GEii
Génie Electrique
et Informatique Industrielle

UVSQ
université PARIS-SACLAY
IUT de Vélizy-Rambouillet
CAMPUS DE VÉLIZY-VILLACOUBLAY

LANGAGE C
MANUEL DE SURVIE

D. AUBERT ET M. FLEURY

2021-2022

TABLE DES MATIERES

CHAP I : Programme et langage

Un programme est un ensemble de directives nommées instructions qui spécifient les opérations élémentaires à exécuter et la manière dont elles s'enchaînent. L'ordinateur met en mémoire le programme qu'on lui fournit puis il l'exécute. Pour cela il dispose d'un répertoire limité d'opérations élémentaires qu'il sait exécuter rapidement. Pour s'exécuter, un programme a en général besoin de données (exemple calcul de la moyenne de plusieurs points, les points seront les données du programme). Lors de l'exécution l'ordinateur retourne un résultat (dans notre exemple il s'agira de la moyenne calculée).

L'échange données-résultats se fait par ce que l'on appelle la communication. Toutes les données ne sont pas forcément données par un utilisateur. Elles peuvent provenir d'un fichier. On parlera alors d'archivage. Ces données peuvent résulter de l'exécution d'un autre programme. Nous voyons donc que des résultats peuvent devenir des données et réciproquement.

I. Mise en forme de l'information (Notion de codage) :

L'ordinateur ne peut traiter des informations que sous la forme binaire (suite de 0 et de 1). Toutes les informations qui sont transmises à l'ordinateur doivent l'être sous forme binaire. De la même manière, lorsqu'il transmet une information il doit la coder de manière à ce que l'homme puisse la comprendre.

L'unité de base en informatique est donc le bit (b) qui peut prendre 2 valeurs : '0' ou '1'. Il existe des multiples classiques de cette unité. Un octet (o), ou byte en anglais (B) est constitué de 8 bits. Il est ainsi possible de coder 2^8 = 256 valeurs différentes sur un octet. Nous avons ensuite le kilo-octet (ko = 1000 o), le mega-octet (Mo = 1000 ko), le giga_octet (Go = 1000 Mo), …

Nous pouvons donc dire que l'ordinateur et l'homme codent les informations. Pour pouvoir stocker une information, il faut prévoir un nombre de bits suffisant pour coder toutes les valeurs possibles. La manière diffère et nous parlerons alors de type.

II. Comment dialoguer avec l'ordinateur :

L'ordinateur comprend un seul langage codé en binaire que l'on appelle le langage machine. La difficulté d'écriture (et de lecture) d'un programme en langage machine est heureusement détournée par l'utilisation d'un autre langage plus évolué et plus explicite que l'on appellera le langage assembleur. Ce langage est propre à chaque processeur (Intel, Motorola, Power PC...).

Par exemple en langage machine une instruction s'écrira 0101010011011010. Sa signification pourra être additionner les valeurs situées à deux «adresses» différentes (adresse : emplacement mémoire). Pour l'exprimer de manière plus parlante on pourra écrire en langage assembleur : ADD A,B. On utilise les capacités de l'ordinateur lui même pour coder le langage assembleur en langage machine. Le langage assembleur est plus lisible mais le faible nombre d'instructions dont il dispose le rende délicat d'emploi.

Pour simplifier la tâche de tout programmeur on a développé des langages plus généraux appelés langages évolués. Parmi ceux-ci citons : Fortran, Basic, Forth, ADA, Pascal, C, C++...

Une instruction écrite en langage C pourra donner ceci :
Y = A*X + B ; sa signification est assez évidente, son écriture sera similaire dans les autres langages.

Un programme écrit en langage C est constitué d'une suite d'instructions, qui seront exécutées de façon séquentielle : chaque instruction est exécutée à son tour, dans l'ordre dans lequel elle apparait dans le programme. Une fois l'instruction exécutée et le résultat stocké en mémoire, celle-ci est ensuite ignorée. Il est ainsi primordial, pour que le programme fonctionne correctement, que les instructions soient écrites correctement et dans le bon ordre.

CHAP II : Les variables, les types de données et l'instruction d'affectation

I. La notion de variable :

C'est un nom. Il sert à repérer un emplacement en mémoire. La valeur de cette variable peut évoluer au cours du temps ou durant le déroulement du programme.

Le nom donné à une variable peut être choisi avec une grande liberté. Il faut toutefois respecter quelques contraintes :

Le choix des caractères se fera obligatoirement parmi les 26 lettres MAJUSCULES, les 26 lettres minuscules, les chiffres de 0 à 9 et le caractère (_). Par conséquent les caractères accentués ne sont pas admis, ainsi que les autres séparateurs de caractères. De plus, le premier caractère ne peut pas être un chiffre.

Exemples de noms corrects : A1, Auto, Groupe_c, nombre_1_et_demi...

Exemples de noms incorrects : Groupe c (il comporte un espace), nombre-1_et_demi (il comporte -), carré (il comporte un caractère accentué).

II. Les types de données :

Trois principales familles de variables existent en C : les variables entières, réelles et pointeurs (un chapitre spécial sera consacré aux pointeurs). De plus ces variables peuvent être codées dans la machine sur 8, 16, 32 ou 64 bits :

char	nombres entiers sur 8 bits (en général réservé aux caractères ASCII)
unsigned char	Nombres entiers naturels sur 8 bits (souvent utilisé pour les registres physiques 8 bits)
int	nombres entiers relatifs sur 16 ou 32 bits (dépend du compilateur)
unsigned int	nombres entiers naturels sur 16 ou 32 bits.
short int	nombres entiers relatifs sur 16 bits.
long int	nombre entiers relatifs sur 32 bits.
unsigned short int	nombres entiers naturels (non signés (>=0)) sur 16 bits.
unsigned long int	nombres entiers naturels sur 32 bits.
float	approximation de nombres réels sur 32 bits.
double	approximation de nombres réels sur 64 bits
long double	approximation de nombres réels sur 80 bits

II.1. Le type entier : int

Suivant les valeurs susceptibles d'être stockées dans la variable, il faudra choisir un des types précédents sachant que la gamme de représentation est la suivante :

sur 8 bits	[-128, +127]	ou	[0 , 255]	(si unsigned)
sur 16 bits	[-32768, +32767]	ou	[0 , 65535]	(si unsigned)
sur 32 bits	[-2147483648, +2147483647]	ou	[0, 4294967295]	(si unsigned)

Attention : si l'extension short ou long n'est pas précisée la variable sera sur 16 ou 32 bits suivant le compilateur utilisé et le système d'exploitation (sur les compilateurs modernes int passe généralement sur 32 bits).

II.2. Le type réel : float et double

Ces nombres ont pour bornes :
sur 32 bits [$3,4*10^{-38}$, $3,4*10^{38}$]
sur 64 bits [$1,7*10^{-308}$, $1,7*10^{-308}$]
sur 80 bits [$1,1*10^{-4932}$, $1,1*10^{4932}$]

Les valeurs des types réels représentent leur précision. Elles peuvent aussi, bien évidemment être négatives.

Décimale : on écrit les nombres avec un point (il remplace notre virgule)
21.54 -0.51 -.51 2. .365
Notez que si on écrivait «2» au lieu de «2.» il serait considéré comme entier.

Exponentielle : On utilise comme mantisse n'importe quel nombre décimal et on lui ajoute un exposant (puissance de 10) sous la forme de E suivi d'un nombre.
2.51E2 25.1E1 2.52E-1 .252E-2
24.E2 2.4E3 24.0E2

De la même manière que pour les entiers, nous pouvons étendre la place mémoire réservée à la variable en utilisant le type double sur 8 octets soit 64 bits, voir 80 bits.

Exemple :

```
float Longueur, Largeur;         /* variables réelles sur 32 bits
*/
double Longueur, Largeur;        /* variables réelles sur 64 bits
*/
long double Longueur, Largeur ; /* variables réelles sur 80 bits */
```

II.3. Le type caractère : char

Ce type permet de représenter un caractère comme une lettre, un chiffre ...
Sur 8 bits [-128, 127] ou [0, 255] si non signé.

Pour bien saisir l'utilité d'un tel type voyons les exemples suivants :
Réponse à une question sous la forme O (oui) ou N (non).

Lecture d'un mot (plusieurs caractères) et comptage du nombre de lettres qu'il contient.

Le type char permet de représenter différents caractères sur un et un seul octet. Les 256 combinaisons possibles permettent de représenter bien plus que les lettres de l'alphabet. Le code généralement employé s'appelle code ASCII (American Standard Code for Information Interchange). Par exemple le 0 aura le code 48.

L'écriture d'une constante caractère se fait d'une manière bien spécifique pour éviter toute ambiguïté.
La notation est la suivante : 'a', 'b' etc... Les caractères s'inscrivent entre quotes ou apostrophes. Les caractères non imprimables possèdent eux aussi une représentation, '\n' indique, par exemple, le saut de ligne, '\b' le retour arrière. Le caractère \ (anti-slash) précède le caractère proprement dit et signifie que ce qui suit n'est pas à prendre comme un caractère mais comme un code.

III. Instruction d'affectation :

On supposera que les variables sont du type int.

Les opérateurs utilisés sont les suivants :

+	addition
–	la soustraction
*	la multiplication
/	la division (il s'agit d'une division entière. Donc 15 divisé par 2donne 7)

Pour s'affranchir des priorités entre opérateurs, il suffit d'utiliser des parenthèses.

`%` est un opérateur supplémentaire appelé " de modulo". Il correspond au reste de la division entière. Par exemple dans la division de 15 par 2, `15%2` vaut 1.

Dans l'instruction d'affectation suivante : `a = 2 ;`
on indique que la variable *a* se voit affectée la valeur 2.

Dans l'instruction suivante **:** `b = a*3 – 4 ;`
on affecte à la variable b la valeur de `a*3` à laquelle on retranche 4.

De manière générale une instruction d'affectation aura pour rôle :

- calcul de la valeur de l'expression à droite du signe égal; dans le premier exemple l'expression se réduit à une simple constante (2); dans le deuxième exemple il faut effectivement réaliser des calculs.

- rangement du résultat obtenu dans la variable à gauche du signe égal.

Important lorsqu'on détermine la valeur d'une expression on ne modifie pas les valeurs des variables qui apparaissent dans cette expression.

III.1. les expressions mixtes :

Il est possible de mélanger différents types dans une même expression, cela s'appelle alors une expression mixte.

```
int Multiplicateur ;
char  Decale ;
float Longueur;

Longueur =Multiplicateur * (Longueur + decale ) ;
```

Dans l'exemple ci-dessus, nous déclarons une variable `Multiplicateur` de type entier, une variable `Decale` de type char et une variable `Longueur` de type float.

Bien que les trois variables soient de types différents, l'expression est parfaitement valide. En fait, le compilateur va commencer par convertir les variables `Multiplicateur` et `Decale` en float afin que l'opération se fasse correctement.

Dans certains cas, il peut y avoir perte d'information pouvant entraîner des erreurs graves (exemples : échec du premier lancement de la fusée Ariane V...) :

```
Decale = Multiplicateur * Longueur ;
```

La variable `Multiplicateur` est convertie par le compilateur en variable de type float, le produit se fera correctement. Cependant, si le résultat de la multiplication n'est pas compris entre [-128 , +127] la variable `Décale` ne peut contenir le résultat correct.

