Chapitre 2

Syntaxe de base du langage C

Sommaire

2.1 Les composants élémentaires du C

Un programme en langage `C` est constitué des six **unités lexicales** (groupes de composants élémentaires) suivantes :
- les mots-clefs,
- les identificateurs,
- 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.

2.1.1 Les mots-clefs

Un certain nombre de mots, appelés mots-clefs, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI `C` compte 32 mots clefs 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`

2.1.2 Les identificateurs

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`, `union` ou `enum`,
- une étiquette.

Un identificateur est une suite de caractères parmi :
- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres,
- le "blanc souligné" (_).

Le premier caractère d'un identificateur ne peut pas être un chiffre. Par exemple, `var1`, `tab_23` 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`.

Les majuscules et minuscules sont différenciées.

Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur. Cette limite dépend des implémentations, mais elle est toujours supérieure à 31 caractères. (Le standard dit que les identificateurs externes, c'est-à-dire ceux qui sont exportés à l'édition de lien, peuvent être tronqués à 6 caractères, mais tous les compilateurs modernes distinguent au moins 31 caractères).

2.1.3 Les commentaires

Un commentaire débute par `/*` et se termine par `*/`. Par exemple, `/*` Ceci est un commentaire `*/` On ne peut pas imbriquer des commentaires. Quand on met en commentaire un morceau de programme, il faut donc veiller à ce que celui-ci ne contienne pas de commentaire. Tout ce qui suit les caractères `//` sur la même ligne est également considéré comme un commentaire.

2.2 Structure d'un programme C

Une expression est définie comme toute combinaison qui possède une valeur, cette valeur est nécessairement associée à un type. Exemples d'expression :

- Une constante : `'a', 1, 1.0, 1 E 23, ''TOTO''`
- L'appel à un opérateur : `1 + 2; i >2 ;`
- L'appel à une fonction : `f(3)`

Une instruction est une expression suivie par un `;` (point virgule). Elle contient soit une expression, soit une expression et une affectation, soit une déclaration de variable :

```
1 ;
1 + 2;
i = 2+3;
```

Un bloc d'instructions est un ensemble d'instructions englobées par une paire d'accolades. Cette paire d'accolade définit un contexte dans lequel on peut définir des variables :

```
 { 1 ; int i ; }
```

Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs séparés par une virgule est une déclaration.Par exemple,

```
int a;
int b = 1, c;
double x = 2.38e4;
char message[80];
```

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

Un programme C se présente de la façon suivante :

```
[directives au préprocesseur]
[déclarations de variables externes]
[fonctions secondaires]

main()
{déclarations de variables internes
  instructions
}
```

La fonction principale `main` peut avoir des paramètres formels. On supposera dans un premier temps que la fonction `main` n'a pas de valeur de retour. Ceci est toléré par le compilateur mais produit un message d'avertissement quand on utilise l'option `-Wall` de `gcc`.

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale. Une fonction secondaire peut se décrire de la manière suivante :

```
type ma_fonction ( arguments )
{déclarations de variables internes
  instructions
}
```

Cette fonction retournera un objet dont le type sera `type` (à l'aide d'une instruction comme `return objet;`). Les arguments de la fonction obéissent à une syntaxe voisine de celle des déclarations : on met en argument de la fonction une suite d'expressions `type objet` séparées par des virgules. Par exemple, la fonction secondaire suivante calcule le produit de deux entiers :

```
int produit(int a, int b)
{
  int resultat;

  resultat = a * b;
  return(resultat);
}
```

2.3 Les conventions d'écriture d'un programme C

Il existe très peu de contraintes dans l'écriture d'un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles. Aussi existe-t-il un certain nombre de conventions.

- On n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne.

- Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne.
- On laisse un blanc :
  - entre les mots-clefs `if, while, do, switch` et la parenthèse ouvrante qui suit,
  - après une virgule,
  - de part et d'autre d'un opérateur binaire.
- On ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composée.
- Les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées. Le mieux est d'utiliser le mode `C` d'Emacs.

2.4 Les types de base

Les types de bases sont : `char`, `int`, `short`, `float`, `double`. Ces types de base peuvent être étendus en utilisant les mots clefs `union`, `struct`, `typedef`.

Un type définit une taille mémoire et un codage de cet espace. Le mot `sizeof` permet d'avoir le nombre d'octets occupé par un type.

```
int  main(int argc, char **argv)
{
  printf(" La taille d'un short %d \n", sizeof(short));
  printf(" La taille d'un int   %d \n", sizeof(int));
  printf(" La taille d'un long  %d \n", sizeof(long));
  printf(" La taille d'un float %d \n", sizeof(float));
  printf(" La taille d'un double %d \n", sizeof(double));
  printf(" La taille d'un char  %d \n", sizeof(char));
  return 1;
}
```

On peut aussi utiliser le mot `unsigned` qui oblige a n'utiliser que les valeurs positives mais qui changent aussi le codage et son interprétation.

2.4.1 Le type caractère

Le mot-clef `char` désigne un objet de type caractère. Un `char` peut contenir n'importe quel élément du jeu de caractères de la machine utilisée. La plupart du temps, un objet de type `char` est codé sur un octet ; c'est l'objet le plus élémentaire en `C`. Le jeu de caractères utilisé correspond généralement au codage ASCII (sur 7 bits). La plupart des machines utilisent désormais le jeu de caractères ISO-8859 (sur 8 bits), dont les 128 premiers caractères correspondent aux caractères ASCII. Les 128 derniers caractères (codés sur 8 bits) sont utilisés pour les caractères propres aux différentes langues. La version ISO-8859-1 (aussi appelée

ISO-LATIN-1) est utilisée pour les langues d'Europe occidentale. Ainsi, le caractère de code 232 est le è, le caractère 233 correspond au é ...

Une des particularités du type char en `C` est qu'il peut être assimilé à un entier : tout objet de type char peut être utilisé dans une expression qui utilise des objets de type entier. Par exemple, si `c` est de type char, l'expression `c + 1` est valide. Elle désigne le caractère suivant dans le code ASCII. La table précédente donne le code ASCII (en décimal, en octal et en hexadécimal) des caractères imprimables. Ainsi, le programme suivant imprime le caractère 'B'.

```
main()
{
  char c = 'A';
  printf("%c", c + 1);
}
```

Suivant les implémentations, le type `char` est signé ou non. En cas de doute, il vaut mieux préciser `unsigned char` ou `signed char`. Notons que tous les caractères imprimables sont positifs.

2.4.2 Les types entiers

Le mot-clef désignant le type entier est `int` (integer). Un objet de type `int` est représenté par un mot de 32 bits pour une machine 32 bits ou 64 bits.

Le type `int` peut être précédé d'un attribut de précision (`short` ou `long`) et/ou d'un attribut de représentation (`unsigned`).Un objet de type `short int` a au moins la taille d'un `char` et au plus la taille d'un `int`. En général, un `short int` est codé sur 16 bits. Un objet de type `long int` a au moins la taille d'un `int` (32 bits sur une machine 32 bits et 64 bits sur une machine 64 bits).

Le bit de poids fort d'un entier est son signe. Un entier positif est donc représenté en mémoire par la suite de 32 bits dont le bit de poids fort vaut 0 et les 31 autres bits correspondent à la décomposition de l'entier en base 2. Par exemple, pour des objets de type `char` (8 bits), l'entier positif `12` sera représenté en mémoire par `00001100`. Un entier négatif est, lui, représenté par une suite de 32 bits dont le bit de poids fort vaut 1 et les 31 autres bits correspondent à la valeur absolue de l'entier représentée suivant la technique dite du "complément à 2". Cela signifie que l'on exprime la valeur absolue de l'entier sous forme binaire, que l'on prend le complémentaire bit-à-bit de cette valeur et que l'on ajoute 1 au résultat. Ainsi, pour des objets de type `signed char` (8 bits), -1 sera représenté par `11111111`, -2 par `11111110`, -12 par par `11110100`. Un `int` peut donc représenter un entier entre -2^{31} et $(2^{31}-1)$. L'attribut `unsigned` spécifie que l'entier n'a pas de signe. Un `unsigned int` peut donc représenter un entier entre 0 et $(2^{32}-1)$. Sur une machine 64 bits, en fonction de la taille des données à stocker, on utilisera un des types suivants :

- `signed char` $[-2^7, 2^7[$,
- `unsigned char` $[0, 2^8[$,
- `short int` $[-2^{15}7, 2^{15}[$,

– `unsigned short int` $[0, 2^{16}[$,
– `int` $[-2^{31}, 2^{31}[$,
– `unsigned int` $[0, 2^{32}[$,
– `long int` $[-2^{63}, 2^{63}[$,

Plus généralement, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard `limits.h`.

