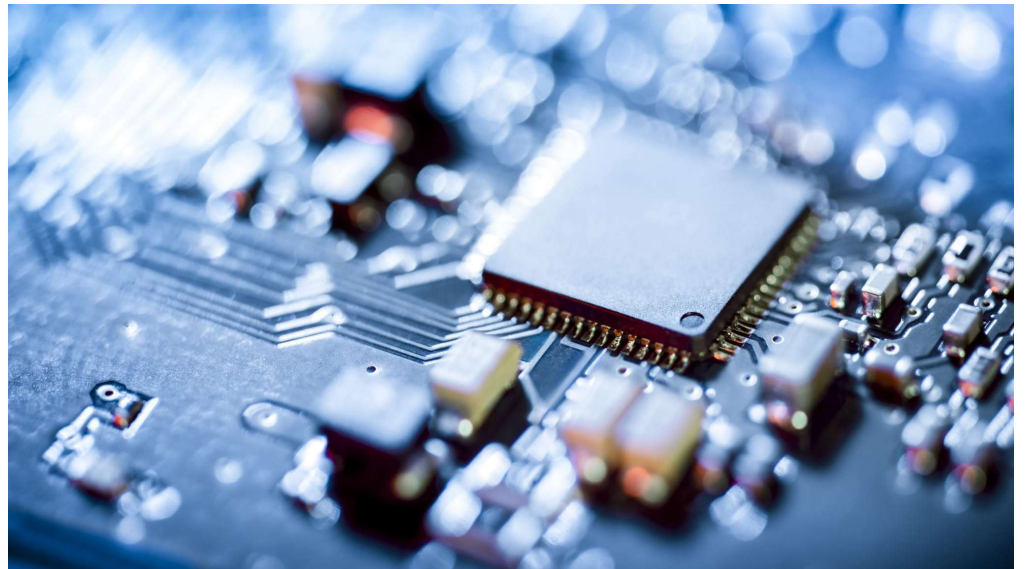
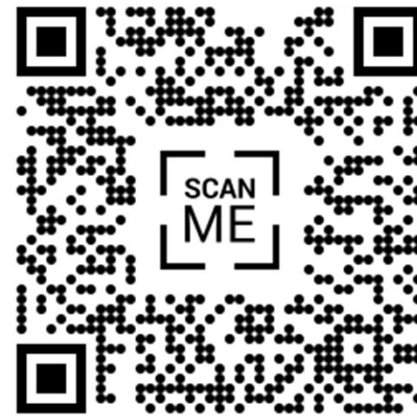


PROGRAMMATION C



Intervenant
Habib NDIAYE



Plan

- Chapitre 1 : Bases de la programmation C
 - Historique et caractéristiques
 - Compilation
 - Composants élémentaires
 - Structures d'un programme
 - Types prédéfinis
 - Opérateurs
 - Fonctions d'entrées et sorties
 - Branchements conditionnels
 - Boucles
 - Branchements non conditionnels

Plan

- Chapitre 2 : Structures de données
 - Tableaux
 - Chaînes de caractères
 - Structures
 - Enumérations
- Chapitre 3 : Pointeurs
- Chapitre 4 : Fonctions
- Chapitre 5 : Fichiers
- Chapitre 6 : Listes chaînées

Introduction aux langages de programmation

Un langage de programmation ressemble à une langue : il dispose de son propre vocabulaire et de ses règles de grammaire et de syntaxe.

Une autre définition est celle-ci : formalisme basé sur une langue et qui définit une manière de donner des ordres à l'ordinateur.

- **programme (ou exécutable)** : algorithme écrit dans un langage de programmation; ensemble d'instructions exécutables par un ordinateur;
- **code source** = fichier texte contenant un ensemble d'instructions écrit dans un certain langage de programmation. En C, un code source est un (ou plusieurs) fichier(s) portant l'extension .c;
- **programmation** : activité de rédaction du code source d'un programme.

On peut regrouper les langages de programmation en deux catégories :

- **les langages de bas niveau** : ils se rapprochent plus du langage machine (langage binaire, langage du processeur) ;

Introduction aux langages de programmation

- **les langages de haut niveau** : ils sont proches du langage humain et indépendants de la machine utilisée (portabilité). Plus un langage est de haut niveau, plus il est proche du langage humain.

Le langage assembleur et le langage machine sont des exemples de langages de bas niveau. Parmi les langages de haut niveau, on retrouve le C, C++, Java, PHP, etc...

Il est possible de faire une traduction du langage de haut niveau vers celui de bas niveau, avec un traducteur. On distingue alors deux types de traducteurs :

- **l'interpréteur** : il traduit le programme instruction par instruction.
- **le compilateur** : il traduit le programme dans son ensemble.

Suivant le type de traducteur utilisé, on parlera alors de langage compilé (cas du C, de Latex, etc.) ou de langage interprété (cas de Ruby, PHP, HTML, etc.). Certains langages, tels que Java, sont intermédiaires.

Historique

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX.

En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre *The C Programming language*.

Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières.

En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent document.

Caractéristiques

Le langage C est un langage de haut niveau(on dit parfois aussi qu'il est le plus bas des langages de haut niveau!!!), **impératif**, **compilé**, **typé** et **déclaratif**.

Avantages

- Grande notoriété et très utilisé, possibilité d'avoir de la documentation et /ou de l'aide ;
- Certain nombre de fonctions prédéfinies dans des bibliothèques ;
- Langage portable et standardisé ;
- Simplicité ;
- Gestion de la mémoire ;
- Vitesse d'exécution.

Inconvénients

- Rigueur d'écriture ;
- Définition de certains concepts par soi-même .

