bfbff01c)	bf	bf	f0	0c

2ème exemple

```
int l1[4] = { 1 , 2 , 3 , 4 };
int l2[3] = { 5 , 6 , 7 };
int l3[1] = { 8 };
int *pt[3] = { l1 , l2 , l3 };
```

Les éléments de `tab` sont les adresses de tableaux ne comportant pas le même nombre d'éléments. `pt` est en fait un tableau dont les 3 lignes n'ont pas la même longueur.



État de la mémoire :

nom	*adresse*	*valeur codée en hexadécimal*			
l1	(bfbff000)	00	00	00	01
	(bfbff004)	00	00	00	02
	(bfbff008)	00	00	00	03
	(bfbff00c)	00	00	00	04
l2	(bfbff010)	00	00	00	05
	(bfbff014)	00	00	00	06
	(bfbff018)	00	00	00	07
l3	(bfbff01c)	00	00	00	08
pt	(bfbff020)	bf	bf	f0	00
	(bfbff024)	bf	bf	f0	10
	(bfbff028)	bf	bf	f0	1c

Chapitre 11

Allocation dynamique de mémoire

Jusqu'à maintenant, lors de la déclaration d'un tableau, il fallait préciser les dimensions, soit de façon explicite :

```
int tab[3][2];
```

soit de façon implicite :

```
int tab[][] = { {0 , 1 } , {2 , 3 } , {4 , 5 } };
```

Dans les 2 cas, on a déclaré un tableau de 3 fois 2 entiers.

Mais si l'on veut que le tableau change de taille d'une exécution à une autre, cela nous oblige à modifier le programme et à le recompiler à chaque fois, ou bien à déclarer un tableau de 1000 fois 1000 entiers et n'utiliser que les *n* premières cases mais ce serait du gâchis.
Pour éviter cela, on fait appel à l'**allocation dynamique de mémoire** : au lieu de réserver de la place lors de la compilation, on la réserve pendant l'exécution du programme, de façon interative.

11.1 La fonction `malloc()`

Pour l'utiliser il faut inclure la bibliothèque `<stdlib.h>` en début de programme.
`malloc( N )` renvoie l'adresse d'un bloc de mémoire de `N` octets libres (ou la valeur `0` s'il n'y a pas assez de mémoire).

Exemple :

```
int *p;
p = malloc(800); /* fournit l'adresse d'un bloc de 800 octets libres */
               /* et l'affecte au pointeur p */
```

11.2 L'opérateur `sizeof()`

D'une machine à une autre, la taille réservée pour un `int`, un `float`,... change. Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine á l'autre, nous avons besoin de la taille effective d'une donnée de ce type.
L'opérateur `sizeof` nous fournit ce renseignement.
`sizeof nom_variable` fournit la taille de la variable *nom_variable*

`sizeof nom_constante` ournit la taille de la constante *nom_constante*
`sizeof (type)` fournit la taille pour un objet du type *type*
Ainsi, les instructions suivantes :

```
int a[10];
char b[5][10];
printf("taille de a : %d\n",sizeof a);
printf("taille de b : %d\n",sizeof b);
printf("taille de 4.25 : %d\n",sizeof 4.25);
printf("taille de Bonjour ! : %d\n",sizeof "Bonjour !");
printf("taille d'un float : %d\n",sizeof(float));
printf("taille d'un double : %d\n",sizeof(double));
```

produiront à l'exécution sur un PC :

```
taille de a : 40
taille de b : 40
taille de 4.25 : 8
taille de Bonjour ! : 9
taille d'un float : 4
taille d'un double : 8
```

11.3 Allocation dynamique pour un tableau à 1 dimension

On veut réserver de la place pour un tableau de `n` entiers, où `n` est lu au clavier :

```
int n;
int *tab;
printf("taille du tableau :\n");
scanf("%d", &n);
tab = (int *)malloc(n*sizeof(int));
```

Les `(int *)` devant le `malloc` s'appelle un **cast**. Un cast est un opérateur qui convertit ce qui suit selon le type précisé entre parenthèses.
Exemple :

```
int n1 , n2;
float x = 4.5;
n1 = ((int) x)*10;
n2 = (int)(x*10);
```

Pour `n1`, on convertit d'abord `x` en entier et on multiplie le résultat par 10 : n1 = 40
Pour `n2`, on convertit en entier `x*10` : n2 = 45

La fonction `malloc` renvoie juste l'adresse de début d'un bloc de `n` fois `sizeof(int)`. Elle renverra donc la même chose pour un tableau de 400 `char` que pour 100 `int` : l'adresse d'un bloc de 400 octets. C'est pourquoi le **cast** est nécessaire : pour préciser le type de données sur leqsuelles `tab` va pointer.