Le mot-clef `sizeof` a pour syntaxe `sizeof(expression)` où `expression` est un type ou un objet. Le résultat est un entier égal au nombre d'octets nécessaires pour stocker le type ou l'objet. Par exemple

```
unsigned short x;

taille = sizeof(unsigned short);
taille = sizeof(x);
```

Dans les deux cas, `taille` vaudra 4.

Pour obtenir des programmes portables, on s'efforcera de ne jamais présumer de la taille d'un objet de type entier. On utilisera toujours une des constantes de `limits.h` ou le résultat obtenu en appliquant l'opérateur `sizeof`.

2.4.3 Les types flottants

Les types `float`, `double` et `long double` servent à représenter des nombres en virgule flottante (floating point). Ils correspondent aux différentes précisions possibles.
– Floattants simple précision (single precision) : `float`, codés sur 32 bits, correspondent environ à une précision de 7 décimales dans une représentation décimale.
– Floattants double précision (double precision), `double`, codés sur 64 bits, correspondent environ à une précision de 16 décimales dans une représentation décimale.
– Floattant en précision étendue (extended precision), `long double`, codés sur 80 bits.

Les flottants sont généralement stockés en mémoire sous la représentation de la virgule flottante normalisée. On écrit le nombre sous la forme `signe 0,mantisse Bêxposant`. En général, B=2. Le digit de poids fort de la mantisse n'est jamais nul.

Un flottant est donc représenté par une suite de bits dont le bit de poids fort correspond au signe du nombre. Le champ du milieu correspond à la représentation binaire de l'exposant alors que les bits de poids faible servent à représenter la mantisse.

2.5 Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, énumération. Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

2.5.1 Les constantes entières

Une constante entière peut être représentée de 3 manières différentes suivant la base dans laquelle elle est écrite :

- décimale : par exemple, `0` et `2437348` sont des constantes entières décimales.
- octale : la représentation octale d'un entier correspond à sa décomposition en base `8`. Les constantes octales doivent commencer par un zéro. Par exemple, les représentations octales des entiers `0` et `255` sont respectivement `00` et `0377`.
- hexadécimale : la représentation hexadécimale d'un entier correspond à sa décomposition en base 16. Les lettres de `a` à `f` sont utilisées pour représenter les nombres de `10` à `15`. Les constantes hexadécimales doivent commencer par `0x` ou `0X`. Par exemple, les représentations hexadécimales de 14 et 255 sont respectivement `0xe` et `0xff`.

Par défaut, une constante décimale est représentée avec le format interne le plus court permettant de la représenter parmi les formats des types `int`, `long int` et `unsigned long int` tandis qu'une constante octale ou hexadécimale est représentée avec le format interne le plus court permettant encore de la représenter parmi les formats des types `int`, `unsigned int`, `long int` et `unsigned long int`.

On peut cependant spécifier explicitement le format d'une constante entière en la suffixant par `u` ou `U` pour indiquer qu'elle est non signée, ou en la suffixant par `l` ou `L` pour indiquer qu'elle est de type `long`. Par exemple :

constante	type
`1234`	`int`
`02322`	`int` /* octal */
`0x4D2`	`int` /* hexadécimal */
`123456789L`	`long`
`1234U`	`unsigned int`
`123456789UL`	`unsigned long int`

2.5.2 Les constantes réelles

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.

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

constante	type
`12.34`	`double`
`12.3e-4`	`double`
`12.34F`	`float`
`12.34L`	`long double`

2.5.3 Les constantes caractères

Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par ex. `'A'` ou `'$'`). 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 `\"`. Les caractères non imprimables peuvent être désignés par `\code-octal` où `code-octal` est le code en octal du caractère. On peut aussi écrire `\xcode-hexa` où `code-hexa` est le code en hexadécimal du caractère. Par exemple, `'\33'` et `\x1b` désignent le caractère escape. Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

- `\n` nouvelle ligne
- `\r` retour chariot
- `\t` tabulation horizontale
- `\f` saut de page
- `\v` tabulation verticale
- `\a` signal d'alerte
- `\b` retour arrière

2.5.4 Les constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple, `"Ceci est une chaîne de caractères"`. Une chaîne de caractères peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple,

```
"ligne 1 \n ligne 2"
```

A l'intérieur d'une chaîne de caractères, le caractère " doit être désigné par ⁘ Enfin, le caractère suivi d'un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple,

```
"ceci est une longue longue  longue longue longue longue longue longue\
chaîne de caractères"
```

2.6 Les variables

Une variable est un quadruplet :

- *Type* : représente le type de la variable et donc il définit la taille mémoire occupée par la variable ainsi que le codage de cette taille mémoire.
- *Nom* : représente l'identifiant de la variable.
- *Valeur* : représente la valeur de la variable.
- *Adresse mémoire* : représente l'emplacement de la variable.

Exemple :

```
int i = 1;
```

La variable de nom `i` est de type `int`, elle a pour valeur 1 et se situe à l'adresse `0x2333`. Pour évoquer la valeur de la variable il suffit de l'appeler :

```
printf("La valeur est %d ", i);
```

Les variables ont une **portée**, c'est-à-dire qu'elles n'existent qu'à partir de leur définition et uniquement dans le bloc où elles ont été définies. On appelle **variable globale**, les variables qui sont définies en dehors de tout bloc. Une variable peut *masquer* la définition d'une autre variable. Les **variables locales** (variables dans un bloc, variables passées en paramètre) sont toutes stockées dans la pile.

```
int var_globale = 1; // une variable globale
int  main(int argc, char **argv)
{
  // argc, et argv sont des variables locales qui sont valables
  // pour toute la fonction main;
  int var_loc = var_globale;
  // var_loc est une variable locale
  // on peut l'affecter avec var_globale qui est aussi présente.
  { // un nouveau bloc
    char var_loc = 'c'; // on redéfinit la variable locale
    // on ne peut plus atteindre int var_loc
  }
  // char var_loc n'existe par contre int var_loc est de nouveau accessible
}
```

2.7 Les opérateurs

2.7.1 L'affectation

L'affectation consiste à changer la valeur d'une variable ;

```
Variable = Expression ;
I = 1 ;
```

Si le type de l'expression est compatible avec le type de la variable alors les bits de l'expression vont devenir les bits de la variable avec possiblement une conversion (*cast*).

Si les types ne sont pas compatibles on peut forcer la conversion mais c'est au risque et péril de l'utilisateur. 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 :

```
int main(void)
```

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

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.

Pour qu'un type soit compatible avec un autre type il faut qu'il est une taille de codage supérieure.

Par exemple, `short` est compatible avec `int` et `long`. Quand on convertit un `short` en `int` ou en `long`, on parle de *promotion*.

```
short k = 1; long int j = k; // promotion de k en long int pour affectation.
```

Quand on force la conversion d'un type de codage plus grand vers un type de codage plus petit, par exemple `float` vers `short`, on parle de *cohercision*.

```
double k= 12343443,34553; short i = (short) k;
```

2.7.2 Les opérateurs arithmétiques

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

- `+` addition,
- `-` soustraction,
- `*` multiplication,
- `/` division,
- `%` reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes : Contrairement à d'autres langages, le `C` ne dispose que de la notation `/` pour désigner à la fois la division entière et la division entre flottants. 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. Par exemple :

```
float x;
x = 3 / 2;
```

affecte à `x` la valeur 1. Par contre

```
x = 3 / 2.;
```

affecte à `x` la valeur 1.5.

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.

Il n'y a pas en `C` d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction `pow(x,y)` de la librairie `math.h` pour calculer x^y.

2.7.3 Les opérateurs relationnels

`<`, `>`, `<=`, `>=`, `==` et `!=` (différent).

Leur syntaxe est

```
expression-1 op expression-2
```

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 à ne pas confondre l'opérateur de test d'égalité `==` avec l'opérateur d'affection `=`. Ainsi, le programme :

```
main()
{
  int a = 0;
  int b = 1;
  if (a = b)
    printf("\n a et b sont egaux \n");
  else
    printf("\n a et b sont differents \n");
}
```

imprime à l'écran `a` et `b` sont egaux !

2.7.4 Les opérateurs logiques booléens

- `&&` et logique
- `||` ou logique
- `!` négation logique

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.

Dans une expression de type

```
expression-1 op-1 expression-2 op-2 ...expression-n
```

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans

```
int i;
int p[10];

if ((i >= 0) && (i <= 9) && !(p[i] == 0))
...
```

la dernière clause ne sera pas évaluée si `i` n'est pas entre 0 et 9.

Il existe également six opérateurs qui permettent de manipuler les entiers au niveau du bit. Nous ne les détaillons pas dans ce cours.

2.7.5 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont :

```
+=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
```

Pour tout opérateur `op`, l'expression

```
expression-1 op= expression-2
```

est équivalente à `expression-1 = expression-1 op expression-2` Toutefois, avec l'affection composée, `expression-1` n'est évaluée qu'une seule fois.