La compilation

Le C est un langage **compilé**(par opposition aux langages **interprétés**). Cela signifie qu'un programme C est décrit par un fichier texte, appelé fichier source. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé compilateur. La compilation se décompose en fait en 4 phases successives:

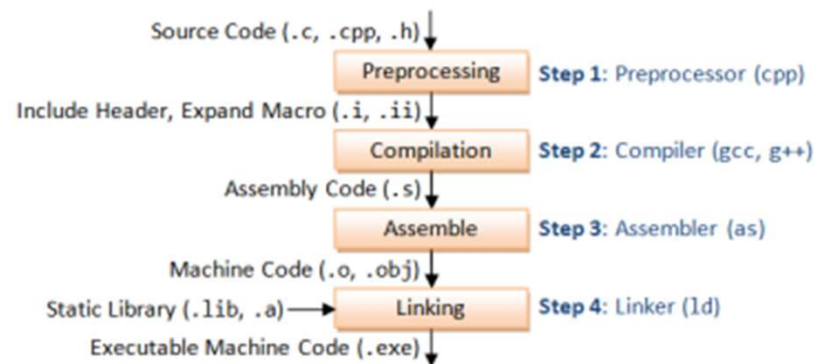
- **la pré-compilation (pre-processing)** : le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...).
- **la compilation** proprement dite (**compiling**) : la compilation proprement dite traduit le fichier généré par le préprocesseur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.

La compilation

- **l'assemblage (assembling)** : cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé fichier objet.
- **l'édition des liens (linking)** : un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des bibliothèques de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit exécutable.

La compilation

GCC(GNU Compiler Collection) est un compilateur en ligne de commande prenant en charge divers langages de programmation, dont le C. Il n'existe pas de compilateur par défaut installé sur Windows. Mais on peut utiliser des versions de **gcc** sur Windows tels que MinGW ou Cygwin.



Les composants élémentaires : Identificateur

Un programme en langage C est constitué des six groupes de composants élémentaires suivants:

- les identificateurs,
- les mots-clés,
- les constantes,
- les chaînes de caractères,
- les opérateurs,
- les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui sont enlevés par le préprocesseur.

Identificateur

Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner :

- un nom de variable ou de fonction;
- un type défini par **typedef**, **struct** ou **enum**.

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées);
- les chiffres;
- le "blanc souligné" (_).

NB: Le premier caractère d'un identificateur ne peut pas être un chiffre et les majuscules et minuscules sont différenciées (sensibilité à la casse, les accents sont interdits)

Exemple: **var1**, **tab23** ou **_deb** sont des identificateurs valides; par contre, **1i** et **i:j** ne le sont pas. Il est cependant déconseillé d'utiliser **_** comme premier caractère d'un identificateur car il est souvent employé pour définir les variables globales de l'environnement C.

Mots clés

Il existe une série de mots réservés (aussi appelés mots-clés) au langage C, c'est-à-dire qu'on ne peut pas les utiliser comme identificateurs (de variables, de fonctions ou de nouveaux types). Ils sont au nombre de 32 :

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

que l'on peut ranger en catégories

– les spécificateurs de stockage

auto register static extern typedef

– les spécificateurs de type

char double enum float int long short

signed struct union unsigned void

– les qualificateurs de type

const volatile

– les instructions de contrôle

break case continue default do else for goto if

switch while

– divers

return sizeof

Commentaires

Dans un programme, il est souvent nécessaire de documenter les parties importantes (par exemple pour indiquer l'auteur du code, la date de création, le rôle d'une fonction ou d'un bloc d'instructions). Pour cela, on utilise la notion de commentaire. Le texte placé dans un commentaire est ignoré par le compilateur lors de l'exécution. On distingue deux sortes de commentaires :

- les commentaires portant sur une ligne : on utilise `//`;
- les commentaires portant sur plusieurs lignes : ils débutent par `/*` et se terminent par `*/`.

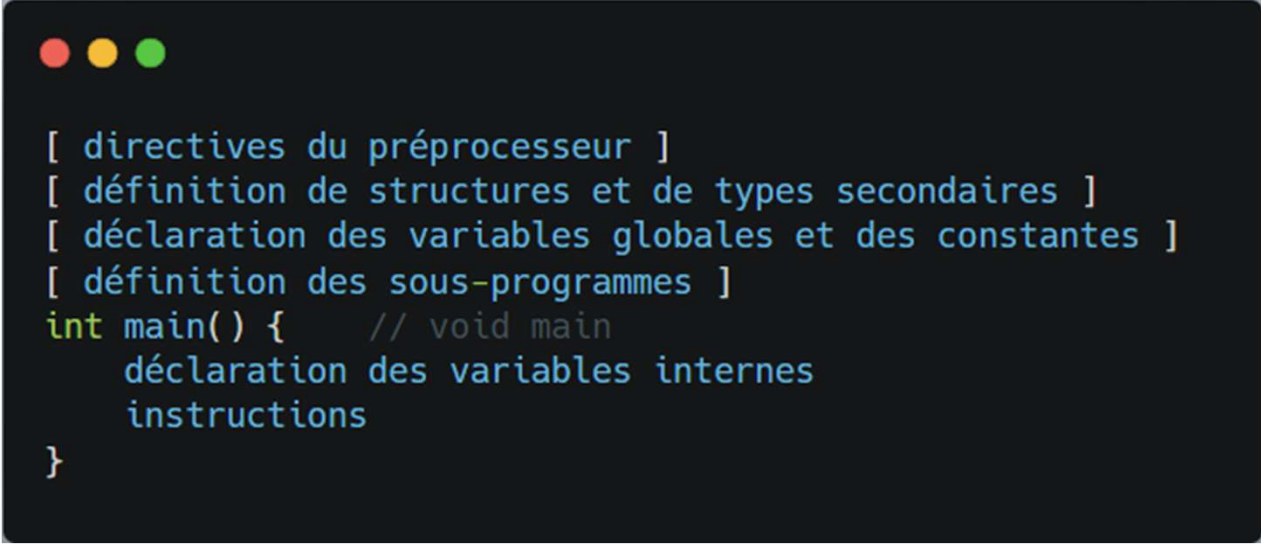
```
#include<stdio.h>
/*
Ci-dessous, vous avez la fonction principale du programme, appelée main.
C'est par cette fonction que tous les programmes commencent.
Ici, ma fonction se contente d'afficher Hello World à l'écran.
*/

int main()
{
    printf("Hello World"); // Cette instruction affiche Bonjour à l'écran
    return 0;              // Le programme renvoie le nombre 0 puis s'arrête
}
```

Structure d'un programme C

Un programme C est décrit par un ou plusieurs fichiers d'extension .c (appelés fichiers source) et est constitué d'un programme principal pouvant faire appel à des sous-programmes qui réalisent chacun une certaine tâche.

