CHARLES CARNUS
LYCÉE TECHNIQUE
ENSEIGNEMENT SUPÉRIEUR

Langage C

Les pointeurs & les références

Prof : **Kamal Boudjelaba**

14 août 2023

B T S S N

LTP Charles Carnus
Avenue de Saint Pierre - Bourran - 12000 RODEZ, France

IR–EC

Table des matières

1. Rappel

1.1 Les opérateurs de manipulation des bits

Le langage C définit six opérateurs permettant de manipuler les bits :

— l'opérateur "et" : `&`
— l'opérateur "ou" : `|`
— l'opérateur "ou exclusif" : `^`
— l'opérateur de négation (de complément) : `~`
— l'opérateur de décalage à droite : `>>`
— l'opérateur de décalage à gauche : `<<`

Remarque 1.1 :

Ne pas confondre les opérateurs de manipulation des bits «et» (`&`) et «ou» (`|`) avec «et» (`&&`) et «ou» (`||`) logiques. Il s'agit d'opérateurs totalement différents au même titre que les opérateurs d'affectation (=) et d'égalité (==).
De même, l'opérateur de manipulation des bits «et» (`&`) n'a pas de rapport avec l'opérateur d'adressage (`&`), ce dernier n'utilisant qu'un opérande.

Exemple 1.1

```c
#include <stdio.h>

int main(void)
{
    int a = 0x63; // 0x63 == 99 == 0110 0011
    int b = 0x2A; // 0x2A == 42 == 0010 1010

    // 0110 0011 & 0010 1010 == 0010 0010 == 0x22 == 34
    printf("%2X\n", a & b);
    // 0110 0011 | 0010 1010 == 0110 1011 == 0x6B == 107
    printf("%2X\n", a | b);
    // 0110 0011 ^ 0010 1010 == 0100 1001 == 0x49 == 73
    printf("%2X\n", a ^ b);
    return 0;
}
```

1.1.1 Les opérateurs de décalage

L'opérateur de décalage à gauche

```c
#include <stdio.h>

int main(void)
{
    // 0000 0001 << 2 == 0000 0100
    int a = 1 << 2;
    // 0010 1010 << 2 == 1010 1000
    int b = 42 << 2;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

L'opérateur de décalage à droite

```c
#include <stdio.h>

int main(void)
{
    // 0001 0000 >> 2 == 0000 0100
    int a = 16 >> 2;
    // 0010 1010 >> 2 == 0000 1010
    int b = 42 >> 2;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

1.1.2 Affichage des nombres en binaire

Il n'existe pas de format de la fonction `printf()` qui permet d'afficher la représentation binaire d'un nombre (la représentation hexadécimale est disponible).

Cet exemple réalise une fonction capable d'afficher la représentation binaire d'un `unsigned int`.

```c
#include <stdio.h>

void affiche_bin(unsigned n)
{
    unsigned mask = ~(~0U >> 1);
    unsigned i = 0;

    while (mask > 0)
    {
        if (i != 0 && i % 4 == 0)
            putchar(' ');

        putchar((n & mask) ? '1' : '0');
        mask >>= 1;
        ++i;
    }
    putchar('\n');
}

int main(void)
{
    affiche_bin(10);
    affiche_bin(50);
    return 0;
}
```

Cet exemple réalise la conversion d'un nombre écrit en décimal vers le binaire.

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

  int tab[10], nbr, i;

  printf("Entrer le nombre à convertir: ");
  scanf("%d",&nbr);

  for(i=0; nbr > 0; i++)
  {
    tab[i] = nbr%2;
    nbr = nbr/2;
  }

```

```
16    printf("\nLe nombre en binaire est : ");
17    printf("0b");
18    for(i=i-1; i >= 0; i--)
19    {
20      printf("%d",tab[i]);
21    }
22    printf("\n");
23
24    return 0;
25  }
```

1.2 scanf et gets

scanf	gets
`scanf` est une fonction C permettant de lire une entrée depuis l'entrée standard (clavier) jusqu'à ce qu'il rencontre un espace, une nouvelle ligne ou un EOF.	`gets` est une fonction C permettant de lire une entrée depuis une entrée standard jusqu'à ce qu'il rencontre une nouvelle ligne ou un EOF. Il considère les espaces comme une partie de l'entrée.
La fonction `scanf` prend une chaîne de formatage et la liste des adresses des variables. par exemple. `scanf("%d", &nbr)` ;	La fonction `gets` prend le nom de la variable pour stocker la valeur reçue. Par exemple. `gets(var)` ;
`scanf()` peut lire plusieurs valeurs de différents types de données.	`gets()` obtiendra uniquement les données de type chaîne de caractères.

1.3 Les structures

Exemple 1.2: Structures