2.7.6 Les opérateurs d'incrémentation et de décrémentation

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 se sera la nouvelle. Par exemple,

```
int a = 3, b, c;
b = ++a;      /* a et b valent 4 */
c = b++;      /* c vaut 4 et b vaut 5 */
```

2.7.7 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

```
expression-1, expression-2, ... , expression-n
```

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple, le programme :

```
main()
{
  int a, b;
  b = ((a = 3), (a + 2));
  printf("\n b = %d \n",b);
}
```

imprime `b = 5`.

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule. En particulier l'évaluation de gauche à droite n'est pas garantie.

2.7.8 L'opérateur conditionnel ternaire

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

```
condition   ? expression-1 : expression-2
```

Cette expression est égale à `expression-1` si condition est satisfaite, et à `expression-2` sinon. Par exemple, l'expression :

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre. De même l'instruction

```
m = ((a > b) ? a : b);
```

affecte à `m` le maximum de `a` et de `b`.

2.7.9 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé cast, permet de modifier explicitement le type d'un objet. On écrit

```
(type) objet
```

Par exemple,

```
main()
{
  int i = 3, j = 2;
  printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

2.7.10 L'opérateur adresse

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

```
&objet
```

2.7.11 Règles de priorité des opérateurs

Il existe un ordre de priorité sur les opérateurs (qu'on ne détaillera pas ici. On préferera toutefois mettre des parenthèses en cas de doute...

2.8 Instructions de branchement Conditionnel

2.8.1 Le test if - else

```
if (expression booleenne) instruction;
```

Si l'expression booléenne est vraie alors l'instruction est évaluée. En `C` la valeur 0 est fausse. Toutes les autres valeurs sont vraies. Néanmoins, il est préférable de montrer explicitement les conditions pour des raisons de lisibilités. Par exemple :

```
if(i%2) // pas très clair
   printf("le nombre i est impair \n");
```

Il vaut mieux écrire :

```
if(i%2 != 0) // plus clair
  printf("le nombre i est impair \n");
```

On peut aussi avoir une clause à réaliser si la condition booléenne est fausse, on utilise alors `else`.

```
if (condition)
   intruction;
else
   instruction.
```

Un exemple plus complexe :

```
int main(int argc, char **argv)
{
  int i = 3;
```

```
  if( i %2 == 0)
    printf(" La valeur de i est paire \n");
  if(i % 3 == 0)
    printf(" La valeur de i est un multiple de 3 \n");
  else
    {
      printf(" La valeur de i n'est pas un multiple de 3");
      printf(" \n");
    }
}
```

2.8.2 Le branchement multiple `switch case`

L'instruction `switch case` permet d'associer une suite d'instructions en fonction de la valeur d'une constante. On ne peut associer aux clauses `case` que des constantes. Par exemple, sur l'exemple ci-dessous, en fonction de la valeur de la variable `i`, on peut passer en mode "francais" ou en mode "anglais".

```
int  main(int argc, char **argv)
{
  int i = atoi(argv[1]);
  char mode = 'f';
  switch(i)
    {
    case 1:
      printf(" Le mode francais est activé \n");
      mode = 'f';
       break;
    case 2:
      printf("Le mode anglais est activé \n");
      mode = 'a';
      break;
    default :
      printf("Le mode n'est pas prevu");
    }
}
```

Le fonctionnement du `switch`/`case` est le suivant : l'ensemble des clauses `case` est parcouru selon l'ordre de leur déclaration (du premier `case` au dernier `case`). La première clause `case` dont la constante convient avec la valeur du `switch` est exécutée. Le programme continue à moins qu'il ne soit interrompu par une instruction `break`. S'il n'y a pas d'instruction `break` entre les clauses `case`, toutes les instructions des clauses `case` sont exécutées. Si aucune des constantes associées aux clauses `case` ne convient, la clause `default` est exécutée si elle est présente, sinon rien ne sera exécuté.

2.9 Les boucles

Il existe trois boucles en `C` : `for`, `while`, `do while`

2.9.1 Boucle `for`

Syntaxe :

```
for( init ; condition ; evolution)
   Instruction ;
```

Exemple :

```
for(int i = 0, int j = 10; i < j; i++,j--)
   printf(" je suis dans une boucle for \n");
```

1. La clause `init` est effectuée avant de rentrer dans la boucle.
2. La clause `condition` est vérifiée avant de commencer la boucle.
3. La clause `evolution` est effectuée après l'instruction de boucle et avant l'évaluation de la condition.

2.9.2 Boucle `while`

Syntaxe :

```
while(condition)
   Instruction ;
```

Exemple :

```
int i = 0; int j = 10;
while(i < j) {
   printf(" je suis dans une boucle while \n");
   i++; j--;
}
```

Seule la clause `condition` est obligatoire pour une boucle `while`. Mais l'on voit qu'il y a une partie `initialisation` et aussi une partie `evolution` à l'intérieur de la boucle. La condition est évaluée si elle est vraie alors les instructions du `while` sont exécutées, sinon on arrête la boucle. Si la condition n'est pas vrai on ne rentre pas dans la boucle.

2.9.3 Boucle `do while`

Syntaxe :

```
do
   instruction ;
while(condition) ;
```

Exemple :

```
int i = 0; int j = 10;
do {
   printf(" je suis dans une boucle while \n");
   i++; j--;
}while(i < j);
```

Les instructions du `do/while` sont, quoiqu'il arrive, exécutées au moins une fois, ensuite la boucle recommence si la condition du `while` est vérifiée.

2.10 Les instructions d'échappement `break; continue;`

L'instruction `continue` arrête l'itération courante quand elle rencontre cette instruction et l'itération recommence en début de boucle. Dans le cas, d'une boucle `for` les instructions correspondant à l'évolution sont exécutées et l'on recommence à évaluer la condition.

L'instruction `break` fait sortir purement et simplement de la boucle.

```
int  main(int argc, char **argv)
{
  for(int i=0; i < 50; i++){
      if(i%2 == 0)
continue;
      if(i == 21)
break;
      printf(" Valeur de l'indice %d \n", i);
    }
  int i = 0;
  while(i <= 50){
      i++;
      if(i%2 == 0)
continue;
      if(i == 21)
break;
      printf(" Valeur de l'indice %d \n", i);
    }
}
```

2.11 les types composés

2.11.1 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.

La déclaration d'un tableau à une dimension se fait de la façon suivante :

```
type nom-du-tableau[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
```

Les tableaux sont un type particulier. On peut donc définir des variables de ce type. Par exemple :

```
int tab[10] ;
```

déclare la variable de nom `tab`, dont le type est `int []`. La valeur d'une variable tableau est son adresse. On ne peut changer la valeur d'un tableau. Lors de la déclaration d'un tableau, la taille mémoire nécessaire est égale à la taille mémoire d'un élément multipliée par le nombre d'éléments, l'ensemble de la mémoire est constitué d'octets consécutifs. Sur cet exemple, la taille de `tab` est égale à `10 * sizeof(int)`.

Pour un tableau de 10 éléments, les indices pour accéder aux éléments vont de 0 à 9. Pour évoquer la valeur d'un élément d'un tableau on utilise `tab[3]`, `tab[i]`, ...

Attention le compilateur ne vérifie pas la validité de l'indice d'un tableau, c'est à l'utilisateur de le faire. ATTENTION AUX DEBORDEMENTS qui peuvent entrainer des "bugs" dont les conséquences n'apparaissent que beaucoup plus tard.

On peut directement initialisé un tableau sans avoir à définir sa taille. Exemple :

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

On peut aussi créer des tableaux bidimensionnel. `int tab[2][5]` initialise un tableau de 2 lignes, chaque ligne à 5 colonnes. Les éléments du tableau bidimensionnel sont consécutifs. On verra plus tard que cette implémentation ne correspond pas à un tableau de tableau.

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

```
#include <stdio.h>


void afficheTableau(int tab[], int taille) {
  for(int i = 0; i < taille; i++)
    printf("L'élément %d du tableau a pour valeur %d \n", i+1, tab[i]);
}

int  main(int argc, char **argv)
{
  int tab1[] = {1,2,3,4};
  printf("L'adresse du tableau est %x \n",tab1);
  printf("Le contenu d'un tableau est %x \n",&tab1);
  afficheTableau(tab1, 4);

  printf("\t** Remise des valeurs à 0 **\n");
  for(int i = 0; i < sizeof(tab1)/sizeof(int); i++)
    tab1[i] = 0;
  afficheTableau(tab1, 4);
}
```

2.11.2 Les structures.

Définition et utilisation d'une structure

La déclaration de structures permet de créer un nouveau type qui est composé de plusieurs champs. Par exemple, la déclaration suivante :