III.2. Les variables non définies :

Au début d'un programme si nous déclarons une variable sans lui affecter de valeur particulière elle ne sera pas égale à zéro mais possédera une valeur quelconque, fruit des opérations antérieures. Afin d'éviter le risque d'utiliser une variable dont la valeur serait quelconque il est recommandé de l'initialiser en début de programme.

CHAP III : L'affichage et la lecture : Comment communiquer.

Dans un premier temps, nous nous limiterons à l'écriture des informations sur l'écran et la saisie se fera par l'intermédiaire du clavier.

I. Affichage avec l'instruction printf :

L'affichage des informations se fera sur l'écran grâce à l'instruction *printf*. L'écriture de cette instruction se fait sous le format suivant :

```
printf ("format", liste d'items);
```

`"format"` est un ensemble de caractères. Cet ensemble est composée de spécifications qui indiquent le type des variables que l'on va lire, ainsi que d'éventuelles informations, sous forme de texte par exemple.
`liste d'items` représente soit une variable, dans ce cas c'est la valeur de la variable qui est écrite, soit une valeur.

quelques exemples :
`printf ("Voici un exemple");` affiche Voici un exemple.
`printf ("La somme de %d et de %d est %d", a, b, c);`
affiche (avec a = 5, b = 10 et c = a + b), La somme de 5 et de 10 est 15

Les `%d` (pourcent d) indique la première variable rencontrée, ici `a`, sera de type `int`.

Les autres types rencontrés précédemment seront *:*
`%u` pour unsigned int,
`%f` pour float,
`%lf` pour double,
`%e` pour float mis sous la forme scientifique,
`%c` pour char et
`%s` (string) pour une variable de type texte (nous en reparlerons plus loin).

Pour affiner la présentation des informations nous disposons d'outils que nous appellerons gabarit, précision et changement de lignes.

Dans l'exemple suivant :
`printf ("%10.4f", x) ;`
Nous allons afficher la valeur de la variable x avec un minimum de 10 chiffres et dans tous les cas 4 chiffres après la virgule (. en C) en type float.

Pour éviter d'afficher toutes les informations du programme les unes derrière les autres sans séparations nous pourrons utiliser des séparateurs et des sauts à la ligne.

L'écriture se fera ainsi :

`printf ("%10.4f\n", x) ;` ici nous aurons un saut de ligne après l'affichage de la valeur de x (\n).

II. Ecriture d'un programme, forme générale:

Tout programme en C s'écrit sur le modèle suivant :

```
#include <stdio.h>

void main()
{
  instructions de déclaration
  instructions exécutables
}
```

Nous expliquerons la première ligne ultérieurement.

`void main()` est l'étiquette du programme principal, celle par laquelle le compilateur sait que c'est le début du programme. Les accolades `{` pour l'ouverture du programme et `}` pour la fermeture sont les limites du programme, les bornes par lesquelles commence et finit le programme.

A noter : les instructions d'affectation doivent apparaître en début.
Exemple de programme :

```
#include <stdio.h>
void main()
{
  float val = 25.14, carre, cube;
  int entier;
  carre = val*val;
  printf ("La valeur %f a pour carré %f \n et pour cube %f \n",
           val, carre, cube=val*carre);
  entier = cube;
  printf ("La partie entière de son cube est : %d", entier);
}
```

Ce programme affichera à l'écran :

```
La valeur 25.14 a pour carré 632.0196 et pour cube 15888.97274
La partie entière de son cube est : 15888
```

II.1. Mise en oeuvre du programme :

Le programme écrit sous forme de texte sur une feuille de papier devra être saisi grâce à un éditeur. Le programme écrit sous forme de fichier texte est le programme source.

Ensuite il doit être traduit à l'aide du compilateur. Le compilateur produit un programme exécutable en langage machine.

Nous pourrons ensuite exécuter ce programme en tapant simplement à l'invite du DOS, son nom.

Dans tous les cas afin que le programme soit compréhensible par soi même à la relecture ou par une tierce personne, il est nécessaire de commenter toutes les instructions. Un commentaire se trouve entre ces délimiteurs /* */, comme le montre l'exemple suivant :

```
int a, b, c ;  /* Déclaration des variables*/
```

III. Lecture des informations, instruction scanf :

Cette instruction lit des données rentrées au clavier et les range en mémoire.

Le format de `scanf` est le suivant (attention prenez l'habitude de ne rentrer qu'une seule information à la fois même s'il est théoriquement possible d'en rentrer plusieurs à la fois) :

```
scanf ( "format", liste_adresses) ;
```

`"format"` est à l'identique de `printf`, le texte d'accompagnement en moins.

`liste_adresses` est la liste des adresses des variables à lire, on doit mettre l'opérateur *&* (adresse de) devant chaque variable à lire.

Les spécifications de type sont les mêmes que pour `printf`.

La lecture de caractères est un peu particulière. Nous utiliserons `%c`, mais ce code `%c` ne saute aucun séparateur et prend le premier caractère qui se présente, même s'il s'agit d'un espace ou d'une fin de ligne. Pour forcer `scanf` à sauter les séparateurs nous placerons un caractère "espace" dans le format à l'image de l'exemple suivant :

```
scanf ("%d %c", &a, &b) ;
```

Par mesure de précaution, on se limitera à la saisie d'un seul nombre par fonction scanf(). A la forme précédente on préfèrera :

```
scanf ("%d", &a) ;
scanf ("%c", &b) ;
```

CHAP IV : Faire des choix, effectuer des répétitions : les instructions if, do...while, for.

Avec les instructions de base étudiées précédemment nous exécutions séquentiellement les instructions. C'est à dire nous exécutions les instructions dans l'ordre où elles apparaissaient dans le programme.

Pour exploiter pleinement la puissance d'un ordinateur tous les langages évolués disposent de possibilités pour effectuer des choix et des répétitions au sein d'un programme. En C ces structures de contrôle s'appellent structures de choix et structures de répétition.

I. Structure de choix : l'instruction if :

Cette instruction permet de tester une condition et d'exécuter une instruction en fonction du résultat. Elle se présente sous le format suivant :

```
if (expression)
{
  instruction 1 ;
}
else
{
  instruction 2 ;
}
```

`expression` est une expression conditionnelle. Si `expression` est vraie (non nulle) alors `instruction 1` est exécutée. Si `expression` est fausse (nulle) c'est `instruction 2` qui est exécutée.

L'instruction `else` n'est pas obligatoire. Nous pouvons ainsi avoir le format suivant :

```
if (expression)
{
  instruction 1;
}
instruction 2 ;
```

Si `expression` est vraie alors l'`instruction 1` est effectuée puis l'`instruction 2` sinon on passe directement à l'`instruction 2` sans effectuer l'`instruction 1`.

De manière générale pour éviter de devoir se limiter à une seule instruction après `if` ou/et `else`, nous les ferons suivre de groupe encadré entre des accolades comme l'exemple ci-dessous :

```
int nbre_1, nbre_2, min, max ;
if (nbre_1>= nbre_2)
{
  max = nbre_1;
  min = nbre_2 ;
```

```
}
else
{
  max = nbre_2 ;
  min = nbre_1 ;
}
```

Dans cet exemple, après avoir défini des variables de type int nous comparons deux nombres entre eux à l'aide de >= (supérieur ou égal) puis suivant le résultat nous affectons le plus grand à max et le plus petit à min.

En C les conditions sont les suivantes :

==	égal (deux fois le signe égal)
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal
!=	non égal ou différent

Nous pouvons aussi relier plusieurs conditions simples par des opérateurs logiques :

&&	et
||	ou (inclusif)
!	non

Voici quelques exemples :

`(a<b) && (c<d),`	prend la valeur vraie si les deux expressions a<b et c<d sont toutes deux vraies sinon elle prend la valeur faux.
`(a<b) || (c<d),`	prend la valeur vraie si l'une au moins des conditions est vraie.
`!(a<b)`	vraie si $a<b$ est fausse, c'est équivalent à $a>=b$.

I.1. Les choix imbriqués :

Les instructions qui se trouvent dans les blocs d'instructions qui suivent *if* ou *else* sont quelconques. Nous pouvons y trouver un ou plusieurs *if* qui se suivent ou sont imbriqués les uns dans les autres (toujours garder à l'esprit l'idée d'un programme lisible, et pour cela éviter de trop grand niveaux d'imbrications).

```
int nbre_1, nbre_2, min, max ;
if (nbre_1>= nbre_2)
{
    if (nbre_1 == 0)
     printf ("Variables nulles \n");
    else
    {
     max = nbre_1;
     min = nbre_2 ;
    }
}
```

```
else
{
    max = nbre_2 ;
    min = nbre_1 ;
}
```

A noter : `else` se rapporte toujours au `if` le plus proche, ceci pour le cas où il y aurait moins de `else` que de `if`.

I.2. Les choix multiples :

L'instruction `switch` permet d'effectuer des choix multiples. Elle se présente sous le format suivant :

```
switch (expression) {
  case valeur 1 :  instruction 1 ;
           break;
  case valeur 2 :  instruction 2 ;
           break;

  case valeur n :  instruction n ;
           break;
  default  :     instruction d ;
           break;
}
```

Où `expression` doit être de type entier, caractère ou tout autre type énumératif. Elle ne peut donc être de type réel ou string.

Instruction 1, instruction n peuvent être des instructions simples ou des blocs d'instructions.

L'instruction `switch` détermine la valeur de l'expression. Elle compare ensuite cette valeur aux différentes valeurs mentionnées après `case`.

Si cette valeur est égale à valeur 1 le programme effectue instruction 1, si cette valeur est égale à valeur n le programme effectue l'instruction n si elle n'est égale à aucune des valeurs qui suivent les différents `case` alors le programme effectue l'instruction qui suit `default`.

L'instruction `break` signifie que l'on sort de `switch` dès que l'instruction précédent `break` est effectuée (à condition que la valeur soit égale à celle suivant le `case`).

II. Structures de répétition :

Nous parlerons indifféremment de répétitions ou de boucles. Ces répétitions sont de deux sortes.

-Les instructions conditionnelles ou indéfinies, la poursuite ou l'interruption de la répétition dépend d'une certaine condition.

-Les instructions inconditionnelles ou définies ou avec compteur sont répétés un certain nombre de fois.

Le premier cas peut se faire à l'aide des instructions `do... while` et `while`.

II.1. L'instruction while :

Son format se présente comme ci-dessous :

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

`Expression` est une instruction conditionnelle qui représente la condition de maintien de la boucle.

`Instructions` est une instruction simple ou un bloc d'instructions. Le fonctionnement se traduit de la manière suivante :

On évalue l'expression :

- Si `l'expression` est vraie (non nulle) on exécute `instructions` et on retourne évaluer `l'`*`expression`*.
- Si `l'expression` est fausse, le programme poursuit son exécution à partir de l'instruction suivante.

Attention : Bien veiller à ce que l'ensemble des instructions exécutées dans la boucle puisse avoir une influence sur l'expression conditionnelle. Dans le cas contraire, la boucle ne s'arrête pas!!! Exemple : `while(1)` crée une boucle infinie.

Si `l'`*`expression`* est fausse dès le premier passage dans la boucle, `instructions` n'est pas effectuée.