Un programme C se présente de la façon suivante :



```
[ directives du préprocesseur ]  
[ définition de structures et de types secondaires ]  
[ déclaration des variables globales et des constantes ]  
[ définition des sous-programmes ]  
int main() {    // void main  
    déclaration des variables internes  
    instructions  
}
```

Chaque instruction dans le code source se termine par un ;

Les directives du préprocesseur

Le préprocesseur est un programme exécuté lors de la précompilation (première étape de la compilation). Il effectue des modifications textuelles sur le fichier source à partir de directives. Une directive du préprocesseur est une ligne du code source commençant par **#**. Il existe trois manières de définir une directive du préprocesseur :

- **#include**<nomFichier> : indique l'utilisation d'une librairie prédéfinie et se trouvant dans le fichier *nomFichier*;
- **#define** : pour la définition de constantes symboliques et de macros
- **#if**, **#ifdef**, **#ifndef** : directives de compilation conditionnelle.


Les directives du préprocesseur

La directive `#include`

Elle permet d'inclure des fichiers (représentant des bibliothèques) dont les éléments (fonctions, objets prédéfinis, etc...) seront utilisés dans le programme. On peut utiliser cette directive de deux façons :

- `#include<nomFichier>` : indique l'utilisation d'une bibliothèque prédéfinie et se trouvant dans le fichier *nomFichier* ;
- `#include "nomFichier"` : Elle est plutôt destinée aux fichiers créés par le programmeur lui-même.

Concrètement, la directive `#include` va inclure le contenu du fichier indiqué dans le code source.



```
#include<stdio.h>
#include "myLibrary.h"
```

Les directives du préprocesseur

La directive `#define`

Elle permet de définir des **constantes** dites **symboliques** (ou constantes du préprocesseur). Une constante est une variable particulière qui ne change pas de valeur.

Syntaxe :

`#define nomConstante valeur`

où *valeur* peut être n'importe quelle chaîne de caractères. Concrètement, dans tout le code source, toute occurrence de *nomConstante* sera remplacée par *valeur*.

```
#define TVA 12
#define BJR "Bonjour"
#define AFF printf
#define NBRE ELEMENTS 5
#define NB LIGNES 2
#define NB COLONNES 3
#define TAILLE NB LIGNES * NB COLONNES
#define DEBUT TEST if
```

- Par convention, les noms des caractères sont toujours écrits en toutes majuscules.
- On peut définir les constantes d'une autre manière (avec le mot-clé **const**. Par exemple : **const int TAXE = 75;**). Mais il est recommandé de toujours les définir comme des constantes symboliques avec la directive `#define`.

La fonction main

Un programme C doit contenir au minimum une fonction spéciale appelée **fonction main**. La fonction main est le point d'entrée du programme, c'est-à-dire que c'est la première fonction qui sera invoquée lors de l'exécution. Sa présence est obligatoire.



```
#include<stdio.h>

int main(){
    printf("Hello World");
    return 0;
}
```



```
#include<stdio.h>

void main(){
    printf("Hello World");
}
```

Éléments de base : Types

Le C est un langage **typé**. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire.

La mémoire de l'ordinateur se décompose en une suite continue d'octets. Chaque octet de la mémoire est caractérisé par son adresse, qui est un entier. Deux octets contigus en mémoire ont des adresses qui diffèrent d'une unité. Quand une variable est définie, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type.

La taille mémoire correspondant aux différents types dépend des compilateurs ; toutefois, la norme ANSI spécifie un certain nombre de contraintes.

Les types de base en C concernent :

- les entiers;
- les flottants (nombres réels).
- les caractères ;

Type caractère

Pour déclarer une variable de type caractère, on utilise le mot-clé **char**.

En C, la notion de caractère dépasse celle de caractère imprimable, c'est à dire auquel est associé un graphisme (lettres, chiffres, ponctuation, caractères spéciaux, etc).

Le type *char* a une particularité : en effet, on considère qu'une donnée de type **char** n'est rien d'autre qu'un entier codé sur un octet. Et donc tout objet de type *char* peut être utilisé dans une expression qui utilise des objets de type *int*. Par exemple, si une variable **c** est de type char, l'expression **c + 1** sera correcte.

Le ASCII(American Standard Code for Information Interchange) donne les correspondances entre les caractères et les entiers qui les représentent.

Quand un caractère est imprimable il est noté entre les **quotes simples** : **'a'**, **'+'**, **'5'**, **' '**, etc.

Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe, qui sont respectivement désignés par **** et **\'**. Le point d'interrogation et les guillemets peuvent aussi être désignés par les notations **\?** et **\"**.

Type caractère

Les caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère antislash(\).

<code>\n</code>	nouvelle ligne	<code>\r</code>	retour chariot
<code>\t</code>	tabulation horizontale	<code>\f</code>	saut de page
<code>\v</code>	tabulation verticale	<code>\a</code>	signal d'alerte
<code>\b</code>	retour arrière		

```
#include<stdio.h>

int main(){
    printf("Hello \n");
    printf("\t World");
}
```

Il existe deux façons d'affecter une valeur à une variable de type char :

- soit en lui donnant directement la valeur du caractère mise entre **quotes**
- soit en utilisant le code ASCII correspondant à ce caractère.

```
#include<stdio.h>

int main(){
    char c1, c2;
    c1 = 'r';
    c2 = 114;
    printf("c1 = %c ; c2 = %c", c1, c2);
}
```

Remarque :

Tous les caractères imprimables sont **non-signés(unsigned)**.

Type entier

Pour déclarer une variable de type entier, on utilise le mot-clé `int`. Le type entier se définit de plusieurs manières :

- en précisant si c'est un entier **signé** (`signed`) ou **non signé** (`unsigned`). Le bit de poids fort de l'entier correspond au signe (0 pour positif, 1 pour négatif). Par défaut, les entiers sont signés;
- et/ou en précisant l'intervalle de valeurs avec les mots-clés : `short` ou `long`. (il existe aussi `long long` sur certains compilateurs).

Le tableau ci-dessous résume les types entiers.