```
struct complex {
  double reel;
  double imaginaire;
};
```

crée un nouveau type qui s'appelle `struct complex`. Une fois le nouveau type déclaré, on peut déclarer des variables de ce type, par exemple :

```
struct complex c1 = {0.0,1.0};
```

Cette définition crée une nouvelle variable `c1`, qui est de type `struct complex`. La taille mémoire de cette variable est au moins de `2*sizeof(double)`. Pour initialiser une structure, on peut utiliser la même syntaxe que pour l'initialisation des tableaux. Sur cet exemple, le champ réel de la variable `c1` sera égal à 0.0 et le champ imaginaire de la variable `c1` sera égal à 1.0. Pour accéder au champ `x` d'une variable `v` qui est de type structure, on utilise la notation suivante : `v.x;`. Par exemple, pour la variable `struct complex c1` on peut écrire : `c1.imaginaire = c1.reel = 1;`

Affectation de structures

Pour l'affectation de structures, les bits de la première structure sont copiés dans la seconde structure. Si on déclare `struct complex c1, c2;` et si l'on écrit : `c1 = c2;`, ce code est équivalent à `c1.imaginaire = c2.imaginaire;` et `c1.reel = c2.reel;`. Il en est de même pour le passage de structures qui se fait par valeur et aussi pour la structure correspondant à un retour de fonction. Par exemple :

```
struct complex additionComplexe(struct complex c1, struct complex c2)
{
  struct complex tmp = {0,0};
  tmp.reel = c1.reel + c2.reel;
  tmp.imaginaire = c1.imaginaire + c2.imaginaire;
  return tmp;
}
```

Par exemple, on peut écrire le code suivant :

```
struct complex c1 = {1,1}
struct complex c2 = {2,2};
struct complex resultat = {0,0};
resultat = additionComplexe(c1,c2);
```

Comparaison de structures

Si on peut affecter une structure en une seule fois, ce qui correspond à une affectation champ par champ, on ne peut pas comparer deux structures directement. Par exemple, l'opération `c1==c2` n'est pas correcte syntaxiquement. Il faut faire alors une comparaison champ par champ. Il faut donc écrire le code suivant :

```
(c1.reel == c2.reel ) && (c1.imaginaire == c2.imaginaire)
```

2.11.3 Les unions

Une union désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Les déclarations et les opérations sur les objets de type `union` sont les mêmes que celles sur les objets de type `struct`. Dans l'exemple suivant, la variable hier de type `union jour` peut être soit un entier,

```
soit un caractère.
```

```
union jour
{
  char lettre;
  int numero;
};

main()
{
  union jour hier, demain;
  hier.lettre = 'J';
  printf("hier = %c\n",hier.lettre);
  hier.numero = 4;
  demain.numero = (hier.numero + 2) % 7;
  printf("demain = %d\n",demain.numero);
}
```

2.11.4 Le mot clef `typedef`

Le mot clef `typedef` permet de définir des types synonymes. Par exemple, la déclaration suivante :

```
typedef int ValeurDiscreteTemperature;
```

crée un nouveau type qui s'appelle `ValeurDiscrete` et qui est équivalent au type `int`. Le code suivant illustre la synonymie de type. On peut affecter des `int` à des `ValeursDiscreteTemperature` et réciproquement.

```
int i =1;
ValeurDiscreteTemperature v =1;
  i = v;
  v = i;
```

L'intérêt de l'instruction `typedef` est d'augmenter la lisibilité d'un programme en précisant le type d'une variable. En effet, il est plus lisible de savoir que l'on attend une variable de type `ValeurDiscreteTemperature` qu'une variable de type `int`. De plus il sera plus facile plus tard de changer le codage de `ValeurDiscreteTemperature` en un `long` par exemple. Par le code suivant : `typedef long ValeurDiscreteTemperature;` .

Syntaxiquement : La définition d'un nouveau type correspond à la déclaration d'une variable précédée du mot clef `typedef`. Par exemple, `typedef struct complex Complex;` crée un nouveau type `Complex` qui est un synonyme de `struct complex`.

2.12 Les pointeurs

2.12.1 Notions d'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, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- sa valeur, c'est-à-dire ce qui est stocké à cette adresse.

Comme nous l'avons vu précedemment une variable est également définit par son nom et son type. Dans l'exemple,

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

Si le compilateur a placé la variable `i` à l'adresse `0x123` en mémoire, et la variable `j` à l'adresse `0x124`, on a alors placé la valeur 3 dans les adresses `0x123` et `0x124`

Deux variables différentes ont des adresses différentes. L'affectation `i = j;` 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. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures.

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.

2.12.2 Définition des pointeurs

Nous avons vu qu'une variable est un quadruplet, dont un champ est une adresse. L'idée des pointeurs est d'utiliser une adresse pour accéder au contenu de la mémoire plutôt que de passer par une variable. On va ainsi pouvoir considérer qu'une adresse mémoire peut-être la valeur d'une variable. Un pointeur est un nouveau type. La notation d'un pointeur est alors la suivante :

```
T *pointeur = NULL;
```

Dans ce cas, la variable de nom `pointeur` est de type `T*`. Avec `T` un type existant (`int`, `char`, `struct complexe`, ...). Par exemple,

```
int * tmp;
```

est la déclaration d'une variable `tmp` de type `int *`. Une manière d'initialiser un pointeur est d'utiliser l'adresse d'une variable existante. Pour obtenir l'adresse d'une variable on utilise l'opérateur `&`. Considérons la déclaration :

```
int i =1;
```

supposons que l'adresse de la variable `i` soit `ox123`, le contenu de l'adresse `ox123` contient les bits nécessaires au codage de l'entier 1. La valeur de l'expression `&i` est `ox123` c'est à dire l'adresse de la variable `i`. On peut donc maintenant initialiser un pointeur de la façon suivante.

```
int * tmp = &i;
```

Maintenant la variable `tmp` est définie comme le quadruplet :

```
< nom = tmp, valeur = ox123, adresse = 0x500, taille = "la taille d'un mot mémoire">
```

Tous les pointeurs occuperont donc la même taille mémoire pour coder une adresse. Cette propriété est importante comme on le verra par la suite. En effet le compilateur a besoin de la taille d'un type pour pouvoir compiler le code. Si on évoque la valeur de `tmp`, la valeur retournée sera le contenu de l'adresse du pointeur à savoir : `ox500`.

```
printf("La valeur du pointeur est %x \n", tmp);
```

Quand on utilise le symbole de l'affectation avec comme membre gauche un pointeur on change la référence du pointeur puisqu'on change la valeur du pointeur. Par exemple, si on considère la variable :`int j = 2;`, si la variable `j` est à l'adresse `ox200` alors l'instruction `tmp = &j;` change la valeur de `tmp` et maintenant le contenu de l'adresse `ox500` est `ox200`. Donc, affecter un pointeur implique le changement de la variable qu'il référence.

Maintenant, il existe un opérateur suplémentaire pour les pointeurs. Il s'agit de l'opérateur `*`. Écrire `*tmp` permet d'accéder au contenu du contenu de `tmp`. On a vu que écrire `tmp` consiste à accéder au contenu de `tmp`. Or comme le contenu de `tmp` est une adresse, écrire `*tmp` consiste à accéder au contenu de cette adresse. Donc, si le contenu de tmp est `ox200` alors `*tmp` fait réference au contenu de `ox200`, c'est à dire la valeur 2. L'opérateur `*tmp` peut être utilisé soit :

- en lecture comme pour l'instruction `i = *tmp;`. Dans ce cas, on met le contenu de `j` dans la variable `i`.
- en écriture comme pour l'instruction `*tmp = i;`. Dans ce cas, on met le contenu de la variable `i` dans le contenu de l'adresse `ox200`. C'est à dire qu'on met le contenu de `i` dans la variable `j;`

L'avantage des pointeurs c'est que l'on peut manipuler le contenu de la mémoire comme une valeur. Ce qui permet de changer des valeurs de variables sans connaitre leur nom.

On peut donc dans un programme manipuler à la fois les objets `p` et `*p`. Ces deux manipulations sont très différentes. Autre exemple :

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

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

2.12.3 L'arithmétique des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques.

La valeur d'un pointeur est une adresse. Par contre le contenu de l'adresse pointée dépend du type du pointeur. Et donc de la taille et du codage du type pointé. Pour un pointeur de type `T`, il faut `sizeof(T)` octets pour coder le type. Ainsi les opérateurs d'addition et de soustraction pour les pointeurs sont spécifiques aux pointeurs.

Soit un pointeur déclaré de la manière suivante : `T *tmp;` Si on écrit `tmp + 1` l'interprétation est la suivante, il ne s'agit pas d'avancer d'un mot mémoire mais de passer à l'adresse qui code l'élément suivant dans le pointeur. La valeur de `tmp + 1` est donc égale à la valeur de `tmp + sizeof(T)`.

Par exemple, le programme :

```
main()
{
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche `p1 = 4831835984 p2 = 4831835988`.