II.2. L'instruction do... while :

Le format de cette instruction est le suivant :

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

Comme précédemment `instructions` peut être une instruction simple ou un bloc d'instructions.

`Expression` est, ici encore, une condition qui, si elle est vraie, permet la poursuite de la boucle.

La différence essentielle entre les instructions `while` et `do... while` est que, dans la deuxième `(do... while) instructions` est au moins exécutée une fois, même si `expression` est fausse, dés le premier passage dans la boucle, en effet `instructions` précède `while (expression).`

En général, en programmation, on parle de *répétition tant que*, lorsqu'on examine une condition de poursuite en début de boucle, c'est le cas de l'instruction `while`. Sinon on parlera de *répétition jusqu'à*, lorsqu'on examine une condition de terminaison en fin de boucle, instruction `do... while`.

La notion de compteur est une notion importante en programmation. On peut compter les tours de boucle à l'aide d'une variable entière comme le montre l'exemple ci-dessous :

```
#include <stdio.h>
void main()
{
  int nbre ;   /* Nombre fourni par l'utilisateur*/
  int compt ;  /* Compteur du nombre de valeurs traitées*/

compt = 0 ;    /* Mise à zéro du compteur ou initialisation du
      compteur*/

  do {
      printf (" Donnez un nombre entier ou 0 pour sortir:") ;
      scanf ("%d", &nbre) ;
      printf ("voici ce nombre %d, puis son carré %d\n",
           nbre, nbre*nbre) ;
      compt = compt +1 ;    /* Incrémentation du compteur*/
  } while (nbre != 0) ;

  printf ("Vous avez fourni %d nombres ", compt - 1) ;
       /* -1 pour déduire le zéro de fin*/
}
```

Note: l'instruction `compt = compt + 1;` peut aussi s'écrire `compt ++;` de la même manière si nous avions `compt = compt - 1;` nous pourrions le remplacer par `compt -- ;`

Nous voyons que le compteur que nous avons introduit pourrait nous permettre d'interrompre la boucle au bout d'un certain nombre de tours.

ainsi au lieu d'écrire : `while (nbre != 0) ;`

nous pourrions écrire : while ((nbre != 0) || (compt < 4)) ;

Ainsi la boucle s'interrompra pour le nombre 0 ou lorsque nous aurons franchi 5 fois la boucle (c'est lorsque le compteur passe à 4 que nous sortirons de la boucle).

II.3. L'instruction for :

Sa syntaxe sera la suivante :

```
for ( début ; expression ; fin_de_tour)
{
  instructions;
}
```

Où `début` est une ou plusieurs instructions simples qui seront exécutées avant le premier tour de boucle.

`Expression` est la condition de poursuite de la boucle. Elle sera examinée avant chaque tour de boucle et si elle est fausse nous sortirons de la boucle.

`Fin_de_tour` est une ou plusieurs expressions simples qui seront exécutées à la fin de chaque tour de boucle.

`Instructions` est une instruction unique ou un bloc d'instructions.

L'instruction ci-dessus est rigoureusement équivalente à celle ci-dessous :

```
début ;
```

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

Exemple d'utilisation de l'instruction for ;

```
#include <stdio.h>
void main()
{
  int nbre_val ; /* Nombre de valeurs à traiter*/
      int nbre ; /* Nombre fourni par l'utilisateur*/
  int compt ;  /* Compteur du nombre de valeurs à traiter*/

  printf ("Combien de valeurs à traiter ? ") ;
  scanf ("%d", &nbre_val) ;

  for (compt = 0; compt < nbre_val; compt ++)
  {
      printf ("Donnez un nombre numéro %d :", compt ++) ;
      scanf ("%d", &nbre) ;
      printf ("Voici son carré : %d \n", nbre*nbre) ;
  }
}
```

Exemple 2: `for(i=0,j=1,k=2 ; m<=0 ; i++, j--, k+=10)`

III. Les instructions break, continue et goto :

break : instruction utilisée pour mettre fin à une boucle ou sortir d'une instruction de sélection multiple (for, while, do-while, switch).

`break` sert à interrompre prématurément une boucle, elle sort de la boucle dès qu'on la rencontre (voir aussi son utilisation avec `switch`).

Exemple : /* traitement pour nombre <= 100 sauf si negatif */

```
do
{
  /* saisie du nombre */
  scanf (″%d″,&nombre) ;

  /* sortie si nombre negatif */
  if ( x < 0)
  {
      printf((″Erreur : le nombre doit
                etre positif \n″) ;
      break ;
  }

  /* traitement pour nombre positif */
  …..
}while (nombre <= 100) ;
```

Si `nombre` est négatif, le traitement n'est pas effectué et la boucle est interrompue.

continue : permet de suspendre l'itération en cours dans une boucle mais la boucle ne se termine pas ; utilisable dans les structures de contrôles : for, while, do-while

Cette instruction `continue` dans une boucle permet de passer prématurément au tour de boucle suivant.

EXEMPLE : /* TRAITEMENT POUR NOMBRE <= 100 SAUF SI NEGATIF */

```
do
{
  /* saisie du nombre */
  scanf ("%d",&nombre) ;

  /* sortie si nombre negatif */
  if ( x < 0)
  {
      printf(("Erreur : le nombre doit
                etre positif \n") ;
      continue ;
  }

  /* traitement pour nombre positif */
  …..
}while (nombre <= 100) ;
```

Si nombre est négatif, le traitement pour nombre positif n'est pas effectué **mais** la boucle continue.

goto : permet le branchement en un emplacement quelconque du programme. **Cette instruction ne doit pas servir dans un programme bien fait.** Sa syntaxe est la suivante :

```
goto sortie;
...
sortie : instructions;
```

CHAP V : Les variables dimensionnées ou tableaux

Dans certains programmes on doit parfois utiliser de nombreuses variables de même type. Il est long et fastidieux de leur donner un nom différent à toutes, aussi on utilise une seule variable mais à composantes multiples.

Ces variables ont pour nom variable dimensionnée ou tableau.

Chaque composante peut être représentée en faisant référence au nom de la variable suivi d'un indice mis entre crochets []. L'indice représente la position de la composante dans la variable.

Chaque tableau peut être à une ou plusieurs dimensions.

I. Tableau à une dimension :

L'idée la plus proche pour se représenter ce tableau est de le visualiser sous la forme d'un vecteur. La déclaration d'une telle variable se fait de la façon suivante :

```
Type nom [TAILLE] ;
```

`nom` correspond au nom de la variable dimensionnée, `type` est un type quelconque représentant le type des composantes de la variable et `TAILLE` le nombre de composantes que va contenir la variable.

Attention : l'indice de la première composante correspond à 0 et la dernière aura donc l'indice taille - 1. `TAILLE` ne peut pas être une variable mais une constante.

Exemple de déclaration et d'initialisation de ce tableau :

```
int table [20], compt ;
for (compt = 0 ; compt < 20 ; compt ++)
{
  table [compt] = 0 ;
}
```

Un indice peut être indifféremment une variable, une constante ou une expression.

On peut affecter des valeurs à un tableau de différentes manières, en particulier comme ci-dessous

```
  table [0] = 10 ;
  table [1] = 20 ;
  table [2] = 5 ;
  table [3] = 5 ;
  table [4] = 30 ;
  table [5] = 2 ;
```

ou encore :

```
  int table [20] = {10, 20, 5, 5, 30, 2} ;
```

Les éléments du tableau qui n'ont pas d'affectation comporteront des valeurs quelconques mais pas forcément nulles.

On peut aussi affecter des valeurs aux différents éléments du tableau en utilisant scanf comme le montre l'exemple suivant :

```
for (compt = 0 ; compt < 20 ; compt ++)
{
  scanf ("%d", &table [compt]) ;  /* Ecrit l'élément dans le tableau
*/
}
```

De la même manière nous lirons le tableau à l'aide de printf.

```
for (compt = 0 ; compt < 20 ; compt ++)
{
  printf("Voici l'élément numéro %d : %d", compt +1,table[compt]);
      /* Lit   l'élément dans le tableau*/
}
```

II. Tableau à deux ou plusieurs dimensions :

La déclaration d'une telle variable se fera de la façon suivante :

```
Type nom [taille 1] [taille 2]... [taille n] ;
```

tout élément du tableau pourra être référencé de la manière suivante :
nom [indice 1] [indice 2]... [indice n]

Le nombre total d'éléments du tableau à n dimensions est égal à : taille 1*taille 2*... *taille n.
Un tableau à plusieurs dimensions est aussi appelé matrice.

L'affectation de valeurs à un tableau à plusieurs dimensions peut se faire de différentes façons:

```
int table [3] [4] = {{1,   2,  3,  4},
           {5,  6,  7,  8},
           {9, 10, 11, 12}} ;
int table [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} ;
```

On peut bien évidemment ne pas affecter tous les éléments du tableau avec les mêmes remarques que pour les tableaux à une dimension.

III. Cas particulier : les chaînes de caractères

En C les chaînes de caractères sont des tableaux de caractères. Leur manipulation est donc similaire à celle d'un tableau à une dimension :

Déclaration : `char chaîne [taille];`
Exemple : `char texte [taille];`

Le compilateur réserve (taille -1) places en mémoire pour la chaîne de caractères, en effet, il ajoute toujours le caractère (\0) à la fin de la chaîne. Ce caractère signifiant la fin de la chaîne.

On peut, comme pour les tableaux, initialiser une chaîne de caractères au moment de sa déclaration. Ceci peut se faire de deux manières :

```
char texte [10] = "BONJOUR";
ou   char texte [10] = {'B', 'O', 'N', 'J', 'O', 'U', 'R', '\0'};

                              /* Ne pas oublier le caractère de fin*/
ou   char texte[ ] ="BONJOUR";   /* permet d'initialiser le tableau
à
                              la taille correspondante au nombre de
                              lettres + le caractère de fin \0 */
ou   char *texte = "BONJOUR";
```

Attention : `char texte [10];`
`texte = "BONJOUR"; /* Ecriture` **interdite**, `on ne peut affecter directement une valeur à une chaîne déclarée comme un tableau. Il faudra utiliser une fonction de traitement de chaînes de caractères */`

Pour afficher à l'écran une chaîne de caractères, on utilisera le format `%s`. De même, pour la saisie, on pourra utiliser ce même format plus pratique que le format `%c`, où il faudrait préciser le nombre de caractères.

Remarque : la fonction `gets()`, d'emploi plus simple, permet la saisie d'une chaîne de caractère en écrivant simplement `gets(texte);` de même `puts(texte);` est équivalent à `printf ("%s\n", texte);`

CHAP VI : Les Structures

Le tableau, comme nous l'avons vu précédemment, nous permettait, sous un seul nom, de désigner un ensemble de valeurs de même type (tableau d'entiers, tableau de pointeurs sur des caractères ...). Chacune de ces valeurs étaient alors repérées par un indice.

Cette nouvelle notion qu'est la structure va nous permettre, elle aussi sous un seul nom, de définir plusieurs valeurs mais ces valeurs pourront être toutes différentes. L'accès à chacun des éléments de la structure se fera, non plus par une indication de position, mais par son nom au sein de la structure. Chaque élément de la structure sera appelé un champ.