```c
/* Structures
 * Calcul le nombre de jours entre deux dates saisies sous la forme aaaa/mm/jj
*/
#include <stdio.h>
#include <stdlib.h>

struct date {
    unsigned annee;
    unsigned mois;
    unsigned jour;
};


int main(void)
{
    struct date d1;
    struct date d2;

    printf("Première date (aaaa/mm/jj) : ");

    if (scanf("%u/%u/%u", &d1.annee, &d1.mois, &d1.jour) != 3)
    {
        printf("Saisie incorrecte\n");
        return EXIT_FAILURE;
    }

    printf("Deuxième date (aaaa/mm/jj) : ");

    if (scanf("%u/%u/%u", &d2.annee, &d2.mois, &d2.jour) != 3)
    {
        printf("Saisie incorrecte\n");
        return EXIT_FAILURE;
    }

    d1.mois += d1.annee * 12;
    d1.jour += d1.mois * 30; // On considère que tous les mois ont 30 jours
    d2.mois += d2.annee * 12;
    d2.jour += d2.mois * 30;

    printf("Il y a %u jour(s) de différence.\n",
    d2.jour - d1.jour);
    return 0;
}
```

Exemple 1.3: Structures : Nombres complexes

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

struct complex
{
    int r, i;
};

int main()
{
    struct complex a, b, somme;
    double module;
    printf("Entrer la valeur a et b du 1er nombre complexe (a + ib): ");
    scanf("%d%d", &a.r, &a.i);
    printf("Entrer la valeur c et d du 2eme nombre complexe (c + id): ");
    scanf("%d%d", &b.r, &b.i);

    somme.r = a.r + b.r;
    somme.i = a.i + b.i;
    module = sqrt(a.r*a.r+a.i*a.i);

    printf("La somme des nombres complexes: %d + %di\n", somme.r, somme.i);
    printf("Le module du 1er nombre complexe %d + %di est : %f\n", a.r, a.i, module);

    return 0;
}
```

1.4 Espace mémoire réservé aux différents types de variables

Exemple 1.4: Espace mémoire occupé par les différents types de variables

```c
#include <stdio.h>

int main(void)
{
    double f;

    printf("_Bool : %zu\n", sizeof(_Bool));
    printf("char : %zu\n", sizeof(char));
    printf("short : %zu\n", sizeof(short));
    printf("int : %zu\n", sizeof(int));
    printf("long : %zu\n", sizeof(long));
    printf("float : %zu\n", sizeof(float));
    printf("double : %zu\n", sizeof(double));
    printf("long double : %zu\n", sizeof(long double));

    printf("int : %zu\n", sizeof 5);
    printf("double : %zu\n", sizeof f);
    return 0;
}
```

Exemple 1.5: Espace mémoire occupé par les types de variables

```c
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Taille de int = %lu octets\n",sizeof(a));
    printf("Taille de float = %lu octets\n",sizeof(b));
    printf("Taille de double = %lu octets\n",sizeof(c));
    printf("Taille de char = %lu octet\n",sizeof(d));

    return 0;
}
```

Exemple 1.6: Espace mémoire occupé par une structure donnée

```c
#include <stdio.h>

struct exemple
{
    double flottant;
    char lettre;
    unsigned int entier;
};

int main(void)
{
    printf("Taille de la structe 'exemple' est : %zu octets\n", sizeof(struct exemple));
    return 0;
}
```

2. Les pointeurs

2.1 Introduction

Lorsque une variable est déclarée, de la mémoire est allouée à cette variable, et l'emplacement de cette mémoire ne variera pas tout au long de l'exécution du code.

En plus des types de données tels que les entiers et les nombres à virgule flottante (double), on peut également déclarer des variables de pointeur qui sont des variables qui stockent les adresses d'autres variables.

2.2 Case mémoire et adresse

— Tout objet (variable, fonction ...) manipulé par l'ordinateur est stocké dans sa mémoire, constituée d'une série de cases.

— Pour accéder à un objet (au contenu de la case mémoire dans laquelle cet objet est enregistré), il faut connaître le numéro de cette case. Ce numéro est appelé l'adresse de la case mémoire.

— Lorsqu'on utilise une variable ou une fonction, le compilateur utilise l'adresse de cette dernière pour y accéder.

