Tables des matières LANGAGE C

Tables des matières

Liste des programmes

Introduction

Cette synthèse se veut un guide d'apprentissage efficace du langage C ANSI.

Tous les chapitres ont été peu ou prou réécrits pour être les plus compréhensibles possibles et vous permettront de comprendre la mise en œuvre et l'utilisation du C.

D'autres chapitres ont été ajoutés par rapport à la parution hebdomadaire initiale.

Enfin de nouveaux exemples illustrent cette présentation.

Ce cours est destiné aux débutants, bien qu'il puisse servir de référence aux initiées.

Si vous ne disposez pas de compilateur C, vous trouverez, sur le Forum des Programmeurs Francophones de CompuServe, une distribution du compilateur GNU adaptée au processeur Intel par DJGPP; ce compilateur ligne de commande fournit du code binaire 32 bits de bonne qualité.

L'auteur,

Tondeur Hervé

Réalisation de notre premier programme C

En C, plus qu'avec d'autres langages, le premier programme traditionnel est :

Code 1 : "Hello world"

```
#include <stdio.h>

void main(){
     printf("hello world!\n");
}
```

Pour exécuter ce premier programme C, enregistrez le source ci-dessus dans un fichier et sauvegardez le sous le nom "hello.c" (ou ce que vous souhaitez, mais si vous utilisez le compilateur GNU l'extension doit être ".c"); ensuite dans une fenêtre Dos - si vous êtes sous Windows - ou avec votre shell préféré (zsh, csh, bash, ...), placez vous dans le répertoire qui contient ce fichier et lancez la compilation et l'édition de liens grâce à la commande, si vous êtes sous Dos/Windows :

```
G:\>gcc hello.c -o hello.exe
```

ou si vous êtes sous unix ou Linux.

```
> gcc hello.c -o hello
```

Par cette commande, nous demandons à l'outil "gcc" de compiler notre source "hello.c" et de créer un exécutable nommé "hello.exe" (ou "hello"); l'option "-o" (pour "output") permettant de définir ce nom. Nous obtenons un programme que nous pouvons lancer, sur l'écran (la console) s'affiche :

```
hello world!
```

Détaillons ce petit programme :

- la première ligne inclut le fichier nommé "stdio.h".

  Ce fichier contient les définitions pour les opérations standards d'entrées (input) à partir du clavier ou d'un fichier, et de sorties (output) vers l'écran ou dans un fichier.

  "stdio.h" définit notamment une fonction nommée "printf()". Cette fonction peut être appelée avec une chaîne de caractères (entre guillemets - ou double quote) comme unique paramètre; cette chaîne est alors affichée à l'écran.

- nous trouvons ensuite une unique fonction appelée "main".

  Cette fonction est le point d'entrée du programme; c'est à dire là où débutera son exécution; elle est appelée "fonction principale".

  Le nom de cette fonction - "main" - est imposé, si vous en utilisez un autre, l'éditeur de liens (linkeur) vous dira qu'il n'a pas trouvé de fonction "main" et la construction de l'exécutable échouera.

  "main" est une fonction, nous devons donc placer une paire de parenthèses ouvrante et fermante - "(" et ")" - après son nom. Dans cet exemple simple, la fonction ne reçoit aucun paramètre, rien ne figure donc entre ces parenthèses.

  Une fonction doit définir le type des informations qu'elle retourne, comme ici aucune valeur n'est retournée par la fonction "main", nous utilisons le mot clé "**void**" (littéralement "rien") placé devant le nom de la fonction.

Enfin le corps d'une fonction est toujours défini entre accolades "{" et "}". Ici, il est constitué d'un simple appel à la fonction "printf()" à qui nous communiquons le texte à afficher. Notez dans notre message que le caractère "\n" sert à insérer un retour à la ligne à la fin du texte.

Notez que la syntaxe est importante. Le C et le C++ distinguent les minuscules des majuscules. Dans ce source, seul le texte entre parenthèses aurait pu avoir une syntaxe différente, tout le reste doit être tapé en minuscule.

En effet, "include" et "void" font parti des mots clé du langage et leur syntaxe est imposée. La routine "printf" est déclarée avec cette orthographe dans le fichier "stdio.h" et est implémentée avec ce nom dans la librairie d'entrée/sortie. Enfin la librairie de démarrage (le runtime) s'attends à trouver une fonction nommée "main".

Remarque: le système Dos/Windows ne distinguant pas la casse des noms de fichiers "stdio.h" aurait pu être saisi autrement. Cependant une des forces du C/C++ étant sa portabilité, il est très recommandé de respecter la syntaxe des noms de fichiers afin de pouvoir recompiler vos sources sur d'autres plates-formes (sous unix, par exemple).

Les types du C / C++

Les langages C et C++ sont des langages typés.

Cela signifie que pour utiliser une variable, vous devez, avant de pouvoir l'utiliser, déclarer cette variable en précisant son type.

Il existe trois sortes de types fondamentaux : le type caractère, le type entier et le type réel ou flottant.

Nous allons étudier chacun de ces types. Notons avant cela l'existence d'un autre type particulier et que nous venons de rencontrer : le type **void**, ce type ne peut être utilisé que comme type d'une fonction (qui ne renvoie rien); en effet il est impossible de déclarer une variable qui ne serait stocké sur aucun octet et qui ne permettrait de stocker aucun nombre.

Le type caractère

Le type caractère permet de définir des variables stockant une lettre. Le nom de ce type est **char**; on définit donc ainsi une variable de type caractère :

```
char maLettre;
```

Nous pouvons ensuite fixer la valeur d'une telle variable avec une lettre donnée entre apostrophes (ou simple quote), voici un exemple :

```
maLettre = 'a';
```

Notez dans cet exemple que le signe "=" sert à définir la valeur d'une variable (à lui affecter une valeur). Nous dirons que "=" est l'opérateur d'affectation.

Nous pouvons également déclarer une variable et fixer sa valeur en même temps:

```
char maLettre = 'A';
```

Nous pouvons également définir la valeur d'une variable de type **char** avec une valeur numérique. En effet, dans le dernier exemple la lettre 'A' est un caractère appartenant au jeu de caractères ASCII a pour valeur le nombre 65. Nous pouvons donc coder la ligne ci-dessus :

```
char maLettre = 65;
```

Plus généralement, le type **char** supporte in extenso l'arithmétique des nombres entiers; ceci est très commode et évite les surcharges (modifications de types) ou appels à des fonctions manipulant les caractères comme c'est le cas en Pascal; par exemple le code suivant affecte à un caractère le caractère suivant dans le jeu de caractères utilisé :

```
char uneLettre = 'A';
uneLettre = uneLettre + 1;/* lettre contient 'B'    */
```

Notez que les symboles "/*" et "*/" permettent d'insérer des commentaires.

Un signe de caractère

Le type **char** stockant des caractères est codé sur un octet, il accepte donc 256 valeurs différentes, mais quelles valeurs ?

Il existe en fait deux types **char**, l'un est signé et peut contenir des valeurs comprises entre -128 et +127; le second est non signé et ne contient donc que des valeurs positives soit les nombres compris entre 0 et 255.

Ces deux types sont définis par les noms **signed char** et **unsigned char**.

Le type **char** (sans plus de précision) comme nous l'avons utilisé précédemment est généralement équivalent au type **signed char**, mais cela peut changer selon la machine hôte. En effet, ce type a en quelque chose été défini pour représenter tous les caractères disponibles sur un système donné.

Cette définition date un peu; à cette époque la plupart des systèmes ne définissaient que les caractères dont la valeur est comprise entre 0 et 127, c'est notamment le cas du jeu de caractères ASCII "primordial" et du jeu de caractères EBCDIC (des IBM 370, par exemple), le problème de la représentation du type **char** (signé ou non signé) ne se posait alors pas.

Aujourd'hui tous les compilateurs permettent de définir explicitement un type caractère signé (grâce à **signed char**) et non signé (grâce à **unsigned char**), le type **char** restant le format natif du système hôte. Cette distinction permet, grâce aux contrôles des types effectués par le compilateur de traiter ces types distinctement et ceci peut s'avérer très utile.

Illustrons ce point par un exemple: les chaînes de caractères en C sont des tableaux de **char**. Ces chaînes sont terminées par un caractère de valeur nulle (ou encore de valeur ASCII zéro, on parlera de chaîne ASCIIZ); imaginons un programme qui aurait besoin de stocker et manipuler des chaînes qui, comme en Pascal, contiennent la longueur de la chaîne comme premier caractère, nous pouvons représenter de telles chaînes comme un tableau de **unsigned char**; grâce à cela le compilateur fera bien la différence entre ces deux sortes de chaînes et il nous avertira d'une éventuelle erreur si nous utilisons une chaîne ASCIIZ là où une chaîne Pascal est attendue et vice versa.

La distinction entre une variable signée et non signée n'influence pas le stockage de la valeur de cette variable, ainsi si nous codons :

```
signed char     lettreSigne = 'a';
unsigned char   lettreNonSigne = 'a';
```

alors ces deux variables contiennent, du point de vue binaire, exactement les mêmes données; par contre l'arithmétique appliquée à ces variables sera différente; on distinguera ainsi une arithmétique signée et une autre non signée; expliquons cela :

Soit la définition:

```
signed char     lettreSigne = 127;
unsigned char   lettreNonSigne = 127;
```

si maintenant nous réalisons :

```
lettreSigne     = lettreSigne + 1;
lettreNonSigne  = lettreNonSigne + 1;
```

alors la variable "lettreSigne" est maintenant interprétée par une opération arithmétique comme -128, tandis que la variable "lettreNonSigne" serait interprétée comme 128.

Voici un autre exemple, nous définissons :

```
signed char     lettreSigne = 0;
unsigned char   lettreNonSigne = 0;
```

si maintenant nous réalisons :

```
lettreSigne     = lettreSigne - 1;
lettreNonSigne  = lettreNonSigne - 1;
```

alors la variable "lettreSigne" est interprétée comme -1, tandis que "lettreNonSigne" est interprétée comme +255.

Notez enfin que le type **char** est bien un type distinct de **signed char** et de **unsigned char**; cette règle ne s'applique pas aux types entiers que nous allons détailler et pour lesquels si la spécification **signed** ou **unsigned** est omise, le type est considéré comme **signed**.

Les types entiers

Les types entiers sont tous basés sur le type fondamental **int**; cependant le codage de ce type dépend de la machine et du compilateur utilisé.

Aussi, et bien que l'emploi du mot clé **int** soit suffisante pour déclarer une variable entière, nous utiliserons les modificateurs de types **short** et **long** pour définir le format des nombres entiers d'une manière qui ne dépende plus de l'environnement

Les entiers courts

Le modificateur **short**, utilisé en association avec **int**, permet de définir un entier codé sur 16 bits ou deux octets. Nous pouvons ici aussi utiliser en plus les modificateurs **signed** et **unsigned**, ceci nous conduit aux types suivants :

signed short int nombres compris entre -32 768 et +32767

unsigned short int nombres compris entre 0 et 65 535

pour simplifier l'écriture le symbole **int** peut être omis; ainsi les deux définitions de types précédentes sont équivalentes à **signed short** et **unsigned short**; de plus, comme nous venons de le voir avec le type **char**, la représentation **signed** est implicite pour des entiers si rien n'est précisé; nous utiliserons donc: **short** (nombre de -32 768 à +32767) et **unsigned short** (nombre de 0 à 65 535).

Voici des définitions valides de variables entières :

```
short               unEntier;
signed short        unNombre;       /* équivalent à short unNombre  */
signed short int    unChiffre;      /* équivalent à short unChiffre */
unsigned short      unEntierPositif;
```

Les entiers longs

Le modificateur **long**, utilisé en association avec **int**, permet de définir un entier codé sur 32 bits ou quatre octets, cet entier peut également être signé ou non, soit les types :

signed long int nombres compris entre -2 147 483 648 et 2 147 483 647

unsigned long int nombres compris entre 0 et 4 294 967 295

Comme précédemment, les mots clé **int** et **signed** sont optionnels, nous utiliserons les définitions de type **long** et **unsigned long**.

Voici des définitions valides de variables entières longues :

```
long            unGrandEntier;
signed long     unGrandNombre;       /* équivalent à long unGrandNombre   */
signed long int unGrandChiffre;      /* équivalent à long unGrandChiffre  */
unsigned long   unGrandEntierPositif;
```

Le modificateur **long** peut être appliqué deux fois au type **int**, ainsi nous définissons un type entier codé sur 64 bits ou 8 octets, soit les types :

signed long long int

unsigned long long int

que nous utiliserons comme **long long** et **unsigned long long**; le type **long long** accepte des valeurs comprises entre -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807; soit presque de quoi compter le nombre de particules présentes dans tout l'univers.

Les types flottants

Ces types servent à représenter des nombres en virgule flottante, c'est à dire des réels.

La précision de telles variables (c'est à dire le nombre de chiffres significatifs) et leurs valeurs absolues minimale et maximale dépendent du processeur sur lequel tourne le code ainsi que, dans certains cas, du compilateur.

Les types définis sont :

float	flottant simple précision
double	flottant double précision
long double	flottant quadruple précision

Le type **float** correspond généralement à la taille des registres du processeur, le type **double** est deux fois plus large (nécessite deux fois plus d'octets) et le type **long double** quatre fois plus.

Ainsi sur les processeurs Intel récents, le type **float** est codé sur 4 octets et permet de stocker des nombre dont la valeur absolue est comprise entre 1.403e-45 et -3.403e+38. Le type **double** est lui codé sur 8 octets et représente une valeur absolue comprise entre 4.941e-324 et 1.798e+308.

Par contre, sur un Cray utilisant des processeurs 64 bits le type **float** est stocké sur 8 octets (et correspond donc au type **double** utilisable avec un processeur Intel) tandis que le type **double** est codé sur 16 octets.

Enfin le type **long double** est totalement dépendant du compilateur (car certain ne le définisse pas) et du processeur utilisé (il peut ainsi être codé sur 10, 16 ou 32 octets).

Notez que les modificateurs **signed** et **unsigned** ne sont pas applicables aux types flottants; ceux-ci définissent toujours des variables ayant un signe.

Conversion de types

Les langages C et C++ supportent la modification de types (typecasting) des valeurs.

Cette modification est implicite (automatique) entre les types fondamentaux que nous venons d'étudier; ainsi le code suivant est valide :

```
float      monReel   = 97.85;
short      monEntier = monReel;      /* monEntier vaut 97 */
char       maLettre  = monEntier;    /* maLettre est 'a'  */
```

cependant le compilateur risque d'émettre des avertissements durant la compilation de ce code afin de nous avertir des risques de pertes de données.

Pour éviter ces messages et lorsque nous savons que le codage réalisé est correct, nous pouvons réaliser une conversion explicite (demandée) des données, ainsi nous écririons:

```
float      monReel   = 97.85;
short      monEntier = (short) monReel;
char       maLettre  = (char) monEntier;
```

avec une telle écriture nous demandons une conversion du type de données de la variable vers le type précisé. La variable d'origine et son type demeure inchangés (monReel est toujours une variable de type **float**) mais l'expression résultat que représente son évaluation (c'est à dire sa valeur) est convertie lors de la réalisation de l'affectation. Répétons-le c'est exactement ce que le compilateur a réalisé lors d'une conversion implicite mais maintenant la compilation est plus propre puisque qu'aucun avertissement ne sera affiché pour ces lignes, ce qui nous aidera à apporter plus d'attention aux autres messages du compilateur.

Le type Pointeur

Les variables de type pointeur permettent de stocker non plus une donnée (comme une lettre, un entier ou un nombre réel) mais l'adresse d'une autre variable.

Pour utiliser une telle variable, nous définirons donc sa valeur avec l'adresse d'une variable existante; pour cela nous utiliserons l'opérateur "&" (opérateur référence) qui appliqué à une variable fournit son adresse; c'est à dire l'endroit où la valeur de cette variable est stockée en mémoire.

Si nous avons défini :

```
short unEntier;
```

alors :

```
&unEntier
```

fournit l'adresse de cette variable.

Notez que l'opérateur "&" ne peut pas être appliqué sur des expressions évaluées. Par exemple "&(unEntier + 1)" est interdit, de même que "&32"; en effet, dans ces expressions ("unEntier + 1" ou le nombre 32) n'existent pas comme des emplacements mémoires définis, il est donc impossible d'en évaluer l'adresse.

La variable pointeur elle-même sera définie en utilisant le symbole "*" qui associé à un type définit le pointeur sur ce type. Nous dirons en effet que cette variable "pointe" sur une donnée, car contrairement à une variable non-pointeur, elle ne contient pas directement la donnée mais seulement l'adresse où cette donnée est stockée.