Par contre, le même programme avec des pointeurs sur des objets de type `double` :

```
main()
{
  double i = 3;
  double *p1, *p2;
  p1 = &i;
```

```
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche `p1 = 4831835984 p2 = 4831835992`.

Comme on peut adresser un pointeur en utilisant les crochets on a l'équivalence suivante `tmp[i]` est équivalente `*(tmp + i)`. L'expression `tmp + i` donne comme résultat l'adresse contenue dans `tmp + i*sizeof(T)`. L'opération de soustraction avec un entier produit le même résultat. L'expression `tmp - i` donne comme résultat l'adresse contenue dans `tmp - i*sizeof(T)`. Les opérateurs `++` et `--` donnent exactement les mêmes résultats. La derniére opération est la soustraction entre pointeur, soit les déclarations suivantes.

```
int tab[10];
int *tmpDebut = tab + 1;
int *tmpFin = tab +5;
```

La valeur `tmpFin - tmpDebut` doit représenter le nombre d'élements entre `tmpFin` et `tmpDebut`. Le résultat de cette opération est donc le suivant : `tmpFin - tmpDebut = (tab + 5*sizeof(int) - (tab + sizeof(int)) / sizeof(int) = 4* sizeof(int) / sizeof(int) = 4.`

```
int main(int argc, char **argv)
{
  int tab[] = {1,2,3,4,5,6,7,8, 9};
  int *tmpInt = (void *) 0;
  printf(" SIZEOF(int) = %d \n", sizeof(int));
  printf("La valeur de tmpInt est %x la valeur de tmpInt+1 est %x \n",
 tmpInt, tmpInt+1);
  printf ("La valeur de ++tmpInt est %x \n", ++tmpInt);
  char *tmpChar = (void *) 0;
  printf(" SIZEOF(char) = %d \n", sizeof(char));
  printf("La valeur de tmpChar est %x la valeur de tmpChar+1 est %x \n",
 tmpChar, tmpChar+1);

  printf ("La valeur de tmpChar++ est %x \n", ++tmpChar);
  int *tmpIntDebut = (void *) 0;
  int *tmpIntFin   = tmpIntDebut + 4;
  printf("La valeur de tmpIntDebut est %x la valeur de tmpIntFin est %x \n
          le nombre d'élement entre ces deux pointeurs est %d \n ",
          tmpIntDebut, tmpIntFin, tmpIntFin - tmpIntDebut);
}
```

2.12.4 La mémoire 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.

Pour l'instant, pour utiliser de la mémoire il est donc nécessaire de déclarer des variables. Ce qui implique que l'on connait au moment de la compilation toutes les variables nécessaires ou alors il existe un risque que le programme ne puisse plus continuer faute de mémoire où alors qu'il consomme trop de mémoire ce qui peut le pénaliser en terme de performance. Supposons par exemple, qu'un programme stocke des formulaires de connexion journalier. Le nombre de connexion varie d'un jour sur l'autre et il est difficile de savoir à l'avance combien de connexions sont disponibles, deux solutions sont alors possibles pour définir la variable `JournalConnexion`. Soit `Connexion journalConnexion [1000]` et il est possible que le programme s'arrête si il y a plus de 1000 connexions. Soit `Connexion journalConnexion [100000000000]` et ce programme risque de passer son temps à "swaper" sur le disque, ce qui le rendra inutilisable.

La solution consiste à allouer de la mémoire dynamiquement c'est à dire pendant que le programme s'exécute. Il existe trois types de mémoires différentes pour un processus :

- *La mémoire statique*. C'est la mémoire qui est utilisée par les variables globales ou les variables rémanentes d'un programme. Cette mémoire est attachée au code, elle est contenue dans le code compilé.
- *La mémoire de pile* qui est utilisée pour les variables locales et qui est consommée au fur et à mesure de l'exécution du programme. Cette mémoire n'est valable que tant que la fonction qui l'a définie n'est pas terminée.
- *La mémoire dynamique* est créée au fur et à mesure des besoins du programme, une fois cette mémoire allouée elle continue à être utilisable tant qu'elle n'est pas libérée. Elle continue donc à exister même si la fonction qui a alloué cette mémoire est terminée.

Les primitives de bases pour la gestion de la mémoire dynamique sont :

```
void *malloc(size_t nbOctets).
```

Cette fonction alloue un espace mémoire qui peut contenir au moins `nbOctets` consécutifs de mémoire utilisable. La valeur retournée par cette fonction est l'adresse du premier octet utilisable. La mémoire n'est pas du tout initialisée par cette fonction. Dans la norme, le pointeur garantie de retourner une adresse qui correspond à tout type (incluant des structures) sinon un pointeur `NULL` est retourné. On peut donc initialisé un pointeur vers un entier de la façon suivante :

```
#include <stdlib.h>
int *p;
p = malloc(sizeof(int));
```

Dans ce cas, `malloc` convertir implictement la valeur retournée en un pointeur entier. Néanmoins pour s'assurer d'une meilleure lisibilité de code (et parfois d'une meilleure compatibilité entre compilateurs, Pour initialiser des pointeurs vers des objets, il vaut mieux toujours convertir le type de la sortie de la fonction malloc à l'aide d'un cast :

```
#include <stdlib.h>
int *p;
p = (int*) malloc(sizeof(int));
```

La fonction `malloc` permet également d'allouer un espace pour plusieurs objets contigus en mémoire. On peut écrire par exemple :

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int i = 3;
  int j = 6;
  int *p;
  p = (int*)malloc(2 * sizeof(int));
  *p = i;
  *(p + 1) = j;
  printf("p = %ld \t *p = %d \t p+1 = %ld \t *(p+1) = %d \n",p,*p,p+1,*(p+1));
}
```

On a ainsi réservé, à l'adresse donnée par la valeur de `p`, 8 octets en mémoire, qui permettent de stocker 2 objets de type `int`. Le programme affiche :

```
p = 5368711424   *p = 3   p+1 = 5368711428   *(p+1) = 6 .
```

La fonction `calloc` a un rôle similaire :

```
 void *calloc(size_t nbElements, size_t tailleElements)
```

même chose que `malloc` mais la mémoire est initialisée à zéro. Par exemple : `calloc(100,sizeof(int))` alloue dynamiquement un tableau de 100 entiers.

```
void *realloc(void *origine, size_t nbOctets)
```

cette fonction peut agrandir ou retrécir une zone mémoire origine déjà allouée dynamiquement. La variable `nbOctets` représente alors la taille de la nouvelle zone. Comme pour `malloc` la valeur retournée est l'adresse du premier octet utilisable. De plus, le contenu pointé par `origine` est copié à l'adresse retournée. On garde ainsi les informations qui étaient présentes dans l'ancien espace mémoire. Par contre, l'ancienne mémoire qui est pointée par `origine` a été libérée, elle n'est donc plus valide et ne peut plus être utilisée. Par exemple : `int *tmp = malloc(100*sizeof(int));.... tmp = realloc(tmp, 200*sizeof(int));`

La mémoire allouée vi`malloc` est rémanente (persistente) : elle continuera à être occupée jusqu'à l'arrêt du programme ou la libération explicite de la mémoire. La fonction `void free(void *memoireDynamique)` sert à libérer la mémoire allouée dynamiquement.

2.12.5 Les pointeurs et les tableaux

Un tableau est un pointeur particulier. En effet, la valeur d'une variable de type tableau est égale à l'adresse de la variable elle même. On a donc pour une déclaration `int tab[3];` l'expression `tab == &tab` est toujours vraie. On ne peut par contre changer le contenu de la variable `tab` car il faut que la propriété précédente soit toujours vraie pour un tableau. On ne peut donc pas écrire une instruction comme `tab = &x` ou bien comme `tab++;`. Par contre on peut très bien utiliser un pointeur pour référencer un tableau. Une instruction comme `int *tmp = tab` est tout à fait correcte. Dans ce cas, l'adresse de `tab` qui est aussi la valeur de `tab` est maintenant référencée par le pointeur `tmp`. Le fait qu'un tableau soit un pointeur particulier explique pourquoi les effets de bords sont possibles lorsqu'une fonction prend en paramètre un tableau. C'est la cas par exemple de la fonction `viderTableau`. Enfin, on peut manipuler les tableaux en utilisant la notation `*` et on peut également manipuler un pointeur en utilisant la notation `[]`. Tous ces éléments sont résumés sur le programme suivant.

```
#include <stdio.h>
void videTableau(int tab[], int taille)
{
 for(int i = 0; i < taille; i++)
      tab[i] = 0;
}


void afficheTableau(int *tab, int taille)
{
 for(int i = 0; i < taille; i++)
     printf("Le %d éléments du tableau a pour valeur %d \n", i+1, tab[i]);
}
int main (int argc, char **argv)
{
  int tab[] = {1,2,3,4,5,6,7};
  printf("l'adresse de la variable tab est %x sa valeur est %x \n", &tab, tab);
  printf(" Le contenu du premier élément du tableau tab %d \n",  tab);
  int *tmp = tab;
  for(int l = 0; l < 7; l++)
    tmp[l] = tmp[l] * 2;
  afficheTableau(tab, 7);
  videTableau (tmp, 7);
  afficheTableau(tab, 7);
}
```

2.13 Les Fonctions

2.13.1 Définition d'une fonction

Une fonction est constituée de :
- Un *nom* ;
- Un *type de retour* ;
- Une *suite de paramètres* ;
- La *définition* de la fonction c'est à dire son code.

