Les bases de la programmation en C

Table des matières

1 Un peu d'histoire

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 sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre The C Programming language [6]. 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.

2 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 :

1. Le traitement par le préprocesseur : le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...).
2. La compilation : 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.
3. L'assemblage : 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.
4. L'édition de liens : un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies 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.

Les différents types de fichiers utilisés lors de la compilation sont distingués par leur suffixe. Les fichiers source sont suffixés par .c, les fichiers prétraités par le préprocesseur par .i, les fichiers assembleur par .s, et les fichiers objet par .o. Les fichiers objets correspondant aux librairies pré-compilées ont pour suffixe .a.

Le compilateur C sous UNIX s'appelle **cc**. On utilisera de préférence le compilateur **gcc** du projet GNU. Ce compilateur est livré gratuitement avec sa documentation et ses sources. Par défaut, gcc active toutes les étapes de la compilation. On le lance par la commande

gcc [options] fichier.c [-llibrairies]

Par défaut, le fichier exécutable s'appelle **a.out**. Le nom de l'exécutable peut être modifié à l'aide de l'option -o.
Les éventuelles librairies sont déclarées par la chaîne -llibrairie. Dans ce cas, le système recherche le fichier liblibrairie.a dans le répertoire contenant les librairies pré-compilées (généralement /usr/lib/). Par exemple, pour lier le programme avec la librairie mathématique, on spécifie -lm. Le fichier objet correspondant est libm.a. Lorsque les librairies pré-compilées ne se trouvent pas dans le répertoire usuel, on spécifie leur chemin d'accès par l'option -L.

Les options les plus importantes du compilateur gcc sont les suivantes :

-c : supprime l'édition de liens ; produit un fichier objet.
-E : n'active que le préprocesseur (le résultat est envoyé sur la sortie standard).
-g : produit des informations symboliques nécessaires au débogueur.
-Inom-de-répertoire : spécifie le répertoire dans lequel doivent être recherchés les fichiers en-têtes à inclure (en plus du répertoire courant).

-Lnom-de-répertoire : spécifie le répertoire dans lequel doivent être recherchées les librairies précompilées (en plus du répertoire usuel).
-o nom-de-fichier : spécifie le nom du fichier produit. Par défaut, le exécutable fichier s'appelle a.out.
-O, -O1, -O2, -O3 : options d'optimisations. Sans ces options, le but du compilateur est de minimiser le coût de la compilation. En rajoutant l'une de ces options, le compilateur tente de réduire la taille du code exécutable et le temps d'exécution. Les options correspondent à différents niveaux d'optimisation : -O1 (similaire à -O) correspond à une faible optimisation, -O3 à l'optimisation maximale.
-S : n'active que le préprocesseur et le compilateur ; produit un fichier assembleur.
-v : imprime la liste des commandes exécutées par les différentes étapes de la compilation.
-W : imprime des messages d'avertissement (warning) supplémentaires.
-Wall : imprime tous les messages d'avertissement.

3 Les composants élémentaires du C

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

- les identificateurs,
- les mots-clefs,
- 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.

3.1 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.

ATTENTION : ***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).

3.2 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 :

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

3.3 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.

4 Structure d'un programme C

Une *expression* est une suite de composants élémentaires syntaxiquement correcte, par exemple
x = 0
ou bien

```
(i >= 0) && (i < 10) && (p[i] != 0)
```

Une *instruction* est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte ``évaluer cette expression". Plusieurs instructions peuvent être rassemblées par des accolades { et } pour former une instruction composée ou bloc qui est syntaxiquement équivalent à une instruction. Par exemple,

```
if (x != 0)
{
  z = y / x;
  t = y % x;
}
```

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 = 2, c;
double x = 2.38e4;
char message[80];
```

ATTENTION : ***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);
}
```

5 Les types prédéfinis

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 caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants :

char int float double short long unsigned

5.1 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.

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. Ainsi, le programme suivant imprime le caractère 'B'.

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

5.2 Les types entiers

Le mot-clef désignant le type entier est ***int***. Un objet de type int est représenté par un mot ``naturel" de la machine utilisée, 32 bits pour un DEC alpha ou un PC Intel.

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 (64 bits sur un DEC alpha, 32 bits sur un PC Intel).

5.3 Les types flottants

Les types ***float***, double et long double servent à représenter des nombres en virgule flottante. Ils correspondent aux différentes précisions possibles.