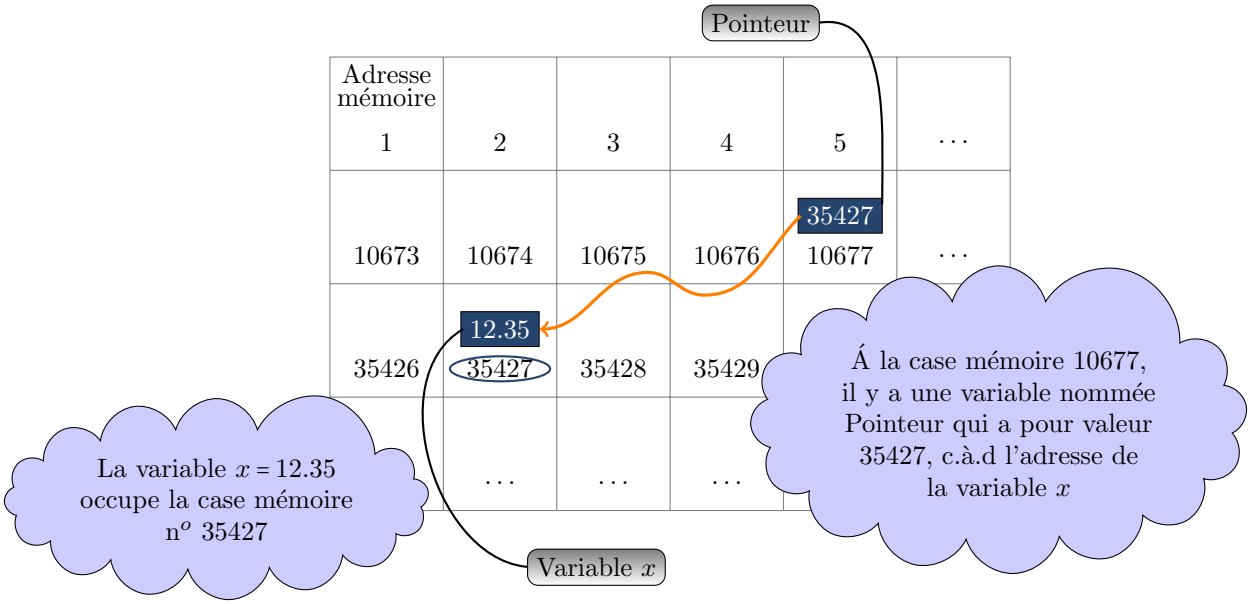

Figure 1. Pointeur

Un pointeur est une variable qui contient l'adresse d'une autre variable (objet).

Utilité des pointeurs :

— le passage de références à des fonctions ;

— la manipulation de données complexes ;

— l'allocation dynamique de mémoire.

Notes :

— `int* x` : Déclare un pointeur `x` vers un entier.
  - x contient l'adresse mémoire où est stocké un entier
  - `x` vaut donc n'importe quelle adresse en RAM (car dans cet exemple le pointeur n'est pas initialisé)

— L'opérateur `&` : Si `a` est un entier, `&a` renvoie l'adresse mémoire de la variable `a`

```
int a;
int *x;
x = &a;
```

Dans cet exemple, on a copié l'adresse où est stockée en mémoire la variable `a` dans le pointeur `x`. Donc, on a copié une adresse vers un entier, pas un entier. On dit que `x` pointe vers la variable a.

— L'opérateur de déréférencement `*` : Si `x` est un pointeur vers un entier, `*x` sera l'entier pointé par `x`.

```
int a = 12;
int *x;
x = &a;
*x = 12;
```

Comme `x` est un pointeur vers `a`, `*x` désigne la variable `a`.
L'instruction `*x = 12;` copie un entier dans un autre et non une adresse. On copie donc l'entier 12 dans la variable `a`.

2.3 Syntaxe : Déclarer un pointeur

Pour déclarer un pointeur, on procède de la même manière que pour les variables, c.à.d. déclarer :
— le type
— le nom, précédé par *

```
int *pointeur;
```

On peut aussi utiliser cette syntaxe :

```
int* pointeur;
```

Inconvenient : ne permet pas de déclarer plusieurs pointeurs sur la même ligne.

```
int* p1, p2, p3, p4;
```

Seul $p1$ sera un pointeur, les autres variables seront des entiers standards.

Exemples

```
double* p_x;     // p_x est un pointeur vers une variable à virgule flottante double précision

int* p_i;        // p_i est un pointeur vers une variable entière
```

```
#include <stdio.h>

int main()
{
    int x = 10;
    printf("%d",&x);  // printf("%p",&x);
    return 0;
}
```

En C, le symbole pour obtenir l'adresse d'une variable est `&`. Pour afficher l'adresse de la variable `x`, on doit écrire `&x`.
Après exécution du code, la console affiche `0x7ffeefbff538`, qui correspond à l'adresse (en hexadécimal) de la case mémoire contenant la variable `x`.

```
#include <stdio.h>
#include <stddef.h> // Pour utiliser NULL
int main()
{
    int x = 10;
    printf("%d", &x);
    int *p_x = NULL;     // ou int *p_x(0);
                         ou = (int *)0;
    p_x = &x;
    printf("\n%d", p_x);
    return 0;
}
```

Important : Les pointeurs doivent être déclarés en les initialisant à 0.
La console affiche :

```
0x7ffeefbff538
0x7ffeefbff538
```

L'adresse de x est sauvegardée dans le pointeur p_x.
On dit alors que le pointeur p_x pointe sur x.

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

int main()
{
    int a, b, c;
    int *x = NULL, *y = NULL;

    a = 98;
    x = &a;
    c = *x + 5;
    y = &b;
    *y = a + 10;

    printf("b = %d", b);
    printf("\nc = %d \n", c);
    return 0;
}
```

La console affiche :

```
b = 108
c = 103
```

- a est initialisé à 98.
- x = &a ; affecté à x l'adresse de a.
  x est un pointeur vers a.
- $*x$ est la variable pointée pas x, c.à.d a (=98).
- c = $*x$ + 5 ; permet de transférer $98+5=103$ dans la variable c.
- y = &b ; permet de mettre dans la variable y l'adresse de la variable b. y un pointeur vers b.
  $a+10=98+10=108$.
- $*y$ = a + 10 ; permet de transférer dans la variable pointée par y la valeur de a + 10 = 108.
  On stocke 108 dans b de manière indirecte via le pointeur y.
- Affichage des valeurs de b et c.

Si une variable `p_x` a été déclarée comme pointeur vers un nombre double, alors il est important de distinguer :

(i) l'emplacement de la mémoire vers lequel pointe ce pointeur (noté `p_x`) ; et

(ii) le contenu de cette mémoire (noté `*p_x`).

L'opérateur astérisque (`*`) dans `*p_x` est appelé déréférencement de pointeur et peut être considéré comme l'opposé de l'opérateur `&`.
Le code ci-dessous montre comment des pointeurs vers des variables réelles peuvent être combinés avec des variables.