I. Définitions :

Comment déclare t-on une structure ?

```
struct etat_civil
{
  char nom[21];
  char prenom[21];
  int age;
};
```

Dans cet exemple nous avons défini une structure de nom `etat_civil` et qui contient trois champs : deux tableaux de caractères et un entier.

Ici nous n'avons pas réservé de variables comme dans le cas d'un tableau. Le nom `struct` nous indique la déclaration d'une "super" variable de type `struct etat_civil.`

Une fois que nous avons déclaré ce type de variable nous pouvons déclarer des variables du type correspondant à savoir: une structure de nom `struct etat_civil` qui contient trois champs, deux tableaux de caractères et un entier.

Pour désigner cette variable il suffit d'écrire :

`struct etat_civil nom_1, nom_2;`

nous avons alors deux variables `nom_1` *et* `nom_2` qui sont de type `struct etat_civil`.

II. Utilisation d'une structure :

On peut utiliser une structure de deux manières :
- en travaillant individuellement sur ses champs
- en travaillant de manière globale sur l'ensemble de la structure.

Utilisation des champs d'une structure :

Chaque champ peut être manipulé comme une variable du type correspondant. Pour utiliser un champ il faut faire suivre le nom de la variable structure de l'opérateur point (.) suivi du nom du champ comme l'exemple ci-dessous :

`nom_1.age = 10;` nous donnons la valeur 10 à la variable `age` de la structure `nom_1`.

`printf("%d", nom_1.age);` affiche la valeur contenue dans cette variable.

Utilisation globale d'une structure :

On peut affecter à une structure le contenu d'une autre structure de même type, c'est à dire définie à partir du même modèle.

Exemple : nom_1 = nom_2;

Cette manière de faire peut permettre de contourner le problème qui se pose avec les tableaux, à savoir l'impossibilité de faire une affectation globale entre tableaux. Ici, avec les structures, c'est possible il suffit de déclarer une structure contenant comme champ un tableau, de donner deux noms de variables de ce type et d'affecter une des structures à l'autre.

Initialisation d'une structure :

On utilise les mêmes règles que pour les autres types de variables, à savoir :

En l'absence d'initialisation explicite, les structures de classe statique sont par défaut initialisées à zéro. (A noter toutefois que cela dépend du compilateur)

Il est possible d'initialiser explicitement une structure, lors de sa déclaration. (On s'inspirera de ce qui se fait avec les tableaux).

III. Simplification de la déclaration de types (typedef) :

`typedef` permet de définir des synonymes. Cette déclaration s'applique à tous les types et pas seulement aux structures, comme le montre les exemples ci-dessous :

`typedef int entier;` ceci permettra de remplacer le mot `int` par `entier`.
`typedef int* point;` ceci permettra de remplacer le mot `int*` par `point`.
`typedef int vecteur[3];` ceci permettra de remplacer le mot `int []` par `vecteur`.

Considérons par exemple les lignes ci-dessous.

```
struct etat_civil
{
  char nom[21];
  char prenom[21];
  int age;
};
typedef struct etat_civil s_etat;
s_etat nom_1, nom_2;
```

Que l'on peut écrire plus simplement :

```
typedef struct etat_civil
{
  char nom[21];
  char prenom[21];
  int age;
} s_etat;

s_etat nom_1, nom_2;
```

Tableau de structure ou structure comportant d'autres structures :

```
struct point
```

```
{
  char nom[21];
  int x;
  int y;
} ;

struct point courbe[100];
```

La structure `point` peut servir à représenter un point d'un plan, point qui sera défini par son nom (caractère) et ses deux coordonnées.

Une structure peut contenir comme champ une ou plusieurs structures. regardons les lignes suivantes :

```
typedef struct date
{
  int jour;
  int mois;
  int annee;
} s_date;

typedef struct personne
{
  char nom[21];
  char prenom[21];
  float heure;
  s_date date_embauche;
  s_date date_poste;
} employe_1,employe_2;
```

La notation `employe_1.date_embauche.annee` représente l'année d'embauche correspondant à la structure `employe_1`.

On peut effectuer des affectations de ce type :

```
employe_2.date_embauche = employe_1.date_embauche;
```

Remarque : comme les autres variables, une variable de type structure peut être locale ou globale suivant l'emplacement où elle est déclarée (après ou avant `main()`)

IV. Le type enum :

Un type `enum` est défini par la donnée d'un ensemble fini de valeurs représentées par des identificateurs appelés constantes d'énumération et spécifiés lors de la déclaration du type.

Les deux syntaxes suivantes sont équivalentes :

```
#define LUNDI  0
#define MARDI  1
#define MERCREDI2
#define JEUDI  3
```

```
#define VENDREDI   4
#define SAMEDI  5
#define DIMANCHE   6
```

équivaut à

```
typedef enum { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE } jours;
```

La déclaration suivante :

```
enum couleur { BLEU, BLANC, VERT } coul;
```

a les effets suivants :

- Elle crée le type enum `couleur` dont les valeurs sont `BLEU, BLANC, VERT`.
- Elle définit la variable `coul` qui prend ses valeurs dans l'ensemble { `BLEU`, `BLANC, VERT`}; On peut donc lui donner une valeur par des affectations telle que `coul = VERT`.

On peut définir d'autres variables de type `enum couleur` par des déclarations de la forme suivantes : `enum couleur coul_1, coul_2 ...`

CHAP VII : Les Fonctions

Lorsqu'on écrit un programme un tant soit peu complexe, il devient nécessaire de le découper en un ensemble de sous problèmes moins complexes (Eux mêmes pouvant être découpés en sous problème ...). Ce découpage est réalisé au moyen de fonctions. Ainsi une fonction pourra appeler plusieurs autres fonctions. Un programme en langage C est donc constitué d'un ensemble de fonctions. Une fonction est constituée d'une suite d'instructions.

La création d'une fonction, implique deux choses :
La déclaration de la fonction (appelé prototype de la fonction).
La définition de la fonction.

Exemple de programme:

```
#include <stdio.h>
int triple(int);  /*prototype ou déclaration de la fonction triple*/

void main()      /* Fonction principale*/
{
int i,j;       /*  variables locales à la fonction main */
  i=2;
  j = triple(i); /*  appel de la fonction nommée triple */
}     /*  fin de main */

int triple (int nb)   /*  définition de la fonction triple
                 et du type de l'argument */
{
  return (nb*3); /*  valeur retournée de type int */
}
```

La **déclaration** d'une fonction se fait en **début** de fichier (elle sera alors connue de toutes les autres fonctions du fichier) , alors que la définition peut être placée à n'importe quel endroit dans le fichier.

I. Prototype :

Le prototype est utilisé par le compilateur pour vérifier si l'utilisation de la fonction est correct, lors de son appel. La première ligne de l'exemple précédent est appelée fonction prototype. Elle donne le nom de la fonction, le type des arguments passés à la fonction ainsi que la nature de la valeur retournée. Ici, la fonction reçoit deux variables de type entière, et renvoie une valeur de type entière. D'une façon générale une fonction prototype s'écrit:

```
type_valeur_retournée   nom_fonction (type_arg1, ...,type_argk) ;
```
Ne pas oublier le '**;**' à la fin.

exemple :
```
void fct (float, double);
char fct (void);
```

La déclaration d'une fonction est en général placée en début de fichier (elle est alors connue de tout le fichier) mais elle peut être également placée au début de la fonction `main()` avec les déclarations de variables (dans ce cas elle ne sera connue que de la fonction main()).

Même si la déclaration d'une fonction n'est pas obligatoire, il est conseillé de toujours la déclarer.

II. Définition de la fonction :

La définition d'une fonction possède la structure suivante :

```
---------------------------------->   type de la valeur retournée
|   --------------------------> nom de la fonction
|   | ------------------->      type de l'argument 1
|   | |   ---------------->     nom de l'argument 1
|   | |   |   ------------>     type de l'argument 2
|   | |   |   | ------>   nom de l'argument 2
|   | |   |   | |   |->   pas de point virgule
int Somme (int A,  int     B)
{
  corps de la fonction
  return( .....) ;
}
```

On remarque que la première ligne ressemble à la déclaration de la fonction, à deux exceptions près :

- On donne un nom aux arguments transmis (ici A et B).
- Pas de point virgule à la fin.

Le corps de la fonction est entre deux accolades {}, il contient le programme de la fonction.

Attention : une fonction ne peut retournée qu'une seule valeur.

III. Appel de la fonction :

Pour appeler une fonction, il suffit d'écrire le nom de la fonction suivi entre parenthèses de la liste des variables transmises à la fonction :

```
j = Somme (i , 5 );
```

Cela signifie : la fonction *Somme* est appelée, on lui passe comme argument la valeur de la variable *i* et la valeur *5*. La valeur rendue par la fonction est stockée dans la variable j

Les expressions suivantes sont également correctes :

```
j = Somme(i , 5)* Somme (i, 10) - 8;
j = Somme(i , Somme(j , 4));
```

Remarque importante : Dans l'exemple 1, la variable `i` dans la fonction `main` est copiée dans la variable `A` de la fonction `Somme`. De même la valeur `5` est copiée dans la variable `B`.

De ce fait, même si la fonction `Somme()` modifie la variable A, la variable `i` du programme principal se trouverait inchangée.

On dit que le passage des arguments se fait par **valeur**. En C, et à la différence d'autres langages (Pascal par exemple), le passage d'argument se fait toujours par valeurs.

Autre exemple pour les noms de variables :

```
#include <stdio.h>
short boucle ( short);     /* Fonction boucle reçoit une valeur de
                 type short et renvoie une valeur de type
                 short */
void main()
{
short j,i;     /* Déclaration de variables locales de main */

  for ( i=0; i<3; i++)
  {
      j= boucle(i);    /* Appel de la fonction boucle*/
      printf("%d ",j); /* Affichage successif de 6,7,8 */
  }
}

short boucle (short nb)          /*Fonction boucle*/
{
short i, nb2;  /*Déclarations des variables locales de boucle*/

  nb2 = nb;
  for (i=1;i<4;i++)
  {
      nb2 = nb2+i;     /* 1+2+3 */
  }
  return nb2;       /* On retourne nb+6 */
}
```

Dans cet exemple, ce n'est pas tellement les valeurs retournées qui sont importantes mais le fait que l'on utilise 2 fois une variable nommée i, les deux fois étant complètement indépendantes l'une de l'autre.

Les noms des arguments qui figurent dans l'entête de la fonction se nomment : arguments formels ou muets. Ils servent, au sein du corps de la fonction, à décrire ce qu'elle doit faire.

Les arguments utilisés lors de l'appel de la fonction s'appellent des arguments effectifs : ils représentent les valeurs que les paramètres prendront effectivement lors de l'exécution de la fonction.

L'instruction `return` peut mentionner n'importe quelle expression, mais elle ne peut s'exécuter qu'une seule fois. Dès son exécution, la fonction prend fin. Toutefois cette instruction peut apparaître plusieurs fois dans une fonction.

Exemple :

```
double somme (double x, double y)
{
  double som ;
```

```
    som = x + y ;
    if (som > 0)
        return (som) ;
    else
        return (- som) ;
}
```

Il est cependant conseillé de s'arranger pour ne faire apparaître l'instruction `return` qu'une seule fois et cela tout à la fin, juste avant l'accolade de fin de fonction.