On parle de *déclaration* de fonction, lorsqu'on ne donne pas le code de la fonction. On emploie le mot *signature*, *prototype*, *déclaration* lorsque l'on ne donne pas le code de la fonction. Exemple :

```
int min (int a, int b);
```

Cette fonction a pour nom `min`, elle prend deux paramètres : le type du premier paramètre est `int` et le type du deuxième paramètre est aussi un `int`. Cette fonction retourne un `int`. La définition de la fonction est alors la suivante :

```
int min (int a, int b)
{
   return a>b?b:a;
}
```

2.13.2 Appel d'une fonction

L'appel d'une fonction se fait par l'expression

```
nom-fonction(para-1,para-2,...,para-n)
```

Pour appeler la fonction `min`, il faut lui communiquer deux paramètres. Voici, deux exemples pour l'appel de la fonction `min` :

```
min (5, 6); int i = 1; int j = 3; min(i,j);
```

Les variables `a` et `b` de la fonction `min` sont des variables locales à cette fonction. Lors de l'appel de la fonction `min`, avant l'appel, les données qui sont communiquées à la fonction sont placées dans la pile d'appel. Les paramètres sont passés par valeur, mais l'on verra plus tard que l'on peut passer des références, c'est à dire des valeurs qui représente des adresses mémoires. Cela sera abordé avec les pointeurs. Le passage par valeur consiste a recopier les bits qui correspondent au codage du paramétre dans la pile.

Le retour d'une information par une fonction, se fait en utilisant le mot clef `return` qui est une instruction d'échapement.

2.13.3 Déclaration d'une fonction

Le `C` n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale `main`. Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction `main`), elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son prototype, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

```
type nom-fonction(type-1,...,type-n);
```

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction `main`. Par exemple, on écrira :

```
int puissance (int, int );

int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}

main()
{
  int a = 2, b = 5;
  printf("%d\n", puissance(a,b));
}
```

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction `main` et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le protype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype. Ainsi les fichiers d'extension `.h` de la librairie standard (fichiers headers) contiennent notamment les prototypes des fonctions de la librairie standard. Par exemple, on trouve dans le fichier `math.h` le prototype de la fonction `pow` (élévation à la puissance) :

```
extern  double pow(double , double );
```

La directive au préprocesseur

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

permet au préprocesseur d'inclure la déclaration de la fonction `pow` dans le fichier source. Ainsi, si cette fonction est appelée avec des paramètres de type `int`, ces paramètres seront convertis en `double` lors de la compilation.

Par contre, en l'absence de directive au préprocesseur, le compilateur ne peut effectuer la conversion de type. Dans ce cas, l'appel à la fonction `pow` avec des paramètres de type int peut produire un résultat faux !

2.13.4 Durée de vie des variables

Les variables manipulées dans un programme `C` ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.

Les variables permanentes (ou statiques)

Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée segment de données. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef `static`.

Les variables temporaires

Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée segment de pile. Dans ce cas, la variable est dite automatique. Le spécificateur de type correspondant, `auto`, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

La durée de vie des variables est liée à leur portée, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

Variables globales

On appelle variable globale une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, `n` est une variable globale :

```
int n;
void fonction();
```

```
void fonction()
{
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

La variable `n` est initialisée à zéro par le compilateur et il s'agit d'une variable permanente. En effet, le programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

Variables locales

On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant :

```
int n = 10;
void fonction();

void fonction()
{
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
```

```
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

affiche

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef `static` :

```
static type nom-de-variable;
```

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, `n` est une variable locale à la fonction secondaire fonction, mais de classe statique.

```
int n = 10;
void fonction();

void fonction()
{
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Ce programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

On voit que la variable locale `n` est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

Utilisation du mot clef `static` pour les variables et fonction globales

L'utilisation du mot clef `static` change la visibilité d'une variable en la réduisant au fichier où elle est déclarée. Par exemple, la définition suivante dans le fichier `static.c` :

```
static int i = 0;
```

réduit la visibilité de la variable `i` au seul fichier `static.c`, elle ne peut être utilisée à l'extérieur de ce fichier. On parle dans ce cas, de "variable globale locale".

On peut maintenant distinguer les variables :
- Globale qui sont déclarées dans l'interface, définit dans un seul fichier ".c" et utilisable depuis n'importe quel autre fichier.
- Locale qui sont déclarées/définies dans un bloc appartenant à une définition de fonction.
- Globale/locale qui sont définies dans un fichier ".c" qui sont précédées du mot clef `static` et qui peuvent être accéder seulement dans le fichier où elles sont définies.

Le même concept est applicable pour les fonctions. On peut faire précéder la déclaration/définition d'une fonction du mot clef `static`. Par exemple, la déclaration de la fonction `usage` qui vérifie que les paramètres de la ligne de commande sont corrects n'a pas de raison d'être appelée en dehors du fichier ".c" dans lequel la fonction `main` est définie. Pour ce style de fonctions, on utilisera le même qualificatif que pour les variables et l'on parlera alors de fonctions globales/locales.

On a vu dans le chapitre précédent que l'interface contenait les déclarations d'informations qui sont utilisables depuis l'extérieur du module. Principalement l'interface d'un module contient un ensemble de déclaration de fonctions. Le rôle de l'implémentation est de fournir du code pour les fonctions déclarées dans l'interface. Il est parfois utile de définir des variables qui servent à l'implémentation de plusieurs des fonctions déclarées par l'interface. Ces variables n'ont pas à être déclarées dans l'interface et comme elles ne sont utiles qu'à l'implémentation elles doivent être déclarées globales/locales. De même, il est aussi utile de factoriser le code entre plusieurs fonctions déclarées dans l'interface. Ces fonctions de factorisation de code n'ont de raison d'être que dans l'implémentation et elles n'ont pas à faire partie de l'interface. Elles aussi doivent être globale/locale. Ces notions seront reprises largement dans le prochain chapitre.

2.13.5 Transmission des paramètres d'une fonction

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée. On dit que les paramètres d'une fonction sont transmis par valeurs. Par exemple, le programme suivant :

```
void echange (int, int );

void echange (int a, int b)
{
  int t;
  printf("debut fonction :\n a = %d \t b = %d\n",a,b);
  t = a;
  a = b;
  b = t;
  printf("fin fonction :\n a = %d \t b = %d\n",a,b);
  return;
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(a,b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

imprime

```
debut programme principal :
 a = 2   b = 5
debut fonction :
 a = 2   b = 5
fin fonction :
 a = 5   b = 2
fin programme principal :
 a = 2   b = 5
```

Les passages de références dans les appels de fonctions Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```
void echange (int *, int *);

void echange (int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
  return;
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(&a,&b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

Rappelons qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction. Par exemple, le programme :

```
#include <stdlib.h>

void init (int *, int );

void init (int *tab, int n)
{
  int i;
  for (i = 0; i < n; i++)
    tab[i] = i;
  return;
}

main()
{
  int i, n = 5;
  int *tab;
  tab = (int*)malloc(n * sizeof(int));
  init(tab,n);
}
```

initialise les éléments du tableau `tab`.

Une première utilisation des pointeurs est de permettre de passer des valeurs de références à des fonctions. On a vu que le passage des paramètres en `C` se fait par valeur. Mais quand

on passe une valeur qui représente une référence, on a un passage par *référence*.

Par exemple, si on considère la fonction suivante :

```
void remiseAZero(int *tmp)
{
  *tmp = 0;
}
```

Cette fonction prend le contenu du contenu de `tmp`. Comme `tmp` représente une adresse, on met le contenu de cette adresse à zéro. Si on considère la variable `i` précédente : `int i =1;`. Supposons que l'adresse de la variable `i` soit `ox123`, le contenu de l'adresse `ox123` contient les bits nécessaire au codage de l'entier 1. L'appel `remiseAZero(&i);` consiste à exécuter la fonction `remiseAZero` avec la valeur `ox123` qui représente l'adresse de la variable `i`. Le contenu de la variable `tmp` de la fonction `remiseAZero` est donc `ox123`, c'est à dire la valeur passée en paramètre à savoir l'adresse de la variable `i`. Comme précédemment, en écrivant, `*tmp = 0`, on met le contenu de l'adresse `ox123` à zéro. Donc, la fonction `remiseAzero` modifie la valeur d'une variable pésente dans l'appelant. On parle dans ce cas d'effet de bord.