```c
double y, z;     // y, z stockent les nombres en double précision
double* p_x;     // p_x stocke l'adresse d'un nombre double

z = 3.0;
p_x = &z;            // p_x stocke l'adresse de z
y = *p_x + 1.0;      // *p_x est le contenu de la mémoire p_x, c'est-à-dire la valeur de z
```

Avertissements sur l'utilisation des pointeurs

Un pointeur de variable ne doit pas être utilisé avant d'avoir d'abord reçu une adresse valide.
Par exemple, le code suivant peut entraîner des problèmes difficiles à localiser.

```c
1 double* p_x ; // p_x peut stocker l'adresse d'un nombre double - on ne connait pas l'adresse
                                                encore
2
3 *p_x = 1,0 ; // essaie de stocker la valeur 1.0 dans un emplacement mémoire non spécifié
```

Dans le code ci-dessus, on n'a pas spécifié l'emplacement de la variable `double` vers laquelle pointe `p_x`. Il peut donc pointer n'importe où dans la mémoire de l'ordinateur. Changer le contenu d'un emplacement non spécifié dans la mémoire d'un ordinateur, comme cela est fait à la ligne 3 du code ci-dessus, a clairement le potentiel de causer des problèmes qui peuvent être difficiles à localiser.

Une autre raison d'utiliser les pointeurs avec précaution est indiquée dans le code ci-dessous. La première fois que `y` est imprimé (à la ligne 5) il prend la valeur 3 : la deuxième fois que `y` est imprimé (à la ligne 7) il prend la valeur 1 même si `y` n'est pas explicitement modifié dans le code entre ces deux lignes. En effet, la ligne entre les instructions `printf`, ligne 6, a modifié la valeur de `y`, peut-être involontairement, en utilisant la variable de pointeur `p_x` (qui contient l'adresse de `y`) pour modifier la valeur de `y`.

```c
double y;
double* p_x;
y = 3.0;
p_x = &y;
printf("y = %lf", y);
*p_x = 1.0; // Cela change la valeur de y
printf("\n y = %lf", y);
```

Une situation où le contenu de la même variable peut être accédé en utilisant des noms différents, comme dans le code ci-dessus, est connue sous le nom d'aliasing. En C, cela est plus susceptible de se produire lorsque des pointeurs sont impliqués, soit lorsque deux pointeurs pointent la même adresse en mémoire, soit lorsqu'un pointeur fait référence au contenu d'une autre variable. Lorsqu'un ou plusieurs pointeurs permettent d'accéder à la même variable en utilisant des noms différents, l'aliasing est appelé aliasing de pointeur.

Remarque 2.1 :

Pour afficher l'adresse mémoire d'une variable en hexadécimal, on utilise le format %p

```c
#include <stdio.h>

int main()
{
    int a = 10;
    printf("L'adresse mémoire de la variable 'a' est : %p\n",(void*)&a);
    return 0;
}
```

2.4 Exemples

Exemple 2.1: Accès à la valeur de la variable "a" via le pointeur "p"

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int a = 10;
    int *p = &a;

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

    return 0;
}
```

Exemple 2.2: Modification de la variable "a" à l'aide du pointeur "p"

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int a = 10;
    int *p = &a;

    *p = 12;
    printf("a = %d\n", a);

    return 0;
}
```

Exemple 2.3: Pointeur dans une fonction – V1

```c
#include <stdio.h>

void foisDeux(int *p_x);

int main(int argc, char *argv[])
{
    int nombre = 15;

    foisDeux(&nombre);          // Envoie l'adresse de nombre à la fonction
    printf("%d\n", nombre);     // Affiche la variable nombre.
                                //La fonction a directement modifié la valeur de la variable
                                                car elle connaissait son adresse

    return 0;
}

void foisDeux(int *p_x)
{
    *p_x *= 2;  // Multiplie par 2 la valeur de nombre
}
```

Exemple 2.4: Pointeur dans une fonction – V2

```c
#include <stdio.h>

void foisDeux(int *p_x);

int main(int argc, char *argv[])
{
    int nombre = 15;
    int *p_x = &nombre;     // p_x prend l'adresse de nombre

    foisDeux(p_x);          // Envoie p_x (adresse de nombre) à la fonction
    printf("%d\n", *p_x);   // Affiche la valeur de nombre avec *p_x

    return 0;
}

void foisDeux(int *p_x)
{
    *p_x *= 2;  // Multiplie par 2 la valeur de nombre
}
```

Exemple 2.5: Plusieurs pointeurs dans une fonction