En plus de donner le résultat, l'instruction `return` interrompt l'exécution de la fonction et retourne dans la fonction qui l'a appelée.

La fonction `return` peut indifféremment contenir une valeur ou une opération.

Exemple : return (x = 2*y + b);

En règle générale il vaut mieux éviter de mettre une opération dans l'expression `return` et ceci pour une meilleure lisibilité du programme. Dans le cas ou la fonction ne retourne aucune valeur, il n'y a pas d'instruction `return` et le retour au programme ou à la fonction appelante se fait automatiquement à la fin de la fonction.

Remarque : La fonction `return` peut retourner comme résultat un code d'erreur indiquant au programme appelant si la fonction appelée s'est déroulée correctement.

IV. Fonctions sans valeur de retour et sans argument :

Si une fonction ne renvoie pas de résultat, il est nécessaire de le préciser dans l'entête et dans sa déclaration. Le mot clé utilisé pour cela s'appelle `void` (rien).

On peut aussi rencontrer une fonction qui ne reçoit aucun argument mais qui retourne une valeur.

Exemple de déclaration *:* float nomb_aleatoire (void);

On peut aussi trouver des fonctions ne recevant aucun argument et ne renvoyant aucune valeur.

Exemple de déclaration : `void message (void);`

V. Notion de variable locale et de variable globale :

Avant d'aborder plus précisément ces notions, il faut parler de la notion de fichier.

Le fichier en-tête standard `stdio.h` contient les déclarations de toutes les fonctions et variables de la bibliothèque d'entrée/sortie du langage C. Pour l'inclure dans un programme, il suffit d'ajouter la ligne suivante en tête du fichier source.

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

On fera de même avec tous les fichiers spécifiques dont nous aurons besoin. Par exemple un fichier pourra contenir des fonctions que vous aurez développées pour des besoins

spécifiques. Lors de telles opérations il est important de choisir comment sera définie une fonction.

Est-ce qu'elle sera locale à une fonction, à un fichier ou plus ?

Une variable locale est une variable qui n'est connue que dans une fonction. Elle est locale à cette fonction. Sa valeur est contenue dans la pile du microprocesseur et dès la fin de l'exécution de la fonction elle est détruite. Pour pallier cet inconvénient, il est possible de garder l'emplacement mémoire de cette variable, uniquement dans la fonction, en utilisant le mot clé `static`. Exemple : `static int nbre;` la variable `nbre` se verra attribuer un emplacement mémoire fixe, mais ceci ne sera vrai que dans la fonction où elle est déclarée.

Une variable globale est une variable qui porte le même nom dans toutes les fonctions et qui donc est connue dans toutes les fonctions. Elle est rangée dans une zone mémoire que l'on appelle « tas ». La particularité des variables globales est double:

- connues partout avec une seule déclaration, ceci est plutôt un avantage.
- elles peuvent être modifiées facilement dans une fonction, attention ceci est dangereux car vous pouvez modifier une variable de partout et vous pouvez vous retrouver avec des incohérences voir des résultats aléatoires. Afin qu'une variable globale ne soit pas connue par tous les fichiers il est possible de se servir de `static`, à ce moment la variable n'est plus globale que pour le fichier dans lequel elle est déclarée.

Déclaration des variables globales:

```
#include <stdio.h>
short triple (short);           /*  Prototype  de  la  fonction
triple*/

/* déclaration des variables globales */
short tableau[2][3];          /*  variable  globale  à  toutes  les
fonctions
                  du fichier */

void main()
{
short i;       /* variable locale à la fonction main */

  tableau[0][1] = 10; /* Modification de la valeur de tableau
                 [0][1] à l'intérieur de main*/
  n = triple(5);
  printf ("Valeur de tableau [0][1] %d", tableau [0][1]);
              /* Ici ce sera 20, la valeur donnée dans la
              fonction triple et non pas celle donnée
               précédemment dans main*/
}

short triple (n)
{
short i;       /*  variable locale à la fonction triple */

  tableau [0][1] = 20;      /* Modification de la valeur donnée
                 dans main*/
```

```
}
```

Attention : Si une variable globale et une variable locale dans une fonction f portent le même nom, la variable globale n'est pas accessible dans la fonction f. Seule la variable locale sera affectée.

VI. Les fonctions récursives :

En C on peut utiliser la récursivité des appels de fonctions. Cette récursivité peut prendre deux aspects :

- récursivité directe : une fonction comporte, dans sa définition, au moins un appel à elle même.
- récursivité croisée : l'appel d'une fonction entraîne celui d'une autre fonction qui, à son tour, appelle la fonction initiale.

Exemple :

```
long  fact (int n)
{
  if (n>1)
      return (fact(n-1)*n);
  else
      return (1);
}
```

Remarque : La récursivité est une démarche intellectuellement satisfaisante. Elle est très utilisée en algorithmique car elle permet de rédiger des algorithmes très concis et efficace. C'est pour cela qu'il est important de comprendre ce mécanisme.

Par contre, dans un programme elle peut être très dangereuse. En effet, en général le système, pour chaque appel, empile les paramètres de la fonction ainsi que l'adresse de retour de la fonction dans sa « pile ». Or cette « pile » est une zone mémoire et n'est donc pas illimitée.

Donc si un grand nombre d'appels de fonctions sont imbriquées, il y a un risque important de faire déborder la zone allouée à la « pile », ce qui aurait pour conséquence possible d'écraser des données.

CHAP VIII : Les pointeurs

I. Définitions, notations :

En C, nous pouvons définir une variable destinée à contenir l'adresse d'une autre variable. On parle de pointeur. (la taille réservée en mémoire dépendra du processeur sur lequel tourne le programme).

Si P est un pointeur (donc contenant une adresse), *P permettra d'accéder et de manipuler le contenu de l'adresse dont la valeur est dans P ou adresse pointée par P.

Pour manipuler *P, il faut connaître le type de la variable située à l'adresse P (ou variable pointée) ;

Le pointeur sera donc déclaré en indiquant le type de la variable pointée.

Important: le type d'un pointeur doit correspondre au type de la variable sur laquelle il pointe.

Déclaration : type* nom_pointeur;

exemple : `int* point ;` déclare un pointeur `point` est de type `int*,` c'est à dire contenant
l'adresse d'un entier.
`point` contiendra l'adresse d'un entier, `*point` représentera le contenu
de cet entier pointé

Comment indiquer à un pointeur l'endroit où il doit pointer ?

Soit une variable `nombre` de type entier : `int nombre;`

pour que le pointeur `point` «pointe» sur la variable `nombre` on lui transmet l'adresse de la variable de la façon suivante : `point = &nombre;`

L'adresse d'un objet étant définie par un pointeur, l'accès à cet objet est INDIRECTEMENT réalisé par l'intermédiaire du pointeur. On dit qu'il y a un niveau d'indirection.

Remarque : on peut également initialiser le pointeur lors de la déclaration

```
int nombre ;
int* point = &nombre ;
```

Comment transmettre le contenu de `nombre` (sa valeur) en se servant du pointeur `point` ?

Si `reserve` est un autre entier déclaré , on écrit simplement: `reserve = *point;`

`reserve` contient maintenant la même valeur que celle contenue dans `nombre`.

Inversement en écrivant : `*point = reserve;`

Ce sera la variable sur laquelle pointe `point` (c'est à dire `nombre`) qui va changer de valeur et prendre celle de `reserve.`(on aura fait l'équivalent de `nombre = reserve;`)

Exemple : int nombre, reserve;

```
      int *point;
```

```
point = &nombre;
nombre = 20;
reserve = *point;     /* reserve = 20 */
reserve = 30;
*point = reserve;          /* nombre = 30  */
```

II. Cas particulier: le pointeur NULL :

La déclaration d'un pointeur réserve la place mémoire nécessaire à la mémorisation d'une adresse. **Par contre, cela ne réserve pas de place en mémoire pour l'objet sur lequel il pointera.** Un pointeur non initialisé pointe donc n'importe où, ce qui peut conduire à des erreurs très graves. En effet utiliser un pointeur non initialisé revient à aller ″squatter″ une case mémoire occupée par une autre variable , surement très utile par ailleurs.

Pour pallier ce problème, on peut initialiser un pointeur avec NULL, cela revient à le faire pointer sur «rien», ce qui sans éviter le problème de la variable non initialisée, permet au moins de ne pas aller utiliser une case déjà prise.

III. Incrémentation. Décrémentation :

Cela signifie : augmenter (ou diminuer) la valeur de l'adresse contenue dans le pointeur pour examiner l'objet suivant(ou précédent). L'amplitude de déplacement est donc directement liée au type d'objet pointé.

Exemple :

```
float nb[5] = {1.5, 2.3, 7.0, 2.25, 3.345} ;
float* pvar ;
pvar = &nb[0] ;  // pvar pointe sur nb[0], *pvar contient 1.5
pvar++ ;     // pvar pointe maintenant sur nb[1], *pvar
         // contient 2.3
```

Dans l'exemple ci-dessus, les données manipulées sont des réels. Un réel est codé sur 4 octets.

Si l'adresse de nb[0] est 0x1000, alors nb[1] est à l'adresse 0x1004 et nb[2] est à l'adresse 0x1008.

Faire l'instruction pvar++ (si pvar est un float*) revient à augmenter la valeur de pvar de 4.

On peut aussi utiliser les opérateurs d'assignation.

```
point += i;
point -= i;
```

IV. Comparaison :

Il est possible d'utiliser les opérateurs relationnels courants avec les pointeurs :
<, <=, >, >=, ==, !=

Ces possibilités sont intéressantes pour manipuler divers pointeurs sur une même zone mémoire.

Exemple :

```
float nb[5] = {1.5, 2.3, 7.0, 2.25, 3.345} ;
float* pvar ;
pvar = &nb[0] ;  // pvar pointe sur nb[0], *pvar contient 1.5
while (pvar < &nb[4])  // attente de parcours de la fin du
                // tableau
```

V. Un nom de tableau est une adresse:

Si nous effectuons la déclaration suivante : `int tab[10];`

La notation `tab` est alors complètement équivalente à `&tab[0];` L'identificateur `tab` est considéré comme étant de type pointeur sur le type correspondant aux éléments du tableau (ici `int`). Voici quelques exemples de notations équivalentes :

```
tab +1   &tab[1]
tab + i      &tab[i]
*(tab + i)    tab[i]
```

Entraînez vous à utiliser ces notations équivalentes.

Cas des tableaux à plusieurs indices :

Comme pour les tableaux à un indice, l'identificateur du tableau, employé seul, représente toujours son adresse de début. Si on s'intéresse à son type exact, il ne s'agit plus d'un pointeur sur des éléments du tableau.

`int tab [3][4] ;`

`tab` désigne un tableau de 3 éléments, chacun de ces éléments est lui même un tableau de 4 entiers. Autrement dit *tab* est du type pointeur sur des blocs de 4 entiers.

On a par exemple `tab` ≡ `&tab[0][0]` mais

si tab [0] ≡ &tab [0][0] on aura

`tab [1]` ≡ `&tab [1][0]` donc attention à l'incrémentation et à la décrémentation.