Le programme complet est alors le suivant :

```
#include <stdlib.h>
#include <stdio.h>
static
void remiseAZero(int *tmp)
{
  *tmp = 0;
}


int main(int argc, char **argv)
{
  int i = 1;

  printf("La valeur de i avant l'appel %d \n", i);
  remiseAZero(\&i);
  printf("La valeur de i après l'appel %d \n", i);

}
```

Une autre illustration des effets de bords est la fonction *swap* :

```
void swap(int *x, int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Cette fonction commence par sauvegarder le contenu de l'adresse référencée par le pointeur `x`, puis elle change le contenu de l'adresse référencée par le pointeur `x` en lui mettant le contenu de l'adresse référencée par le pointeur `y`. Et enfin, elle change le contenu de l'adresse référencée par le pointeur `y` en l'affectant à `tmp`. Le programme suivant est une illustration de l'utilisation de la fonction `swap`.

```
int main(int argc, char **argv)
{
  int a = 1;
  int b = 2;
  printf ("la valeur de a est %d la valeur de b est %d \n", a, b);
  swap(&a,&b);
  printf ("la valeur de a est %d la valeur de b est %d \n", a, b);
}
```

Lorsqu'on passe un tableau en paramètre à une fonction, c'est la valeur du tableau qui est passée à la fonction. Comme on l'a vu dans la section précédente la valeur d'un tableau est l'adresse du tableau. On passe l'adresse du tableau c'est à dire une référence aux éléments du tableau. Dans ce cas, si on change la valeur d'un élément du tableau cela aura des conséquences sur la variable qui a été passée en paramétre.

```
void videTableau(int tab[], int taille){
  printf(" L'adresse d'un tableau est %x \n", tab);
  for(int i = 0; i < taille; i++)
      tab[i] = 0;
}
```

La fonction `videTableau` prend en paramètre un tableau, c'est à dire une référence à une adresse mémoire. Par exemple, l'appel à cette fonction avec `videTableau(tab1, 4);` fera passer les quatre premiers éléments de `tab1` à 0.

2.14 Les paramètres de la fonction `main`

En principe le prototype de la fonction `main` est le suivant :

```
int main (int argc, char **argv)
```

En principe le type de retour est un `int` pour communiquer au système d'exploitation, le bon ou le mauvais fonctionnement du programme. Les paramètres `argc` et `argv` permettent de représenter les arguments de la ligne de commande au moment de l'appel.

Le paramètre `argc` est un entier qui représente le nombre de paramètres de la ligne de commande y compris le nom de l'exécutable. Le deuxième paramètre est un tableau de chaîne de caractères et chaque chaîne correspond à un paramètre.

```
int  main(int argc, char **argv)
{
  for (int i = 0; i < argc; i++)
  printf(" L'argument %d est %s \n", i, argv[i]);
}
```

Le programme précédent permet de vérifier le fonctionnement des arguments `argc` et `argv`. La ligne de commande `principal toto 1 tutu 3 titi` produit le résultat suivant :

```
 L'argument 0 est principal
 L'argument 1 est toto
 L'argument 2 est 1
 L'argument 3 est tutu
 L'argument 4 est 3
 L'argument 5 est titi
```

Les informations passées à un programme permettent de paramètrer son comportement. La plupart du temps, les informations passées sont des modes de fonctionnement du programme. En principe, il existe une fonction : `int usage (int argc, char **argv)` qui a en charge de vérifier que les paramètres fournis sont corrects et qui doit aussi fixé le comportement du programme en fonction des paramètres passés.

Autre exemple. Ainsi, le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :

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

int main(int argc, char *argv[])
{
  int a, b;

  if (argc != 3)
    {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
  return(EXIT_SUCCESS);
}
```

On lance donc l'exécutable avec deux paramètres entiers, par exemple,

```
a.out 12 8
```

Ici, `argv` sera un tableau de 3 chaînes de caractères `argv[0], argv[1]` et `argv[2]` qui, dans notre exemple, valent respectivement "`a.out`", "`12`" et "`8`". Enfin, la fonction de la librairie standard `atoi()`, déclarée dans `stdlib.h`, prend en argument une chaîne de caractères et retourne l'entier dont elle est l'écriture décimale.

Lorsqu'elle est utilisée sans arguments, la fonction main a donc pour prototype `int main(void);` On s'attachera désormais dans les programmes à respecter ce prototype et à spécifier les valeurs de retour de main.

2.14.1 Le pointeur générique `void *`.

On a vu dans la section précédente que tous les pointeurs occupaient la même place mémoire. A savoir le nombre d'octets nécessaire pour adresser la mémoire. On a vu également que pour effectuer des opérations comme l'addition ou la soustraction avec un entier ou bien la différence entre deux pointeurs, il fallait connaître la taille du type pour effectuer correctement ces opérations. Il existe un type de pointeur particulier qui est le type `void *`. Ce type de pointeur est appelé pointeur générique.

Ce type de pointeur permet d'adresser la mémoire comme tout autre pointeur mais on ne peut pas utiliser ce pointeur pour faire les opérations de déplacement dans la mémoire. Ni même évoquer son contenu puisqu'on ne connait pas la taille du type qui est associé à l'adresse mémoire. Par exemple, `void *tmp;` On ne doit pas écrire `tmp++`, `*tmp`, `tmp +1`, ce sont des instructions dangereuses voir interdites.

L'intérêt du type `void *` est de pouvoir unifier tous les types de pointeurs. En effet, on peut convertir tous les pointeurs vers le type `void *` et réciproquement.

```
int main(int argc, char **argv)
{
  void *tmp = (void *) 0;
  int *tmpInt = (void *) 0;
  tmp = tmpInt;
  tmpInt = tmp;
}
```

On verra l'intérêt de ce type de pointeur pour la factorisation de code.

2.14.2 Pointeur sur une fonction

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction. Cette procédure permet en particulier d'utiliser une même fonction pour différents usages. Pour cela, on utilise un mécanisme de pointeur. Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction. Un pointeur sur une fonction ayant pour prototype

```
type fonction(type_1,...,type_n);
```

est de type

```
type (*)(type_1,...,type_n);
```

Ainsi, une fonction `operateur_binaire` prenant pour paramètres deux entiers et une fonction de type `int`, qui prend elle-même deux entiers en paramètres, sera définie par :

```
int operateur_binaire(int a, int b, int (*f)(int, int))
```

Sa déclaration est donnée par

```
int operateur_binaire(int, int, int(*)(int, int));
```

Pour appeler la fonction `operateur_binaire`, on utilisera comme troisième paramètre effectif l'identificateur de la fonction utilisée, par exemple, si `somme` est une fonction de prototype

```
int somme(int, int);
```

on appelle la fonction `operateur_binaire` pour la fonction `somme` par l'expression

```
operateur_binaire(a,b,somme)
```

Notons qu'on n'utilise pas la notation `&somme` comme paramètre effectif de `operateur_binaire`.

Pour appeler la fonction passée en paramètre dans le corps de la fonction `operateur_binaire`, on écrit `(*f)(a, b)`. Par exemple

```
int operateur_binaire(int a, int b, int (*f)(int, int))
{
  return((*f)(a,b));
}
```

Ainsi, le programme suivant prend comme arguments deux entiers séparés par la chaîne de caractères plus ou fois, et retourne la somme ou le produit des deux entiers.