Voici des définitions de variables pointeurs :

```
short*          adresseDunEntier;
char*           adresseDunCaractere;
unsigned long*  adresseDunEntierPositif;
```

Notez que le fait de coller le symbole "*" à droite du nom du type est arbitraire, en effet les déclarations suivantes sont rigoureusement identiques aux précédentes :

```
short           *adresseDunEntier;
char            *adresseDunCaractere;
unsigned long   *adresseDunEntierPositif;
```

Selon votre propre préférence vous utiliserez la première en explicitant littéralement l'instruction "**short*** adresseDunEntier" comme "adresseDunEntier" est une variable dont le type est "**short***" c'est à dire "pointeur sur un **short**"; ou bien considérant la forme "**short** *adresseDunEntier" vous pourrez vous dire "*adresseDunEntier" est une variable pointeur qui pointe sur une variable de type "**short**"; les deux sont tout à fait équivalents.

Pour stocker l'adresse de notre variable "unEntier" nous faisons donc :

```
short     unEntier;
short*    adresseDunEntier;

adresseDunEntier = &unEntier;
```

Comme pour une variable non-pointeur, nous pouvons déclarer notre variable pointeur et fixer sa valeur en même temps, soit :

```
short     unEntier;

short*    adresseDunEntier = &unEntier;
```

Pour lire la valeur pointée par une variable pointeur, nous utiliserons le signe "*" à gauche du nom de la variable pointeur; cette opération s'appelle **déréférencer** le pointeur et le symbole "*" est "l'opérateur d'adressage".

Ainsi pour obtenir la valeur pointée par "adresseDunEntier" nous utilisons :

```
*adresseDunEntier
```

cette écriture est à l'origine de la préférence souvent rencontrée pour la seconde forme de définitions des pointeurs vue ci-dessus.

Appliquons cela :

```
short     unEntier;       /* déclaration d'un entier court     */
short     unNombre;       /* un autre entier court             */
short*    unPointeur;     /* un pointeur sur un entier court   */

unEntier = 5;             /* fixe la valeur de unEntier        */
unPointeur = &unEntier;   /* stocke l'adresse de unEntier */
unNombre = *unPointeur;   /* récupère la donnée pointée: 5     */
```

Notez que ce code est équivalent à :

```
short     unEntier;       /* déclaration d'un entier court     */
short     unNombre;       /* un autre entier court             */

unEntier = 5;             /* fixe la valeur de unEntier        */
unNombre = unEntier;      /* prend directement la valeur       */
```

Cet exemple montre un cas où il semble inutile d'utiliser un pointeur, dans un vrai code son usage se révélerait indispensable; imaginez par exemple qu'au lieu d'affecter la variable "unPointeur" avec l'adresse de "unEntier" nous devons réaliser un test pour choisir parmi plusieurs variables celle dont nous stockerons l'adresse; l'utilisation d'un pointeur permet alors d'utiliser la donnée de la variable choisie sans devoir réaliser ce test à chaque fois.

De plus nous pouvons utiliser la variable pointeur pour modifier le contenu de la variable pointée, par exemple :

```
short     entier1;        /* déclaration d'un entier court     */
short     entier2;        /* un autre entier court             */
short     entier3;        /* un autre entier court             */
etc...
short*    unPointeur;     /* un pointeur sur un entier court   */

un test conduirait par exemple à :
unPointeur = &entier2;    /* stocke l'adresse de entier2       */

*unPointeur = 5;          /* fixe la donnée pointée à 5        */
maintenant entier2 vaut 5
```

Une autre mise en oeuvre incontournable des pointeurs est l'utilisation de paramètres variants avec une fonction. Paramètre variant signifie que vous désirez passer une variable à une fonction et vous souhaitez que cette fonction puisse modifier le contenu de cette variable.

Un exemple pourrait être une fonction utilisée lors d'un tri; nous souhaitons écrire une fonction qui reçoit deux nombres "a" et "b" et le cas échéant permute ces deux nombres afin que "a" soit inférieur ou égal à "b".

Si nous définissons simplement le type des paramètres transmis à cette routine comme étant des types non pointeur, nous ne recevrons que la valeur de la variable et le traitement effectué sera perdu à la sortie de la fonction.

Ainsi le programme suivant ne marche pas !

Code 2 : Mauvais passage de paramètres variants

```
#include <stdio.h>

void permute(short a, short b) {
      short temp;
      if (a > b){
            temp = a;
            a = b;
            b = temp;
      }
}

void main(){
      short x, y;               /* déclaration de deux entiers courts */

      x = 5;
      y = 2;
      printf("%hd %hd\n", x, y);/* affiche les nombres          */
      permute(x, y);                  /* appel de notre fonction*/
      printf("%hd %hd\n", x, y);/* affiche les nombres          */
}
```

A l'exécution, nous obtenons deux fois l'affichage de :

```
5    2
```

Remarque : ne vous effrayez pas de l'expression "`printf("%hd %hd\n", x, y)`". Cette commande permet d'afficher les deux nombres "x" et "y"; nous étudierons cela plus loin.

Pour obtenir, le résultat souhaité nous devons transmette à notre fonction l'adresse des variables afin que ce soit bien le contenu de ces variables qui soit échangé.

Pour cela notre routine déclare des paramètres de type **short*** et non plus **short**, nous sommes donc obligé de déréférencer ces variables afin de lire la valeur pointée et lors de l'appel à cette routine nous devons transmettre les adresses grâce à l'opérateur "&".

Voila la version corrigée du programme :

Code 3 : Bon passage de paramètres variants

```
#include <stdio.h>

void permute(short* a, short* b) {
      short temp;
      if (*a > *b){
            temp = *a;
            *a = *b;
            *b = temp;
      }
}

void main(){
      short x, y;               /* déclaration de deux entiers courts */

      x = 5;
      y = 2;
      printf("%hd %hd\n", x, y);/* affiche les nombres          */
      permute(&x, &y);                /* appel de notre fonction*/
      printf("%hd %hd\n", x, y);/* affiche les nombres          */
}
```

A l'exécution, nous obtenons l'affichage suivant :

```
5    2
2    5
```

Le type Tableau

Un tableau est une variable qui permet de stocker non plus une seule valeur mais un nombre fini et déterminé de valeurs.

Il est possible de créer des tableaux d'éléments quelconques, par exemple des tableaux de caractères - nous y reviendrons en détail plus loin - ou encore des tableaux d'entiers ou de réels; nous pouvons également définir des tableaux de variables pointeurs et même des tableaux de fonctions, mais commençons par le début.

Les tableaux statiques

Un tableau statique est un tableau dont la taille (le nombre d'éléments) est figée et déclarée dans le source.

Un tel tableau est défini en utilisant les symboles "[" et "]" après le nom de la variable et en faisant figurer le nombre d'éléments désirés entre ces crochets; par exemple :

```
short tableau_d_entier[5];
```

définit une variable nommée "tableau_d_entier" qui est un tableau de 5 entiers de type **short**. Les éléments du tableau sont numérotés de 0 à (taille du tableau - 1); par exemple nous pouvons réaliser :

```
short tableau_d_entier[5];

tableau_d_entier[0] = 1;
tableau_d_entier[1] = 3;
tableau_d_entier[2] = 5;
tableau_d_entier[3] = 3;
tableau_d_entier[4] = 1;
```

Notez que nous pouvons définir les valeurs des éléments du tableau lors de sa déclaration, pour cela nous placerons les différentes valeurs entre accolades en les séparant par des virgules; la définition suivante est identique au tableau précédent :

```
short tableau_d_entier[5] = {1, 3, 5, 3, 1};
```

Nous pouvons également définir des tableaux multi-dimensions en répétant la spécification de taille faites avec les crochets, par exemple voici une matrice 2*2 :

```
float matrice[2][2];

matrice[0][0] = 1.0;
matrice[0][1] = 2.0;
matrice[1][0] = 3.0;
matrice[1][1] = 4.0;
```

cette variable représente en quelque sorte la matrice : $\begin{vmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{vmatrix}$

Notez qu'ici aussi nous pouvons définir les valeurs des éléments lors de la déclaration, nous imbriquerons simplement des accolades pour définir chacune des dimensions du tableau; ainsi la définition suivante est identique à la matrice précédente :

```
short matrice[2][2] = {{1.0, 2.0}, {3.0, 4.0}};
```

Il existe une forte similitude en C/C++ entre un type tableau et le type pointeur sur ce même type. Ainsi étant défini le "tableau_d_entier" ci-dessus nous pouvons réaliser :

```
short* ptr_entier = tableau_d_entier;
```

ou encore, c'est identique :

```
short* ptr_entier = &tableau_d_entier[0];
```

En effet une variable tableau contient l'adresse du premier élément de ce tableau.

Les tableaux dynamiques

Un tableau dynamique est un tableau dont la taille n'est pas figée dans le source.

La remarque précédente concernant la similitude entre une variable tableau et un pointeur sur un type équivalent nous permet de déclarer aisément un tel tableau de taille inconnue, nous utiliserons simplement un pointeur et stockerons l'adresse de ce tableau dans la variable pointeur.

Pour illustrer cela, nous devons savoir que le fichier d'en-tête (ou header) "stdlib.h" déclare la fonction "malloc()" avec le prototype (l'API) suivant :

```
void* malloc(size_t size);
```

cela signifie qu'elle attend un paramètre de type "**size_t**" (qui est également défini dans ce fichier "stdlib.h" comme l'équivalent de **unsigned long int**) qui indique le nombre d'octets à allouer et qu'elle renvoie un pointeur **void***.

Nous connaissions le type **void** que signifie **void*** ? Littéralement c'est un pointeur sur rien du tout, dans la pratique c'est le contraire : cela décrit un pointeur sur n'importe quelle type de donnée, un pointeur universel en quelque sorte.

Il est important de noter que le paramètre taille transmis à malloc() (la variable nommée "size" dans le prototype) est une taille en octets. Ainsi si nous ne souhaitons pas un tableau d'octets - de **char** - nous devrons préciser le nombre d'éléments multiplié par la taille d'un élément. Pour cela nous pouvons utiliser l'opérateur "sizeof()" - ce n'est pas une fonction car le résultat est évalué à la compilation - qui fournit la taille du type ou de la variable utilisé comme paramètre. Par exemple, l'expression "sizeof(**short**)" est remplacée par la valeur 2 durant la phase de compilation sur machine Intel.

Enfin, pour stocker le résultat (de type **void***) de la fonction malloc() dans notre variable, nous devons réaliser une conversion explicite de **void*** vers le type qui nous intéresse.

Voyons comment réaliser tout cela :

Code 4 : Création d'un tableau de taille quelconque

```
#include <stdio.h>        /* pour la fonction printf() */
#include <stdlib.h>       /* pour la fonction malloc() */

void main(){
     short i;             /* une variable utilisée pour une boucle */
     short taille;        /* mémorise la taille du tableau */
     long* tableau;       /* pointeur pour stocker l'adresse du tableau  */

     taille = 5;          /* saisi par l'utilisateur dans un vrai code   */

/*   allocation (dynamique) de notre tableau   */
     tableau = (long*) malloc(taille * sizeof(long));

/*   remplissage du tableau, on utilise les indices 0 à (taille -1) */
     for (i = 0; i < taille; i++)
          tableau[i] = i;

/*   affichage du contenu du tableau */
     for (i = 0; i < taille; i++)
          printf("%lu\n", tableau[i]);

/*   destruction du tableau : libération de la mémoire réservée */
     free(tableau);

}
```

Notez que tout bloc de mémoire alloué avec malloc() doit être libéré par un appel à free(). Le prototype de free() est "**void** free(**void*** ptr);" ici il n'est pas besoin de convertir explicitement notre tableau de type **long*** en **void*.** En effet, un type pointeur générique peut recevoir n'importe quel pointeur puisque cette conversion perd de l'information.

Les tableaux de caractères

Dans notre premier programme nous avons utilisé la fonction printf() avec une chaîne de caractères; notez qu'en C/C++ il n'existe pas de type "chaîne de caractères" équivalent au type "String" du Pascal; ce type est obtenu en utilisant un tableau de caractères.

Pour déclarer une chaîne statique, nous utiliserons donc un tableau - dont la taille sera ou non précisée. Si elle ne l'est pas le compilateur réservera un tableau pour y stocker toute la chaîne, si par contre une taille "N" est précisé le compilateur ne retiendra que les "N" premiers caractères de cette chaîne; exemple :

```
char chaine1[] = "hello";      /* déclaration correcte */
char chaine2[5]= "bonjour";    /* correct mais chaîne2 contient "bonjo"  */
```

Que font ces instructions ?

Lorsque nous déclarons "char chaine1[] = "hello"", le compilateur réserve dans une zone mémoire (généralement dans le segment de données) six octets et il y range notre chaîne en y ajoutant un caractère ascii zéro marquant la fin de cette chaîne :

h	e	l	l	o	0

ensuite il fixe la valeur de "chaine1" avec l'adresse du premier caractère de la chaîne.

Notez que la déclaration :

```
char* chaine1 = "hello";
```

est complètement équivalente à la définition de "chaine1" précédente.

Excepté cet ajout automatique d'un caractère ascii zéro de terminaison de chaînes, les tableaux de caractères sont semblables aux tableaux d'autres types.

Nous pourrons par exemple créer des chaînes dynamiquement en utilisant malloc(); nous devrons cependant dans ce cas être vigilant à ajouter nous même le caractère zéro lors de la construction de nos chaînes.

Les opérateurs

Les opérateurs sont nombreux en C et C++, ceci est dû au fait que certaines opérations réalisées par des fonctions dans d'autres langages sont faites ici par des opérateurs et également au fait que le C et C++ permettent des formes d'écritures d'opérations impossibles dans d'autres langages.

Les opérateurs mathématiques

les opérateurs disponibles sont :

Opérateur	Syntaxe	Description
+	a + b	calcule la somme de a et b
-	a - b	calcule la différence de a moins b
*	a * b	calcule la multiplication de a par b
/	a / b	calcule la division de a par b
%	a % b	calcule le reste de la division entière de a par b
-	- a	évalue l'opposé de a

Notez que le résultat d'une opération numérique dépend du type de la variable résultat, en effet si ce nombre dépasse l'intervalle de définition du type de la variable résultat il sera tronqué, exemple:

```
unsigned short  a = 160, b = 500;
unsigned short  c = a * b;

printf("%hu\n", c);        /*   affiche 14464 soit 80000 - 65536 */
```

cette opération aurait dû être codée :

```
unsigned short  a = 160, b = 500;
unsigned long   c = a * b;

printf("%u\n", c);         /*   affiche 80000   */
```

De plus l'utilisation de nombres signés ou non signés peut également produire des résultats différents, par exemple :

```
signed short    a = 80, b = 500;
signed short    c = a * b;
unsigned short  d = a * b;

printf("%hd\n", c);        /*   affiche -25536  */
printf("%hu\n", d);        /*   affiche 40000   */
```

Dans cet exemple les variables c et d contiennent en fait exactement la même valeur, c'est l'interprétation que l'on en fait qui provoque la différence des résultats.

Enfin, le résultats dépend du type des variables opérandes, ainsi le code :

```
short      a = 80, b = 500;
float      c = b / a;

printf("%g\n", c);         /*   affiche 6 au lieu de 6.25 */
```

n'affiche pas le (vrai) résultat de la division (non entière) de b par a; en effet a et b étant entiers, une division entière est réalisée.

Pour obtenir le résultat correct il faut, soit introduire un nombre flottant dans les arguments de la division, exemple :

```
short      a = 80, b = 500;
float      c = 1.0 * b / a;

printf("%g\n", c);          /*    affiche 6.25    */
```

soit retyper une des variables **signed short** en **float**, ce changement de type est réalisé en indiquant avant la variable à altérer un nom de type entre parenthèses :

```
short      a = 80, b = 500;
float      c = (float) b / a;

printf("%g\n", c);          /*    affiche 6.25    */
```

Les opérateurs logiques et relationnels

Les opérateurs suivant produisent tous un résultat logique “vrai” (généralement codé 1) ou “faux” (codé 0).