```c
#include <stdio.h>

void fonction(int *pa, int *pb)
{
    *pa = 1;
    *pb = 2;
}
int main(void)
{
    int a;
    int b;
    int *pa = &a;
    int *pb = &b;

    fonction(&a, &b);
    fonction(pa, pb); // Réalise la même chose que la ligne 15
    printf("a = %d, b = %d\n", a, b);
    printf("a = %d, b = %d\n", *pa, *pb);
    return 0;
}
```

Exemple 2.6: Pointeur de pointeur

```c
#include <stdio.h>

int main(void)
{
    int a = 12;
    int *pa = &a;
    int **pp = &pa;

    printf("a = %d\n", **pp);
    return 0;
}
```

Exemple 2.7: Permutation de deux variables

```c
#include <stdio.h>

void permutation(int *, int *);

int main(void)
{
    int x = 5;
    int y = 3;

    permutation(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}

void permutation(int *px, int *py)
{
    int tmp = *px;
    *px = *py;
    *py = tmp;
}
```

Exemple 2.8: Structures et pointeurs

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

struct date {
    unsigned annee;
    unsigned mois;
    unsigned jour;
};

int main(void)
{
    struct date d;
    struct date *p = &d;

    p->annee = 2022; // (*p).annee = 2022;
    p->mois = 9;
    p->jour = 21;

    printf("La date est : %u/%s%u/%u\n",d.annee,(d.mois<10)?"0":"",d.mois,d.jour);
    return 0;
}
```

Exemple 2.9: Fonction pour afficher un tableau 1D

```c
#include <stdio.h>

void affiche_tableau1D(int *tableau, unsigned taille)
{
    for (unsigned i = 0; i < taille; ++i)
        printf("tableau[%u] = %d\n", i, tableau[i]);
}

int main(void)
{
    int tableau[5] = { 2, 5, 12, 22, 20 };

    affiche_tableau1D(tableau, 5);
    return 0;
}
```

Exemple 2.10: Fonction pour afficher un tableau 2D

```c
#include <stdio.h>

void affiche_tableau2D(int (*tab)[2], unsigned n, unsigned m)
{
    for (unsigned i = 0; i < n; ++i)
        for (unsigned j = 0; j < m; ++j)
            printf("tab[%u][%u] = %d\n", i, j, tab[i][j]);
}

int main(void)
{
    int tab[2][2] = { { 11, 12 }, { 21, 22 } };

    affiche_tableau2D(tab, 2, 2);
    return 0;
}
```

3. Les références

Une alternative à l'utilisation de pointeurs est d'utiliser des variables de référence : ce sont des variables qui sont utilisées à l'intérieur d'une fonction et qui sont un nom différent pour la même variable que celle envoyée à une fonction. Lors de l'utilisation de variables de référence, toute modification à l'intérieur de la fonction aura un effet en dehors de la fonction. Ceux-ci sont beaucoup plus faciles à utiliser que les pointeurs : il suffit d'inclure le symbole `&` devant le nom de la variable dans la déclaration de la fonction et du prototype — cela indique que la variable est une variable de référence. C'est en fait le cas où les références se comportent comme des pointeurs, mais sans que le programmeur ait à convertir en une adresse avec `&` sur l'appel de la fonction et sans avoir à déréférencer à l'intérieur de la fonction — elles fournissent une syntaxe pour alléger le fardeau du programmeur.

- Une référence peut être vue comme un alias d'une variable (utiliser la variable ou une référence à cette variable est équivalent).
- On peut modifier le contenu de la variable en utilisant une référence.
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Toute autre affectation modifie en fait la variable référencée.
- Une référence ne peut donc référencer qu'une seule variable.
- Les références sont principalement utilisées pour passer des paramètres aux fonctions.

3.1 Syntaxe : déclarer une référence

```
type &reference = identificateur;
```

Exemples

```
int x = 0;
int &r_x = x;            // Référence sur la variable x.
r_x = r_x + x;       // Double la valeur de x (et de r_x).
```

```
#include <stdio.h>

int main()
{
    int a = 98, b = 78, c;
    int &x = a;
    c = x + 5;  // équivalent à :  c = a + 5;
    int &y = b;
    y = a + 10; // équivalent à :  b = a + 10;
    printf("b = %d", b);
    printf("\n c = %d", c);
    return 0;
}
```

- `int &x = a;` permet de déclarer une référence x vers la variable a.
- `c = x + 5;` permet donc de transférer 103 dans la variable c.
- `int &y = b;` permet de déclarer une référence y vers la variable b.
- `y = a + 10;` permet de transférer 108 dans la variable b.

La console affiche :

```
b = 108
c = 103
```

3.2 Fonctions : Passage de paramètres par valeur, par pointeur et par référence)

Il existe trois manières de transmettre des arguments ou des paramètres aux fonctions :

- Passage par valeur : une copie des arguments réels est transmise aux arguments formels
- Passage par référence : l'emplacement (adresse) des arguments réels est transmis à des arguments formels, toute modification apportée aux arguments formels se reflétera également dans les arguments réels
- Passage par pointeur : généralement semblable au passage de paramètres par références.