`tab` contient alors l'adresse de début d'un bloc de `n` entiers et on accède à la i^{ème} valeur du tableau par `tab[i]`.

Jusqu'à maintenant, on a vu des pointeurs qui contenaient l'adresse d'une variable en mémoire. Ici, on a l'exemple d'un pointeur qui contient l'adresse d'un bloc contenant des données. Celles-ci ne sont accessibles que via un pointeur.

Ètat de la mémoire avant l'allocation :

nom	*adresse*	*valeur codée en hexadécimal*				
n	(bfbff000)	00	00	00	04	l'utilisateur a entre 4 comme valeur de n
tab	(bfbff004)	00	00	00	00	

Ètat de la mémoire après l'allocation :

nom	*adresse*	*valeur codée en hexadécimal*				
n	(bfbff000)	00	00	00	04	
tab	(bfbff004)	80	00	00	00	
	(80000000)	00	00	00	00	(tab[0])
	(80000004)	00	00	00	00	(tab[1])
	(80000008)	00	00	00	00	(tab[2])
	(8000000c)	00	00	00	00	(tab[3])

11.4 Allocation dynamique pour un tableau à plusieurs dimensions

On veut réserver de la place pour un tableau de `n` fois `m` entiers, où `n` et `m` sont lus au clavier : On a vu que pour manipuler des tableaux à plusieurs dimensions, il fallait utiliser des tableaux de pointeurs (*i.e.* des pointeurs de pointeurs).

```
int i,j,n,m;
float **tab; /* (1) */
scanf("%d%d",n,m);
tab = (float **)malloc(n*sizeof(float *)); /* (2) */
for (i=0; i<n; i++){
     tab[i] = (float *)malloc(m*sizeof(float)); /* (3) */
}
for (i=0; i<n; i++){
     for (j=0; j<m; j++){
          tab[i][j] = 10*i+j; /* (4) */
     }
}
```

Explications :

`(1)` Un tableau de pointeurs étant un pointeur de pointeurs, on peut déclarer au choix un tableau de pointeurs : `int *tab[3]` ou un pointeur de pointeur : `**tab`. Dans le cas de l'allocation dynamique de mémoire, comme on ne connait pas la taille du tableau dont on aura besoin, on déclare donc un pointeur de pointeur.

`(2)` `tab` étant en fait un tableau de pointeurs de flottants, on réserve un bloc pouvant contenir `n` pointeurs de `float`. `tab` contient alors l'adresse de ce bloc.

`(3)` Les `tab[i]` sont des sous-tableaux de `tab`. On réserve alors pour chacun d'eux de la place pour `m` flottants. Au total, on a bien réservé de la place pour `n` fois `m` flottants.

`(4)` On manipule les éléments de `tab` comme ceux d'un tableau "normal".

Ètat de la mémoire après l'étape `(2)` :

nom	*adresse*	*valeur codée en hexadécimal*			
n	(bfbff000)	00	00	00	03
m	(bfbff000)	00	00	00	02
tab	(bfbff004)	80	00	00	00
	(80000000)	00	00	00	00
	(80000004)	00	00	00	00
	(80000008)	00	00	00	00

Ètat de la mémoire à la fin du programme :

nom	*adresse*	*valeur codée en hexadécimal*				
n	(bfbff000)	00	00	00	03	
m	(bfbff000)	00	00	00	02	
tab	(bfbff004)	80	00	00	00	
	(80000000)	80	00	01	00	(adresse du ss-tableau tab[0])
	(80000004)	80	00	0a	00	(adresse du ss-tableau tab[1])
	(80000008)	80	00	0d	00	(adresse du ss-tableau tab[2])
	(80000100)	00	00	00	00	(valeur de tab[0][0])
	(80000104)	00	00	00	01	(valeur de tab[0][1])
	(80000a00)	00	00	00	0a	(valeur de tab[1][0])
	(80000a04)	00	00	00	0b	(valeur de tab[1][1])
	(80000d00)	00	00	00	14	(valeur de tab[2][0])
	(80000d04)	00	00	00	15	(valeur de tab[2][1])

Note : Les différentes allocations de mémoire ne se font pas en même temps. C'est pour cela que les différents blocs mémoire ne sont pas contigüs.