Opérateur	Syntaxe	Description
==	a == b	évalue a égal b
!=	a != b	évalue a différent de b
&&	a && b	effectue un “ET logique” entre a et b
||	a || b	effectue un “OU logique” entre a et b
!	!a	évalue la négation logique de a
>	a > b	évalue a plus grand que b
>=	a >= b	évalue a plus grand ou égal à b
<=	a <= b	évalue a plus petit ou égal à b
<	a < b	évalue a plus petit que b

On notera que les opérateurs relationnels (>, >=, <= et <) ont une priorité supérieure aux opérateurs logiques (==, !=, && et ||), ainsi :

```
(a > b && c < d)
```

est équivalent à :

```
( (a > b) && (c < d) )
```

De plus tous les opérateurs ont une priorité inférieure aux opérateurs mathématiques, ainsi :

```
(a < b - 1 && c % d)
```

est équivalent, puisque qu’un test logique est “vrai” pour toutes valeurs différentes de zéro, à :

```
( ( a < (b - 1) ) && ((c % d) != 0) )
```

L'opérateur unaire de négation logique "!" appliqué à une valeur logique "vrai" ou "faux" renvoie son contraire; appliqué à une valeur numérique ou une adresse non nulle il renvoie 0, enfin appliqué à un opérande nul il renvoie 1; cet opérateur est couramment utilisé dans des tests :

```
long*      a_ptr;
....
a_ptr = (long*) malloc( 10 * sizeof(long) );
if (!a_ptr) {
     /* l'allocation a échouée       */
     /* a_ptr est un pointeur nul    */
}


short      c;
....
if (!c){        /*   équivalent à: if (c == 0) */
     ....
}
```

Les opérateurs d'incrémentation et décrémentation

L'opérateur "++" est l'opérateur d'incrémentation: il ajoute 1 à la variable sur lequel il est appliqué. L'opérateur "--" est l'opérateur de décrémentation: il diminue de 1 la valeur de la variable sur lequel il est appliqué.

Ces deux opérateurs produisent deux résultats différents selon qu'ils sont placés à gauche ou à droite d'une expression.

Placés à gauche, ils agissent avant que la variable ne soit utilisée dans l'instruction qui la contient; placés à droite, ils agiront après que la variable ait été utilisée.

Illustrons ceci :

```
short a = 5;    /* déclare "a" avec la valeur 5      */

a++;            /* fixe la valeur de "a" à 6              */

short b = a++;  /* fixe "b" à "a" soit 6                  */
                /*   PUIS incrémente "a" ("a" égal 7)     */

short c = ++a;  /* incrémente "a" (soit 8)           */
                /*   PUIS affecte "c" ("c" égal 8)        */
```

La position de l'opérateur (avant ou après l'opérande) induit donc des effets de bord (ou effets à rebond), ces effets peuvent être très sensibles et produiront un résultat imprévisible lorsque l'opérande apparaît plusieurs fois dans l'expression; par exemple l'instruction suivante cherche à remplir l'élément d'indice "i" d'un tableau avec cette valeur de "i" :

```
short a[10];     /* un tableau de 10 entiers courts */
....
short i = 3;
a[ i ] = i++;    /* qu'avons-nous réalisé ??? */
```

après cette dernière instruction, "i" vaut 4 mais a t'on fixé a[3] ou a[4] ?

En fait cela dépend du compilateur et la seule réponse certaine est qu'il ne faut pas abuser des opérateurs ou pour le moins qu'une telle expression ne doit jamais être codée.

Les opérateurs binaires

Ces opérateurs opèrent sur la représentation binaire des nombres, ils ne peuvent donc être utilisés qu'avec des opérandes de type entiers.

Opérateur	Syntaxe	Description
&	a & b	effectue un ET bit à bit
|	a | b	effectue un OU inclusif bit à bit
^	a ^ b	effectue un OU exclusif bit à bit (appelé aussi XOR)
<<	a << b	effectue un décalage à gauche de b bits sur le nombre a
>>	a >> b	effectue un décalage à droite de b bits sur le nombre a
~	~a	effectue un complément à un (négation binaire)

Les opérateurs de décalage de bits seront utilisés avec prudence sur les types signés, en effet lors d'un décalage à droite les bits décalés seront remplis par des zéros pour un opérande non signé alors qu'un nombre signé sera complété de bits de signe (décalage arithmétique) ou par des zéros (décalage logique) en fonction de la machine utilisée et / ou du compilateur.

Rappel pour les personnes non rompues aux opérations binaires par une longue pratique d'assembleur:

un décalage de bits à gauche revient à multiplier le nombre par une puissance de 2; formellement "a << b" est équivalent à "a * 2^b"; ainsi 5 << 2 est identique à (5 * 2^2 = 5 * 4); cette écriture apporte une vitesse de traitement optimale : le décalage est réalisé par le microprocesseur beaucoup plus rapidement qu'une multiplication n'est réalisé par l'unité de traitement des entiers; cependant si nous n'êtes pas habitué à ces techniques, laissez le compilateur faire son œuvre : la plupart savent reconnaître les expressions pouvant bénéficier d'un décalage de bits en lieu et place d'une multiplication, ainsi le code :

```
a = b * 80
```

sera souvent codé :

```
a = (b * 64) + (b * 16)
```

soit :

```
a = (b << 6) + (b << 4)
```

Note : rappelez-vous qu'en assembleur plus le code est long, plus ça va vite…

Rappel des tables vérités des opérations binaires :

a	~a	b	a & b	a | b	a ^ b
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	1	1	0

Les opérateurs contractés

La plupart des opérations de la forme a = a **opérateur** b peuvent être écrites de manière plus concise sous la forme a **opérateur**= b.

Ainsi:

`a = a + b`	peut s'écrire	`a += b`
`a = a - 5`	peut s'écrire	`a += 5`
`a = a * (b + 2)`	peut s'écrire	`a *= b + 2`
`a = a & 0x0F`	peut s'écrire	`a &= 0x0F`

Remarque : il n'existe pas de forme contractée pour les opérateurs "<<" et ">>", ni, évidemment, pour les opérateurs unaires.

Les pièges à éviter

Lors de votre apprentissage du langage C et C++ vous risquez de rencontrer des bugs (incompréhensibles) dûs à des inversions entre différents opérateurs ressemblants.

Le premier piège à éviter, surtout si vous avez l'expérience du Pascal, est la confusion entre les opérateurs de tests ("==") et d'affectation ("="), ainsi si vous écrivez

```
a == 5;
```

au lieu de

```
a = 5;
```

le compilateur ne détectera aucune erreur (sauf pour une expression comme `short a == 5;`) et votre initialisation se résumera à un test (dont le résultat est ignoré).

Cette erreur est rarement commise, par contre l'inverse peut se produire, par exemple lors d'un test :

```
if (a = 2) ...  /* possible erreur de codage */
```

ici, non seulement le test sera toujours vérifié pour des valeurs différentes de zéro, mais en plus vous altérez le contenu de la variable; certains compilateurs émettent un avertissement ("affectation non désirée possible") pour ce type d'écriture, d'autres ne vous diront rien.

Une solution pour contourner ce problème est d'écrire, quand cela est possible, la constante à gauche du signe, ainsi :

```
if (2 = a) ...  /* possible erreur de codage MAIS erreur de syntaxe */
```

provoquera toujours une erreur de compilation; cependant vous comparerez souvent deux variables entre elles et dans ce cas cette astuce est inutilisable; dans d'autres cas vous voulez précisément réaliser cette affectation et tester si la valeur transmise est non nulle, c'est le cas notamment lorsque vous stockez le résultat d'une fonction retournant un code d'erreur, on écrirait alors :

```
if ( err = maFonction() ) ...
```

ceci est valide et pour le coup on sera peut être agacé par les avertissement émis par le compilateur, la meilleure façon d'écrire une telle instruction serait :

```
if ( ( err = maFonction() ) == noErr ) ...
                                        /* où noErr est une constante */
```

L'autre erreur souvent commise et l'inversion entre les opérateurs ET et OU logiques versus binaires, ainsi:

```
short a = 5, b = 3;
if (a & 4  &&  b & 1) ...
```

est évalué comme if (4 "et" 1) soit if ("vrai" "et" "vrai"), donc le test est vérifié. Par contre :

```
short a = 5, b = 3;
if (a && 4  &  b && 1) ...
```

est évalué comme if (5 "et" (4 "et binaire" b) "et" 1) soit if (5 "et" 0 "et" 1), donc le test n'est pas vérifié.

Les instructions de tests

L'instruction "if"

L'instruction de test "**if** ... **else** ..." sert à adapter un traitement à toutes sortes de conditions pouvant se ramener à une expression "vrai" ou "faux"; notez que les mots clés "**if**" et "**else**" doivent être saisis en minuscule, comme tous les mots clés.

La syntaxe de l'instruction "**if**" est :

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

Remarque :

- la condition test est toujours placée entre parenthèses.
- la clause "**else**" et le bloc d'instruction lié sont optionnels.
- toute instruction se termine par un point virgule, y compris celle précédant l'instruction "**else**" (contrairement à la syntaxe du Pascal).

Nous supposons dans les exemples suivants que "A", "B" et "C" désignent des valeurs booléennes. Voici un premier exemple de test simple :

```
if ( A )
     printf("A est vrai\n");
else
     printf("A est faux\n");
```

Les blocs d'instructions exécutés si le test est vérifié ou non peuvent être constitués de plusieurs instructions, nous utiliserons alors des accolades pour délimiter ces blocs; exemple :

```
if ( A ) {
     printf("A\n");
     printf("est vrai\n");
}
else {
     printf("A\n");
     printf("est faux\n");
}
```

Souvent nous aurons à déterminer le traitement à réaliser parmi plus de deux choix, pour cela nous pouvons enchaîner les tests "**if** ... **else**"; exemple :

```
if ( A )
     printf("A est vrai\n");

else if ( B )
     printf("B est vrai\n");

else
     printf("A et B sont faux\n");
```

Lorsque nous enchaînons plusieurs tests, il est important d'être vigilant sur la portée de la clause "**else**", considérons par exemple le bloc suivant :

```
if ( A )
     if ( B )
          printf("A et B sont vrais\n");
else if ( C )
     printf("C est vrai, A est-il vraiment faux ?\n");
else
     printf("A et C seraient faux\n");
```

Les propositions faites par ce code sont fausses; en effet le test “else if (C)” s’applique si “B” est faux et non si c’est “A” qui est faux.

Correctement indenté code ci-dessus est en fait équivalent à :

```
if ( A ) {
     if ( B )
          printf("A et B sont vrais\n");

     else if ( C )
          printf("C est vrai, A est-il vraiment faux ?\n");

     else
          printf("A et C seraient faux\n");
}
```

avec cette écriture nous voyons clairement que le test est mal codé, nous devons écrire:

```
if ( A ) {
     if ( B )
          printf("A et B sont vrais\n");
}
else if ( C )
     printf("C est vrai, A est vraiment faux !\n");

else
     printf("A et C sont faux\n");
```

ce codage est meilleur et nous permet de voir que le cas “A” vrai, “B” faux a été oublié de l’analyse; la bonne implémentation sera donc :

```
if ( A ) {
     if ( B )
          printf("A et B sont vrais\n");
     else
          printf("A est vrai mais B est faux\n");
}
else if ( C )
     printf("C est vrai, A est faux\n");

else
     printf("A et C sont faux\n");
```

L’expression condition

La condition appliquée à l’instruction “**if**” peut être une expression plus compliquée qu’une simple valeur booléenne mais doit pouvoir être évaluée comme une valeur booléenne; notons que les compilateurs C/C++ considèrent comme “faux” la valeur zéro et comme “vrai” toute autre valeur, ce qui signifie que mis à part le résultat d’une fonction de type **void**, tout peut être interprété comme une valeur booléenne; par exemple le code suivant affichera “vrai” (c’est un exemple extrême : qui coderait cela ?).

```
if ( "une chaîne" )
     printf("vrai\n");
else
     printf("faux\n");
```

Les exemples suivants illustrent un peu plus ce principe :

```
short n;
.......
if ( n )
     printf("n est non nul\n");
else
     printf("n est nul\n");
```

est équivalent à :

```
short n;
.......
if ( n != 0 )
     printf("n est non nul\n");
else
     printf("n est nul\n");
```

Des conditions avec “et” et “ou”

La condition test peut également être constituée de plusieurs conditions associées par des critères “et logique” et “ou logique”.

L’opérateur “et logique” s’écrit “&&”; l’opérateur “ou logique” s’écrit “||” ([Alt Gr] + 6); exemple :

```
if ( A && B )
     printf("A et B sont vrais\n");

else if ( A || C )
     printf("A ou C ou les deux sont vrais\n");

else
     printf("A et C sont faux, B est peut être vrai\n");
```

L’opérateur ternaire “?:”

On parle d’opérateur “ternaire” car celui-ci nécessite trois arguments, contrairement aux opérateurs binaires qui prennent deux arguments (exemple l’opérateur de test “==”, on écrit: arg1 == arg2) et aux opérateurs unaires qui n’utilisent qu’un argument (exemple l’opérateur de négation “-”, on écrit: -arg).

Cet opérateur s’écrit: (condition) **?** (instruction_1) **:** (instruction_2)

si la condition est vérifiée le bloc “instruction_1” est exécuté, sinon c’est le bloc “instruction_2” qui l’est.

Cet opérateur se prête bien aux affectations conditionnelles, par exemple nous voudrions écrire :

```
short choix;

if ( A )
     choix = 1;
else
     choix = 2;
```

ce code peut être écrit grâce à l’opérateur “**? :**” :

```
short choix = ( A ) ? 1 : 2;
```

Pour des raisons de lisibilité, on évitera son emploi dans des blocs de codes trop compliqués. Retenez cependant que partout où nous utilisons une séquence :

```
if ( condition ) {
     expression 1
}
else {
     expression 2
}
```

nous pouvons écrire :

```
( condition ) ? { expression 1 } : { expression 2 }
```

L'instruction "switch"

Imaginons que nous souhaitons écrire un bloc de tests portant de nombreuses conditions, avec des critères "ou logique" et des tests imbriqués, par exemple :

```
short X;
...
if ( X == 1 )
     [instruction A];

else if ( X == 2 || X == 3 )
     [instruction B];

else if ( X == 4 || X == 5) {
     [instruction C];
     if ( X == 4 )
          [instruction D];
}
else
     [instruction E];
```

pour simplifier l'écriture de tel bloc nous utiliserons l'instruction "**switch**(…) { … }" associée à plusieurs étiquettes "**case**".

Nous utilisons comme argument de la clause "**switch**" l'expression dont nous aurions testée la valeur; chaque valeur-test fait l'objet d'une étiquette "**case**" suivie d'un bloc d'instructions et du mot clé "**break**" dès que le traitement à réaliser pour cette valeur-test est effectué; enfin l'étiquette "**default**" permet de gérer tous les cas n'ayant pas été traités.

Cela conduit à :

```
short X;
...
switch ( X ) {
     case 1:
          [instruction A];
          break;
     case 2:
     case 3:
          [instruction B];
          break;
     case 4:
          [instruction C];
     /*   et continue avec l'instruction suivante */
     case 5:
          [instruction D];
          break;
     default:
          [instruction E];
}
/* l'instruction "break" nous mène ici */
```

Explications :

- notez la présence d'un ":" (deux points) après la valeur des étiquettes **case**
- l'étiquette "**case** 1:" intercepte le cas où X est égal à la valeur 1, le bloc [instruction A] est alors exécuté puis l'instruction **break** nous fait sortir du bloc **switch**. Cette instruction **break** est indispensable, sans elle l'exécution continuerait linéairement avec le bloc [instruction B].
- l'étiquette "**case** 2:" atteinte si X est égal à 2 ne réalise aucune opération si ce n'est de poursuivre avec les instructions suivantes.
- l'étiquette "**case** 3:" est atteinte si X est égal à 3 mais également pour X égal 2 puisqu'à l'étiquette précédente nous n'avons pas effectué de **break**.

- l'étiquette "**case** 4:" provoque l'exécution de l' [instruction C]; nous ne faisons pas suivre ce bloc d'instruction du mot clé **break**, l'exécution se poursuit donc avec le bloc [instruction D].
- l'étiquette "**case** 5:" est atteinte directement si X est égal à 5 et par poursuite du traitement si X était égal à 4; nous réalisons l'exécution de l' [instruction D] et nous quittons le bloc switch.
- enfin, tous les cas non traités (c'est à dire dans tous les cas où X est plus petit que 1 ou plus grand que 5) nous effectuons le bloc [instruction E].