Type entier	Taille en octets	intervalle pour <code>signed</code>	intervalle pour <code>unsigned</code>
<code>char</code>	1	$[-2^7, 2^7[$	$[0, 2^8[$
<code>int</code>	4	$[-2^{31}, 2^{31}[$	$[0, 2^{32}[$
<code>short int</code>	2	$[-2^{15}, 2^{15}[$	$[0, 2^{16}[$
<code>long int</code>	8	$[-2^{63}, 2^{63}[$	$[0, 2^{64}[$

Type réel ou flottant

Les flottants (les réels) sont désignés par les mots-clés `float`, `double` et `long double`.

Type flottant	Taille en octets
<code>float</code>	4
<code>double</code>	8
<code>long double</code>	16

Remarques

- Pour les flottants, pas besoin d'utiliser les mots-clés `signed` et `unsigned`.
- Le plus souvent, on utilisera juste `float` pour déclarer une variable de type flottant.
- Pour les nombres à virgule, le point remplace la virgule.

Les constantes réelles sont représentées par la notation classique par mantisse et exposant.

L'exposant est introduit par la lettre `e` ou `E` ; il s'agit d'un nombre décimal éventuellement signé.

Type réel ou flottant

Par défaut, une constante réelle est représentée avec le format du type **double**. On peut cependant influencer sur la représentation interne de la constante en lui ajoutant un des suffixes **f** (indifféremment **F**) ou **l** (indifféremment **L**). Les suffixes **f** et **F** forcent la représentation de la constante sous forme d'un float, les suffixes **l** et **L** forcent la représentation sous forme d'un long double.

```
#include<stdio.h>

int main(){
    float x = 12.34;
    double pi = 3.14;
    double a = -56.6782107288;
    double z = 12.3e-4; //constante réelle
}
```

Type void

Il existe un type spécial, appelé **void**, qui exprime l'idée de "*aucune valeur*" ou "*vide*". On l'utilise par exemple :

- comme type de retour d'une fonction ne retournant pas de valeur;
- dans le prototype de fonctions sans paramètres;
- pour déclarer un pointeur générique.

NB:

Le type booléen n'existe pas explicitement en C.

Il est plutôt défini par le langage C++ (type `bool`). Dans la pratique, si on veut définir une variable booléenne en C, on passera par le type `int` : **0** pour **false**, **1** pour **true**.

La meilleure façon de définir un booléen est d'utiliser la définition introduite par la norme du langage C99.

```
#include <stdbool.h>
```

L'opérateur d'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe `=`. Sa syntaxe est la suivante :

`variable = expression ;`

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer *expression* et d'affecter la valeur obtenue à variable. De plus, cette expression possède une valeur, qui est celle expression. Ainsi, l'expression `i = 5` vaut 5.

L'affectation effectue une conversion de type implicite: la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant:

imprime pour `x` la valeur 6.5 (et non 7), car dans l'instruction `i = j + x`, l'expression `j + x` a été convertie en entier.

```
#include<stdio.h>

int main(){
    int i, j = 2;
    float x = 2.5;
    i = j + x;
    x = x + i;
    printf("\n %f \n",x);
}
```

Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires :

+	Addition
-	Soustraction
*	Multiplication
/	Division entière
%	Division modulo

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes:

- Si les deux opérandes sont de type entier, l'opérateur / produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant.

Les opérateurs arithmétiques

```
#include<stdio.h>

int main(){
    float x;
    x = 3 / 2;
    printf("x vaut %f \n", x);
    x = 3 / 2.;
    printf("x vaut %f \n", x);
    return 0;
}
```

- L'opérateur % ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

Les opérateurs relationnels

Les opérateurs relationnels sont :

>	Supériorité stricte
>=	Supériorité ou égalité
<	Infériorité stricte
<=	Infériorité ou égalité
==	Egalité
!=	Difference

Leur syntaxe est : **expression1 op expression2**

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type int (il n'y a pas de type booléen en C); elle vaut **1** si la condition est vraie, et **0** sinon.

Attention: Ne pas confondre l'opérateur de test d'égalité **==** avec l'opérateur d'affectation **=**.

Les opérateurs logiques

Les opérateurs logiques permettent de combiner le résultat de plusieurs conditions en une seule expression logique

&&	ET Logique
	OU Logique
!	NON Logique

Leur syntaxe est : **expression1 op expression2**

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un **int** qui vaut **1** si la condition est vraie et **0** sinon.

```
#include<stdio.h>

int main(){
    int a = 5, b;
    b = (a > 4) && (a < 10);
    printf("b = %d", b);
    b = (a < 3) || (b > 15);
    printf("b = %d", b);
}
```


Les opérateurs d'incrementation et décré.

Les opérateurs d'incrémentation ++ et de décrémentation -- s'utilisent aussi bien en suffixe (i++) qu'en préfixe (++i). Dans les deux cas la variable i sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe, ce sera la nouvelle. Par exemple:

```
#include<stdio.h>

int main(){
    int x, y, z;
    x = 7;
    y = ++x;
    printf("x = %d et y = %d\n", x,y);
    z = y++;
    printf("y = %d et z = %d\n", y,z);
    return 0;
}
```

NB:

L'opérateur d'incrément (resp. de décrémentation) est un opérateur unaire et n'est valable que pour les entiers.

Les opérateurs de conversion de types (cast)

Les conversions de types (ou **casts**) permettent de passer d'un type de base à un autre type de base. La conversion peut être :

- **implicite** : elle est alors effectuée directement par le compilateur. Par exemple, la somme d'une variable de type float et d'une autre de type int sera automatiquement de type float; pas besoin donc de le signaler au compilateur.
- **explicite** : en utilisant la syntaxe suivante : **(unType)objet**; où *objet* peut être un nom de variable ou une expression à convertir dans le type *unType*.