Exemple de comparaison de pointeurs avec un tableau :

```
int tab [10]; /* tableau de 10 entiers*/
int* point;
for (point = tab; point < tab +10; point ++)
  *point = 1;
```

Ici on met à un tous les éléments du tableau.

CHAP IX : Fonctions (suite) : Passage de paramètres par « adresse »

Rappelez vous une fonction ne transmet et ne reçoit que des valeurs. Il est donc impossible de modifier une variable transmise à une fonction.

Que faire donc si on veut écrire une fonction qui doit retourner plusieurs résultats ? Impossible de faire passer plus d'un résultat par l'instruction `return`.

Donc pas de solutions ... Si bien sur, il suffit d'utiliser les pointeurs ...

Car une adresse est une valeur, et si on passe comme paramètre l'adresse d'une variable, on pourra , grâce aux pointeurs accéder au contenu de l'adresse transmise .

I. Exemple: fonction de permutation

Il est évident que cette fonction recevra deux paramètres en donnée, qu'elle devra les permuter et retourner ces 2 valeurs dans les mêmes paramètres. Essayez de le faire sans l'aide de pointeurs ... impossible , les valeurs sont bien échangées au niveau de la fonction mais dès le retour dans le programme d'appel, on constate que rien n'a changé.

Alors qu'à l'aide des pointeurs ...

```
#include <stdio.h>

void echange (int*, int*) ;

void main ()
{
int a = 10, b = 20;
  printf (″Avant appel %d, %d \n″, a, b);
  echange (&a, &b);
  printf (″Après appel %d, %d″, a, b);
}

void echange (int* ad_1, int *ad_2)
{
int x;
  x = *ad_1;
  *ad_1 = *ad_2;
  *ad_2 = x;
}
```

Les arguments effectifs de l'appel de `echange` sont les adresses des variables `a` et `b` et non plus leurs valeurs. Mais la transmission se fait toujours par valeur.

Exercice : Ecrire une fonction qui fournit en retour la somme des valeurs d'un tableau de flottants à deux indices dont les dimensions sont fournies en argument.

L'intégrer dans un programme simple où on rentre les valeurs au clavier (ce peut être une fonction de remplissage). Afficher la somme.

II. Les tableaux transmis en argument :

Si on place le nom d'un tableau en argument effectif de l'appel d'une fonction, on transmet l'adresse du tableau à la fonction. On peut donc effectuer toutes les manipulations voulues sur ses éléments.

II.1. Tableau à un indice :

Exemple : mettre à un tous les éléments d'un tableau de 10 entiers sans fonction, puis avec fonction.

```
void  fct (int tab [10])
{
int i;
  for ( i = 0; i < 10; i++)
  {
      tab[i] = 1;
  }
}
```

Exemples d'appel de cette fonction :

```
int tab1[10], *tab2;
fct (tab1);
fct (tab2);
```

Le prototype de la fonction peut s'écrire indifféremment

```
void fct (int tab[10]) ;
void fct (int *tab) ;
void fct (int tab[]) ;
```

Avec des tableaux à une seule dimension il est facile de réaliser une fonction capable de travailler avec un tableau de dimension quelconque, à condition de lui en transmettre la taille en argument.

Exemple :

```
int somme (int tab[], int nbre)
{
int s = 0, i;
  for (i = 0; i < nbre; i++)
      s+ = tab[i];
  return s;
}
```

II.2. Tableaux à plusieurs indices :

- Tableau fixe : Exemple d'une fonction qui place la valeur 1 dans chacun des éléments d'un tableau de dimension 10 et 15.

```
void raun (int tab[][])
{
int i, j;
  for (i = 0; i < 10; i++)
  {
      for (j = 0; j < 15; j++)
       tab[i][j] = 1;
  }
}
```

III. Transmission de l'adresse d'une structure en argument d'une fonction :

Nous avons dit précédemment qu'une transmission des arguments se faisait par valeur. Ceci implique une recopie de l'information transmise à la fonction. Comme pour les variables de type simple, nous pourrons utiliser des pointeurs pour affecter les valeurs, dans la fonction, à la structure désirée.

```
#include <stdio.h>

void fonction (struct reel *);

struct reel  {
  float a;
  float b;
};

void main()
{
struct reel first;

  first.a = 1;
  first.b = 2.5;
  printf("%f, %f",first.a, first.b);
  fonction (&first);
  printf("%f, %f",first.a, first.b);
}

void fonction(struct reel *third)
{
  third->a = third->b;     /*   Notez   l'opérateur   ->   il   est
équivalent
                à la   notation (*third).a ou (*third).b */
}
```

IV. Quelques fonctions de manipulations de chaînes de caractères :

La fonction `strcat()` :

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

void main ()
{
char chaîne_1[ ] = ″Bonjour ″;
char* chaîne_2 = ″Monsieur″;
puts (chaîne_1);/* Bonjour*/
strcat (chaîne_1, chaîne_2);
puts (chaîne_1);/* Bonjour Monsieur*/
}
```

Cette fonction `strcat()` recopie la seconde chaîne à la suite de la première après en avoir effacé le caractère de fin.

La fonction `strncat()` :

Elle fonctionne de manière similaire à la précédente mais en donnant la possibilité de contrôler le nombre de caractères concaténés à la première chaîne.

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

void main ()
{
  char chaîne_1[ ] = "Bonjour ";
  char* chaîne_2  = "Monsieur";
  puts (chaîne_1); /* Bonjour*/
  strncat (chaîne_1, chaîne_2, 6);
  puts (chaîne_1); /* Bonjour Monsi*/
}
```

La fonction `strcmp()` : fonction de comparaison de chaînes

`strcmp (chaîne_1,chaîne_2)` donne un résultat positif si `chaîne_1` arrive après `chaîne_2` au sens de l'ordre défini par le code des caractères, négatif si `chaîne_1` arrive avant et nulle si les deux chaînes sont les mêmes.

La fonction `strncmp(chaîne_1, chaîne_2, longueur)` effectue la même chose mais sur une longueur définie par `longueur`.

Pour ne pas tenir compte des majuscules et minuscules, on utilisera respectivement `stricmp()` et `strnicmp()`.

La fonction `strcpy` (respectivement `strncpy`) :

`strcpy (chaîne_1, chaîne_2)` ou `strncpy(chaîne_1, chaîne_2, longueur)` recopie `chaîne_2` à l'adresse de `chaîne_1` (nécessité d'avoir une taille suffisante en `chaîne_1` pour pouvoir recevoir l'intégralité de `chaîne_2`. (Respectivement définie par longueur)

cette liste n'est pas exhaustive, l'aide du compilateur vous permettra d'utiliser d'autres chaînes telles que `strlen()`, `strrev()`...

CHAP X : LISTES CHAÎNÉES

I. Pré-requis : Allocation dynamique :

Lorsque nous déclarons une variable, une place mémoire est allouée et cela pour toute la durée du programme : on parle d'allocation de mémoire statique.

Mais il existe des fonctions qui permettent de demander au système d'allouer une zone mémoire d'une certaine taille en cours de déroulement du programme.

Ces fonctions recoivent en paramètre la taille de la place mémoire demandée et retourne l'adresse de la première case de cette zone. Ces fonctions retournent un pointeur puisqu'il s'agit d'une adresse.

Sans être exhaustif en voici deux parmi les plus importantes:

la fonction malloc() :

```
char* c;
int* i, * j, * k;
float* r;
c = (char*) malloc (10);   /*On réserve 10 cases mémoire, soit la place pour 10 caractères*/
i = (int*) malloc (16);    /* On réserve 16 cases mémoire (ce sont des int donc méfiance)*/
r = (float*) malloc (24);  /* On réserve 24 emplacements mémoires, soit le place pour 6réels*/
j = (int*) malloc (sizeof (int));      /* On réserve la taille d'un entier en mémoire, utile si on ne connaît pas les caractéristiques du compilateur*/

k = (int*) malloc (3*sizeof (int));  /* On réserve la place en mémoire pour 3 entiers*/
```

Les places réservées en mémoire sont logées dans une zone de la mémoire vive (RAM) appelée le «tas». Ce tas est spécifique au compilateur utilisé. La zone de mémoire est de taille limitée. Il n'est pas possible d'allouer une zone de mémoire illimitée. Il faut donc, comme pour tous les appels systèmes, vérifier que l'allocation mémoire s'est bien passée.

Si l'allocation n'a pas été possible, la fonction retourne la valeur NULL comme adresse. Il est donc impératif de tester systématiquement la valeur de retour de malloc().

```
k = (int*) malloc (3*sizeof (int)) ;
if( k == NULL)
{
  perror ( "pb malloc") ;
  exit(-1) ;
}

ou

if( (k = (int*) malloc (3*sizeof (int)) ) == NULL)
{
  perror ( "pb malloc") ;
```

```
  exit(-1) ;
}
```

La fonction `perror()` permet d'afficher le texte donné en paramètre et de donner le code d'erreur système qui a empêché le bon déroulement de l'appel. Il est très souvent inutile de continuer le programme après une erreur de ce type , d'où l'utilisation de la fonction `exit()`.

La fonction calloc() :

Contrairement à `malloc()`, la fonction `calloc()` remet à zéro chacun des octets de la zone mémoire allouée. En général, l'allocation par `calloc()` de p blocs de n octets conduit à utiliser un peu moins de mémoire que la même opération par `malloc()`.

Sur l'exemple suivant on peut voir l'écriture nécessaire avec `calloc()` et la légère différence avec `malloc()` :

```
    k = (int*) calloc (3, sizeof (int));      /* On réserve la place
en
                                mémoire pour 3 entiers */
```

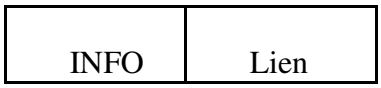
Ces deux fonctions font appel à `stlib.h` et/ou `alloc.h`.

Pour libérer l'espace mémoire alloué lorsque l'on en n'a plus besoin se fait à l'aide de la fonction free:

```
    free(k); /* On libère la place en mémoire pour 3 entiers*/
```

II. Introduction :

Cas particulier de la gestion dynamique de la mémoire, la liste chaînée. Nous allons nous intéresser à des emplacements en mémoire réunis par un lien. Ces emplacements en mémoire se découpent comme suit .



où lien fait référence à un autre emplacement ou bloc de données. Lorsque tous les blocs, sauf le dernier, contiennent un pointeur sur le bloc suivant (ou lien) on parle de liste chaînée ou liste séquentielle. Les blocs d'adresse mémoire d'une liste chaînée sont généralement appelés éléments ou nœuds de la liste. Ce sont des structures de même type et de même taille pour une liste donnée. A l'exception du dernier chaque nœud contient un pointeur sur son successeur immédiat et est désigné, sauf le premier, par son successeur immédiat.

Un tableau est l'exemple le plus simple d'un ensemble d'objets liés entre eux. C'est une suite de variables de même type, placées en mémoire de manière contiguë, et auxquelles on accède par la notation simple de l'indexation. Malheureusement les tableaux ne sont pas adaptés au stockage de toutes les données à traiter dans un programme. En particulier :

Si la taille des données s'accroît en fonction des résultats et si on ne connaît pas au départ la place mémoire qui sera nécessaire.

Ou bien lorsqu'il faut insérer un nouvel objet à une place déterminée dans une liste d'objets déjà stockés.