Remarques :

- l'étiquette "**default**" est optionnelle.
- une étiquette ne peut pas définir un intervalle de valeur, par exemple "**case** 1..3:" est interdit; on écrira à la place plusieurs "**case**" comme pour les valeurs 2 et 3 dans l'exemple ci-dessus.
- la valeur d'une étiquette doit pouvoir être évaluée au moment de la compilation et être une valeur entière. C'est donc une valeur immédiate (comme une constante), une opération sur des constantes (exemple: "maConstante + 2"), mais cela ne peut pas être le résultat d'une fonction ou tout autre expression évaluée à l'exécution uniquement.
- les valeurs numériques des étiquettes n'exigent ni d'être ordonnées (de la plus petite valeur à la plus grande) ni de former une série discrète (1 puis 2 puis 3 ...).

Les instructions de boucles

L'instruction "while"

L'instruction de boucle **while** permet de réaliser un traitement tant qu'une condition booléenne est vraie.

La syntaxe de l'instruction **while** est :

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

Remarque : la condition test est toujours placée entre parenthèses.

Si le bloc instruction est composé de plusieurs instructions, on placera ces instructions entre les symboles de début et de fin de bloc: "{" et "}" soit :

```
while ( condition ) {
     instruction  1;
     instruction  2;
     ...
}
```

Exemple de boucle :

Code 5 : Boucle "while(..) { ... }"

```
#include <stdio.h>

/*   Saisie d'un nombre avec écho de ce nombre sur la console  */

void main() {
     short nombre = 1;
     while ( nombre != 0 ) {
          printf("Tapez un nombre (0 pour sortir) : ");
          scanf("%hd", &nombre);
          printf("Votre nombre est %hd\n", nombre);
     }
}
```

Le contenu de la boucle (affichage du message, saisie du nombre et réaffichage du nombre) est répété tant que l'utilisateur ne rentre pas la valeur zéro.

L'instruction "break"

L'instruction de sortie de boucle **break** permet de quitter la boucle en cours. Ceci peut être utile si le traitement est irréalisable ou si une erreur se produit dans la boucle.

Notez cependant que l'instruction **break** quitte la boucle immédiatement parente; mais en cas de boucle imbriquée il reste dans la boucle précédente.

Réécrivons le code précédent pour limiter le traitement à des nombres inférieurs à 100 :

Code 6 : Boucle "while(..) { ... }" avec sortie par "break"

```
#include <stdio.h>

void main() {
     short nombre = 1;
     while ( nombre != 0 ) {
          printf("Tapez un nombre (0 pour sortir) : ");
          scanf("%hd", &nombre);
          if (nombre > 100)
               break;
          printf("Votre nombre est %hd\n", nombre);
     }
}
```

Bien sûr, nous aurions pu écrire "**while (** nombre != 0 && nombre <= 100**)** ...", nous serions sortis de la boucle si "nombre" est plus grand que cent. Cependant l'instruction "printf("Votre nombre ..." aurait été également exécuté. Ici ce n'est pas dramatique mais on peut facilement imaginer un code où le traitement n'aurait pas dû être effectué.

L'instruction "do ... while( )"

Un bloc d'instruction contrôlé par une instruction "while" peut ne jamais être exécuté, en effet si la condition de test est (logiquement) fausse, l'exécution se poursuit avec l'instruction immédiatement suivante au bloc, par exemple :

```
#include <stdio.h>

void main() {
     short nombre;
     ...
     nombre = 0;          /* un traitement met "nombre" à zéro */

     ...
     while ( nombre != 0 ) {
          printf("Tapez un nombre (0 pour sortir) : ");
          scanf("%hd", &nombre);
          printf("Votre nombre est %hd\n", nombre);
     }
     printf("Vous avez raté la boucle\n");
}
```

n'affichera jamais rien d'autre que simplement "Vous avez raté la boucle".

L'instruction "do ... while ( )" permet de réaliser des boucles dont le test de sortie est évalué à la fin; ceci implique que le corps de la boucle sera exécuté au moins une fois.

La syntaxe de l'instruction "**do** ... **while** ( )" est :

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

Ici aussi, on placera les instructions entre les symboles de début et de fin de bloc: "{" et "}" si plusieurs instructions doivent être exécutées, soit :

```
do {
     instruction  1;
     instruction  2;
     ...
     }
while  ( condition );
```

Reprenons notre exemple, il devient :

Code 7 : Boucle "do { ... } while(..)"

```
#include <stdio.h>

void main() {
     short nombre;   /* l'initialisation n'est plus nécessaire */
     do {
          printf("Tapez un nombre (0 pour sortir) : ");
          scanf("%hd", &nombre);
          printf("Votre nombre est %hd\n", nombre);
     } while ( nombre != 0 )
}
```

ces boucles (avec contrôle à la fin) sont moins utilisées, cependant il existera toujours un problème pour lequel le meilleur algorithme utilisera cette forme de boucle.

L'instruction "for"

L'instruction de boucle **for** est peut être la plus souple du langage C, on dit couramment que n'importe quel programme C peut se ramener à une boucle **for**; la syntaxe de l'instruction **for** est :

```
for  ( initialisation ; test ; incrémentation )
     instruction ;
```

Au nouveau, si plusieurs instructions doivent être exécutées à chaque itération, on écrira :

```
for  ( initialisation ; test ; incrémentation ) {
     instruction  1;
     instruction  2;
     ...
}
```

Les trois expressions d'initialisation, de tests et d'incrémentation sont facultatives, ainsi le bloc :

```
for  ( ; ; ) { }
```

est un bloc d'instruction valide, vide et répété à l'infini.

L'expression d'initialisation

La première instruction sert à initialiser des variables, vous pouvez aussi définir une variable lors de cette initialisation (en C++ et sous certaines conditions en C), par exemple une boucle contrôlée par une variable "compteur" commencerait par :

```
for  ( short compteur = 0 ; . . .
```

Remarque : dans le cas où la variable est définie dans l'instruction **for**, celle-ci n'est plus visible dès que l'on sort de la boucle (son utilisation provoquera une erreur signalée par le compilateur). De plus les dernières normes du langage C++ recommandent ou imposent l'emploi d'une variable locale comme variable de contrôle de l'instruction **for** et la visibilité de cette variable est réduite à cette boucle; ainsi dans :

```
for  ( short compteur = 0 ; . . .) {
     quelque chose;
}
for  ( short compteur = 0 ; . . .) {
     autre chose
}
```

l'identificateur "compteur" décrit bien deux variables différentes, chacune n'étant utilisable que dans son instruction **for**; cependant encore de nombreux compilateurs ignorent cette norme et signaleraient, par un avertissement ou une erreur, que l'identificateur "compteur" est déjà déclaré lors de la compilation de la seconde boucle.

L'expression de test

La condition de test est vérifiée au début de l'exécution du bloc, comme c'était le cas pour l'instruction **while**, ainsi si notre boucle commence par :

```
for  ( short compteur = 0 ; compteur < 0 ; . . .
```

le corps de la boucle ne sera jamais exécuté car compteur (initialisé à zéro) ne vérifie pas la condition test (zéro n'est pas strictement inférieur à zéro).

De plus le test peut être complexe et mettre en jeu des opérateurs logiques, par exemple nous pouvons définir :

```
for  ( short compteur = 0 ; compteur < ValeurMax && variableLogique ; . .
```

L'expression d'incrémentation

La troisième partie de l'instruction **for** est généralement utilisée pour incrémenter ou décrémenter la variable de contrôle, ainsi pour réaliser cinq itérations nous codons:

```
for  ( short compteur = 0 ; compteur < 5 ; compteur++ ) {
     printf("%hd\n", compteur);
}
```

ce code affichera: 0, 1, 2, 3, 4 et 5.

Nous pouvons également faire varier le compteur avec un pas négatif, ainsi pour aller de 10 à 2 par pas de -2 nous codons :

```
for  ( short compteur = 10 ; compteur >= 2 ; compteur -= 2 ) {
     printf("%hd\n", compteur);
}
```

ce code affichera: 10, 8, 6, 4 et 2.

Cette expression d'incrémentation peut en fait contenir toute expression devant être exécutée à la fin de la boucle, avant que l'itération suivante ne soit réalisée.

Remarque : il n'est pas possible d'insérer un bloc d'instruction délimité par les signes "{" et "}" comme expression d'initialisation, de test ou d'incrémentation d'une boucle **for**. Il est néanmoins possible de combiner plusieurs tests dans l'expression de test en utilisant les opérateurs logiques ("et logique" et "ou logique"). De même, l'expression d'incrémentation peut être constituée de plusieurs instructions, il suffit pour cela de séparer les différentes instructions par des virgules à la place du point-virgule habituel; cela définira alors une seule instruction valide.

Voila un exemple avec notre programme de saisie de nombre

Code 8 : Boucle "for"

```
#include <stdio.h>

void main() {
     short pairs = 0;
     short impairs = 0;
     short compteur, nombre;

/* Saisie de 10 nombres maximum */
/* avec décompte des nombres pairs et impairs */

     for ( compteur = 0;
           compteur < 10 && nombre != 0;
           compteur++ , ((nombre % 2) == 0) ? pairs++ : impairs++ )
     {
          printf(
               "Tapez votre %hd%s nombre (0 pour sortir) : ",
               compteur, (compteur == 0) ? "ier" : "ième");
          scanf("%hd", &nombre);
          printf("Votre nombre est %hd\n", nombre);
     }
     printf("Vous avez tapé %hd nombre(s) pair(s)", pairs);
     printf(" et %hd nombre(s) impair(s)\n", impairs);
}
```

Dans cet exemple, l'expression test combine deux vérifications "nombre inférieur à 10" et "nombre différent de zéro" grâce à l'opérateur "et logique" noté "&&".

De même, l'expression d'incrémentation incrémente compteur ("compteur++") et, selon le résultat du test "nombre % 2" qui calcule nombre modulo 2 et donc retourne 0 si nombre est pair et 1 s'il est impair, incrémente "pairs" ou "impairs".

Revenons à notre exemple d'origine, avec une boucle **for** nous écrivons :

```
void main() {
      short nombre = 1
      for ( ; nombre != 0 ; ) {
            printf("Tapez votre nombre (0 pour sortir) : ");
            scanf("%hd", &nombre);
            printf("Votre nombre est %hd\n", nombre);
      }
}
```

ici nous n'avons aucune opération d'initialisation ou d'incrémentation à réaliser, seule l'expression de test est alors définie.

Pour illustrer la polyvalence de la boucle **for**, remarquons que l'expression :

```
for  ( expression 1 ; .expression 2 ; expression 3 ) {
      instructions;
      ...
}
```

est équivalente à l'expression :

```
expression 1;
while  ( expression 2 ) {
      instructions;
      ...
      expression 3;
}
```

L'instruction "continue"

Nous avons vu tout à l'heure que l'instruction "break" quittait la boucle en cours d'exécution; si en fait nous souhaitions de pas appliquer un traitement pour certaines valeurs mais autoriser la poursuite de la boucle, nous pouvons utiliser l'instruction **continue** pour forcer à évaluer l'itération suivante sans quitter la boucle (dans l'exemple présenté ici, nous pouvions certes utiliser un test), exemple :

Code 9 : Boucle "for" avec "break" et "continue"

```
#include <stdio.h>

/* Saisie de 10 nombres maximum avec écho de ce nombre sur la console    */
/* si le nombre saisi est supérieur à 10, il n'est pas affiché      */
/* si le nombre saisi est supérieur à 100, la boucle est interrompue*/

void main() {
      short compteur;
      short nombre = 1;
      for ( compteur = 0; compteur < 10 ; compteur++ ) {
            printf("Tapez votre nombre (0 pour sortir) : ");
            scanf("%hd", &nombre);

            if (nombre > 100)
                  break;     /* finit le programme pour une valeur > 100 */
            if (nombre > 10)
                  continue;  /* finit cette itération pour une valeur > 10 */

            printf("Votre nombre est %hd\n", nombre);

      /*    l'instruction "continue" nous mène ici, "compteur++" va être
            exécuté, le test "compteur < 10" sera évalué et le cas échéant
            l'itération suivante sera exécutée.  */
      }
/*    l'instruction "break" nous mène ici */
}
```

Les fonctions

Les fonctions sont omniprésentes en C; elles présentent le moyen de diviser des taches complexes en opérations plus simples et, avec le temps, les bibliothèques de fonctions que vous aurez écrites constitueront votre "capital-codage" qui vous permettra de construire de nouveaux programmes sans passer votre temps à tout reconstruire à chaque fois.

Pour gérer efficacement vos bibliothèques de fonctions, nous vous conseillons de maintenir des fichiers pas trop volumineux où les fonctions sont regroupées par thème.

Pour construire une bibliothèque, vous devrez créer deux fichiers:

1. Un fichier d'en-tête (avec l'extension ".h") sera utilisé pour stocker les déclarations - ou prototypes - des fonctions.
2. Un fichier de définition (ayant le même nom mais utilisant l'extension ".c" s'il s'agit de C pur et ".cpp" s'il s'agit de C++) contiendra l'implémentation de ces fonctions.

Cette organisation assurera, à vos bibliothèques, une compatibilité maximale avec le plus grand nombre de compilateurs.

Déclaration

La déclaration générique d'une fonction est :

```
type_resultat   nom_de_fonction ( type_parametre [nom_parametre] );
```

1. le type_resultat est un type fondamental, un type utilisateur ou un type pointeur sur l'un de ces types. S'il s'agit d'un type utilisateur il doit être connu - c'est à dire avoir été déclaré - au moment de la définition de la fonction. La valeur **void** précise que la fonction ne renvoie rien (c'est alors une procédure au sens pascalien du terme); ce "type_resultat" définit le "type de la fonction".

2. le type des paramètres reçus est quelconque mais doit également être connu au moment de la définition; on ne peut pas transmettre **void** mais **void*** est valide et décrit un pointeur universel.

3. le nom des paramètres est facultatif dans la déclaration de la fonction : vous pouvez vous contenter de fournir uniquement leur type. De plus le nom que vous utiliserez votre décrire ce paramètre dans la définition de la fonction (son codage) peut être différent du nom que vous utiliseriez ici. Une bonne règle est de choisir un nom décrivant au mieux ce que représente ce paramètre.

Le fichier de déclaration que vous allez créer sera inclus dans tous les fichiers de codages (les ".c" ou ".cpp") qui auront besoin d'appeler ces fonctions; pour éviter qu'un fichier ne soit inclus plusieurs fois - par suite d'inclusion en cascade de fichiers utilisant les mêmes définitions - ce qui pourrait provoquer des erreurs de redéfinition de symboles, nous utilisons une définition de constantes décrivant ce fichier.

Pour cela nous utiliserons "#ifndef" ("if not defined") afin de tester si un symbole - une directive - est défini, "#define" pour définir un symbole et "#endif" pour finir le "#ifndef".

Remarque: il existe aussi "#ifdef" pour tester l'existence d'une directive.

Grâce à ces "instructions" nommées pragma, nous pourrons en quelque sorte signer notre fichier de définition en définissant une constante dont l'existence ne dure que le temps de la compilation d'un fichier, l'habitude veut que cette signature soit le nom du fichier en majuscules avec un double souligné au début et à la fin de ce nom.

Nous souhaitons définir une bibliothèque de fonctions géométriques, pour cela nous allons créer deux fichiers “geometry.h” et “geometry.c”; voici le fichier de définition :

Code 10 : Fichier de déclaration : “geometry.h”

```
/* Déclaration de la bibliothèque géométrique  */

#ifndef __GEOMETRY__
#define __GEOMETRY__

/* Définition de constantes     */
/* on utilise: #define symbole_a_définir valeur     */
/* la ligne ne comporte pas de point-virgule        */

#define    CERCLE     1
#define    CARRE      2


/* Définition de types utilisateurs  */
/* on utilise: typedef type_connu type_a_définir    */
/* la ligne est complétée par un point-virgule      */
/* les types utilisateurs finissent souvent par _t   */

typedef unsigned short int forme_t;
/* forme_t est maintenant un synonyme de unsigned short int    */

/* Définition de fonctions*/
/* const signifie que le paramètre est constant */

float surface_cercle( const float rayon );
float surface_carre ( const float cote  );
float surface_forme ( const float taille, forme_t forme );

#endif /* __GEOMETRIE__ */
```

Définition

La définition générique d’une fonction est :

```
type_resultat   nom_de_fonction ( type_parametre [nom_parametre] )
{
     déclarations de variables
     instructions
     [return <expression de type "type_resultat">]
}
```

1. le prototype de la fonction doit être identique à celui déclaré dans le fichier de déclaration.
2. le nom des variables paramètres est facultatif si vous n’utilisez pas ce paramètre dans la routine, de plus ce nom (uniquement le nom) peut être différent de celui utilisé lors de la définition.
3. l’instruction “return quelque chose” est obligatoire si la fonction n’est pas de type void.

