

## Compléments

### 12.1 Objectif

Ce chapitre vous propose d'aborder quelques notions plus complexes qu'il est nécessaire de comprendre mais où seule la pratique sera votre véritable alliée.

### 12.2 Conversions de type

Pour convertir un type en un autre, on réalise ce que l'on appelle un « cast ». Imaginons que nous souhaitions convertir un nombre du type `float` en un entier du type `int`. Voici comment procéder :

```
int main () {
    float f;
    int i;

    f=3.1415;
    i=(int) f; /* résultat dans i : 3 */
    /* donc la partie décimale est perdue... */
}
```

```

return 0 ;
}

```



Sans la conversion explicite, le programme aurait donné la même chose : un `float` est converti en `int` avant d'être stocké dans une variable de type `int` (le compilateur pourrait néanmoins émettre un avertissement).

## 12.3 Usage très utile des conversions de type

Considérons le programme suivant<sup>1</sup> :

```

int main () {
    printf("Résultat : %f",3/4);
    return 0 ;
}

```

Celui-ci affichera `Résultat : 0.0!!!`

En effet, l'opérateur « / » réalise une division entière de l'entier 3 par l'entier 4 ce qui vaut 0 (les deux opérands étant entières, le résultat est converti automatiquement en entier).

Il existe au moins deux solutions pour remédier à ce problème :

1. écrire `3.0` à la place de `3` ce qui va forcer le programme à faire une division en flottants :

```

#include <stdio.h>

int main () {
    printf("Résultat : %f",3.0/4);
    return 0 ;
}

```

2. convertir le nombre 3 en un flottant :

```

#include <stdio.h>

int main () {
    printf("Résultat : %f", (float)3/4);
    return 0 ;
}

```

La division se fera alors en flottants<sup>2</sup>.

1. Là aussi, vous pourriez avoir un avertissement du compilateur.

2. L'opérateur de cast (`<type>`) (où `<type>` est un type quelconque) est prioritaire sur les opérateurs binaires. L'expression `(float)3/4` est donc évaluée comme `((float)3)/4` et non comme `(float)(3/4)`.



On notera que la solution suivante ne fonctionnerait pas :

```
#include <stdio.h>

int main () {
    printf("Résultat : %f", (float) (3/4));
    return 0 ;
}
```

En effet, nous réalisons tout d'abord la division de 3 par 4 (avec pour résultat l'entier 0), puis la conversion de ce dernier en flottant.

Le résultat est donc :

**Résultat : 0.0**

## 12.4 Fonction putchar

Le programme suivant :

```
#include <stdio.h>

int main () {
    char c='A';
    putchar(c);

    return 0 ;
}
```

fait la même chose que :

```
#include <stdio.h>

int main () {
    char c='A';
    printf("%c",c);
    return 0 ;
}
```

Nous constatons que `putchar` affiche un unique caractère à l'écran.

## 12.5 Allocation dynamique de mémoire

### 12.5.1 Fonctions `malloc` et `sizeof`

La fonction `malloc`, déclarée dans `stdlib.h` permet de réserver un bloc mémoire au cours de l'exécution du programme.

Ainsi, `malloc(N)` fournit l'adresse d'un bloc en mémoire de `N` octets libres ou la valeur `NULL` (c'est à dire 0) s'il n'y a pas assez de mémoire.

Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de moins de 4000 caractères.

1. nous créons un pointeur `tab` vers un caractère (`char *tab`).
2. nous exécutons l'instruction : `tab = malloc(4000) ;` qui fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à `tab`.

S'il n'y a plus assez de mémoire, c'est la valeur `NULL` (c'est à dire 0) qui est affectée à `tab`.

Si nous souhaitons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine / d'un système à l'autre, nous aurons besoin de la grandeur effective d'une donnée de ce type. Il faut alors utiliser l'opérateur `sizeof` afin de préserver la portabilité du programme :

- `sizeof(<var>)` : fournit la taille, en octets, de la variable `<var>`
- `sizeof(<constante>)` : fournit la taille de la constante `<const>`
- `sizeof(<type>)` : fournit la taille pour un objet du type `<type>`

Exemple :

Soit la déclaration des variables suivantes :

```
short tab1[10];
char tab2[5][10];
```

Instructions	Valeurs retournées	Remarques
<code>sizeof(tab1)</code>	20	
<code>sizeof(tab2)</code>	50	
<code>sizeof(double)</code>	8	Généralement !
<code>sizeof("bonjour")</code>	8	Pensez au zéro final des chaînes
<code>sizeof(float)</code>	4	Généralement !