Conversion Implicite

```
#include<stdio.h>

int main(){
    float a = 9.5;
    int b = 7;
    printf("a + b = %f \n", a+b); //Affiche 16.5
    return 0;
}
```

Conversion Explicite

```
#include<stdio.h>

int main(){
    int a = 22 ,b = 7;
    printf("a/b = %f \n", (float)a/b); //Affiche
3.142857
    return 0;
}
```

Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

$+=$

$-=$

$*=$

$/=$

$\%=$

Pour tout opérateur *op*, l'expression

$\text{expression-1 } op= \text{expression-2}$

est équivalente à

$\text{expression-1} = \text{expression-1 } op \text{ expression-2}$

Toutefois, avec l'affectation composée, *expression-1* n'est évaluée qu'une seule fois

L'opérateur conditionnel ternaire et d'adresse

L'opérateur conditionnel `?` est un opérateur ternaire. Sa syntaxe est la suivante :

`condition ? expression1 : expression2;`

Cette expression est égale à *expression1* si *condition* est satisfaite, et à *expression2* sinon.

```
#include<stdio.h>

int main(){
    int x, y, z;
    printf("Entrez deux entiers");
    scanf("%d%d",&x,&y);
    z = (x>y) ? x : y ; //affecte a z le max entre x et y
    printf("z = %d", z);
}
```

```
#include<stdio.h>

int main(){
    int x;
    printf("Entrez un entier");
    scanf("%d",&x);
    //Affiche la valeur absolue de x
    printf("x = %d", ((x>=0) ? x : -x));
}
```

L'opérateur d'adresse `&` appliqué à une variable retourne l'adresse mémoire de cette variable. La syntaxe est:

`&objet`

Fonctions d'entrée et sorties

La fonction `printf` est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est :

`printf("chaîne de contrôle ", expression-1, ..., expression-n);`

La chaîne de contrôle contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications de format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère %, suivi d'un caractère désignant le format d'impression.

format	conversion en	écriture
%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée
%f	double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

Fonctions d'entrée et sorties

La fonction `scanf` permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonction.

`scanf("format",argument-1,...,argument-n);`

On indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de `\n`). Comme pour `printf`, les conversions de format sont spécifiées par un caractère précédé du signe `%`. Les formats valides pour la fonction `scanf` diffèrent légèrement de ceux de la fonction `printf`.

Exemple:

```
scanf("%d", &a); // a est un entier
scanf("%c", &b); // b un caractère
scanf("%f", &c); // c un float
```

format	type d'objet pointé	représentation de la donnée saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères

Impression et lecture de caractères

Les fonctions **getchar** et **putchar** permettent respectivement de lire et d'imprimer des caractères. Il s'agit de fonctions d'entrées-sorties non formatées.

La fonction **getchar** retourne un **int** correspondant au caractère lu. Pour mettre le caractère lu dans une variable **char**, on écrit :

```
caractere = getchar();
```

Lorsqu'elle détecte la fin de fichier, elle retourne l'entier EOF (End Of File : *valeur définie dans la librairie **stdio.h***). En général, la constante EOF vaut -1.

La fonction **putchar** écrit caractère sur la sortie standard:

```
putchar(caractere);
```

Elle retourne un **int** correspondant à l'entier lu ou à la constante EOF en cas d'erreur.

Branchement conditionnel: If..Else

La forme la plus générale est celle-ci :

```
if ( condition1 ){  
    instructions;  
} else if ( condition2 ) {  
    instructions;  
}  
  
...  
} else if ( conditionN ) {  
    instructions;  
} else {  
    instructions;  
}
```

avec un nombre quelconque de *else if* (...). Le dernier *else* est toujours facultatif.

La forme la plus simple est :

```
if ( condition ){  
    instructions;  
} else {  
    instructions;  
}
```

```
if ( condition ){  
    instructions;  
}
```

Les conditions seront évaluées en fonction de leur ordre d'apparition. Une condition vraie correspond à l'exécution de l'action correspondante.

A Faire: Ecrire un programme C qui teste si un nombre saisi par l'utilisateur est nul ou non.

Branchement multiple: Switch

Sa forme la plus générale est celle-ci :

```
switch(expression){  
    case constante1:  
        instructions;  
        break;  
    case constante2:  
        instructions;  
        break;  
    .....  
    .....  
    case constanteN:  
        instructions;  
        break;  
    default:  
        instructions;  
        break;  
}
```

Si la valeur de **expression** est égale à l'une des **constantes**, la liste d'instructions correspondant est exécutée. Sinon la liste d'instructions correspondant à **default** est exécutée. L'instruction **default** est facultative.

```
#include<stdio.h>  
  
int main(){  
    char rep;  
    printf("Entrer le caractere 'N' pour Non ou 'O' \n");  
    rep = getchar();  
    switch(rep){  
        case 'N': printf("Vous avez repondu Non\n");  
                   break;  
        case 'O': printf("Vous avez repondu Oui\n");  
                   break;  
        default:  
            printf("Caractere non pris en compte");  
            break;  
    }  
}
```

Les boucles : For

La syntaxe de for est :

```
for(exp1; exp2; exp3){  
    instructions;  
}
```

Par exemple, pour imprimer tous les entiers de 1 à 25, on écrit :

```
#include<stdio.h>  
  
int main(){  
    int i;  
    for(i=1;i<=25;i++){  
        printf("%d \t", i);  
    }  
}
```

A la fin de cette boucle, i vaudra 26.

A faire : Ecrire un programme qui permet de calculer la factorielle d'un entier donné par l'utilisateur.

Les boucles : While

La syntaxe de `while` est la suivante :

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

Tant que `expression` est vérifiée, `instruction` est exécutée. Si `expression` est nulle au départ, `instruction` ne sera jamais exécutée. `instruction` peut évidemment être une instruction composé. Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
#include<stdio.h>  
  
int main(){  
    int i;  
    i = 1;  
    while(i < 10){  
        printf("%d \n", i);  
        i++;  
    }  
}
```

Les boucles: Do ... while

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle **do---while**. Sa syntaxe est :

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

L'action est exécutée tant que *expression* reste vraie. A noter qu'il y'a au moins une exécution !

```
#include<stdio.h>

int main(){
    int b;
    do{
        printf("Entrez un nombre entre 1 et 10 \n");
        scanf("%d",&b);
    } while((b <= 0 ) || ( b > 10));
}
```

Branchements non conditionnels : Break

On a vu le rôle de l'instruction **break** au sein d'une instruction de branchement multiple *switch*. L'instruction **break** peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, break fait sortir de la boucle la plus interne. Par exemple, le programme suivant :

```
#include<stdio.h>

int main(){
    int i;
    for (i = 0; i < 5; i++){
        printf("i = %d\n", i);
        if (i == 3)
            break;
    }
    printf("A la sortie de la boucle, i = %d\n", i);
    return 0;
}
```

Branchements non conditionnels : Continue

L'instruction **continue** permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
#include<stdio.h>

int main(){
    int i;
    for (i = 0; i < 5; i++){
        if (i == 3)
            continue;
        printf("i = %d\n", i);
    }
    printf("A la sortie de la boucle, i = %d\n", i);
    return 0;
}
```

Structures de données : Tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contigües. La déclaration d'un tableau à une dimension se fait de la façon suivante:

```
type nomTableau[nombre-éléments];
```

Où *nombre-éléments* est une expression constante entière positive.