Notre fichier de définitions va implémenter toutes nos fonctions, afin de connaître les déclarations faites dans le fichier .h nous inclurons celui-ci.

Nous pouvons également inclure d’autres fichiers de définitions si cela est nécessaire.

Le fichier doit définir toutes les fonctions déclarées, dans le cas contraire il demeurera parfaitement compilable mais si un programme utilise ce fichier il risque de ne pas pouvoir être construit du fait du manque de nos fonctions; dans un tel cas, il est préférable de ne pas inclure les fonctions non encore codées dans le fichier déclaration; elles y seront ajoutées lorsque leur codage sera réalisé.

Voici notre fichier définition pour notre librairie géométrique :

Code 11 : Fichier de définition : "geometry.c"

```
/* Déclaration de la bibliothèque géométrique  */

#ifndef __GEOMETRY__
#include "geometry.h"
/* les fichiers utilisateurs sont précisés entre guillemets    */
#endif

#include <math.h>
/* les fichiers de système sont précisés entre symboles < et > */

/* Définition des fonctions non exportées */

/* calcul du carré d'un nombre */
inline float sqr(const float x)
/*    inline signifie que la fonction ne sera pas réellement appelée
      mais que son code sera recopiée là où la fonction est appelée,
      cela accélère le traitement pour des petites fonctions */
{
      return ( x * x );
}

/* Définition des fonctions publiques */

float surface_cercle( const float rayon )
{
/*    M_PI est défini dans math.h */
      return M_PI * sqr( rayon );
}

float surface_carre( const float cote  )
{
      return sqr( cote );
}

float surface_forme( const float taille, forme_t forme )
{
      switch ( forme ){
      case CERCLE:
            return surface_cercle( taille );
      case CARRE:
            return surface_carre( taille );
      default:
            return 0.0;
      }
}
```

Ces deux fichiers définissent notre bibliothèque. Le fichier "geometry.h" sera inclus par tout fichier ayant besoin de ses fonctions, tandis que "geometry.c" devra être inclus dans la liste des fichiers du projet. C'est à dire présent sur la ligne de commande lors de l'appel au compilateur GNU ou inséré dans votre projet si vous utilisez un gestionnaire de projets.

Librairie

Nous pouvons simplifier un peu cela en construisant une véritable librairie.

Pour cela, nous compilons (une fois pour toute - jusqu'à sa prochaine modification) le fichier "geometry.c" en utilisant l'option "-c" de GNU gcc afin de demander seulement la compilation du fichier (et non la création d'un exécutable) :

```
> gcc -c geometry.c -o geometry.o
```

ceci créé un fichier "geometry.o"; l'option "-o geometry" utilisée pour lancer gcc est optionnelle selon l'environnement; nous convertissons maintenant ce fichier binaire au format "objet" en une librairie :

```
> ar -rc geometry.a geometry.o
```

Sur système unix et Linux, vous devrez peut être exécuter en plus :

```
> ranlib geometry.a
```

Ceci a créé notre librairie, voyons comment l'utiliser, pour cela nous créons un petit programme - ce sera un programme et non plus une bibliothèque car il contiendra une fonction **main** - ce fichier doit inclure "geometry.h" et utiliser "geometry.a" durant sa construction.

Voici le programme, enregistrez-le sous "test-geo.c" :

Code 12 : Utilisation de la librairie : "geometry.a"

```
/* Utilisation de la bibliothèque géométrique  */

#include <stdio.h>
#include "geometry.h"

void main()
{
      float carre  = surface_forme ( 10.0, CARRE );
      float cercle = surface_forme ( 5.0, CERCLE );
      float rapport= cercle / carre;

      printf( "un cercle inscrit dans un carre "
            "occupe %g %% de sa surface\n", rapport );
}
```

Notez que la différenciation selon la casse permet de définir une variable "carre" qui est bien un symbole différent de la constante "CARRE" définie dans "geometry.h".

Nous compilons ce programme en incluant notre librairie dans la liste des fichiers :

```
> gcc test-geo.c geometry.a -o test-geo.exe
```

Nous pouvons alors lancer ce programme et nous obtenons :

```
> un cercle inscrit dans un carre occupe 0.7854 % de sa surface
```

Notez que notre programme aurait pu appeler directement les fonctions "surface_cercle" et "surface_carre" puisque celles-ci sont exportées par le fichier de déclaration. Par contre la fonction "sqr", implémentée dans "geometry.c", mais non définie dans l'en-tête reste cachée et ne peut être utilisée qu'à l'intérieur de "geometry.c".

La fonction “main”

La fonction principale “main” est obligatoire pour créer un programme. Elle doit être unique et ce nom (ainsi que la casse) est imposé.

Code de retour

La valeur retournée par la fonction main est soit de type **int** soit de type **void.**

En fait main retourne toujours une valeur car le système qui a lancé votre programme attend cette valeur de compte rendu d’exécution; si vous avez déclaré **void**, la valeur retournée est imprévisible ou dépend de votre compilateur.

Sous Dos vous pouvez tester la valeur retournée avec la variable Dos “errorlevel”, en voici un exemple :

Code 13 : Valeur retournée par la fonction “main”

```
int main()
{
     return 1;
}
```

Enregistrez ce fichier sous “testmain.c” et compilez-le pour créer “testmain.exe”.

Nous utilisons ensuite ce programme dans un fichier batch :

Code 14 : Lecture du code retourné par la fonction “main”

```
@echo off

testmain
if errorlevel 2 goto option2
if errorlevel 1 goto option1
if errorlevel 0 goto option0

:option0
echo l'option 0 est atteinte
goto fin

:option1
echo l'option 1 est atteinte
goto fin

:option2
echo l'option 2 est atteinte
goto fin

:fin
```

Enregistrez ce fichier sous “testIt.bat” et lancez-le, nous obtenons :

```
> l'option 1 est atteinte
```

Cette fonctionnalité permet de faire des programmes de type menu ou de construire des systèmes de traitement qui exécutent leurs étapes via plusieurs programmes séparés. On testera alors le code de sortie de chaque programme avant de lancer le suivant; la valeur zéro est généralement utilisée pour signifier une exécution correcte.

Remarque: ce principe existe également sur unix et Linux, reportez-vous à la documentation du shell utilisé pour savoir comment utiliser ce code retour.

Paramètres reçus

La fonction main accepte plusieurs interfaces (prototypes) permettant ou non de recevoir des informations depuis l'interpréteur (shell).

Les deux interfaces suivantes sont reconnues par la plupart des compilateurs :

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

avec un code de retour de type **int** ou **void**.

La première version nous est familière; la seconde permet, en déclarant une variable de type **int** et un tableau de chaînes de caractères (**char* []**), de récupérer les arguments passés au programme via la ligne de commande.

Une troisième version, disponible sur certains systèmes, permet également de récupérer les variables d'environnement de ce système :

```
main(int argc, char* argv[], char* env[])
```

Voici un exemple de lecture des différents paramètres :

Code 15 : Paramètres reçus par la fonction "main"

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

void main(int argc, char* argv[], char* env[])
{
     int i;

     printf("Nombre de parametres recus : %d\n", argc);

     printf("\nLes arguments de la ligne de commande sont:\n");
     for (i = 0; i < argc; i++)
          printf("argument[%d] = %s\n", i, argv[i]);

     printf("\nLes variables d'environnements sont:\n");
     for (i = 0; env[i] != NULL; i++)
          printf("variable[%d] = %s\n", i, env[i]);
}
```

Enregistrez ce programme sous "testarg.c" et compilez-le pour créer "testarg.exe"; nous pouvons alors tester la lecture des arguments; notez avant cela que le caractère espace utilisé sur la ligne de commande sert à séparer les différents arguments; si vous désirez transmettre une phrase incluant des espaces, vous devez la quoter (avec des simples quotes - apostrophes - ou des doubles quotes - guillemets) :

```
> testarg premier "plusieurs mots" 3 4
```

affichera :

```
Nombre de parametres recus : 5

Les arguments de la ligne de commande sont:
argument[0] = c:/src/cours/cpp/testarg.exe
argument[1] = premier
argument[2] = plusieurs mots
argument[3] = 3
argument[4] = 4

Les variables d'environnements sont:
variable[0] = DIRCMD=/OGN /L
variable[1] = TMP=C:\SYS\SYSTEME\TEMP
...
variable[8] = GNU_PATH=c:\dev\gnu
etc....
```

La sortie des données

La fonction “printf”

Nous avons déjà utilisé la fonction “printf()”, sa définition réelle (dans “stdio.h”) est :

```
int printf(const char* format[, argument, ...]);
```

Cela signifie que cette fonction attend un moins un paramètre de type “**const char***” (le modificateur “**const**” sert ici à préciser que le paramètre est traité comme constant, il ne pourra pas être modifié par la fonction); après ce paramètre obligatoire on peut ajouter un nombre arbitraire d’autres paramètres, ils correspondent à des valeurs (un nombre, une chaîne, une constante, ...) ou des variables dont on souhaite afficher la valeur.

La valeur retournée par la fonction est le nombre de caractères affichés si l’opération se déroule correctement, et la constante “EOF” (généralement -1) en cas d’échec.

Le premier paramètre, la chaîne de caractère, sert à préciser la nature des valeurs constituant les autres paramètres (le type de ces variables ou le type selon lequel traiter ces valeurs) ainsi que le format à utiliser pour afficher ces valeurs; cette chaîne inclura également le texte devant être affiché en plus des variables.

Le nombre de valeurs affichées dépend du nombre de descriptions présents dans cette chaîne de formatage et non du nombre de paramètres fournis; si vous fournissez plus de valeurs qu’il n’y a des descriptions, les valeurs supplémentaires seront ignorées, si vous fournissez plus de descriptions qu’il n’y a de valeurs le résultat est imprévisible (généralement des valeurs aléatoires seront affichées).

Le format générique d’une description de valeur est le suivant (les arguments entre crochets sont optionnels) :

```
% [modificateur] [largeur] [.précision] [N|F|h|l|L] caractère_de_type
```

Définition des données

le “caractère de type” sert à définir le type de la donnée à afficher, en voici la liste :

valeur	**signification**
d *ou* i	valeur entière (décimale) signée
u	valeur entière (décimale) non signée
o	valeur entière octale non signée (basée sur une arithmétique n’utilisant que les chiffres de 0 à 7)
x	valeur entière hexadécimale non signée (basée sur une arithmétique utilisant les chiffres de 0 à 9 et les lettres ‘a’ à ‘f’)
X	valeur entière hexadécimale non signée (basée sur une arithmétique utilisant les chiffres de 0 à 9 et les lettres ‘A’ à ‘F’)
f	nombre réel affiché sous la forme [+/-] xxxx.xxxx
e	nombre réel affiché sous la forme [+/-] x.xxxx e[+/-]xxx
g	laisse la fonction choisir entre le format “f” et “e” selon la valeur réelle
E	nombre réel affiché sous la forme [+/-] x.xxxx E[+/-]xxx
G	idem format “g” avec le caractère “E” pour l’exposant du format “e”
c	un caractère unique (variable de type **char** ou caractère entre apostrophes)
s	une chaîne de caractère (variable **char*** ou chaîne entre guillemets)
p	un pointeur sous la forme segment:offset ou offset seul selon la spécification de taille “N” ou “F”

Le format d'une description de valeur commençant par le symbole "%", vous devez utiliser le caractère de type "%" pour afficher le symbole pour-cent lui-même, voici un exemple :

```
printf( "Un tiers est 33.33 %%" );
```

ceci affichera bien :

```
Un tiers est 33.33 %
```

Voici d'autres exemples avec différents caractères de types :

```
printf( "%d", 12 );
```

affiche: 12

```
printf( "%d %d", 12, 34 );
```

affiche: 12 [espace] 34

```
printf( "%d\t%d\n", 12, 34 );
```

affiche: 12 [tabulation] 34 [retour ligne]

```
int  a = 12, b = 34;
printf( "%d %d", a, b );
```

affiche: 12 [espace] 34

```
int  a = 12, b = 34;
printf( "%d plus %d font %d", a, b, a + b );
```

affiche: 12 plus 34 font 46

```
int  a = 62;
printf("%d vaut %x en hexa", a, a);
```

affiche: 62 vaut 3e en hexa

```
int   un = 1, deux = 2;
char* eyes[2] = {"oeil", "yeux"};
printf("%d %s, %d %s", un, eyes[un-1], deux, eyes[deux-1]);
```

affiche: 1 oeil, 2 yeux

N'oubliez pas que les tableaux commencent à l'indice zéro.

Modificateur de description

valeur	signification
-	la valeur sera alignée à gauche et complétée à droite par des espaces; par défaut, la valeur est alignée à droite en insérant des zéros ou des espaces avant la donnée; l'option largeur doit être utilisée avec ce modificateur
+	les types signés seront précédés du caractère "+" ou "-"
(espace)	les valeurs positives seront précédées d'un espace et non de "+"
#	l'argument "caractère_de_type" subira une interprétation différente, ainsi : avec "d", "i", "u", "c" et "s" aucune modification d'interprétation n'est réalisée avec "o" le caractère "0" (zéro) est affiché au début du nombre avec "x" et "X" les nombres sont affichés avec le préfixe "0x" ou "0X" avec "e", "E" et "f" le point décimal est toujours affiché, habituellement il l'est uniquement s'il existe au moins un chiffre significatif (non nul) avec "g" et "G" le point décimal sera également ajouté ainsi que des zéros

Voici des exemples d'utilisation de modificateurs :

Le code :

```
int  a = 12, b = -34;
printf( "<%+d>\n<%+d>", a, b );
```

affiche:

```
<+12>
<-34>
```

Le code :

```
int  a = 12, b = -34;
printf( "<%+u>\n<%+u>", a, b );
```

affiche:

```
<12>
<4294967262>
```

Le modificateur "+" a été ignoré car il ne s'applique que sur des types signés; or, "u" provoque une interprétation non signé. De même "b" a été interprété comme non-signé, il est ici exprimé comme un **unsigned long int** puisque **int** est équivalent à **short int** sur système 16 bits et **long int** sur système 32 bits.

```
int  a = 12, b = -34;
printf( "<% d>\n<% d>", a, b );
```

affiche:

```
< 12>
<-34>
```

Le code :

```
int  a = 62;
printf("%d vaut %#X en hexa et %#o en octal", a, a, a);
```

affiche:

```
62 vaut 0X3E en hexa et 076 en octal
```

Largeur d'affichage

L'option largeur permet de fixer le nombre minimum de caractères à afficher, si le nombre précisé commence par un zéro, les données, si elles sont alignés à droite, sont complétées par des zéros sinon des espaces sont insérés. La largeur peut également être fournie comme parmi la liste des variables en utilisant ici un astérisque "*" (cf plus loin).

Le code :

```
int  a = 1, b = -12, c = 123;
printf( "<%5d>\n<%5d>\n<%5d>", a, b, c );
```

affiche:

```
<    1>
<  -12>
<  123>
```

Le code :

```
int  a = 1, b = -12, c = 123;
printf( "<%-5d>\n<%-5d>\n<%-5d>", a, b, c );
```

affiche:

```
<1    >
<-12  >
<123  >
```

Précision d'affichage

L'option précision, toujours précédée d'un point, permet de fixer le nombre maximum de caractères à afficher.

La précision par défaut est de 1 pour les types entiers décimaux (d, i et u), octal (o) et hexadécimal (x et X), de 6 pour les types réels exposants (e et E) et flottant (f), tous les chiffres significatifs pour les formats g et G, tous les caractères (jusqu'au caractère nul) pour les chaînes (s) enfin il est sans effet pour un caractère.

Vous pouvez préciser :

.0 (zéro) pour supprimer le point décimal affiché avec les formats e, E et f

.n où n est un nombre, pour afficher n décimales ou les n premiers caractères d'une chaîne

* comme pour la spécification de la largeur, la précision figurera, sous forme d'entier, parmi les arguments et sera placée juste avant la donnée utilisant le format en cours de description; dans le cas où la largeur est aussi fournie en paramètre, on fournira la largeur, la précision puis la donnée à afficher

Le code :

```
char* question = "2 + 2 font 4 !";
printf("%.10s ?\n", question);
printf("%s\n", question);
```

affiche:

```
2 + 2 font ?
2 + 2 font 4 !
```

Le code :

Code 16 : Affichage formaté de nombres décimaux

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

void main() {
     int   larg, prec;
     char* blanc = "        ";

     printf("largeur");
     for (larg = 8; larg <= 10; larg++) {
     /*   affichage centré sur 2 colonnes parmi 'larg' */
          prec = (larg - 2) / 2;
          printf(" | %.*s%2d%.*s",
               prec, blanc, larg, prec + larg % 2, blanc);
     }
     printf(" |\n");

     for (prec = 3; prec < 7; prec++) {
          printf("prec: %d", prec);
     /*   affichage aligné à gauche sur 'larg' colonnes
          et 'prec' chiffres significatifs */
          for (larg = 8; larg <= 10; larg++)
               printf(" | %-*.*f", larg, prec, M_PI);
          printf(" |\n");
     }
}
```

affiche:

```
largeur |     8    |     9     |     10     |
prec: 3 | 3.142    | 3.142     | 3.142      |
prec: 4 | 3.1416   | 3.1416    | 3.1416     |
prec: 5 | 3.14159  | 3.14159   | 3.14159    |
prec: 6 | 3.141593 | 3.141593  | 3.141593   |
```

Modificateur de taille

Ce modificateur optionnel est choisi parmi: N, F, h, l (L minuscule) ou L

valeur	signification
N	l'argument pointeur (de type %p ou %s) est un pointeur proche (NEAR).
F	interprète l'argument pointeur comme un pointeur long (FAR). par défaut les pointeurs correspondent au modèle mémoire utilisé et sont donc des pointeurs longs avec GNU qui génèrent du code 32 bits.
h	uniquement avec les caractères de type d, i et u, il spécifie que la donnée est un entier court, soit **short int**
l	signifie qu'il s'agit d'un type long, soit **long int** si le caractère de type spécifie un entier (d, i, u, o, x ou X) ou un **double** si le caractère de type spécifie un réel (e, E, f, F, g, G) sans ce modificateur les réels sont assimilés à **float**
L	signifie un double long, c'est à dire un entier **long long int** ou un réel **long double** selon le caractère de type

Bien qu'optionnel, le modificateur de taille sera très souvent utilisé car sans lui les variables autres que **int** et **float** pourraient être mal interprétées, ou leur interprétation présumerait du compilateur ou de l'environnement.