6 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.

6.1 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.

On peut 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

6.2 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) 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. Par exemple :

Constante	type
12.34	double
12.3e-4	double
12.34F	float
12.34L	long double

6.3 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 (cf. page X). 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

6.4 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"
```

7 Les opérateurs

7.1 L'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

```
main()
{
  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.

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.

Notons enfin qu'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.

7.3 Les opérateurs relationnels

>	supérieur
>=	supérieur ou égal
<	strictement inférieur
<=	inférieur ou égal
==	égal
!=	différent

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 : ***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 !

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

expression1 op1 expression2 op2 ...expressionN

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.

7.5 Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (short, int ou long), signés ou non.

&	et
|	ou inclusif
^	ou exclusif
~	complément à 1
<<	décalage à gauche
>>	décalage à droite

7.6 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

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

Pour tout opérateur op, l'expression

expression1 op = expression2

est équivalente à

expression1 = expression1 op expression2

Toutefois, avec l'affection composée, expression1 n'est évaluée qu'une seule fois.

7.7 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 */
```

7.8 L'opérateur virgule

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

expression1, expression2, ... , expressionN

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.

7.9 L'opérateur conditionnel ternaire

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. 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.

7.10 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.

7.11 L'opérateur adresse

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

&objet

8 Les instructions de branchement conditionnel

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les instructions de branchement et les boucles. Les instructions de branchement permettent de déterminer quelles instructions seront exécutées et dans quel ordre.

8.1 Branchement conditionnel « if---else »

La forme la plus générale est celle-ci :

```
if (expression1 )
  instruction1
else if (expression2 )
  instruction2
    ...
else if (expressionN )
  instructionN
else
  instructionM
```

avec un nombre quelconque de else if (...). Le dernier else est toujours facultatif. La forme la plus simple est

```
if (expression )
  instruction
```

Chaque instruction peut être un bloc d'instructions.

8.2 Branchement multiple « switch »

Sa forme la plus générale est celle-ci :

```
switch (expression )
  {case constante1:
          liste d'instructions 1
          break;
  case constante2:
          liste d'instructions 2
          break;
    ...
  case constanteN:
          liste d'instructions N
          break;
  default:
          liste d'instructions M
          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
M correspondant à default est exécutée. L'instruction default est facultative.

9 Les boucles

Les boucles permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

9.1 Boucle « while »

La syntaxe de **while** est la suivante :

while (expression)
  instruction

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

```
i = 1;
while (i < 10)
  {
    printf("\n i = %d",i);
    i++;
  }
```

9.2 Boucle « 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
  instruction
while (expression);

Ici, instruction sera exécutée tant que expression est non nulle. Cela signifie donc que instruction est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int a;

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

9.3 Boucle « for »

La syntaxe de for est :

for (expr 1 ;expr 2 ;expr 3)
  instruction

Une version équivalente plus intuitive est :

```
expr 1;
 while (expr 2 )
   {instruction
    expr 3;
   }
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

```
for (i = 0; i < 10; i++)
  printf("\n i = %d",i);
```

A la fin de cette boucle, i vaudra 10. Les trois expressions utilisées dans une boucle for peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois. Par exemple, pour calculer la factorielle d'un entier, on peut écrire :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; i++)
  fact *= i;
printf("%d ! = %d \n",n,fact);
```

On peut également insérer l'instruction fact *= i; dans la boucle for ce qui donne :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; fact *= i, i++);
printf("%d ! = %d \n",n,fact);
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

10 Les instructions de branchement non conditionnel

10.1 Branchement non conditionnel « 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 :

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

imprime à l'écran

```
i = 0
i = 1
i = 2
i = 3
valeur de i à la sortie de la boucle = 3
```

10.2 Branchement non conditionnel « continue »

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

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

imprime

```
i = 0
i = 1
i = 2
i = 4
valeur de i à la sortie de la boucle = 5
```

11 Les fonctions d'entrées-sorties classiques

Il s'agit des fonctions de la librairie standard **stdio.h** utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran. Sur certains compilateurs, l'appel à la librairie stdio.h par la directive au préprocesseur

#include <stdio.h>

n'est pas nécessaire pour utiliser printf et scanf.