Ou lorsqu'il faut retirer un objet sans laisser de «trou» dans la liste.

Ou encore pour stocker des objets qui ont entre eux des liens complexes (fiches d'identité par exemple).

La donnée de l'adresse de la première structure donne accès à tous les éléments de la liste. La fin de la liste est indiquée par un pointeur sur »rien» (NULL). Sur l'exemple suivant nous allons nous intéresser à une liste de personnes désignées par leur nom et classées par ordre alphabétique. Une telle liste entraîne les manipulations courantes suivantes : affichage, recherche d'un élément et insertion d'un nouvel élément.

```
typedef struct personne{
  char* Nom;
  struct personne *suivant;
} personne;
```

III. Création :

La fonction suivante retourne un pointeur `person` sur le type personne avec initialisation de `person->Nom` au nom donné en argument.

```
personne* Cree (char* PNom)
{
  personne* person;
  person = malloc(sizeof(personne));
  if (person == NULL)
  {
      perror(″pb malloc person ″) ;
      exit(-1) ;
  }
  person-> Nom = malloc( strlen(Nom)+1);    /*Ne pas oublier le
                       caractère de fin de chaîne*/
  strcpy(person-> Nom, PNom);    /*Copie de PNom dans person->Nom*/
  return person;
}
```

Le test sur la valeur retournée par `malloc` dans `person` est indispensable pour savoir si l'allocation dynamique a été faite correctement. Si la mémoire n'a pas réussie à être allouée correctement, rien ne sert de poursuivre (`exit()`). La fonction `perror()` permet d'afficher le texte indiqué entre ″ et de donner également la cause de l'erreur .

Une liste chaînée de personnes est un ensemble de pointeurs sur le type personne avec les trois caractéristiques suivantes :

-L'un des pointeurs, appelé couramment tête ou début, a pour valeur l'adresse de la première structure de la liste.

-Pour chaque pointeur p de cet ensemble, le pointeur `p->suivant` a pour valeur l'adresse de la structure suivante de la liste.

-Le dernier pointeur q de la liste se reconnaît par le fait que le pointeur `q->suivant` vaut `NULL`.. La fonction suivante affiche des noms contenus dans une liste de personnes. On définit un pointeur auxiliaire Pinter qui prend successivement pour valeur toutes les adresses des structures de la liste.

IV. Affichage :

```
void Affiche (personne* Tete)
{
  personne* Pinter;
  for (Pinter = Tete; Pinter != NULL; Pinter = Pinter ->suivant)
      printf ("%s\n", Pinter ->suivant);
}
```

La ligne du for est une manière classique de parcourir une liste chaînée.

V. Insertion :

Dans la fonction qui suit nous allons nous attacher à introduire une fiche supplémentaire dans la liste. On supposera que la liste est déjà classée par ordre alphabétique. Ici nous devons prendre en compte trois éventualités à savoir :

-Si la liste n'existe pas encore, il faut la créer.

-Si la liste existe déjà et si `person->Nom` précède `Tete->Nom` dans l'ordre alphabétique, il faut placer `person` en tête de liste.

-dans les cas restants, il faut parcourir la liste pour insérer person à la bonne place. On utilise pour cela un pointeur intermédiaire Pinter qui parcourt la liste et compare à chaque fois `person->Nom` avec `Pinter->Nom`.

```
personne* Insere (personne* Tete, personne* person)
{
  personne* Pinter;/*Pointeur auxiliaire*/
  if (Tete == NULL)
  {
      person -> suivant = Tete;
      return person;  /*Cas d'initialisation de la liste*/
  }

  /*Cas du changement de tete de liste*/
  if (strcmp (person -> Nom, Tete -> Nom) <= 0)  /*Comparaison,
                si antérieur la nouvelle fiche passe devant*/
  {
      person -> suivant = Tete;
      return person;
  }

  /* Cas général, insertion dans la liste*/

  Pinter = Tete; /* Parcours de la liste*/
  while (Pinter -> suivant) /*Tant que différent de NULL*/
  {
      if (strcmp(person -> Nom, Pinter -> suivant -> Nom) >0)
      {
       Pinter = Pinter->suivant;  /* On avance dans la liste*/
      }
      else
```

```
      { /* Insertion de person entre Pinter et Pinter -> suivant*/
      person -> suivant = Pinter -> suivant;
      Pinter -> suivant = person;
      return Tete;
      }
  }

  /* Si on arrive ici il faut placer person en bout de liste*/
  Pinter -> suivant = person;
  person -> suivant = NULL;
  return Tete;
}
```

CHAP XI : LES ENTRÉES SORTIES

I. Introduction :

Toutes les fonctions d'entrée-sortie ont une mise en œuvre qui suit un principe simple. Un transfert de fichiers quelle que soit sa destination est un transfert d'une suite d'octets. Ceci est vrai aussi pour les dialogues avec les terminaux tels que l'écran ou le clavier, de même pour une imprimante ou tout autre périphérique. Ceci correspond à la nature des fichiers sous des systèmes d'exploitation tels que UNIX ou DOS. Pour ces derniers un fichier n'est qu'une suite non structurée d'octets. Grâce à ceci les mêmes fonctions sont utilisées quelle que soit l'unité physique concernée.

Lorsqu'un programme doit réaliser une opération d'entrée-sortie, il doit d'abord ouvrir le fichier correspondant. Toutes les opérations de lecture ou d'écriture s'effectuent au moyen d'un pointeur sur une structure (appelé aussi pointeur de fichier) obtenu lors de l'ouverture.

Les prototypes de toutes les fonctions d'entrée-sortie sont déclarés dans le fichier stdio.h.

Attention: Il est possible de spécifier si un fichier est ouvert en mode texte (le mode par défaut) ou en mode binaire. Le second mode nécessite la lettre b à la suite des lettres ci-dessous exemple : *ab*.

Par exemple ouvrir un fichier en mode texte permet de traduire les caractères de fin de ligne de DOS au format Borland C.

II. Ouverture, fermeture d'un fichier :

La fonction `fopen()` permet d'ouvrir un fichier, son prototype est le suivant :

```
FILE *fopen (const char *Nom, const char *Mode);
```

`Nom` est un pointeur sur une chaîne de caractères désignant le fichier concerné et `Mode` est un pointeur sur une chaîne de caractères définissant le mode d'accès demandé.

Les principaux modes sont :

- `"r"` ouverture en lecture,
- `"w"` ouverture en écriture avec création du fichier s'il n'existe pas et écrasement s'il existe,
- `"a"` ouverture en ajout à la fin du fichier s'il existe et création s'il n'existe pas.
- `"r+ "` ouverture pour lecture et écriture, mise à jour d'un fichier existant.
- `"w+"` ouverture d'un fichier en lecture et écriture, si le fichier existe il est écrasé.
- `"a+"` ouverture en lecture et écriture d'un fichier auquel on peut ajouter des enregistrements. Si le fichier n'existe pas il est créé

Exemple d'ouverture du fichier toto.doc en lecture :

```
FILE *pf; /*Pour la définition d'un pointeur de fichier*/
pf = fopen ("toto.doc", "r");
```

On affecte à pf l'adresse d'une structure FILE et on initialise les champs de cette structure avec les données adéquates pour la gestion des opérations ultérieures de lectures dans le fichier toto.doc. Si l'ouverture du fichier échoue, fopen() retourne NULL : il faut donc utiliser cette valeur retournée pour effectuer un test de succès de l'ouverture ; sinon il est bien sur interdit et dangereux d'effectuer une quelconque opération sur ce fichier non ouvert.

```
if(pf == NULL)
{
  perror("pb ouverture toto.doc") ;
  exit(-1) ;
}
```

Pour fermer un fichier il suffit de supprimer le lien entre ce fichier et la structure de type FILE qui lui a été associée à l'ouverture. Elle s'effectue au moyen de la fonction fclose avec l'instruction : `fclose (pf);`

avec pf l'adresse de cette structure.

Après r,w,a on peut ajouter "t" ou "b" : "t" signifie qu'on ouvre le fichier en mode texte (valeur par défaut) et "b" signifie qu'on ouvre le fichier en mode binaire. Ce mode est préférable lorsque l'on stocke des nombres.

Exemple : fopen("toto.bin", "w+b") ;

III. Ecriture dans un fichier :

Les deux fonctions principales d'écriture dans un fichier sont fputc (caractère par caractère) et fprintf (écriture formatée). La première a pour prototype :

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

Soit pf le pointeur de fichier obtenu lors de l'ouverture d'un fichier en écriture, l'instruction `fputc (c, pf);` écrit dans le fichier de caractère de code c. La fonction fputc() retourne le code du caractère ou EOF en cas d'erreur.

La fonction fprintf() réalise des écritures formatées dans un fichier quelconque avec la même syntaxe que printf(), à la différence que l'on ajoute en premier argument supplémentaire le pointeur du fichier. Comme printf() cette fonction retourne le nombre de caractères écrits ou EOF en cas d'erreur. Ci-dessous un exemple d'utilisation de cette fonction:

```
fprintf (pf, "2 + 2 = %d\n", i);
```

La fonction fwrite() peut être une alternative intéressante elle permet de transférer des blocs de données, sa syntaxe sera la suivante:

`fwrite (adresse, nombre d'octets, nombre, pointeur)`

avec adresse adresse de départ à partir de laquelle les caractères seront écrits.

nombre d'octets définit la taille des blocs de données.

nombre définit le nombre de blocs à transférer

pointeur pointe sur le fichier d'entrée.

IV. Lecture dans un fichier :

`fgetc()` et `ungetc()` sont deux fonctions couramment employées pour la lecture caractère par caractère et `fscanf()` pour la lecture formatée. Leurs prototypes sont :

```
int fgetc (FILE *pf);
int ungetc (intc, FILE *pf);
```

La première fonction lit un caractère dans le fichier et retourne son code. La deuxième place le caractère de code c dans le flux d'entrée venant du fichier, de sorte qu'il soit le prochain caractère lu.

`fscanf()` a une syntaxe similaire à celle de `scanf()`, on rajoute en premier argument supplémentaire le pointeur du fichier. Cette fonction réalise des lectures formatées dans un fichier quelconque, l'exemple ci-dessous donne une idée de la syntaxe :

```
fscanf (pf, "%d%d\n", &i, &j);
```

De la même manière que `scanf()`, la fonction `fscanf()` retourne le nombre de spécifications de conversions ayant donné lieu à une lecture ou `EOF (end of file)` en cas d'erreur).

La fonction `fread()` peut être, elle aussi comme `fwrite()` plus haut, une alternative intéressante elle permet de transférer des blocs de données, sa syntaxe sera la suivante:

`fread (adresse, nombre d'octets, nombre, pointeur)`

avec `adresse` adresse de départ à partir de laquelle les caractères seront lus.

`nombre d'octets` définit la taille des blocs de données.

`nombre` définit le nombre de blocs à transférer

`pointeur` pointe sur le fichier de sortie.

V. Les entrées-sorties standard :

Lorsque l'exécution d'un programme débute, d'emblée trois structures de type `FILE` sont allouées et associées par défaut au terminal. Chacune est affectée à un certain type d'entrée-sortie. Les adresses de ces structures sont respectivement : `stdin`, `stdout` et `stderr`.