Les fonctions "?printf"

Ils existent des variantes de la fonction printf(), toutes fonctionnent sur le même principe que printf() mais ont des particularités que nous allons étudier.

La fonction fprintf() :

```
int fprintf(FILE* stream, const char* format[, argument, ...]);
```

permet d'écrire dans un fichier; nous utilisons pour cela un descripteur de fichier de type FILE*, ce type et les fichiers sont étudiés plus loin.

Notez que stdio.h définit une variable "stdout" de type FILE* qui est connectée à la sortie standard, c'est à dire habituellement l'écran. Vous pouvez donc utiliser fprintf() à la place de printf(), en précisant "stdout" comme premier paramètre pour afficher à l'écran ou un vrai descripteur de fichier pour enregistrer sur disque; cela vous permet de réutiliser une même routine de sortie indépendamment de la destination réelle.

La fonction sprintf() :

```
int sprintf(char* s, const char* format[, argument, ...]);
```

permet d'écrire la chaîne formatée dans une chaîne passée en premier paramètre; ce tampon doit être assez grand pour contenir l'ensemble des données formatées car il n'existe aucun moyen de préciser le nombre maximum de caractères à y stocker.

Exemple d'utilisation : (n'oubliez pas d'utiliser "geometry.a" avec gcc)

Code 17 : Stockage de données formatées dans une chaîne

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

void main() {
     char buffer[256];
     sprintf(buffer, "Un cercle de rayon %g a pour surface %f * %g^2, "
          "soit %f\n", 1.5, M_PI, 1.5, surface_cercle(1.5));
     printf("%s\n", buffer); /* pour vérifier */
}
```

affiche (et donc stocke dans buffer) :

```
Un cercle de rayon 1.5 a pour surface 3.141593 * 1.5^2, soit 7.068583
```

La saisie des données

La fonction “scanf”

La saisie directe de données est réalisée grâce à la fonction “scanf()” (définie dans “stdio.h”) :

```
int scanf(const char* format[, adresse, ...]);
```

Cette fonction attend un paramètre de type **const char*** (il ne pourra pas être modifié par la fonction); après ce paramètre obligatoire on ajoutera un nombre arbitraire d’arguments; ces arguments sont obligatoirement des pointeurs donnant les adresses des variables dans lesquelles une valeur doit être stockée.

La valeur retournée par la fonction est le nombre de données effectivement lues, éventuellement converties et stockées dans nos variables-arguments. Si aucune donnée n’est stockée, la valeur retournée est zéro; si la fonction rencontre une fin de fichier (variante fscanf()) ou la fin d’une chaîne (variante sscanf()) la constante “EOF” est retournée (généralement définie comme -1).

Le premier paramètre de la fonction scanf(), une chaîne de caractère, sert à préciser la nature des données à lire.

Le nombre de valeurs lues et/ou stockées dépend du nombre de descriptions présentes dans cette chaîne de formatage et non du nombre d’adresses fournies; en effet si vous fournissez plus de pointeurs qu’il n’y a de descriptions, la valeur des variables pointées supplémentaires ne sera pas définie; si vous stockez plus de données que vous ne fournissez de pointeurs, le résultat est imprévisible et dangereux (risque de blocage sévère), en effet la fonction va écrire des valeurs à des adresses mémoires indéfinies.

En résumé, soyez sûrs de fournir une description de format pour chaque donnée à lire et un pointeur pour chaque donnée à stocker; ce nombre peut être différent; en effet nous pouvons demander que des données soient lues mais que celles-ci ne soient pas stockées.

Le format générique d’une description de données est le suivant :

`% [*] [largeur] [F|N] [h|l|L]` caractère_de_type

les arguments entre crochets sont optionnels, détaillons ces arguments :

Définition des données

le “caractère de type” sert à définir le type de la donnée à lire, en voici la liste :

valeur	**signification**
d	pointeur **int*** soit l’adresse d’un entier signé
D	pointeur **long*** soit l’adresse d’un entier long signé (équivalent à “ld”)
u	pointeur **unsigned int*** soit l’adresse d’un entier non signé
U	pointeur **unsigned long*** soit l’adresse d’un entier long non signé
o	pointeur **int***; la valeur lue doit être une valeur octale
O	pointeur **long***; la valeur lue doit être octale
x	pointeur **int***, la valeur lue est hexadécimale et écrite sous la forme 0x????
X	pointeur **int***, la valeur lue est hexadécimale et écrite sous la forme 0X????
i	pointeur **int***, la valeur lue peut être décimale, octale ou hexadécimale
I	pointeur **long***, la valeur lue peut être décimale, octale ou hexadécimale

valeur	signification
e	pointeur **float***, soit l'adresse d'un réel court, la valeur à lire doit être écrite sous la forme [+/-]x.xxxe[+/-]xxx
E	pointeur **float***, la valeur doit être écrite sous la forme [+/-]x.xxxE[+/-]xxx
f	pointeur **float***, la valeur doit être codée sous la forme [+/-]xxx.xxx
g	pointeur **float***, la valeur doit être codée sous la forme [+/-]x.xxxe[+/-]xxx ou [+/-]x.xxxE[+/-]xxx ou [+/-]xxx.xxx
G	pointeur **float***, la valeur doit être codée sous la forme [+/-]x.xxxe[+/-]xxx ou [+/-]x.xxxE[+/-]xxx ou [+/-]xxx.xxx
c	pointeur **char***, par défaut l'interprétation faite est celle de l'adresse d'un caractère (unique) et non pas l'adresse d'un tableau de caractères.
s	pointeur **char***, l'interprétation faite est celle de l'adresse d'une chaîne de caractères; la taille n'est pas vérifiée, vous devez donc fournir un tableau de caractères suffisamment large pour contenir tout le texte à lire
p	pointeur sur type pointeur, soit l'adresse d'un pointeur; une adresse sera lue
n	pointeur **int***, la valeur à lire est écrite sous forme de caractères, le nombre de caractères stockés dépend de la taille réelle de la variable (cf modificateur de taille) **ce type de saisie n'est pas supporté par toutes les implémentations versions de stdio**
%	le caractère "%" lui-même est stocké

Exemple d'utilisation :

Code 18 : Saisie avec écho

```
#include <stdio.h>

void main() {
      char       aChar;
      char       aStr[32 + 1]; /* +1 pour stocker le zéro ascii */
      short      int16;
      long       int32;

      printf("Tapez une lettre        : ");
      scanf ("%c", &aChar);
      printf("votre lettre            : %c\n", aChar);

      printf("Tapez une chaine (32 car. max.) :");
      scanf ("%s", aStr);
      printf("votre chaine            : %s\n", aStr);

      printf("Tapez un entier signe   : ");
      scanf ("%hd", &int16);
      printf("votre entier            : %hd\n", int16);

      printf("Tapez un long non signe : ");
      scanf ("%U", &int32);
      printf("votre nombre            : %lu\n", int32);
}
```

Suppression d'affectation

le caractère "*", dit caractère de suppression d'affectation, permet qu'une donnée lue ne soit pas stockée; ceci est particulièrement intéressant dans le cas où vous n'utilisez qu'une partie des données reçues, aucun pointeur n'est fourni pour une telle donnée. Voir le code 19 - ci-après - pour un exemple d'utilisation.

Largeur de lecture

L'option largeur permet de fixer le nombre maximum de caractères à lire.

Utilisable avec un pointeur sur caractères, cette option permet de contrôler le nombre de caractères transférés dans la chaîne argument.

Cette option ne garantit pas que ce nombre de caractères sera lu; en effet le buffer de saisie peut contenir moins de caractères; mais il se peut également qu'il y ait plus de caractères lus que la valeur demandée. En effet, en cas de caractères espace ou non convertibles, des caractères supplémentaires seront lus. Par contre ces caractères non pris en compte ne sont pas écrits dans votre tampon.

Cette caractéristique a deux usages courants :

1. tout d'abord cela permet de contrôler le nombre de caractères lus vis à vis de la taille du buffer fourni. Ainsi la saisie de texte ci-dessus aurait du être codée :

```
char aStr[32 + 1]; /* +1 pour stocker le zéro ascii */
printf("Tapez une chaine (32 car. max.) :");
scanf ("%32s", aStr);
printf("votre chaine            : %s\n", aStr);
```

2. le fait que le caractère espace ne soit pas pris en compte est également utile pour saisir un caractère qui, justement, doit être différent de espace et imprimable; si nous codons :

```
char aChar;
printf("Tapez une lettre        : ");
scanf ("%c", &aChar);
printf("votre lettre            : %c\n", aChar);
```

   l'utilisateur peut saisir un espace ou taper directement un retour ligne; si par contre nous codons :

```
char aChar;
printf("Tapez une lettre        : ");
scanf ("%1s", &aChar);
printf("votre lettre            : %c\n", aChar);
```

   nous sommes sûrs que aChar sera un caractère valide (et différent d'espace).

Modificateurs de taille

Nous pouvons utiliser deux paramètres de spécification de taille.

Le premier N ou F redéfinit la taille du pointeur reçu en argument; avec N il est considéré comme un pointeur proche (near pointer), avec F il est utilisé comme un pointeur lointain (far pointer).

Le second modificateur redéfinit la taille de la variable pointée :

valeur	**signification**
h	signifie qu'il s'agit d'un pointeur sur un type court, applicable uniquement avec le type **int**; il spécifie donc un pointeur sur un **short**
l	signifie qu'il s'agit d'un pointeur sur un type long, soit **long int*** si le caractère de type spécifie un entier, soit un **double*** s'il s'agit d'un réel
L	signifie qu'il s'agit d'un pointeur sur un double long, c'est à dire un **long long*** ou un **long double*** selon le caractère de type.

Les fonctions “?scanf”

A l’identique de ce que nous avons vu pour les variantes de printf() il existe :

```
int fscanf(FILE* stream, const char* format, ...);
```

qui utilise un premier paramètre supplémentaire de type “FILE” (un descripteur de fichier) pour y lire des données.

Et une fonction :

```
int sscanf(const char* s, const char* format, ...);
```

qui lit des données, non plus issues de la console (le clavier), mais dans un tableau de caractères que nous fournissons en premier paramètre supplémentaire.

Exemple d’utilisation :

Code 19 : Lecture de données formatées depuis une chaîne

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

void main() {
     char       buffer[256];
     float      rayon;
     float      surface;
     double     pi_arrondi;

/* création de la chaine contenant les données */

     sprintf(buffer, "Un cercle de rayon %g a pour surface %f * %g^2, "
          "soit %f\n", 1.5, M_PI, 1.5, surface_cercle(1.5));

/* lecture des données de la chaine */

     sscanf(buffer, "Un cercle de rayon %f a pour surface %lf * %*f^2, "
          "soit %f\n", &rayon, &pi_arrondi, &surface);

/* affichage des données relues */

     printf("Le rayon est   : %g\n", rayon);
     printf("La surface est : %g\n", surface);
     printf("avec pi valant : %lg\n", pi_arrondi);
}
```

affiche :

```
Le rayon est   : 1.5
La surface est : 7.06858
avec pi valant : 3.14159
```

Remarque: nous lisons et stockons le rayon (“%f”), la valeur de pi (“%lf”), lisons mais ignorons la répétition du rayon (“%*f”) et enfin nous lisons et stockons la surface.

Les chaînes de caractères

Un nombre signé ou non

Nous avons annoncé que le type caractère peut être considéré comme un nombre. Pour illustrer cette équivalence, voici un code qui affiche les caractères dont le code ASCII est compris entre 32 (l'espace) et 127 (non compris) :

Code 20 : Table ASCII restreinte

```
#include <stdio.h>

void main() {
     char  c;
     char* separateur[2] = {"\t", "\n"};
     for (c = ' '; c < 127; c++)
          printf("%hd = '%c' %s",
               c, c, separateur[(c - 31) % 5 == 0]);
}
```

Explications :

1. dans l'instruction "for" nous initialisons "c" avec une lettre et testons sa valeur par rapport à un nombre
2. le caractère "c" est affiché comme un entier (format "%hd") et comme la lettre qu'il représente (format "%c")
3. le test sur le reste de la division entière de (c - 31) par 5 renvoie 0 ou 1 qui est utilisé comme indice du tableau de chaînes afin d'insérer un retour à la ligne ("\n") ou une marque de tabulation ("\t").

Pourquoi ne pas afficher le caractère 127 en réalisant une boucle qui testerait "c <= 127" (ou "c < 128") ?

Voila un cas où la distinction signé, non signé du type **char** pourrait produire des effets indésirables. Supposons que le compilateur traite le type **char** comme **signed char** (c'est le cas de GNU C/C++) : dans un tel cas le test "c <= 127" serait toujours vérifié, puisque le compilateur effectue un test signé entre un nombre prenant les valeurs comprises entre -128 et +127 et le nombre 127, et le programme partirait dans une boucle infini. En effet après avoir imprimé le caractère 127, l'instruction "c++" affecterait -128 à "c" qui vérifie bien "c <= 127", imprimerait ce caractère -128 (?!) puis -127, -126 etc. jusqu'à +127 ... et on repart pour un tour ... indéfiniment.

Pour imprimer tous les caractères imprimables du jeu de caractères ASCII étendu, sans risquer la mauvaise surprise, on coderait :

Code 21 : Table ASCII complète

```
#include <stdio.h>

void main() {
     unsigned char c;
     char* separateur[2] = {"   ", "\n"};
     for (c = ' '; c < 255; c++)
          printf("%3hd = '%c' %s",
               c, c, separateur[(c - 31) % 6 == 0]);
}
```

ici encore on ne cherchera pas à imprimer le caractère de code ASCII 255 (qui d'ailleurs n'est pas imprimable) sous peine de calculer 255 + 1 soit 0 et de produire une autre boucle infinie.

Remarque : cette dernière remarque ne signifie nullement que l'on ne puisse pas coder des boucles s'exécutant sur tout le domaine de définition d'un nombre; il conviendrait juste d'écrire la boucle autrement, par exemple :

Code 22 : Table ASCII vraiment complète

```
#include <stdio.h>

void main() {
     unsigned char c, x;
     char* separateur[2] = {"\t", "\n"};

     for (c = 0; ; c++) {
          x = ((c > 6 && c < 11) || c == 13 || c == 26) ? ' ' : c;
          printf("%3hd = '%c' %s",
               c, x, separateur[(c + 1) % 6 == 0]);
          if (c == 255)
               break;          /* sort de la boucle */
     }
}
```

Les tableaux de caractères

Les chaînes de caractères du C/C++ sont des pointeurs ! Nous l'avons déjà dit et répété mais ceci est peut être la première source d'erreur lorsque l'on débute en C/C++, donc insistons.