	Passage par valeur	**Passage par référence**
Variable	Une copie de la variable est transmise	La variable elle-même est transmise
Effet	La modification d'une copie de variable ne modifie pas la valeur d'origine de la variable en dehors de la fonction	La modification de la variable affecte également la valeur de la variable en dehors de la fonction

Exemple 3.1: Fonctions – Passage par pointeur

```c
#include<stdio.h>

void minmax(int i, int j, int* min, int* max)
{
    if(i<j)
        {*min=i; *max=j;}
    else
        {*min=j; *max=i;}
}

int main()
{
    int a, b, x, y;
    printf("Taper la valeur de a : ");
    scanf("%d",&a);
    printf("Taper la valeur de b : ");
    scanf("%d",&b);
    minmax(a, b, &x, &y);
    printf("Le plus petit vaut : %d\n",x);
    printf("Le plus grand vaut : %d\n",y);

    return 0;
}
```

On a une fonction `minmax` qui a comme paramètres : 2 entiers `i` et `j` et 2 pointeurs vers des entiers `min` et `max`. Cette fonction calcule le plus petit de `i` et de `j` et le met dans l'entier pointé par `min`. Elle calcule le plus grand des 2 entiers et le copie dans l'entier pointé par `max`.
Dans la fonction `main()`, on déclare 4 entiers `a`, `b`, `x`, et `y`. On demande à l'utilisateur de saisir au clavier les entiers `a` et `b`. Lors de l'appel de fonction `minmax(a,b,&x,&y)`, on copie la valeur de `a` dans `i`, la valeur de `b` dans `j`. On copie la valeur de `&x` (un pointeur vers `x`) dans `min` et on copie `&y` (un pointeur vers `y`) dans `max` :
`min` pointe donc vers `x` et `max` vers `y`. Lors de l'appel, on va donc récupérer dans `x` le plus petit des entiers `a` et `b` et dans `y` le plus grand de ces 2 entiers.

Exemple 3.2: Fonctions – Passage par référence

```c
#include<stdio.h>

void minmax(int i, int j, int& min, int& max)
{
    if(i<j)
    { min=i; max=j; }
    else
    { min=j; max=i; }
}

int main()
{
    int a, b, x, y;
    printf("Taper la valeur de a : ");
    scanf("%d",&a);
    printf("Taper la valeur de b : ");
    scanf("%d",&b);
    minmax(a, b, x, y);
    printf("Le plus petit vaut : %d\n",x);
    printf("Le plus grand vaut : %d\n",y);
    return 0;
}
```

La fonction `minmax` possède 4 paramètres : 2 entiers `i` et `j` passés par valeur et 2 entiers `min` et `max` passés par référence. `i` et `j` sont les paramètres en entrée de la fonction `minmax`. `min` et `max` sont les paramètres en sortie de cette fonction.
Lors de l'écriture de la fonction `minmax`, on note le symbole `&` placé après le type qui indique que le paramètre est passé par référence.
Lors de l'appel de `minmax`, on note qu'elle s'écrit `minmax(a,b,x,y);` sans symbole particulier. `a` et `b` sont passés par valeur et `x` et `y` sont passés par référence.

Exemple 3.3: Fonctions – Passage par valeur

```c
#include <stdio.h>

void permutePassageParValeur(int, int);

int main()
{
  int a = 3, b = 5;

  permutePassageParValeur(a, b);
  printf("nombre1 : %d, nombre2 : %d\n", a, b);
}

void permutePassageParValeur(int x, int y)
{
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
//Résultat : nombre1 : 3, nombre2 : 5
```

Exemple 3.4: Fonctions – Passage par référence (pointeur)

Quand on utilise le passage par référence, l'adresse de l'argument est transmise à la fonction, et les modifications effectuées à l'intérieur de la fonction sont également répercutées en dehors de la fonction.

```c
#include <stdio.h>

void permutePassageParReference(int*, int*);

int main()
{
  int a = 3, b = 5;
  // les arguments réels seront modifiés
  permutePassageParReference(&a, &b);
  printf("nombre1: %d, nombre2: %d\n", a, b);
}

void permutePassageParReference(int *x, int *y)
{
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}
//Résultat : nombre1 : 5, nombre2 : 3
```

Exemple 3.5: Fonctions – Passage par valeur et par référence (pointeur)

```c
#include <stdio.h>

int passage_valeur(int a)
{
    a = a*2;
    return a;
}
void passage_reference(int *b)
{
    *b = *b*2;
}
void passage_reference2(int *a, int *b)
{
    *a = *a+10;
    *b = *b+100;
}

int main()
{
    int x = 3, y = 4, z;
    printf("Initialement : x = %d, y = %d\n",x,y);
    z = passage_valeur(x);
    printf("Fonction 1 : z = %d \n",z);
    passage_reference(&y);
    printf("Fonction 2 : y = %d \n",y);
    passage_reference2(&x,&y);
    printf("Fonction 3 : x = %d, y = %d \n",x,y);
    return 0;
}
```

Note 3.1. *Si on ne veut transmettre que la valeur de la variable, on utilise le "Passage par valeur" et si on veut voir la modification de la valeur originale de la variable, on utilise le "Passage par référence".*

Exemple 3.6: Calcul de la partie réelle et imaginaire d'un nombre complexe

On donne un nombre complexe sous forme polaire, $z = re^{i\theta}$, on veut écrire une fonction qui renvoie la partie réelle, notée par la variable `x`, et la partie imaginaire, notée par la variable `y`, de ce nombre.
Donc on utilise une fonction pour calculer les parties réelles et imaginaires d'un nombre complexe donné sous forme polaire en utilisant des références au lieu des pointeurs.