TABLE 12.1 - Déclaration de variables



Si nous souhaitons réserver de la mémoire pour `x` valeurs de type `int`, la valeur de `x` étant lue au clavier :

```
int x;
int *pNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &x);
pNum = malloc(x*sizeof(int));
```

Certains compilateurs imposent d'écrire

```
pNum = (int *) malloc(x*sizeof(int));
```

En effet, étant donné que `malloc` renvoie un pointeur quelconque (`void *`), nous devons convertir cette adresse par un *cast* en un pointeur sur un entier, d'où l'usage de `(int *)`.

Voici d'autres exemples :

```
char * pointeur_sur_chaine;
char * pointeur_sur_float;

pointeur_sur_chaine = (char *) malloc(1000*sizeof(char));
pointeur_sur_float = (float *) malloc(10000*sizeof(float));
```



Le programme suivant lit 20 phrases au clavier, recherche des blocs de mémoire libres assez grands pour la mémorisation et passe les adresses aux composantes du tableau `phrases[]`. S'il n'y a pas assez de mémoire pour une chaîne, le programme affiche un message d'erreur et s'interrompt avec le code d'erreur -1. Nous devons utiliser une variable d'aide `phrase` comme zone intermédiaire (non dynamique). Pour cette raison, la longueur maximale d'une phrase est fixée à 100 caractères<sup>1</sup>.

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

#define LONGUEUR_MAX_PHRASE 100 /* longueur maximum d'une phrase */
#define NOMBRE_MAX_PHRASES 20 /* nombre maximum de phrases */

int main() {

    char phrase[LONGUEUR_MAX_PHRASE];
    char *phrases[NOMBRE_MAX_PHRASES];

    int i;

    for (i=0; i<NOMBRE_MAX_PHRASES; i++) {
        printf("Entrez une phrase SVP...");
        gets(phrase);

        /* Réserve de la mémoire */
        phrases[i] = (char *) malloc(strlen(phrase)+1);

        /* S' il y a assez de mémoire, ... */
        if (phrases[i]!=NULL) {
            /* copier la phrase à l'adresse renvoyée par malloc, ... */
            strcpy(phrases[i],phrase);
        }
        else {
            /* sinon faire une sortie "catastrophe" */
            printf("Attention! Plus assez place en mémoire !!! \n");
            exit(-1);
        }
    }

    return 0;
}
```

1. À la fin de ce programme, nous devrions libérer la mémoire (voir paragraphes suivants).

## 12.5.2 Fonction free

Lorsque nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de `malloc`, nous pouvons le libérer à l'aide de la fonction `free` déclarée dans `<stdlib.h>`.

L'appel à `free(<Pointeur>)` libère le bloc de mémoire désigné par `<Pointeur>`. L'appel à la procédure n'a pas d'effet si le pointeur passé en paramètre possède la valeur zéro.



- La fonction `free` peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par `malloc`.
- La fonction `free` ne change pas le contenu du pointeur ; il est conseillé d'affecter la valeur `NULL` au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
- Si nous ne libérons pas explicitement la mémoire à l'aide de `free`, celle-ci peut être libérée automatiquement à la fin du programme (mais cette façon de faire est à éviter). Ce processus de libération de la mémoire dépend du système d'exploitation utilisé. Une telle fuite de mémoire s'appelle « Memory Leak ».



Par exemple :

```
char * pointeur_sur_chaine;
pointeur_sur_chaine = (char *) malloc(1000*sizeof(char));
...

/* à présent, nous n'avons plus besoin de cette zone mémoire : */
free(pointeur_sur_chaine);
pointeur_sur_chaine=NULL;
```

## 12.6 Avez-vous bien compris ceci ?

Considérons le programme suivant :

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i;
5.     int * p;
6.     printf ("%d\n", i);
7.     *p = 12;
8.     return 0;
9. }
```

- À la ligne 6, le programme affichera n'importe quoi car la variable `i` n'a pas été initialisée.
- La ligne 7 a de fortes chances de faire bugger le programme ! En effet `p` n'a pas été initialisé. En particulier si on faisait un `printf ("%p", p)`, on pourrait constater que sa valeur est indéfinie. En effet, `p`, n'est qu'un pointeur, c'est-à-dire une variable contenant une valeur. Donc `p` désigne une adresse quelconque qui peut être celle d'une autre variable. Ainsi, le bug sera dû au fait que `p` pointe n'importe où et que vous essayez d'initialiser ce n'importe où (qui ne vous appartient peut-être pas).