Une chaîne statique est créée et initialisée de la façon suivante :

```
char* chaine = "mon texte entre guillemets";
```

ou

```
char chaine[] = "mon texte entre guillemets";
```

dans les deux cas, le compilateur réserve la place nécessaire pour stocker les caractères présents **et** un caractère de valeur nulle inséré à la fin de la chaîne; et affecte à la variable "chaine" l'adresse mémoire du premier caractère.

Toutes les opérations sur les chaînes que vous utilisez (le cas échéant) habituellement en Pascal ou Basic sont ici non valides; par exemple si sont définis :

```
char* str1 = "premiere chaine";
char* str2 = "second message";
```

vous ne pouvez pas définir le contenu de "str1" avec celui de "str2" en utilisant l'opérateur d'affectation:

```
str1 = str2;
```

ceci fait pointer "str1" sur l'emplacement mémoire où est stocké "second message"; "str1" et "str2" deviennent donc deux pointeurs équivalents (pointant sur la même donnée).

De même, des affectations hors initialisations modifient l'adresse de ce pointeur, mais pas le contenu de la chaîne pointée, exemple :

```
str1 = "autre chaine";
```

provoque la réservation d'une nouvelle zone mémoire par le compilateur durant la phase de compilation pour y stocker les caractères de la chaîne et, lors de l'exécution, le pointeur "str1" est changé pour désigner cet autre emplacement.

Ceci montre clairement que nous ne pouvons pas utiliser une telle syntaxe pour modifier le contenu de chaînes utilisées, par exemple, comme paramètres d'une routine ayant pour fonction de définir ce contenu.

Voici un exemple pour illustrer cela :

```
#include <stdio.h>

void main() {
     char* str1 = "premiere chaine";
     char* str2 = "second message";

     printf("ptr\tcontenu\n");
     printf("%p\t%s\n", str1, str1);

     str1 = str2;
     printf("%p\t%s\n", str1, str1);

     str1 = "autre chaine";
     printf("%p\t%s\n", str1, str1);
}
```

l'exécution donne (les adresses de la première colonne peuvent être différentes) :

```
ptr  contenu
1550 premiere chaine
1560 second message
1583 autre chaîne
```

où on observe que la variable "str1" change de valeur, c'est à dire pointe sur un emplacement différent.

Vous pourriez penser que tout cela n'a pas trop d'importance puisque la chaîne affichée est bien celle que nous attendions; tant que nous manipulons des chaînes statiques cela semble en effet ne pas être gênant; cependant, comme nous l'avons déjà dit, si "str1" bien que statique est supposée pointer toujours sur le même bloc de caractères dont nous cherchons à modifier le contenu, cela ne fonctionne plus; un problème plus grave va apparaître avec les chaînes dynamiques - dont l'espace mémoire utilisé pour stocker les caractères est alloué durant l'exécution et non réservé durant la compilation. Voyons cela.

Une chaîne est créé dynamiquement en utilisant "malloc( )" :

```
char* aStr = (char*) malloc(32);              /* alloue 32 octets  */
```

Notre chaîne dynamique est créée, nous utilisons alors l'opérateur d'affectation comme dans l'exemple précédent :

```
aStr = "autre chaine";
```

ceci a modifié l'adresse décrite par "aStr" ... comment allons-nous maintenant libérer la zone de 32 octets précédemment allouée ? ... impossible ! et à force votre code finira en "out of memory".

Comment alors fixer le contenu d'une chaîne avec une autre chaîne ? nous devons recopier un à un tous les caractères de la nouvelle chaîne dans le bloc mémoire représentant la première.

Voici un codage possible d'une routine strcpy (String Copy) qui réaliserait une telle tâche :

```
char* strcpy(char* destination, const char* source)
{
     int i;
     for (i = 0; source[ i ] != 0; i++)
          destination[ i ] = source[ i ]; /* recopie un caractère   */
     destination[ i ] = 0;                /* marque la fin          */
     return destination;                  /* retourne ce pointeur   */
}
```

cette routine est un peu sale : on ne vérifie pas que "source" et "destination" existe et on suppose que le bloc pointé par "destination" contient suffisamment de place pour stocker tous les caractères de "source".

De plus cette routine peut être largement optimisée, les accès aux i-ième éléments des chaînes sont coûteux en cpu; une première amélioration consisterait à utiliser l'arithmétique des pointeurs : en effet puisque "source" est un pointeur sur caractère, "*source" est le premier caractère et "*(source + 1)" le second, etc.

Ainsi nous pouvons remplacer l'accès par indice par une déréférence d'un pointeur parcourant la chaîne, comme nous ne pouvons modifier le pointeur "source" puisqu'il est constant, et que nous voulons conserver la valeur de "destination" pour la retourner à la fin du traitement, nous créons deux variables locales de type **char*** et nous balayons ensuite ces pointeurs :

```
char* strcpy(char* destination, const char* source)
{
     char* sour = (char*) source;    /* conversion explicite en char* */
     char* dest = destination;

     for (; *sour; sour++, dest++)
          *dest = *sour;             /* recopie un caractère          */
     *dest = 0                       /* insertion du zéro terminal    */
     return destination;
}
```

Remarques :

1. le test de sortie "*sour" est équivalent à "*sour != 0" soit "est-ce que le caractère dont l'adresse est stocké dans "sour" est différent de zéro".

2. l'instruction d'incrémentation utilise l'opérateur virgule de liaison d'instruction pour réaliser les deux actions : incrémenter le pointeur "sour" et le pointeur "dest".

Vous avez suivi ? alors allons y franchement, vous vous rappelez que l'opérateur "++" placé à gauche d'une variable incrémente son contenu avant son évaluation; pour tirer le meilleur parti du cache du micro processeur nous coderons, afin d'évaluer les termes qu'une seule fois par itération :

```
char* strcpy(char* destination, const char* source)
{
     char* sour = (char*) source - 1;     /* conversion en char*     */
     char* dest = destination - 1;        /* pointeur local         */

     while (*++dest = *++sour)
          ;                               /* instruction nulle      */
     return destination;
}
```

Remarques: les pointeurs locaux sont initialisés avec l'octet précédent l'adresse réelle puisque nous utilisons "++" à gauche; ainsi à la première évaluation les pointeurs seront incrémentés d'un octet et désigneront donc bien le premier caractère.

Sachez que le compilateur optimise une partie du code pour vous et vous n'aurez pas habituellement à chercher à écrire des expressions le plus incompréhensibles possibles.

Cependant :

1. évitez les accès par l'opérateur crochet et préférez l'arithmétique des pointeurs

2. cette recherche de l'expression cryptique est un jeu qui a fait la réputation du C et qui vous permettra aussi de vous familiariser avec le C et de le domestiquer.

Enfin, les librairies fournies avec un compilateur définissent les fonctions utiles aux traitement des chaînes (dont strcpy); vous n'aurez donc pas à écrire toutes ces fonctions.

Les fonctions standards de manipulation de chaînes

Les fonctions suivantes sont définies dans “string.h”.

Comparaison

```
int  strcmp(const char* s1, const char* s2);
```

retourne une valeur négative si s1 < s2, zéro si les chaînes sont identiques, une valeur positive si s1> s2; la comparaison se fait caractère par caractère sur les codes ASCII des caractères.

```
int  stricmp(const char* s1, const char* s2);
```

retourne une valeur négative si s1 < s2, zéro si les chaînes sont identiques, une valeur positive si s1> s2; la comparaison se fait caractères par caractères sans tenir compte de la casse (‘A’ est égal à ‘a’).

```
int  strncmp(const char* s1, const char* s2, size_t n);
```

traitement identique à “strcmp” mais portant que sur les “n” premiers caractères.

Rappel : le type “**size_t”** est défini comme **unsigned long int**.

```
int  strnicmp(const char* s1, const char* s2, size_t n);
```

traitement identique à “stricmp” mais portant que sur les “n” premiers caractères.

Concaténation

```
char*strcat(char* s1, const char* s2);
```

retourne la chaîne formée de s1 et s2; réalloue s1 pour stocker les caractères de s1 et ceux de s2

équivalent Pascal : `s1 := Concat(s1, s2);` *ou* `s1 := s1 + s2;`

```
char*strncat(char* s1, const char* s2, size_t n);
```

retourne la chaîne formée de s1 et d’au maximum “n” caractères de s2; réalloue s1 pour stocker les caractères de s1 et les “n” premiers caractères de s2 (ou la taille réelle de s2).

équivalent Pascal : `s1 := s1 + Copy( s2, 1 n );`

Copie

```
char*strcpy(char* s1, const char* s2);
```

recopie le contenu de “s2” dans le bloc pointé par “s1”; “s1” doit avoir une taille suffisante pour l’opération

équivalent Pascal : `s1 := s2;`

```
char *     strncpy(char* s1, const char* s2, size_t n);
```

recopie au maximum les “n” premiers caractères de “s2” dans le bloc pointé par “s1”; “s1” doit avoir une taille suffisante pour stocker “n” caractères; “s1” peut ne pas être nul terminé si “s2” contient “n” ou plus caractères.

quasi-équivalent Pascal : `s1 := Copy( s2, 1, n );`

```
char*strdup(const char* s);
```

retourne une copie de la chaîne “s”; la routine alloue l’espace nécessaire, l’utilisateur doit libérer ce bloc quand il n’en a plus besoin.

Longueur

```
size_t     strlen(const char* s);
```

retourne le nombre de caractères dans “s” (le caractère nul n’est pas compté)

équivalent Pascal : Length(s)

Recherche

```
char*strchr(const char* s, int c);
```

retourne un pointeur sur la première occurrence du caractère “c” dans “s”; la chaîne retournée est un sous-ensemble de “s”, rien n’est alloué.

équivalent Pascal : `Copy( s, Pos( Chr( c ), s ), Length( s ));`

```
char*strrchr(const char* s, int c);
```

retourne un pointeur sur la dernière occurrence du caractère “c” dans “s”; la chaîne retournée est un sous-ensemble de “s”, rien n’est alloué.

```
size_t     strcspn(const char* s1, const char* s2);
```

retourne le nombre de caractères, pris depuis le début de “s1” ne contenant aucun caractère de “s2”

```
size_t     strspn(const char* s1, const char* s2);
```

retourne une sous chaîne de “s1” formée uniquement par les caractères de “s2”.

```
char*strpbrk(const char* s1, const char* s2);
```

retourne un pointeur sur la première occurrence d’un des caractères de “s2” dans “s1”; la chaîne retournée est un sous-ensemble de “s1”, rien n’est alloué.

équivalent Pascal : `Copy( s, Pos( Chr( c ), s ), Length( s ));`

```
char*strstr(const char* s1, const char* s2);
```

retourne une sous chaîne de “s1” identique à “s2”

Les Fichiers

Les entrées sorties réalisées en C/C++ sur des fichiers ne sont pas très différents de celles réalisées avec la console (c'est à dire le clavier en entrée, l'écran, en mode texte, en sortie).

En effet lorsque nous travaillons avec la console, nous utilisons des périphériques automatiquement pris en compte par le système; c'est à dire que ce système a ouvert les accès en entrée ou sortie, insère ou récupère les données, etc.; pour travailler avec des fichiers nous allons avoir besoin de gérer nous mêmes ces opérations, mais le travail sur le fichier va ensuite être quasi identique à l'utilisation de la console.

Le type "FILE"

Le type "FILE", déclaré dans le fichier "stdio.h", permet de déclarer une référence à un fichier (un flux de données); les fonctions utilisant des références à une variable de ce type sont indépendantes du système; nous manipulerons donc un fichier de la même façon quel que soit le système de gestion implémenté par le système lui-même; le revers de cette transparence est que certaines fonctions ne seront pas disponibles sur certains systèmes (par exemple sous window il est impossible de définir des droits d'accès à un fichier comme cela se fait sous unix).

La première fonction qui va nous intéresser est fopen(); son prototype est :

```
FILE* fopen(const char* nom_du_fichier, const char* mode_d_ouverture);
```

Le second paramètre est une chaîne de caractères qui contiendra tout d'abord une des options suivantes :

"r" ouvre le fichier en mode de lecture, il est donc interdit d'y écrire et le fichier doit exister.

"w" ouvre le fichier en écriture, tout fichier de même nom existant sera écrasé; il est évidemment impossible d'y lire des données puisque le fichier est toujours vide après l'appel de la fonction.

"a" ouvre le fichier en mode ajout ou en mode écriture si aucun fichier de ce nom n'existe; toute écriture se fait à la fin du fichier.

"r+" ouvre un fichier en mode lecture, donc le fichier doit exister et la position dans le fichier est fixée au début de ce fichier, mais il est autorisé d'y écrire. Si l'écriture est réalisé à la fin du fichier, les données seront ajoutées. Si par contre les données sont écrites à partir d'une position différente de la fin, toutes les anciennes données situées entre cette position et la fin sont perdues. En résumé il n'existe aucun moyen d'insérer directement des données dans un fichier.

"w+" ouvre un nouveau fichier en écriture. Ce fichier remplace, comme avec l'option "w", tout autre fichier de même nom. Cependant il sera possible de relire les données du fichier en se repositionnant dans le fichier; cette dernière opération sera nécessaire car évidemment lorsque nous utilisons un fichier en écriture, la position courante après l'ouverture du fichier est toujours la fin de ce fichier.

"a+" ouvre le fichier, ou crée un nouveau fichier, en mode ajout. L'usage est identique que lors de l'utilisation de l'option "r+", à ceci près que la position courante est la fin du fichier au retour de la fonction "fopen".

A chacune de ces options, nous pouvons ajouter, à la fin de la chaîne, le caractère "b" ou "t"; le caractère "t" assignera un comportement "texte" au fichier, tandis que "b" provoquera un mode "binaire".

Cette distinction ne sert qu'à savoir comment sont enregistrés les symboles de fin de ligne dans le fichier; si vous n'inscrivez que des nombres (par exemple quatre octets pour un "**long**") cette option ne vous concerne pas et vous n'êtes pas tenu de préciser "t" ou "b"; si vous enregistrez des fichiers contenant du texte mais que le format de ces fichiers est un format propre à votre application, précisez "b" afin qu'aucune interprétation de ces symboles de fin de ligne ne soit réalisée. Vous retrouverez ainsi le contenu exact de ce que vous avez inscrit, y compris pour une application destinée à s'exécuter sur différentes plates-formes où le symbole n'est pas toujours le même. Si enfin vous enregistrez des fichiers texte qui doivent être relus par n'importe quel éditeur sur n'importe quelle plate-forme, précisez "t".

Cette option (texte ou binaire) est optionnelle car le système a toujours un choix par défaut, désigné par la variable globale déclarée par "fcntl.h" dont le nom est "_fmode"; la valeur par défaut de cette variable est généralement de considérer les fichiers en mode texte, vous pouvez dans votre programme modifier sa valeur pour l'adapter à votre utilisation la plus courante.

Les valeurs autorisées pour cette variable sont les constantes "O_TEXT" et "O_BINARY" définies également dans le fichier "fcntl.h".

Désignation du fichier

Nous avons défini le second paramètre de la fonction fopen(), et le premier alors ?

Vous l'aviez deviné, ce premier paramètre désigne le nom du fichier à créer ou à ouvrir.

Ce nom peut désigner uniquement un nom de fichier ou un nom précédé d'un chemin relatif ou absolu; la chaîne fournie doit bien sûr respecter les règles du système, notamment pour la syntaxe d'un chemin d'accès. Ainsi sous système Unix, le séparateur "/" doit être utilisé entre chaque nom de répertoire. Sur Macintosh, on utilisera le signe ":". Enfin avec DOS/Windows, le séparateur étant "\", on utilisera "\\" car le caractère simple "\" sert à définir des caractères spéciaux tandis que le double anti-slash "\\" désigne le caractère "\" lui-même. La validité du nom porte également sur le nombre de caractères maximal autorisé pour le nom des répertoires et fichiers ainsi que les caractères autorisées dans ce nom.

Enfin, selon le compilateur, vous pourrez ou non profiter des possibilités de codage des noms. Ainsi si vous souhaitez créer des noms de fichiers longs sous Windows 95 ou NT, vous devrez utiliser un compilateur comme MetroWerks CodeWarrior ou Microsoft Visual C++. Le compilateur GNU ne sachant pas gérer de tels noms tronquera le nom fourni à ses huit premiers caractères pour le nom et trois premiers pour l'extension.

Vous avons donc vu que la fonction fopen() sert tout à la fois à créer un nouveau fichier, relire un ancien fichier, en remplacer un, ajouter des données à un fichier existant.