11.1 La fonction d'écriture « printf »

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 **",**expression1**, ..., **expressionN**);**

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. Les formats d'impression en C sont donnés dans la table 1suivante.

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

Table 1: Formats d'impression pour la fonction printf

En plus du caractère donnant le type des données, on peut éventuellement préciser certains paramètres du format d'impression, qui sont spécifiés entre le % et le caractère de conversion dans l'ordre suivant :

- largeur minimale du champ d'impression : %10d spécifie qu'au moins 10 caractères seront réservés pour imprimer l'entier. Par défaut, la donnée sera cadrée à droite du champ. Le signe - avant le format signifie que la donnée sera cadrée à gauche du champ (%-10d).
- précision : %.12f signifie qu'un flottant sera imprimé avec 12 chiffres après la virgule. De même %10.2f signifie que l'on réserve 12 caractères (incluant le caractère .) pour imprimer le flottant et que 2 d'entre eux sont destinés aux chiffres après la virgule. Lorsque la précision n'est pas spécifiée, elle correspond par défaut à 6 chiffres après la virgule. Pour une chaîne de caractères, la précision correspond au nombre de caractères imprimés : %30.4s signifie que l'on réserve un champ de 30 caractères pour imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés (suivis de 26 blancs).

11.2 La fonction de saisie « scanf »

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

scanf("chaîne de contrôle"**,**argument1**,...,**argumentN**)**

La chaîne de contrôle 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 et sont donnés dans la table 2.

Les données à entrer au clavier doivent être séparées par des blancs ou des <RETURN> sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères de la donnée à lire. Par exemple %3s pour une chaîne de 3 caractères, %10d pour un entier qui s'étend sur 10 chiffres, signe inclus.

Exemple :

```
#include <stdio.h>
main()
{
  int i;
  printf("entrez un entier sous forme hexadecimale i = ");
```

```
  scanf("%x",&i);
  printf("i = %d\n",i);
}
```

Si on entre au clavier la valeur 1a, le programme affiche i = 26.

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

Table 2: Formats de saisie pour la fonction scanf

11.3 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 caractere, 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 caractere sur la sortie standard :

putchar(caractere);

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

12 Les types composés

A partir des types prédéfinis du C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés types composés, qui permettent de représenter des ensembles de données organisées.

12.1 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

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. Le programme suivant imprime les éléments du tableau tab :

```
#define N 10
main()
{
  int tab[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i,tab[i]);
}
```

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant. Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation. Par exemple, on ne peut pas écrire ``tab1 = tab2;''. Il faut effectuer l'affectation pour chacun des éléments du tableau :

```
#define N 10
main()
{
  int tab1[N], tab2[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    tab1[i] = tab2[i];
}
```

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

type nom-du-tableau[N] = {constante1,constante2,...,constanteN};

Par exemple, on peut écrire

```
#define N 4
int tab[N] = {1, 2, 3, 4};
main()
{
```

```
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i,tab[i]);
}
```

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à zéro si le tableau est une variable globale (extérieure à toute fonction) ou une variable locale de classe de mémorisation static.

De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale. Notons que le compilateur complète toute chaîne de caractères avec un caractère nul '\0'. Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

```
#define N 8
char tab[N] = "exemple";
main()
{
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %c\n",i,tab[i]);
}
```

Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de
la liste d'initialisation. Ainsi le programme suivant imprime le nombre de caractères du tableau tab, ici 8.

```
char tab[] = "exemple";
main()
{
  int i;
  printf("Nombre de caracteres du tableau = %d\n",sizeof(tab)/sizeof(char));
}
```

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

type nom-du-tableau[nombre-lignes][nombre-colonnes]

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]". Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de
constantes :

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

main()
{
  int i, j;
  for (i = 0 ; i < M; i++)
    {
      for (j = 0; j < N; j++)
        printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
    }
}
```

12.2 Les 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 contiguës 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;

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

```
struct modele
{type1 membre1;
  type2 membre2;
    ...
  typeN membreN;
}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. Par exemple, le programme suivant définit la structure complexe, composée de deux champs de type double ; il calcule la norme d'un nombre complexe.

```
#include <math.h>
struct complexe
{
  double reelle;
  double imaginaire;
};

main()
{
  struct complexe z;
  double norme;
    ...
  norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
  printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
}
```

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :
struct complexe z = {2. , 2.};
En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

```
...
main()
{
```

```
  struct complexe z1, z2;
   ...
  z2 = z1;
}
```

12.3 Les champs de bits

Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (int ou unsigned int). Cela se fait en précisant le nombre de bits du champ avant le ; qui suit sa déclaration. Par exemple, la structure suivante

```
struct registre
{
  unsigned int actif : 1;
  unsigned int valeur : 31;
};
```

possède deux membres, actif qui est codé sur un seul bit, et valeur qui est codé sur 31 bits. Tout objet de type struct registre est donc codé sur 32 bits. Toutefois, l'ordre dans lequel les champs sont placés à l'intérieur de ce mot de 32 bits dépend de l'implémentation.