## 12.7 Sur l'utilité des pointeurs

### 12.7.1 Modifier une variable dans une fonction

Vous savez qu'une fonction n'est pas capable de modifier ses arguments :

```
#include <stdio.h>

void calcule_double (int x){
    x=x+x ;
}

main () {
    int i=2;
    calcule_double(i);
    printf("i vaut à présent :%d",i);/* i vaut toujours 2 !!! */
}
```

La solution consiste donc à faire (voir le déroulé plus loin).

```
#include <stdio.h>

void calcule_double (int * pointeur_int){
    * pointeur_int = (* pointeur_int) + (* pointeur_int) ;
}

main () {
    int i=2;
    calcule_double(&i);
    printf("i vaut à présent :%d",i); /* i vaut bien 4 !!! */
}
```

Imaginons que `pointeur_int` se trouve en mémoire logé à l'adresse 20 et `i` à l'adresse 10 :

Nom de la variable	i	pointeur_int
Contenu	<input type="text"/>	<input type="text"/>
Position en mémoire	10	20

FIGURE 12.1 - Adresse mémoire (a)

À présent, déroulons le programme principal :

1. `int i=2` : *déclaration et initialisation de i à 2*
2. `calcule_double(&i)` : *appel de la fonction calcule\_double, avec la valeur 10 (l'adresse de i)*
3. `void calcule_double (int * pointeur_int)` : *on lance cette fonction et pointeur\_int vaut donc 10*
4. `* pointeur_int = (* pointeur_int)+(* pointeur_int)` : *(\* pointeur\_int) pointe sur i. L'ancienne valeur de i va être écrasée par la nouvelle valeur : 2+2*
5. `printf("i vaut à présent : %d", i)` : *on se retrouve avec 4 comme valeur de i.*

Nom de la variable	i	pointeur_int
Contenu	2	10
Position en mémoire	10	20

FIGURE 12.2 - Adresse mémoire (b)

### 12.7.2 Saisie d'un nombre dans une fonction

Étudions le programme suivant dont le but est simplement de modifier la valeur de `n` :

```
#include <stdio.h>

void saisie (int *pointeur);

int main() {
    int n;
    saisie(&n);
    printf("n vaut : %d",n);

    return 0;
}

void saisie (int *pointeur) {
    printf("Entrez un nombre :");
    scanf ("%d",pointeur);
}
```

Imaginons que `pointeur` se trouve en mémoire logé à l'adresse 100 et `n` à l'adresse 1000 :

Nom de la variable	pointeur	n
Contenu		
Position en mémoire	100	1000

FIGURE 12.3 - Adresse mémoire (c)

À présent, déroulons le programme :

1. `int n` : déclaration de `n`. Il vaut n'importe quoi vu qu'il n'a pas été initialisé !
2. `saisie(&n)` : appel de la fonction `saisie`, avec la valeur 1000 (l'adresse de `n`)
3. `void saisie (int *pointeur)` : `pointeur` vaut donc 1000
4. `printf("Entrez un nombre :")`
5. `scanf ("%d",pointeur)` : la fonction `scanf` va stocker ce que l'utilisateur tape au clavier à partir de l'adresse mémoire 1000 (imaginons que l'on entre la valeur 42).
6. retour dans `main`, la valeur entrée (42) a été stockée à l'adresse 1000, donc dans `n`. Le programme affichera `n vaut : 42`

## 12.8 Un mot sur les warnings

Lorsque vous compilez, vous pouvez obtenir des erreurs, des *warnings* (avertissements) ou parfois les deux.

Les erreurs, vous connaissez : lorsque écrivez par exemple `floatt` au lieu de `float`, c'est une erreur. S'il y a des erreurs à la compilation (`gcc -o essai essai.c`), le compilateur ne générera pas de fichier exécutable (fichier `essai`). En revanche, s'il n'y a que des *warnings*, le fichier `essai` sera créé, mais le compilateur nous alerte au sujet d'une erreur possible.

Il est important de comprendre qu'en langage C, un *warning* équivaut à une « erreur larvée ». Ainsi, vous pouvez effectivement exécuter votre programme et ce malgré les warnings mais le problème est que vous risquez de le « payer » par la suite :



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

void affiche_matrice(int matrice[9][9]) {
    int i, j;

    for(i=1; i<=7; i++) {
        for(j=1; j<=7; j++)
            printf("%d", matrice[i][j]);
        printf("\n");
    }
}

int main() {
    int i;
    int matrice[9][9];
    ...
    affiche_matrice(matrice[9][9]);
}
```

La dernière ligne `affiche_matrice(matrice[9][9])` vous renverra un warning... Pourquoi ? Prenons le cas où cette ligne serait remplacée par :

```
affiche_matrice(matrice[1][1]);
```

Dès lors, on voit bien le problème, cette ligne ne passe pas tout le tableau `matrice` à la fonction `affiche_matrice`, mais uniquement la case `[1][1]`. La solution consisterait donc à écrire :

```
affiche_matrice(matrice);
```

En conclusion, vous devez considérer tous les warnings comme des erreurs et les éliminer (cela ne concerne cependant pas les warnings provenant de l'usage de `gets` même si cette considération dépend de l'usage qui en est fait au sein de votre programme).