Par exemple, la déclaration : `int tab[10];` indique que `tab` est un tableau de 10 éléments de type `int`.

Cette déclaration alloue donc en mémoire pour l'objet `tab` un espace de **10×4 octets** consécutifs. Pour plus de clarté, il est recommandé de donner un nom à la constante `nombre-éléments` par une directive au préprocesseur, par exemple

```
#define nombre-éléments 10
```

On accède à un élément du tableau en lui appliquant l'opérateur `[]`. Les éléments d'un tableau sont toujours numérotés de **0** à **nombre-éléments -1**.

Les structures de données : Tableaux

Initialisation et affichage des éléments d'un tableau

```
#include<stdio.h>

#define TAILLE 10

int main(){
    //Declaration d'un tableau de 10 elements
    int tab[TAILLE];
    int i;

    //Initialisation
    for(i=0; i<TAILLE;i++){
        printf("Entrez l'element %d\n", i+1);
        scanf("%d",&tab[i]);
    }

    //Affichage du tableau
    for(i=0; i<TAILLE;i++){
        printf("%d\t", tab[i]);
    }
}
```

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

type nomTableau[N] = {cons1,const2,...,constN};

```
#include<stdio.h>

int main(){
    /*
    **Declaration et initialisation
    **d'un tableau de 5 elements
    */
    int tab[5]={100, -2, 9, 19, -45};
    int i;

    //Affichage du tableau
    for(i=0; i<TAILLE;i++){
        printf("%d\t", tab[i]);
    }
}
```

Initialisation Implicite
int tab[] = { 1, 2, 3 };

Les structures de données : Tableaux

De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple, pour un tableau à deux dimensions:

`type Tableau[nombreLignes][nombreColonnes];`

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression "`tableau[i][j]`".

```
#include<stdio.h>

#define N 3
#define M 2

int main(){
    //Declaration et initialisation d'un tableau 3L et 2C
    int tab[N][M] = {{1,2}, {3,4}, {5,6}};
    int i,j;
    //Affichage du tableau
    .....
}
```

```
#include<stdio.h>

#define N 5
#define M 3

int main(){
    //Declaration d'un tableau de 5 lignes et 3 colonnes
    int tab[N][M];
    int i, j;

    //Initialisation
    for(i=0; i<N; i++){
        for(j=0; j<M; j++){
            printf("Entrez l'element [%d][%d]\n", i+1,j+1);
            scanf("%d",&tab[i][j]);
        }
    }

    //Affichage du tableau
    for(i=0; i<N; i++){
        for(j=0; j<M; j++){
            printf("%d\t", tab[i][j]);
        }
        printf("\n");
    }
}
```

Structures de données: Chaînes de caractères

Le langage C ne propose pas de type prédéfini pour les chaînes de caractères. Elles y sont représentées soit par des tableaux de char, soit par des pointeurs sur le type char.

Une chaîne de N caractères va occuper en mémoire **N+1** octets : le dernier octet est occupé par le caractère de fin de chaîne `'\0'`.

On peut initialiser une chaîne de caractères de plusieurs manières :

- par une suite de char : `char chaine2[8]={'b','o','n','j','o','u','r','\0'};`
- par une chaîne littérale constante : `char chaine1[8]="bonjour";`
- en précisant une taille supérieure au nombre de caractères de la chaîne :

`char ch3[20]="chaîne";`

- en ne précisant pas la taille (le compilateur va choisir la taille adéquate) :

`char chaine1[]="hello";`

Structures de données: Chaînes de caractères

Remarques:

char MACHAINE[5] = "Hello"; donnera une erreur à l'exécution

char MACHAINE[4] = "Hello"; donnera une erreur à la compilation.

- 'x' désigne un caractère constant et occupe **1 octet** en mémoire.
- "x" désigne une chaîne de caractères constituée de deux caractères ('x' et '\0') et occupe **2 octets** en mémoire.
- "" est la chaîne vide, et " " la chaîne constituée du caractère espace et du caractère '\0'.

Lecture – Ecriture

- Ecriture d'une chaîne : la fonction printf avec le format %s, la fonction puts.
- Lecture d'une chaîne : fonction scanf avec le format %s, la fonction gets

Structures de données: Chaînes de caractères

Exemples:

```
#include<stdio.h>

int main(){
    char nom[25], prenom[30];
    printf("Entrez votre nom\n");
    gets(nom);
    printf("Entrez votre prenom\n");
    gets(prenom);
    printf("Nom = ");
    puts(nom);
    printf("\tPrenom = ");
    puts(prenom);
}
```

```
#include<stdio.h>

int main(){
    char nom[25], prenom[30];
    printf("Entrez votre nom\n");
    scanf("%s", nom);
    printf("Entrez votre prenom\n");
    scanf("%s", prenom);
    printf("Nom = %s \t Prenom = %s ", nom, prenom);
}
```

Remarque :

Contrairement à la fonction **gets**, la fonction **scanf** ne lit pas les espaces/tabulations présents dans une chaîne : la lecture s'arrête dès que la fonction en rencontre un.

Structures de données: Chaînes de caractères

Pour manipuler les chaînes de caractères, il existe une série de fonctions prédéfinies dans la librairie `string.h`.

prototype	action
<code>char *strcpy(char *ch1, char *ch2);</code>	copie le contenu de la chaîne <code>ch2</code> dans la chaîne <code>ch1</code> et retourne la chaîne <code>ch1</code>
<code>char *strncpy(char *ch1, char *ch2, int n);</code>	copie au plus <code>n</code> caractères de la chaîne <code>ch2</code> dans la chaîne <code>ch1</code> et retourne <code>ch1</code>
<code>char *strcat(char *ch1, char *ch2);</code>	concatène la chaîne <code>ch2</code> à la suite de <code>ch1</code> (après avoir supprimé <code>\0</code>) et retourne <code>ch1</code>
<code>char *strncat(char *ch1, char *ch2, int n);</code>	concatène au plus <code>n</code> caractères de la chaîne <code>ch2</code> à la suite de <code>ch1</code> et retourne <code>ch1</code>
<code>char *strrev(char *str);</code>	renvoie une chaîne après avoir inversé la chaîne <code>str</code> .