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

void CalculReelEtImaginaire(double r, double theta, double& reel, double& imaginaire);

int main(int argc, char* argv[])
{
    double r = 3.4;
    double theta = 1.23;
    double x, y;
    CalculReelEtImaginaire(r, theta, x, y);
    printf("Partie réelle = %lf \n", x);
    printf("Partie imaginaire = %lf \n", y);

    return 0;
}

void CalculReelEtImaginaire(double r, double theta, double& reel, double& imaginaire)
{
    reel = r*cos(theta);
    imaginaire = r*sin(theta);
}
```

Exemple 3.7: Fonctions – tableau comme argument

```
#include <stdio.h>

double calculMoyenne(int *arr, int size);

int main ()
{
   int tab[5] = {10, 3, 7, 20, 30};
   double moyenne;
   // passer le pointeur vers le tableau en tant qu'argument
   moyenne = calculMoyenne(tab, 5);

   printf("La valeur moyenne du tableau est : %f\n", moyenne);
   return 0;
}

// définition de fonction
double calculMoyenne(int *tab, int taille)
{
   int  i, som = 0;
   double moy;

   for (i = 0; i < taille; ++i)
   {
      som += tab[i];
   }

   moy = (double)som / taille;
   return moy;
}
```

4. Différence entre pointeurs et références

Les références sont souvent implémentées à l'aide de pointeurs. Donc les deux sont utilisés pour qu'une variable donne accès à une autre. Le caractère «*» désigne un pointeur et le caractère «&» désigne une référence.

- Un pointeur doit être dé-référencé avec l'opérateur «*» pour accéder à l'emplacement de mémoire vers lequel il pointe.
- Un pointeur peut être réaffecté.
```
a = 1, b = 2;
int *p;
p = &a;
p = &b;
```
- Une référence, comme un pointeur, est également implémentée en stockant l'adresse d'un objet.
- Une référence peut être considérée comme un pointeur constant avec une indirection automatique (i.e. le compilateur appliquera automatiquement l'opérateur «*»).
- Une référence ne peut pas être réaffectée et doit être affectée à l'initialisation

	Pointeur	**Référence**
Définition	Le pointeur est l'adresse mémoire d'une variable (objet)	La référence est un alias pour une variable (objet)
Opérateurs	*, ->	&
Retour	Un pointeur renvoie la valeur située à l'adresse stockée dans une variable de pointeur qui est précédée du signe «*»	Une référence renvoie l'adresse de la variable précédée du signe de référence «&»
Initialisation	On peut créer un pointeur sans initialisation Un pointeur peut être initialisée à tout moment dans le programme	On ne peut pas créer une référence sans initialisation Une référence ne peut être initialisée qu'au moment de sa création
Nulle	Un pointeur peut faire référence à NULL	Une référence ne peut jamais faire référence à NULL

5. Bonus

5.1 Tableaux

Exemple 5.1: Tableaux 1D – Fonction pour écrire (remplir) un tableau 1D

```c
#include <stdio.h>

int *Remplir( )
/* Fonction pour remplire le tableau */
{
   static int r[10];
   int i;
   for (i = 0; i < 10; ++i)
   {
      r[i] = i+10;
      printf("r[%d] = %d\n", i, r[i]);
   }
   return r;
}

int main () {
   // Un pointeur vers un entier
   int *p;
   int i;
   p = Remplir();
   printf("\n");
   for ( i = 0; i < 10; i++ )
   {
      printf( "*(p + %d) : %d\n", i, *(p + i));
   }
   return 0;
}
```

Exemple 5.2: Tableaux 1D – Fonctions pour initialiser, écrire et afficher un tableau 1D

```c
#include <stdio.h>

void initTab(int tab[], int taille)
{
    for (int i=0; i<taille; i++)
        tab[i] = 0;
}

void ecritTab(int tab[], int taille)
{
    printf("\n***** Ecriture du tableau *****\n");
    for (int i=0; i<taille; i++)
    {
        printf("Saisir l'élément %d : ",i);
        scanf("%d",&tab[i]);
    }
}

void afficheTab(int tab[], int taille)
{
    printf("\n***** Affichage du tableau *****\n");
    for (int i=0; i<taille; i++)
        printf("%d\t",tab[i]);
    printf("\n");
}

int main()
{
    int T;
    printf("Saisir la taille du tableau : ");
    scanf("%d",&T);
    int tableau[T];
    initTab(tableau, sizeof(tableau)/sizeof(*tableau));
    ecritTab(tableau, sizeof(tableau)/sizeof(*tableau));
    afficheTab(tableau, sizeof(tableau)/sizeof(*tableau));

return 0;
}
```

Exemple 5.3: Tableaux 2D – Fonctions pour initialiser, écrire et afficher un tableau 2D