Ajoutons que la fonction fclose() sert à fermer le fichier passé comme unique argument et nous sommes prêts à créer notre premier fichier; nous allons tout d'abord demander un nom de fichier à l'utilisateur puis nous remplirons le fichier texte ouvert (après sa création ou le remplacement d'un ancien fichier) avec les chaînes de caractères saisies par l'utilisateur; nous nous arrêterons quand nous recevrons une chaîne vide (c'est à dire un simple retour chariot).

Ecriture de fichier

Code 23 : Ecriture de fichier

```
/* Saisie de texte et recopie dans un fichier */

#include <stdio.h>

int main()
{
     char buffer[256];          /* tampon pour la saisie */
     FILE* sortie = NULL; /* périphérique de sortie */

     printf("Nom du fichier a créer : ");

     if (gets(buffer) == NULL || *buffer == 0)
     {    /* pas de nom saisi, arrêtes le programme */
          printf("\nVous deviez saisir un nom pour continuer\n");
          return -1;
     }

/*   Création du fichier avec le nom fourni en mode écriture et texte */
     sortie = fopen(buffer, "wt");

/*   testons si le fichier a pu être créé */
     if (sortie == NULL){
          printf("\nErreur durant la création du fichier %s\n", buffer);
          return -1;
     }

/*   un peu d'aide pour l'utilisateur
     (le compilateur colle les morceaux de chaines tout seul) */
     printf("\nLe fichier a été correctement ouvert,\n"
     "Tapez des lignes de texte qui seront stockées dans ce fichier\n"
     "Pour terminer tapez simplement un retour chariot\n\n");

/*   Saisie de lignes de texte et insertion à la fin du fichier */
     while (gets(buffer) != NULL && *buffer != 0)
     /*   gets(char*) retourne un pointeur sur le buffer en cas de succès
          ce pointeur désigne le caractère zéro si la chaîne est vide */
     {
     /*   nous connaissons déjà "printf" qui écrit sur la console
          "fprintf" écrit dans le fichier précisé en premier paramètre
          les autres paramètres sont identiques à ceux de "printf" */
          fprintf(sortie, "%s\n", buffer);
     }

/*   ferme le fichier */
     fclose(sortie);

     return 0;
}
```

Remarques :

```
if (gets(buffer) == NULL || *buffer == 0)
```

est équivalent à:

```
char* dummy = gets(buffer);
if ( (dummy == NULL) || (dummy[0] == 0) )
```

c'est à dire que nous testons si le pointeur retourné par "gets" est non nul et si le premier caractère du bloc de caractère retourné est différente du caractère zéro de fin de chaîne; ici "gets" retourne le même pointeur que celui transmis comme paramètre, ceci nous dispense d'une variable intermédiaire; avec d'autres fonctions retournant un pointeur différent (un nouveau bloc alloué par exemple) nous nous pourrions pas utiliser un tel code.

Lecture de fichier

En utilisant “fprint()” et “fscanf()” en lieu et place de “prinf()” et “scanf()” nous pouvons travailler sur les fichiers de la même manière qu’avec la console. De même lorsque nous travaillons avec la console, nous pouvons utiliser toutes les fonctions prévues pour les fichiers en utilisant les flux prédéfinis “stdin” pour la saisie et “stdout” pour les sorties.

Voici un autre exemple qui relit un fichier, et recopie son contenu sur la console ou dans un fichier. Notez comment un même code peut s’adapter à différents besoins.

Code 24 : Lecture de fichier

```
/* Lecture d'un fichier et recopie dans un fichier ou sur la console */

#include <stdio.h>

int main()
{
	char buffer[256];		/* tampon pour la lecture */
	char* dummy  = buffer;	/* fgets attends un char* */
	FILE* entree = NULL;	/* périphérique d'entree  */
	FILE* sortie = NULL;	/* périphérique de sortie */

	printf("Nom du fichier a lire : ");

	if (gets(buffer) == NULL || *buffer == 0)
	{	/* pas de nom saisi, arrêtes le programme */
		printf("\nVous deviez saisir un nom pour continuer\n");
		return -1;
	}

/*	Ouverture du fichier avec le nom fourni en mode lecture et texte
/*	et test sur le succès de l'ouverture */
	if ((entree = fopen(buffer, "rt")) == NULL)
	{
		printf("\nErreur durant l'ouverture du fichier %s\n", buffer);
		return -1;
	}

	printf("\nTapez un nom de fichier à créer pour recopier ce fichier\n"
		"ou simplement un retour chariot pour afficher le contenu\n\n");

	if (gets(buffer) == NULL || *buffer == 0)
	{	/* pas de nom saisi, utilises la console */
		sortie = stdout;
	}
	/*	Création du fichier en mode écriture et test de succès */
	else if ((sortie = fopen(buffer, "wt")) == NULL)
	{
		printf("\nErreur durant la création du fichier %s\n", buffer);
		fclose(entree);		/* n'oublions pas de faire le ménage */
		return -1;
	}

/*	Lecture de la source et recopie */
	while (fgets(dummy, 255, entree) != NULL && *dummy != 0)
	/*	fgets(char*, size n, FILE*) retourne un pointeur sur le buffer
		passé en premier paramètre après avoir lu au plus n caractères
		depuis le flux passé en troisième paramètre */
	{
		fprintf(sortie, "%s", dummy); /* le symbole "\n" a été lu par
fgets */
	}

/*	ferme les fichiers */
	fclose(entree);
	fclose(sortie);		/* la fermeture de stdout sera ignoré */

	return 0;
}
```

Les fonctions génériques d'entrée sortie

Nous avons noté que les fonctions fichiers peuvent être utilisées avec la console grâce aux flux C ANSI; cette similitude existe également pour les fonctions d'entrée sortie. En effet, pour la plupart il existe une version fichier de la routine console dont le comportement est identique; dans quasiment tous les cas le nom de la routine travaillant sur un fichier aura pour nom celui de la fonction console préfixé du caractère "f" et recevra un premier paramètre supplémentaire de type FILE*.

Nous trouvons ainsi les fonctions :

```
int   fgetc(FILE* flux);
```

qui retourne un caractère lu depuis "flux" en cas de succès ou la valeur EOF en cas d'erreur. La position courante est incrémentée d'une position; "fgetc(stdin)" lit un caractère saisi au clavier.

```
char*fgets(char* s, int n, FILE* flux);
```

qui remplit le buffer "s" avec au plus "n" caractères lus depuis "flux" et **insère un caractère zéro** après ces "n" caractères (c'est pour cela que nous avons passé la valeur 255 alors que notre buffer avait une taille de 256 caractères); la fonction retourne "s" en cas de succès et la valeur NULL si la fin du flux a été atteinte; "fgets(s, n, stdin)" lira une chaîne au clavier (tout caractère jusqu'à l'appui de la touche "Entrée")

```
int   fputc(int c, FILE* flux);
```

qui insère le caractère "c" dans le flux et retourne ce caractère en cas de succès ou la valeur EOL en cas d'erreur; "fputc(c, stdout)" affiche le caractère sur la console.

```
int   fputs(const char* s, FILE* flux);
```

qui injecte la chaîne "s" dans le flux (qui peut être "stdout") et renvoie une valeur positive ou nulle en cas de succès (cela peut être le nombre de caractères effectivement injectés) et EOF en cas d'erreur.

La fonction n'ajoute pas automatiquement de symbole de retour à la ligne ("\n").

Les fonctions orientées fichiers

Les fonctions suivantes s'appliquent sur des flux correspondant réellement à des fichiers :

```
int   feof(FILE* flux);
```

retourne zéro si la fin du fichier n'a pas été atteinte et une valeur différente de zéro dans le cas contraire.

Remarque : une telle définition de la valeur retournée peut paraître un peu floue, mais c'est la norme ANSI définie pour cette fonction et selon la librairie utilisée la valeur réellement reçue peut être différente; il est donc impératif de ne se baser que sur une valeur nulle pour écrire un code portable. Ainsi nous aurions pu écrire dans notre second exemple :

```
/*   Lecture de la source et recopie */
while (!feof(entree))
{
     if (fgets(dummy, 255, entree) != NULL)    /* lecture de la source   */
          fputs(dummy, sortie);                /* écriture en sortie     */
}
```

ce code serait meilleur pour une recopie de fichier car il ne s'arrête pas sur la première ligne vide que contiendrait la source (en effet, nous ne souhaiterions pas qu'il s'arrête).

Et partant de la définition de “fgets” qui doit renvoyer non NULL tant que la fin du flux n’a pas été atteinte, nous pourrions supprimer le test redondant et coder directement :

```
while (!feof(entree))
     fputs(fgets(dummy, 255, entree), sortie);
```

ceci étant, méfiez-vous du codage réalisé dans les librairies dites “standards” et utilisez, dans les opérations sensibles, la ceinture **et** les bretelles afin de détecter toutes erreurs.

La position courante dans le flux peut être manipulée grâce à :

```
int  fgetpos(FILE* flux, fpos_t* pos);
```

qui stocke dans notre variable “pos” la position courante et renvoie 0 en cas de succès et une valeur différente de zéro en cas d’erreur. La valeur stockée dans “pos” est dite opaque. On ne cherchera pas à l’interpréter et ne sera utiliser qu’avec la fonction :

```
int  fsetpos(FILE* flux, const fpos_t* pos);
```

qui restaurera la position courante avec les informations contenues dans “pos” et retourne 0 ou différent de 0, respectivement en cas de succès ou d’erreur.

Voici un exemple d’utilisation :

```
fpos_t une_position;
...
/* sauvegarde de la position */
if (fgetpos(flux, &une_position) == 0) {
     /* traitement */
     ....
     /* restauration de la position */
     if (fsetpos(flux, &une_position) != 0) {
     /*   la situation est bancale ... pas moyen de restaurer
          l'état initial, prévoir une sortie de secours !!! */
     }
}
```

Un autre moyen de fixer la position est de le faire autoritairement en donnant une distance par rapport à la position courante, au début du fichier ou à la fin de ce fichier; on utilise par cela :

```
int  fseek(FILE* flux, long offset, int mode);
```

à qui on donne le flux sur lequel opérer, le décalage et une constante parmi :

- SEEK_SET pour préciser un décalage par rapport au début du fichier
- SEEK_CUR pour un décalage par rapport à la position courante
- SEEK_END pour un décalage par rapport à la fin du fichier

“fseek” devrait retourner zéro en cas de succès et non zéro dans le cas contraire; malheureusement sous DOS cette valeur de retour n’est pas toujours fiable car le système peut réinitialiser la position en cours.

Nous pouvons également sauvegarder la position courante, par rapport au début du fichier grâce à :

```
long ftell(FILE* flux);
```

qui renvoie la taille en octets ou -1 en cas d’erreur.

Enfin :

```
int  fflush(FILE* flux);
```

force l’écriture des données dans le fichier en vidant le tampon associé,

```
void rewind(FILE* flux);
```

rembobine le fichier en fixant la position courante au début du fichier.

Conclusion

Maintenant que vous connaissez “tout” du C, je vous propose un codage possible pour un carnet d’adresses; étudiez cet exemple et inspirez-vous en pour réaliser des petits programmes du même genre.

Code 25 : Carnet d’adresses

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

/*   Definition d'une structure */

struct StructureAdresse {
     char Nom[30];
     char Prenom[30];
     char Adresse[50];
     long CodePostal;
     char Ville[20];
};

/*   le typedef suivant permet d'utiliser directement
       le type 'TAdresse' a la place de 'struct StructureAdresse' */

typedef struct StructureAdresse TAdresse;

/*   nom du fichier de donnees en constante globale */

const char* gFileName = "Adress.dta";

/*   verification de l'existence d'un fichier */

int ExisteFichier(const char* aFileName)
{
     FILE* dummy = fopen(aFileName, "rt");
     if (dummy != NULL)
     {
          fclose(dummy);
          return 1;
     }
     else
          return 0;
}

/*   saisie d'une ligne de texte via un tampon surdimensionne */

size_t SaisieTexte(char* container, short maxSize)
{
     char      dummy[256];     /* tampon de saisie */
     size_t    size;

/*   verification des parametres */
     if (container == NULL)
          return 0;

/*   le resultat est une chaine vide par defaut */
     *container = 0;
     if (maxSize <= 0)         /*   rien ne pourra etre recopie */
          return 0;

/*   saisie */
     gets(dummy);

/*   taille de la saisie */
     size = strlen(dummy);
     if (size > maxSize - 1)
          size = maxSize - 1;
```

```
/*    recopie des donnees saisies */
      memcpy(container, dummy, size);
      container[size] = 0;
      return size;
}

/*    saisie d'un nombre entier long
      utilise prealablement une saisie en texte puis retourne le nombre
      converti ou 0 si la saisie est vide ou si une erreur se produit

      long atol(const char*) convertit une chaine en long int */

long SaisieLong()
{
      char dummy[32];      /* tampon de saisie */
      if (SaisieTexte(dummy, 32) > 0)
            return atol(dummy);
      else
            return 0;
}

void Affiche(TAdresse* adresse)
{
      printf("%s%s%s%5lu %s\n",
            adresse->Nom, adresse->Prenom,
            adresse->Adresse,
            adresse->CodePostal, adresse->Ville);
}

TAdresse* Relit(TAdresse* adresse, FILE* source)
{
      char dummy[9];

      if (fgets(adresse->Nom, 29, source) == NULL)
            return NULL;

      fgets(adresse->Prenom, 29, source);
      fgets(adresse->Adresse, 49, source);
      fgets(dummy, 8, source);
      adresse->CodePostal = atol(dummy);
      fgets(adresse->Ville, 19, source);

      return adresse;
}

void Ajoute()
{
      TAdresse   adresse;
      FILE*      calepin;

      if ((calepin = fopen(gFileName, "a+t")) == NULL) return;

      printf("\nSaisie d'une nouvelle fiche :\n");

      printf("Nom     (30 car.) : ");  SaisieTexte(adresse.Nom, 30);
      printf("Prenom  (30 car.) : ");  SaisieTexte(adresse.Prenom, 30);
      printf("Adresse (50 car.) : ");  SaisieTexte(adresse.Adresse, 50);
      printf("Code postal       : ");  adresse.CodePostal = SaisieLong();
      printf("Ville   (20 car.) : ");  SaisieTexte(adresse.Ville, 20);

/*    ecriture de adresse dans le fichier */
      fprintf(calepin, "%s\n%s\n%s\n%lu\n%s\n",
            adresse.Nom, adresse.Prenom, adresse.Adresse,
            adresse.CodePostal, adresse.Ville);
      fclose(calepin);
}
```

```
void Liste()
{
     TAdresse   adresse;
     FILE*      calepin;
     long       compteur = 0;

     if ((calepin = fopen(gFileName, "rt")) == NULL) return;

     printf("\nContenu du fichier :\n");
     while (!feof(calepin))
     {
          if (Relit(&adresse, calepin) != NULL)
          {
               printf("fiche %5lu\n----------\n", ++compteur);
               Affiche(&adresse);
          }
     }
     fclose(calepin);
}

void Cherche()
{
     TAdresse   adresse;
     FILE*      calepin;
     char       cherche[30];
     size_t     nbDeCarac;
     long       compteur = 0;

     if ((calepin = fopen(gFileName, "rt")) == NULL) return;

     printf("\nIndiquez le nom (ou le debut du nom) a chercher : ");
     nbDeCarac = SaisieTexte(cherche, 30);
     if (nbDeCarac > 0)
          while (!feof(calepin))
          {
               if (Relit(&adresse, calepin) != NULL)
               /* test sur le nom lu */
                    if (strnicmp(adresse.Nom, cherche, nbDeCarac) == 0)
                    /* le nom lu contient 'cherche' */
                    {
                         printf("\nfiche %5lu\n----------\n",
                              ++compteur);
                         Affiche(&adresse);
                    }
          }
     fclose(calepin);
}

short Menu()
{
     printf("\nCommandes disponibles\n\n");
     printf("1 - Ajout d'une adresse\n");

     if (ExisteFichier(gFileName))
     {
          printf("2 - Recherche d'une adresse\n");
          printf("3 - Liste des adresses\n");
     }
     printf("4 - Fin\n");
     printf("\nVotre choix : ");
     return SaisieLong();
}
```

```
void main()
{
	short choice;

	do {
		switch (choice = Menu()) {
		case 1:
			Ajoute();
			break;
		case 2:
			Cherche();
			break;
		case 3:
			Liste();
			break;
		}
	}
	while (choice != 4);
}
```