Le champ actif de la structure ne peut prendre que les valeurs 0 et 1. Aussi, si r est un objet de type struct registre, l'opération r.actif += 2; ne modifie pas la valeur du champ.

La taille d'un champ de bits doit être inférieure au nombre de bits d'un entier. Notons enfin qu'un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur &.

12.4 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);
}
```

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant
permet de manipuler en même temps les deux champs de type unsigned int d'une structure en les identifiant à un objet de type unsigned long (en
supposant que la taille d'un entier long est deux fois celle d'un int).

```
struct coordonnees
{
  unsigned int x;
  unsigned int y;
};
union point
{
  struct coordonnees coord;
  unsigned long mot;
};

main()
{
  union point p1, p2, p3;
  p1.coord.x = 0xf;
  p1.coord.y = 0x1;
  p2.coord.x = 0x8;
  p2.coord.y = 0x8;
  p3.mot = p1.mot ^ p2.mot;
  printf("p3.coord.x = %x \t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
}
```

12.5 Les énumérations

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-clef **enum** et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet :

enum modele {constante-1, constante-2,...,constante-n};

En réalité, les objets de type enum sont représentés comme des int. Les valeurs possibles constante-1, constante-2,...,constante-n 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.

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

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

```
enum booleen {faux = 12, vrai = 23};
```

12.6 Définition de types composés avec typedef

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de **typedef** :

typedef type synonyme;

Par exemple,

```
struct complexe
{
  double reelle;
  double imaginaire;
};
```

```
typedef struct complexe complexe;

main()
{
  complexe z;
   ...
}
```

13 Les pointeurs

13.1 Introduction

Avant de parler de pointeurs, nous allons brièvement passer en revue les deux modes d'adressage principaux :

Adressage direct : ***Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder directement à cette valeur.***

Adressage direct = Accès au contenu d'une variable par le nom de la variable.

Exemple :

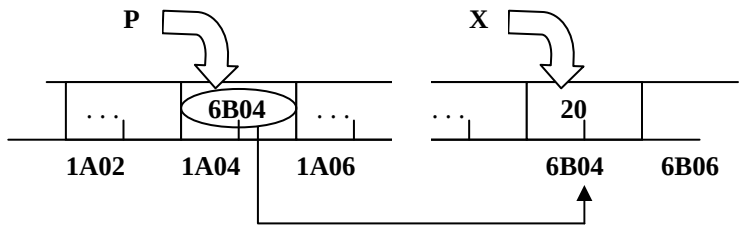
short X ;
X = 20 ;

Adressage indirect : Si nous ne voulons pas ou encore nous ne pouvons pas utiliser le nom d'une variable ***X***, nous pouvons copier l'adresse de cette variable dans une variable spéciale ***P***, appelée ***pointeur***. Ensuite, nous pouvons retrouver l'information de la variable ***X*** en passant par le pointeur ***P***.

Adressage indirect = Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple

Soit ***X*** une variable contenant la valeur 20 et ***P*** un pointeur qui contient l'adresse de ***X***. En mémoire, ***X*** et ***P*** peuvent se présenter comme suit:



La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de pointeurs, c.-à-d. à l'aide de variables auxquelles on peut attribuer les adresses d'autres variables.
On peut donc définir un pointeur comme une ***variable spéciale qui peut contenir l'adresse d'une autre variable***.
En C, chaque pointeur est limité à un type de données.

Si un pointeur ***P*** contient l'adresse d'une variable ***X***, on dit que

*' **P** pointe sur **X** '.*

Attention : *Les **pointeurs** et les **noms de variables** ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :*

- *Un pointeur est une variable qui peut 'pointer' sur différentes adresses.*
- *Le nom d'une variable reste toujours lié à la même adresse.*

13.2 Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin

- d'un opérateur 'adresse de': **&** pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de': ***** pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

13.2.1 L'opérateur 'adresse de' : &

&<NomVariable> fournit l'adresse de la variable <NomVariable>

L'opérateur **&** nous est déjà familier par la fonction *scanf*, qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

Exemple