```c
#include <stdio.h>

int main()
{
    long sommeTab2D(int *TAB, int L, int C, int CMAX);
    void lireDim(int *L, int LMAX, int *C, int CMAX);
    void ecrireTab2D(int *TAB, int L, int C, int CMAX);
    void afficherTab2D(int *TAB, int L, int C, int CMAX);

    int T[10][10]; // Tableau 2-D d'entiers
    int L, C;      // Dimensions du tableau

    lireDim(&L, 10, &C, 10);
    ecrireTab2D( (int*)T, L,C,10);
    printf("Tableau 2-D donné : \n");
    afficherTab2D( (int*)T, L,C,10);
    printf("Somme des éléments du tableau 2-D : %ld\n", sommeTab2D( (int*)T, L,C,10));

    return 0;
}

long sommeTab2D(int *TAB, int L, int C, int CMAX)
{
```

```
24      int I,J;
25      long SOMME = 0;
26
27      for (I=0; I<L; I++)
28          for (J=0; J<C; J++)
29              SOMME += *(TAB + I*CMAX + J);
30      return SOMME;
31  }
32
33  void lireDim(int *L, int LMAX, int *C, int CMAX)
34  {
35      // Saisie des dimensions du tableau 2-D
36      do
37      {
38      printf("Nombre de lignes du tableau 2-D (max.%d) : ",LMAX);
39      scanf("%d", L);
40      }
41      while (*L<0 || *L>LMAX);
42      do
43      {
44      printf("Nombre de colonnes du tableau 2-D (max.%d) : ",CMAX);
45      scanf("%d", C);
46      }
47      while (*C<0 || *C>CMAX);
48  }
49
50  void ecrireTab2D(int *TAB, int L, int C, int CMAX)
51  {
52      int I,J;
53      // Saisie des composantes du tableau 2-D
54      for (I=0; I<L; I++)
55          for (J=0; J<C; J++)
56              {
57              printf("Elément[%d][%d] : ", I, J);
58              scanf("%d", TAB + I*CMAX + J);
59              }
60  }
61
62  void afficherTab2D(int *TAB, int L, int C, int CMAX)
63  {
64      int I,J;
65      // Affichage des composantes du tableau 2-D
66      for (I=0; I<L; I++)
67          {
68          for (J=0; J<C; J++)
69              printf("%7d", *(TAB + I*CMAX + J));
70          printf("\n");
71          }
72  }
```

5.2 Réseaux

Exemple 5.4: Adresses IPv4

```c
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <ifaddrs.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
int main()
{
    struct ifaddrs *addr, *intf;
    char hostname[NI_MAXHOST];
    int family, s;

    if (getifaddrs(&intf) == -1) {
        perror("getifaddrs");
        exit(EXIT_FAILURE);
    }
    for (addr = intf; addr != NULL; addr = addr->ifa_next) {
        family = addr->ifa_addr->sa_family;
    //AF_INET est la famille d'adresses pour IPv4
        if (family == AF_INET) {
          //getnameinfo permet la résolution de noms.
          s = getnameinfo(addr->ifa_addr,
                          sizeof(struct sockaddr_in),
                          hostname,
                          NI_MAXHOST,
                          NULL,
                          0,
                          NI_NUMERICHOST);
          printf("<interface>: %s \t <adresse> %s\n", addr->ifa_name, hostname);
        }
    }
    return 0;
}
```

5.3 Date et heure

Exemple 5.5: Date et heure

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

int main(void)
{
    int h, min, s, day, mois, an;
    time_t now;
    // Heure actuelle
    time(&now);
    // Convertir au format heure locale
    printf("Aujourd'hui : %s", ctime(&now));
    struct tm *local = localtime(&now);
    h = local->tm_hour;
    min = local->tm_min;
    s = local->tm_sec;
    day = local->tm_mday;
    mois = local->tm_mon + 1;
    an = local->tm_year + 1900;
    // Afficher l'heure
    printf("Heure : %02d:%02d:%02d\n", h, min, s);
    // Afficher la date
    printf("Date : %02d/%02d/%d\n", day, mois, an);
    return 0;
}
```

5.4 Lister les fichiers dans un répertoire

Exemple 5.6: Lister les fichiers du répertoire dans lequel le fichier exécutable présent

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

int main()
{
    struct dirent *dir;
    // opendir() renvoie un pointeur de type DIR.
    DIR *d = opendir(".");
    if (d)
    {
        while ((dir = readdir(d)) != NULL)
        {
            printf("%s\n", dir->d_name);
        }
        closedir(d);
    }
    return 0;
}
```

Exemple 5.7: Lister les fichiers V2

```c
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <time.h>

int main (void)
{
   DIR *rep = opendir ("." );

   if (rep != NULL)
   {
      struct dirent *lecture;

      while ((lecture = readdir (rep)))
      {
         struct stat st;

         stat (lecture->d_name, &st);
         {
            /* Modified time */
            time_t t = st.st_mtime;
            struct tm tm = *localtime (&t);
            char s[32];
            strftime (s, sizeof s, "%d/%m/%Y %H:%M:%S", &tm);

            printf ("%-14s %s\n", lecture->d_name, s);
         }
      }
      closedir (rep), rep = NULL;
   }
   return 0;
}
```