Structures de données: Chaînes de caractères

Exemple:

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

int main(){
    char nom[25], nomCopie[25];
    printf("\nEntrez votre nom :");
    gets(nom);
    strcpy(nomCopie, nom);
    printf("Le nom copie = %s", nomCopie);
    return 0;
}
```

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

int main(){
    char str1[50] = "Bonjour, "
    char str2[30] = "Bienvenue dans votre espace";
    strcat(str1, str2);
    printf("%s", str1);
    return 0;
}
```

Structures de données: Chaînes de caractères

prototype	action
<code>int strcmp(char *ch1, char *ch2);</code>	compare les deux chaînes ch1 et ch2 pour l'ordre lexicographique et renvoie un entier (égal à 0 si ch1 est égal à ch2, valeur positive si ch1 supérieure à ch2, valeur négative si ch1 inférieure à ch2)
<code>int strncmp(char *ch1, char *ch2, int n);</code>	compare les n premiers caractères des deux chaînes ch1 et ch2 pour l'ordre lexicographique et renvoie un entier (égal à 0 si ch1 est égal à ch2, valeur positive si ch1 supérieure à ch2, valeur négative si ch1 inférieure à ch2)
<code>char *strchr(char *ch, char c);</code>	recherche depuis le début le caractère c dans la chaîne ch. Elle renvoie NULL si c n'est pas dans ch.
<code>char *strrchr(char *chaine, char c);</code>	idem que strchr, recherche à partir de la fin
<code>char *strstr(char *ch1, char *ch2);</code>	trouve la première occurrence de la chaîne ch2 dans la chaîne ch1 et NULL si ch2 n'y figure pas
<code>int strlen(char *chaine);</code>	retourne la longueur de chaîne (sans le caractère '\0')

Structures de données: Chaînes de caractères

Exemple:

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

int main(){
    char str1[] = "Bonjour"
    char str2[] = "bonJour";
    if(strcmp(str1, str2) == 0){
        printf("Les deux chaines sont identiques\n");
    }else{
        printf("Les deux chaines sont differentes");
    }
    return 0;
}
```

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

int main(){
    char str[25];
    int longueur;
    printf("Entrez une chaine\n");
    gets(str);
    longueur = strlen(str);
    printf("La longueur de la chaine = %d", longueur);
    return 0;
}
```


Structures de données : Structures

Une structure est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contigües en mémoire. Chaque élément de la structure, appelé **membre** ou **champ**, est désigné par un identificateur.

On distingue la déclaration d'un modèle de structure de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est `modele` suit la syntaxe suivante :

```
struct modele{  
    type1: membre1;  
    type2: membre2;  
    ....  
    typeN: membreN;  
};
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe:

```
struct modele objet;
```

Structures de données : Structures

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele{  
    type-1 membre-1;  
    type-2 membre-2;  
    type-3 membre-3;  
    ...  
    type-n membre-n;  
} objet;
```

On accède aux différents membres d'une structure grâce à l'opérateur membre de structure, noté « . ». Le i -ème membre de objet est désigné par l'expression

`objet.membre-i`

On peut effectuer sur le i -ème membre de la structure toutes les opérations valides sur des données de type `type-i`.

Structures de données : Structures

Exemple:

```
#include<stdio.h>

struct Etudiant{
    int id;
    char nom[20];
    char prenom[25];
};

int main(){
    struct Etudiant e;
    printf("Entrez votre identifiant");
    scanf("%d", &e.id);
    printf("Entrez votre nom");
    scanf("%s", &e.nom);
    printf("Entrez votre prenom");
    scanf("%s", &e.prenom);
    printf("Les infos entrees :| \nID = %d \t Nom = %s \t Prenom = %s ", e.id, e.nom, e.prenom);
}
```

Les structures de données : Enumérations

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clé **enum** et un identificateur de modèle, suivi de la liste des valeurs que peut prendre cet objet :

En réalité, les objets de type **enum** sont représentés comme des **int**. Les valeurs possibles *constante1*, *constante2*, ..., *constanteN* sont codées par des entiers de 0 à n-1.

Par exemple, le type **enum booleen** défini dans le programme suivant associe l'entier 0 à la valeur faux et l'entier 1 à la valeur vrai.

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple:

```
enum booleen {faux = 21, vrai = 33};
```

```
enum modele{constante1, constante2,...,constanteN}
```

```
#include<stdio.h>

int main(){
    enum booleen{faux, vrai};
    enum booleen b;
    b = vrai;
    printf("b = %d\n",b);
}
```

Définition de types composés : Typedef

Une définition de type permet de définir un type, c'est-à-dire d'en produire un nouveau ou, plus précisément, de créer un *alias* (un *synonyme*) d'un type existant. Une définition de type est identique à une déclaration de variable, si ce n'est que celle-ci doit être précédée du mot-clé **typedef** (pour type définition) et que l'identificateur ainsi choisi désignera un type et non une variable.

`typedef type synonyme;`

```
#include<stdio.h>

struct complexe{
    double reel;
    double imaginaire;
};
typedef struct complexe complexe;

enum boolean { false, true };
typedef enum boolean boolean;

int main(){
    complexe z;
    boolean b;
    ....
}
```

Pointeurs : Adresse et valeur d'un objet

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle adresse. Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets). Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

On appelle **Lvalue** (**left value**) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une *Lvalue* est caractérisée par :

- son adresse, l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- sa valeur, ce qui est stocké à cette adresse.

Pointeurs : Adresse et valeur d'un objet

```
int i, j;
```

```
i = 3, j = i;
```

Si le compilateur a placé la variable *i* à l'adresse 4831836000 en mémoire, et la variable *j* à l'adresse 4831836004, on a :

objet	adresse	valeur
<i>i</i>	4831836000	3
<i>j</i>	4831836004	3

Deux variables différentes ont des adresses différentes. L'affectation *j* = *i*; n'opère que sur les valeurs des variables.