```
int X;
printf("Entrez un nombre entier : ");
scanf("%d", &X);
```

Soit **P** un pointeur non initialisé et **X** une variable (du même type) contenant la valeur **20** :
Alors l'instruction

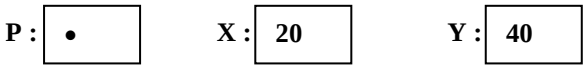
P = &X; affecte l'adresse de la variable **X** à la variable **P**.

13.2.2 L'opérateur 'contenu de' : *

***<NomPointeur>** désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>

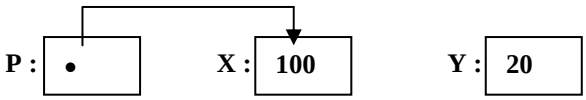
Exemple

```
int X , Y;
int *P ;
```



Soit **X** une variable contenant la valeur **20**, **Y** une variable contenant la valeur **40** et **P** un pointeur non initialisé:
Après les instructions,

```
P = &X;
Y = *P;
*P = 100;
```



- P pointe sur X,
- le contenu de X (référencé par *P) est affecté à Y, et
- le contenu de X (référencé par *P) est mis à 100.

La déclaration d'un pointeur est comme suit :

<Type> *<NomPointeur>
déclare un pointeur <NomPointeur> qui peut recevoir des adresses de variables du type <Type>

Une déclaration comme

int *P_num;

peut être interprétée comme suit:

"*P-num est du type int" ou **"P_num est un pointeur sur int"** ou
"P_num peut contenir l'adresse d'une variable du type int"

13.3 Les opérations élémentaires sur pointeurs

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

Priorité de ***** et **&**

- Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incrémentation ++, la décrémentation --). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

- Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple

Après l'instruction **P = &X;**

les expressions suivantes, sont équivalentes:

Y = *P+1	⟺	Y = X+1
*P = *P+10		X = X+10
*P += 2		X += 2
++*P		++X
(*P)++		X++

13.4 Adressage des composantes d'un tableau

Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de son premier élément. En d'autre termes:

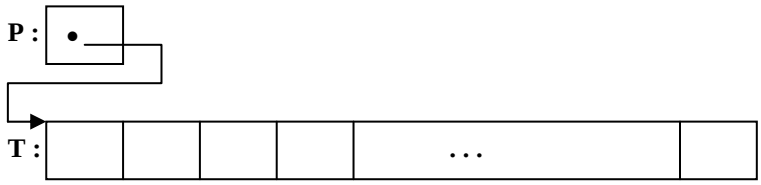
&tableau[0] et tableau sont une seule et même adresse.

En simplifiant, nous pouvons retenir que le nom d'un tableau est un pointeur constant sur le premier élément du tableau, par exemple :

En déclarant un tableau **T** de type *int* et un pointeur **P** sur *int*,

int T[20];

*int *P;*



l'instruction:
P = T; est équivalente à P = &T[0];

Si **P** pointe sur une composante quelconque d'un tableau, alors **P+1** pointe sur la composante **suivante**. Plus généralement,

P+i	pointe sur la **i-ième** composante **derrière P** et
P-i	pointe sur la **i-ième** composante **devant P**.

Ainsi, après l'instruction,

P = T;	le pointeur P pointe sur T[0], et
*(P+1)	désigne le contenu de T[1]
*(P+2)	désigne le contenu de T[2]
...	
*(P+i)	désigne le contenu de T[i]

13.5 Pointeurs et chaînes de caractères

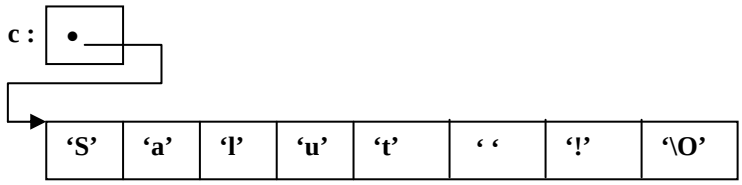
De la même façon qu'un pointeur sur *int* peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur char peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur char peut en plus contenir l'adresse d'une chaîne de caractères constante et il peut même être initialisé avec une telle adresse.

13.5.1 Affectation

On peut attribuer l'adresse d'une chaîne de caractères constante à un pointeur sur char :

Exemple

*char *c;*
c = "Salut !";



Nous pouvons lire cette chaîne constante (par exemple : pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

13.5.2 Initialisation

Un pointeur sur char peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:

```
char *C = "Salut !";
```

Attention ! *: Il existe une différence importante entre les deux déclarations:*

```
char T[] = "Salut !";   /* un tableau  */
char *C  = "Salut !";   /* un pointeur */
```

T ***est un tableau*** qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '**\0**'. Les caractères de la chaîne peuvent être changés, mais le nom **T** va toujours pointer sur la même adresse en mémoire.

C ***est un pointeur*** qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.

14 Quelques conseils pour l'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.
- 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 (ou un autre éditeur de texte équivalent).

Références

[1] Braquelaire (J.-P.). -- Méthodologie de la programmation en C. -- Dunod, 2000, troisième édition.

[3] Cassagne (B.). -- Introduction au langage C. -- http://clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html.

[4] Delannoy (C.). -- Programmer en langage C. -- Eyrolles, 1992.

[5] Faber (F.). -- Introduction à la programmation en ANSI-C. -- http://www.ltam.lu/Tutoriel_Ansi_C/.

[6] Kernighan (B.W.) et Richie (D.M.). -- The C programming language. -- Prentice Hall, 1988, seconde édition.

[7] Léon (L.) et Millet (F.). -- C si facile. -- Eyrolles, 1987.

Exercices facultatifs

printf()	écriture formatée de données	*printf("<format>",<Expr1>,<Expr2>, ...)*
scanf()	lecture formatée de données	*scanf("<format>",<AdrVar1>,<AdrVar2>, ...)*
putchar()	écriture d'un caractère	*putchar('caractère');*
getchar()	lecture d'un caractère	*C = getchar();*

Exercice I.1

Ci-dessous, vous trouvez un simple programme en C. Essayez de distinguer et de classifier autant que possible les éléments qui composent ce programme (commentaires, variables, déclarations, instructions, etc.)

```
#include <stdio.h>
/* Ce programme calcule la somme de 4 nombres entiers
   introduits au clavier.
*/
main()
{
   int NOMBRE, SOMME, COMPTEUR;

   /* Initialisation des variables */
   SOMME = 0;
   COMPTEUR = 0;
   /* Lecture des données */
   while (COMPTEUR < 4)
      {
       /* Lire la valeur du nombre suivant */
       printf("Entrez un nombre entier :");
       scanf("%i", &NOMBRE);
       /* Ajouter le nombre au résultat */
       SOMME += NOMBRE;
       /* Incrémenter le compteur */
       COMPTEUR++;
      }
   /* Impression du résultat */
   printf("La somme est: %i \n", SOMME);
   return 0;
}
```

Exercice I.2

Ecrivez un programme qui lit la date du clavier et écrit les données ainsi que le nombre de données correctement reçues sur l'écran.

Exemple:

```
Introduisez la date (jour mois année): 11 11 1991

données reçues : 3
jour  : 11
mois  : 11
année : 1991
```

* Testez les réactions du programme à vos entrées. Essayez d'introduire des nombres de différents formats et différentes grandeurs.

* Changez la partie format du programme de façon à séparer les différentes données par le symbole '-' .

Exercice I.3 *Lecture d'un caractère*

Ecrire un programme qui lit un caractère au clavier et affiche le caractère ainsi que son code numérique:

a) en employant getchar et printf,
b) en employant getch et printf.

Exercice I.4

Ecrire un programme qui permute et affiche les valeurs de trois variables A, B, C de type entier qui sont entrées au clavier :

A ==> B , B ==> C , C ==> A

Exercice I.5

Ecrire un programme qui affiche le quotient et le reste de la division entière de deux nombres entiers entrés au clavier ainsi que le quotient rationnel de ces nombres.

II Les Tableaux à une dimension - Vecteurs

Exercice II.1 *Produit scalaire de deux vecteurs*

Ecrire un programme qui calcule le produit scalaire de deux vecteurs d'entiers U et V (de même dimension).

Exemple:

```
/         \      /         \
| 3  2  -4 | * | 2  -3  5 | = 3*2+2*(-3)+(-4)*5 = -20
\         /      \         /
```

Exercice III.2 *Maximum et minimum des valeurs d'un tableau*

Ecrire un programme qui détermine la plus grande et la plus petite valeur dans un tableau d'entiers A. Afficher ensuite la valeur et la position du maximum et du minimum. Si le tableau contient plusieurs maxima ou minima, le programme retiendra la position du premier maximum ou minimum rencontré.

III Les Tableaux à deux dimensions - Matrices

Exercice III.1 *Mise à zéro de la diagonale principale d'une matrice*

Ecrire un programme qui met à zéro les éléments de la diagonale principale d'une matrice carrée A donnée.

Exercice III.2 *Matrice unitaire*

Ecrire un programme qui construit et affiche une matrice carrée unitaire U de dimension N. Une matrice unitaire est une matrice, telle que:

```
      1 si i=j
uij =
      0 si i !=j
```

Exercice III.3 *Transposition d'une matrice*

Ecrire un programme qui effectue la transposition **tA** d'une matrice **A** de dimensions N et M en une matrice de dimensions M et N.

a) La matrice transposée sera mémorisée dans une deuxième matrice B qui sera ensuite affichée.

b) La matrice A sera transposée par permutation des éléments.

Rappel:

```
tA =  t | a b c d |  =  | a e i |
       | e f g h |     | b f j |
       | i j k l |     | c g k |
                       | d h l |
```

Exercice III.4 *Multiplication d'une matrice par un réel*

Ecrire un programme qui réalise la multiplication d'une matrice A par un réel X.

Rappel:

```
      | a b c d |      | X*a X*b X*c X*d |
X * | e f g h |   =  | X*e X*f X*g X*h |
      | i j k l |      | X*i X*j X*k X*l |
```

a) Le résultat de la multiplication sera mémorisé dans une deuxième matrice A qui sera ensuite affichée.

b) Les éléments de la matrice A seront multipliés par X.

IV Les Fonctions

Exercice IV.1

Ecrire un programme se servant d'une fonction MOYENNE du type float pour afficher la moyenne arithmétique de deux nombres réels entrés au clavier.

Exercice IV.2

Ecrire une fonction MIN et une fonction MAX qui déterminent le minimum et le maximum de deux nombres réels.

Ecrire un programme se servant des fonctions MIN et MAX pour déterminer le minimum et le maximum de quatre nombres réels entrés au clavier.

Exercice IV.3

Ecrire un programme se servant d'une fonction F pour afficher la table de valeurs de la fonction définie par

$$f(x) = \sin(x) + \ln(x) - \text{sqrt}(x)$$

où x est un entier compris entre 1 et 10.

Exercice IV.4

En mathématiques, on définit la fonction factorielle de la manière suivante:

0! = 1
n! = n*(n-1)*(n-2)* ... * 1 (pour n>0)

Ecrire une fonction FACT du type double qui reçoit la valeur N (type int) comme paramètre et qui fournit la factorielle de N comme résultat.
Ecrire un petit programme qui teste la fonction FACT.

V Les pointeurs

Exercice V.1

Soit P un pointeur qui 'pointe' sur un tableau A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions:

a) *P+2
b) *(P+2)
c) &P+1
d) &A[4]-3
e) A+3
f) &A[7]-P
g) P+(*P-10)
h) *(P+*(P+8)-A[7])

Exercice V.2

Ecrire un programme qui lit un entier X et un tableau A du type int au clavier et élimine toutes les occurrences de X dans A en tassant les éléments restants. Le programme utilisera les pointeurs P1 et P2 pour parcourir le tableau.

Exercice V.3

Ecrire un programme qui range les éléments d'un tableau A du type int dans l'ordre inverse. Le programme utilisera des pointeurs P1 et P2 et une variable numérique AIDE pour la permutation des éléments.

Exercice 4

Ecrire un programme qui lit deux tableaux d'entiers A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A.
Utiliser deux pointeurs PA et PB pour le transfert et afficher le tableau résultant A.

Exercice 5

Ecrire un programme qui lit une chaîne de caractères CH et détermine la longueur de la chaîne à l'aide d'un pointeur P. Le programme n'utilisera pas de variables numériques.