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

void usage(char *);
int somme(int, int);
int produit(int, int);
int operateur_binaire(int, int, int(*)(int, int));

void usage(char *cmd)
{
  printf("\nUsage: %s int [plus|fois] int\n",cmd);
```

```
  return;
}

int somme(int a, int b)
{
  return(a + b);
}

int produit(int a, int b)
{
  return(a * b);
}

int operateur_binaire(int a, int b, int (*f)(int, int))
{
  return((*f)(a,b));
}

int main(int argc, char *argv[])
{
  int a, b;


  if (argc != 4)
    {
      printf("\nErreur : nombre invalide d'arguments");
      usage(argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[3]);
  if (!strcmp(argv[2], "plus"))
    {
      printf("%d\n",operateur_binaire(a,b,somme));
      return(EXIT_SUCCESS);
    }
  if (!strcmp(argv[2], "fois"))
    {
      printf("%d\n",operateur_binaire(a,b,produit));
      return(EXIT_SUCCESS);
    }
  else
    {
      printf("\nErreur : argument(s) invalide(s)");
      usage(argv[0]);
      return(EXIT_FAILURE);
```

```
    }
}
```

Les pointeurs sur les fonctions sont notamment utilisés dans la fonction de tri des éléments d'un tableau qsort et dans la recherche d'un élément dans un tableau `bsearch`. Ces deux fonctions sont définies dans la libriarie standard (`stdlib.h`).

Le prototype de la fonction de tri (algorithme quicksort) est

```
void    qsort(void *tableau, size_t nb_elements, size_t taille_elements,
int(*comp)(const void *, const void *));
```

Elle permet de trier les `nb_elements` premiers éléments du tableau tableau. Le paramètre `taille_elements` donne la taille des éléments du tableau. Le type `size_t` utilisé ici est un type prédéfini dans `stddef.h`. Il correspond au type du résultat de l'évaluation de `sizeof`. Il s'agit du plus grand type entier non signé. La fonction `qsort` est paramétrée par la fonction de comparaison utilisée de prototype :

```
int comp(void *a, void *b);
```

Les deux paramètres `a` et `b` de la fonction `comp` sont des pointeurs génériques de type `void *`. Ils correspondent à des adresses d'objets dont le type n'est pas déterminé. Cette fonction de comparaison retourne un entier qui vaut 0 si les deux objets pointés par `a` et `b` sont égaux et qui prend une valeur strictement négative (resp. positive) si l'objet pointé par `a` est strictement inférieur (resp. supérieur) à celui pointé par `b`.

Par exemple, la fonction suivante comparant deux chaînes de caractères peut être utilisée comme paramètre de `qsort` :

```
int comp_str(char **, char **);

int comp_str(char **s1, char **s2)
{
  return(strcmp(*s1,*s2));
}
```

Le programme suivant donne un exemple de l'utilisation de la fonction de tri `qsort` pour trier les éléments d'un tableau d'entiers, et d'un tableau de chaînes de caractères.

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

#define NB_ELEMENTS 10

void imprime_tab1(int*, int);
void imprime_tab2(char**, int);
```

```
int comp_int(int *, int *);
int comp_str(char **, char **);

void imprime_tab1(int *tab, int nb)
{
  int i;
  printf("\n");
  for (i = 0; i < nb; i++)
    printf("%d \t",tab[i]);
  printf("\n");
  return;
}

void imprime_tab2(char **tab, int nb)
{
  int i;
  printf("\n");
  for (i = 0; i < nb; i++)
    printf("%s \t",tab[i]);
  printf("\n");
  return;
}

int comp_int(int *a, int *b)
{
  return(*a - *b);
}
int comp_str(char **s1, char **s2)
{
  return(strcmp(*s1,*s2));
}

int main()
{
  int *tab1;
  char *tab2[NB_ELEMENTS] = {"toto", "Auto", "auto", "titi", "a", "b",\
"z", "i , "o","d"};
  int i;

  tab1 = (int*)malloc(NB_ELEMENTS * sizeof(int));
  for (i = 0 ; i < NB_ELEMENTS; i++)
    tab1[i] = random() % 1000;
  imprime_tab1(tab1, NB_ELEMENTS);
  qsort(tab1, NB_ELEMENTS, sizeof(int), comp_int);
  imprime_tab1(tab1, NB_ELEMENTS);
  /***********************/
```

```
  imprime_tab2(tab2, NB_ELEMENTS);
  qsort(tab2, NB_ELEMENTS, sizeof(tab2[0]), comp_str);
  imprime_tab2(tab2, NB_ELEMENTS);
  return(EXIT_SUCCESS);
}
```

2.14.3 Fonctions avec un nombre variable de paramètres

Il est possible en `C` de définir des fonctions qui ont un nombre variable de paramètres. En pratique, il existe souvent des méthodes plus simples pour gérer ce type de problème : toutefois, cette fonctionnalité est indispensable dans certains cas, notamment pour les fonctions `printf` et `scanf`.

Une fonction possédant un nombre variable de paramètre doit posséder au moins un paramètre formel fixe. La notation `...` (obligatoirement à la fin de la liste des paramètres d'une fonction) spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes. Ainsi, une fonction ayant pour prototype

```
int f(int a, char c, ...);
```

prend comme paramètre un entier, un caractère et un nombre quelconque d'autres paramètres. De même le prototype de la fonction `printf` est

```
int printf(char *format, ...);
```

puisque `printf` a pour argument une chaîne de caractères spécifiant le format des données à imprimer, et un nombre quelconque d'autres arguments qui peuvent être de types différents.

Un appel à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle autre fonction.

Pour accéder à la liste des paramètres de l'appel, on utilise les macros définies dans le fichier en-tête `stdarg.h` de la librairie standard. Il faut tout d'abord déclarer dans le corps de la fonction une variable pointant sur la liste des paramètres de l'appel ; cette variable a pour type `va_list`. Par exemple,

```
va_list liste_parametres;
```

Cette variable est tout d'abord initialisée à l'aide de la macro `va_start`, dont la syntaxe est

```
va_start(liste_parametres, dernier_parametre);
```

où `dernier_parametre` désigne l'identificateur du dernier paramètre formel fixe de la fonction. Après traitement des paramètres, on libère la liste à l'aide de la `va_end` :

```
va_end(liste_parametres);
```

On accède aux différents paramètres de liste par la macro `va_arg` qui retourne le paramètre suivant de la liste :

```
va_arg(liste_parametres, type)
```

où `type` est le type supposé du paramètre auquel on accède.

Notons que l'utilisateur doit lui-même gérer le nombre de paramètres de la liste. Pour cela, on utilise généralement un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Cette méthode est utilisée dans le programme suivant, où la fonction `add` effectue la somme de ses paramètres en nombre quelconque.

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

int add(int,...);

int add(int nb,...)
{
  int res = 0;
  int i;
  va_list liste_parametres;

  va_start(liste_parametres, nb);
  for (i = 0; i < nb; i++)
    res += va_arg(liste_parametres, int);
  va_end(liste_parametres);
  return(res);
}

int main(void)
{
  printf("\n %d", add(4,10,2,8,5));
  printf("\n %d\n", add(6,10,15,5,2,8,10));
  return(EXIT_SUCCESS);
}
```

2.15 Un exemple récupitulatif. Le module `VecteurUnique`.

Il s'agit de faire un vecteur extensible unique qui permet d'écrire et de lire dans un tableau sans limitation de taille. Cet exemple permet de revoir la plupart des notions présentées dans ce chapitre. On verra dans le chapitre suivant comment le faire évoluer.

VECTEURUNIQUE.H

```
#ifndef _VECTEUR_
#define _VECTEUR_

extern void reInitaliseVecteur(void);
/* Elle remet */

extern int  lireElement(int indice);
/* La fonction LireElement ne marche que si.....
indice est superieur ou egal a zero
et que indice represente un indice valide
(indice inférieur ou egal a max indice).

*/

extern void ecrireElement(int indice, int element);
/*Cette fonction  ne marche que si indice est positif ou nul */


/* extern void rallongerjusquaIndiceMax (int indiceMax);
Cette fonction ne peut pas faire partie de l'interface car elle est dépendante
de l'implémentation */

extern int indiceMax(void);
/*
extern int donneIndice(int element);
Cette fonction redondante qui ne respecte pas la minimalité de l'interface.
 On peut l'implémenter en utilisant les fonctions lireElement et indiceMax.
Néanmoins, il peut-être interessant de l'écrire pour des raisons d'efficacité
 et pour éviter de la duplication de code.
*/
#endif

#include "Vecteur1.h"
#include <assert.h>
#include <malloc.h>

/* DEFINIR LES DONNEES : TYPES, CONSTANTES, VARIABLES....*/

static int *tab = (void *) 0; //NULL
// globale locale....

static const int tailleDefault = 10;

static int maxIndice = -1;
```

```
static void allongerVecteur(int indice)
{
  tab = realloc(tab, (indice + tailleDefault) * sizeof(int));

  for(int i = maxIndice + 1; i < indice + tailleDefault; i++)
    tab[i] = 0;

  maxIndice = (indice + tailleDefault) - 1;
}

void
reInitaliseVecteur(void)
 {
   if(tab != (void *) 0)
     free(tab);

   tab = malloc(sizeof(int)* tailleDefault);
   maxIndice = tailleDefault - 1;

   for(int i = 0; i <= maxIndice; i++)
     tab[i] = 0;
 }


 int
lireElement(int indice)
 {
   assert(!((indice < 0 || indice > maxIndice ) || tab == (void *) 0));
   return tab[indice];
 }


 void ecrireElement(int indice, int element)
 {
   assert( ! (indice < 0 || tab == (void *) 0));
   if(indice >= maxIndice)
     allongerVecteur(indice);
   *(tab + indice) = element;
 }

 int indiceMax(void)
 {
   return maxIndice;
 }
```