L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier quelque soit le type de l'objet considéré.

L'opérateur **&** permet d'accéder à l'adresse d'une variable. Toutefois **&i** n'est pas une *Lvalue* mais une constante : on ne peut pas faire figurer **&i** à gauche d'un opérateur d'affectation.

Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs

Pointeurs : Notion

Un pointeur est un objet dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction:

`type *nom-du-pointeur;`

Où `type` est le type de l'objet pointé. Cette déclaration déclare un identificateur, `nom-du-pointeur`, associé à un objet dont la valeur est l'adresse d'un autre objet de type `type`.

L'identificateur `nom-du-pointeur` est donc en quelque sorte un identificateur d'adresse.

Même si la valeur d'un pointeur est toujours un entier (éventuellement un entier long), le type d'un pointeur dépend du type de l'objet vers lequel il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur.

En effet, pour un pointeur sur un objet de type `char`, la valeur donne l'adresse de l'octet où cet objet est stocké. Par contre, pour un pointeur sur un objet de type `int`, la valeur donne l'adresse du premier des 4 octets où l'objet est stocké.

Pointeurs : Notion

Dans l'exemple suivant, on définit un pointeur **p** qui pointe vers un entier **i** :

```
int main(){
    int i = 3;
    int* p ;
    p = &i;
}
```

On se trouve dans la configuration :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

L'opérateur unaire d'indirection ***** permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si **p** est un pointeur vers un entier **i**, ***p** désigne la valeur de **i**.

```
int main(){
    int i = 3;
    int* p ;
    p = &i;
    printf("*p = %d\n", *p);
}
```

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Pointeurs : Notion

```
int main(){
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    *p1 = *p2;
}
```

```
int main(){
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    p1 = p2;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type:

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Pointeurs : Notion

Après l'affectation `*p1 = *p2`; du premier programme, on a :

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Par contre, l'affectation `p1 = p2` du second programme, conduit à la situation:

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

Pointeurs : Arithmétique

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ;
- la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ;
- la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

NB: Notons que la somme de deux pointeurs n'est pas autorisée.

Pointeurs : Arithmétique

Si i est un entier et p est un pointeur sur un objet de type *type*, l'expression $p + i$ désigne un pointeur sur un objet de type *type* dont la valeur est égale à la valeur de p **incrémentée de $i * \text{sizeof}(\text{type})$** . Il en va de même pour la **soustraction d'un entier à un pointeur**, et pour les opérateurs d'incrément et de décrémentation $++$ et $--$. Par exemple, le programme

```
#include<stdio.h>

int main(){
    int i = 4;
    int *p1, *p2;
    p1 = &i;
    p2 = p1 + 1;
    printf("p1 = %ld\t p2 = %ld\n", p1, p2);
}
```

Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

Si p et q sont deux pointeurs sur des objets de type *type*, l'expression $p - q$ désigne un entier dont la valeur est égale à $(p - q)/\text{sizeof}(\text{type})$.

Pointeurs : Allocation dynamique

Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection `*`, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée `NULL` définie dans `stdio.h`. En général, cette constante vaut `0`. Le test `p == NULL` permet de savoir si le pointeur `p` pointe vers un objet.

On peut initialiser un pointeur `p` par une affectation sur `p`. Par exemple, on peut affecter à `p` l'adresse d'une autre variable. Il est également possible d'affecter directement une valeur à `*p`. Mais pour cela, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate.

L'adresse de cet espace-mémoire sera la valeur de `p`. Cette opération consistant à réserver un espace-mémoire pour stocker l'objet pointé s'appelle allocation dynamique. Elle se fait en C par la fonction `malloc` de la librairie standard `stdlib.h`. Sa syntaxe est :

```
malloc(nombre-octets);
```

Pointeurs : Allocation dynamique

Cette fonction retourne un pointeur de type **char** pointant vers un objet de taille **nombre-octets** octets. Pour initialiser des pointeurs vers des objets qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction **malloc** à l'aide d'un **cast**. L'argument nombre-octets est souvent donné à l'aide de la fonction **sizeof()** qui renvoie le nombre d'octets utilisés pour stocker un objet.

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

int main(){
    int i = 3;
    int *p;
    printf("La valeur de p avant initialisation = %ld\n", p);
    p = (int*)malloc(sizeof(int));
    printf("La valeur de p apres initialisation = %ld\n", p);
    *p = i;
    printf("valeur de *p = %d\n", *p);
}
```

Pointeurs : Allocation dynamique

Avant l'allocation dynamique, on se trouve dans la configuration.

objet	adresse	valeur
i	4831836000	3
p	4831836004	0

A ce stade, ***p** n'a aucun sens. En particulier, toute manipulation de la variable ***p** génèrerait une violation mémoire, détectable à l'exécution par le message d'erreur **Segmentation fault**.

L'allocation dynamique a pour résultat d'attribuer une valeur à **p** et de réserver à cette adresse un espace-mémoire composé de 4 octets pour stocker la valeur de ***p**.

On a alors :

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	? (int)

***p** est maintenant définie mais sa valeur n'est pas initialisée. Cela signifie que ***p** peut valoir n'importe quel entier (celui qui se trouvait précédemment à cette adresse).

L'affectation ***p = i;** a enfin pour résultat d'affecter à ***p** la valeur de **i**. A la fin du programme, on a donc

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	3

Pointeurs : Allocation dynamique

La fonction **calloc** de la librairie **stdlib.h** a le même rôle que la fonction **malloc** mais elle initialise en plus l'objet pointé ***p** à **zéro**. Sa syntaxe est :

```
calloc(nb-objets, taille-objets);
```

Ainsi, si **p** est de type **int***, l'instruction

```
p = (int*)calloc(N,sizeof(int));           ⇔           p = (int*)malloc(N * sizeof(int));  
                                                for (i = 0; i < N; i++)  
                                                    *(p + i) = 0;
```

L'emploi de **calloc** est simplement plus rapide.

Enfin, lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement, il faut libérer cette place en mémoire. Ceci se fait à l'aide de l'instruction **free** qui a pour syntaxe :

```
free(nom-du-pointeur);
```

A toute instruction de type **malloc** ou **calloc** doit être associée une instruction de type **free**.