- le fichier correspondant à `stdin` est appelé entrée standard.
- le fichier correspondant à `stdout` est appelé sortie standard
- le fichier correspondant à `stderr` est appelé sortie erreur standard.

VI. Les fichiers directs :

Ci dessus nous avons vu des gestions de fichiers faites en séquentiel. Il existe un certain nombre de commandes qui permettent l'accès direct aux données.

`fseek()` est une des plus utilisées elle permet le repositionnement d'un pointeur de fichier. Sa syntaxe est la suivante :

`fseek( pointeur, nombre, position)`

Le premier paramètre correspond à un pointeur de fichier. Le déplacement est effectué à partir de position sur nombre (d'octets). L'argument position peut prendre trois valeurs

`SEEK_SET`	début de fichier
`SEEK_CUR`	position courante du pointeur de fichier
`SEEK_END`	fin (réelle) de fichier.

CHAP XII : ANNEXES

Annexe A

Systèmes numériques

Décimal	Binaire	Octal	Hexadécimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Les chiffres en représentation octale sont codés au moyen de huit symboles (les chiffres 0 à 7), tandis que les chiffres en représentation hexadécimale le sont au moyen de seize symboles (les chiffres 0 à 9 et les lettres A à F).

Chaque chiffre octal est équivalent à trois chiffres binaires (ou trois bits) et chaque chiffre hexadécimal équivaut à quatre chiffres binaires (quatre bits). Les nombres exprimés sous forme octale ou hexadécimale fournissent par conséquent un moyen efficace de représenter des motifs binaires de façon compacte. Le motif binaire `10110111`, par exemple, représente `B7` en mode hexadécimal, équivalence qui apparaît plus clairement si l'on sépare les groupes de bits: `1011 0111` est ainsi représenté par `B 7`.

Ce même motif binaire peut être interprété comme le nombre octal `267`. Cette équivalence peut être mise en évidence en imaginant que ce motif binaire est précédé d'un `0`, permettant ainsi de distinguer trois groupes de bits `010`, `110` et `111`, correspond dans le système octal aux chiffres `2`,`6` et `7`.

La plupart des compilateurs font appel à la représentation hexadécimale pour traiter les motifs binaires, mais certains cependant utilisent le système octal.

D'après "Programmation en C". B.S. Gottfried

Annexe B

Séquences d'échappement

Caractère	Séquence d'échappement	Code ASCII
sonnerie (alarme)	\a	007
retour arrière	\b	008
tabulation horizontale	\t	009
tabulation verticale	\v	011
retour à la ligne (LF)	\n	010
nouvelle page *(FF)*	\f	012
retour chariot	\r	013
guilIemets (")	\"	034
apostrophe (')	\'	039
point d'interrogation	\?	063
barre fraction inv.(\)	\\	092
caractère nul	\0	000
nombre octal	`\ooo` (où o représente un chiffre octal)	

En général, la représentation est limitée à trois chiffres en mode octal:
Exemples: \5 \005 \123 \177
nombre hexadécimal `\xhh` (où `h` représente un chiffre hexadécimal)
En général. le nombre de chiffres est indifférent en mode hexadécimal:
Exemples: \x5 \x05 \x53 \x7f

La plupart des compilateurs autorisent l'emploi de l'apostrophe (') et du point d'interrogation (?) dans les chaînes, exprimés aussi bien comme caractère ordinaire que comme séquence d'échappement.

D'après "Programmation en'C ", B.S. Gottfried

Annexe C

Les opérateurs du langage C

Catégories d'opérateurs	Opérateurs	Associativité
fonction, tableau, membre de structure, pointeur sur un membre de structure	( ) [ ] . ->	G → D
opérateurs unaires	- ++ -- ! ~ * & sizeof (type)	D → G
multiplication, division, modulo	* / %	G → D
addition, soustraction	+ -	G → D
opérateurs binaires de décalage	<< >>	G → D
opérateurs relationnels	< <= >= >	G → D
opérateurs de comparaison	== !=	G → D
et binaire	&	G → D
ou exclusif binaire	^	G → D
ou binaire	|	G → D
et logique	&&	G → D
ou logique	||	G → D
opérateur conditionnel	? :	D → G
opérateurs d'affectation	= += -= *= /= %=	D → G
opérateur virgule	,	G → D

Les groupes sont listés ici par ordre décroissant de priorité. Certains compilateurs C comprennent également un plus unaire (+) complémentaire de l'opérateur moins unaire (-). La valeur d'une expression formée d'un plus unaire est cependant celle de son opérande. autrement dit +v a la même .-que v.

D'après "Programmation en C ", B.S. Gottfried

Annexe D

Types de données et règles de conversion

Type de données	Description	Occupation mémoire-type
int	Donnée entière	2 octets ou 1 mot (selon l'ordinateur)
short	Donnée de type entier court (peut contenir moins de chiffres que le type `int`)	2 octets ou 1 mot (selon l'ordinateur)
long	Donnée de type entier long (peut contenir plus de chiffres que le type `int`)	1 ou 2 mots (selon l'ordinateur)
unsigned	Donnée de type entier non signé (valeur positive ou nulle pouvant être . approximativement deux fois plus élevée qu'une valeur de type `int`)	2 octets ou 1 mot (selon l'ordinateur)
char	Caractère simple	1 octet
signed char	Caractère simple dont la valeur numérique varie de - 128 à + 1 27	1 octet
unsigned char	Caractère simple dont la valeur numérique varie de 0 à 255	1 octet
float	Nombre réel (c'est-à-dire contenant un point décimal et un exposant)	1 mot
double	Nombre réel en double précision (c'est-à-dire ayant plus de chiffres significatifs et/ou un exposant éventuellement plus grand)	2 mots
long double	Nombre réel en double précision de grande amplitude (peut prendre des valeurs plus élevées que le type `double`)	2 mots ou plus (selon l'ordinateur)
void	Type de données propre aux fonctions ne retournant aucune valeur	(sans objet)
enum	Constante d'énumération (type particulier de `int`)	2 octets ou 1 mot (selon l'ordinateur)

Note : le qualificateur `unsigned` peut figurer en complément des types `short int` et `long int` ; en outre, `unsigned short int` et `unsigned long int` peuvent être abrégés sous la forme `unsigned short` et `unsigned long` respectivement.

XII.I.1.1 RÈGLES DE CONVERSION

Ces règles s'appliquent aux opérations arithmétiques ayant deux opérandes de même type. Elles peuvent connaître de légères modifications sur certains compilateurs.

1. Si l'un des deux opérandes est de type `long double`, le second est converti dans ce même type ainsi que le résultat de l'opération.
2. Sinon. si l'un des opérandes est de type `double`. le second est converti en `double` ainsi que le résultat.
3. Sinon. si l'un des opérandes est de type `float`, le second est converti en `float` ainsi que le résultat.
4. Sinon. si l'un des opérandes est de type `unsigned long int`, le second est converti en `unsigned long int` ainsi que le résultat.
5. Sinon. si l'un des opérandes est de type `long int` et le second de type `unsigned int`, alors :
   (a) Si la valeur de type `unsigned int` peut être convertie en `long int`, l'opérande de ce type et le résultat ont le type `long int`.
   (b) Dans le cas contraire, les deux opérandes sont convertis en `unsigned long int` ainsi que le résultat.
6. Sinon. si l'un des opérandes est de type `long int`, le second ainsi que le résultat sont convertis en `long int`.
7. Sinon, si l'un des opérandes est de type `unsigned int` le second ainsi que le résultat sont convertis en `unsigned int`.
8. Si les opérandes ne se trouvent dans aucune des situations ci-dessus. ceux-ci sont convertis si nécessaire en `int` et le résultat est alors une valeur de type `int`.

Certains compilateurs C effectuent automatiquement la conversion d'opérandes de type réel en réels en double précision.

XII.I.1.2 RÈGLES D'AFFECTATION

Lorsque les deux opérandes d'une expression d'affectation sont d'un type de données différent, la valeur de l'opérande à droite de l'opérateur est convertie automatiquement dans le type de la donnée de l'opérande receveur (celui de gauche). L'expression dans son ensemble prend également le type de l'opérande de gauche. D'autre part,

1. Une valeur réelle est éventuellement tronquée lorsqu'elle est affectée à un identificateur de type entier.
2. Une valeur en double précision est éventuellement arrondie lorsqu'elle est affectée à un identificateur de type réel.
3. Une valeur entière est éventuellement altérée lorsqu'elle est affectée à un identificateur d'un type entier plus court. ou de type caractère (certains des bits de poids fort sont alors perdus).

D'après "Programmation en C ", B.S. Gottfried

Annexe E

Le jeu de caractères ASCII

Code ASCII	Caractère	Code ASCII	Caractère	Code ASCII	Caractère	Code ASCII	Caractère
0	NUL	32	espace	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	S	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	1
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Les 32 premiers caractères ainsi que le dernier sont des caractères de contrôle. En règle générale, ils ne sont pas affichables

Annexe F

Structures de contrôle

Structure	Syntaxe	Exemple

break	break;	for (n = 1; n <= 100: ++n) {
		scanf("%f", &x);
		if (x < 0) {
		printf("ERREUR - X NE PEUT ETRE NEGATIF");
		break;
		}
		...
		}

continue	continue;	for (n = 1; n <= 100: ++n) {
		scanf("%f", &x);
		if (x < 0) {
		printf("ERREUR - X NE PEUT ETRE NEGATIF");
		continue;
		}
		...
		}

do-while	do
	instruction
	while (expression);
		do
		printf("%d\n",chiffre++);
		while (chiffre <= 9);

for	for (exp1; exp2; exp3) instruction	for (chiffre = 0; chiffre <= 9; ++chiffre) printf("%d\n",chiffre)

goto	goto drapeau;
	drapeau: instruction
		if (x < 0)
		goto drapeau;
		...
		drapeau: printf("ERREUR");

if	if (expression) instruction	if (x < 0) printf("%f", x);

Structure	Syntaxe	Exemple

if-else	if (expression)
	  instruction1
	else
	  instruction2
		if (statut = 'S')
		  cotisation = 0.20 * salaire;
		else
		  cotisation = 0.14 * salaire;

return	return(expression)	return(n1 + n2);

switch	switch(expression) {
	
	case expressionl:
	  instructionl
	  instruction2
	  ...
	  instructionk
	  break;
	
	case expression2:
	  instructionl
	  instruction2
	  ...
	  instructionp
	  break;
	
	...
	
	default:
	  instructionl
	  instruction2
	  ...
	  instructionq
	}
		switch (choix = getchar()) {
		
		case 'R':
		  printf ("ROUGE");
		  break;
		
		case 'V':
		  printf ("VERT");
		  break;
		
		case 'B':
		  printf ("BLEU");
		  break;
		
		default:
		  printf("ERREUR");
		}

while	while (expression)
	  instruction
		while (chiffre <= 9)
		  printf("%d\n",chiffre++);

D'après "Programmation en C", B.S.Gottfried

Annexe G

Caractères de contrôle usuellement employés dans les fonctions `scanf` et `printf`

Caractères de contrôle de `scanf`

Caractère de contrôle	Signification
c	Donnée de type caractère simple
d	Donnée de type entier décimal
e	Donnée de type réel
f	Donnée de type réel
g	Donnée de type réel
h	Donnée de type entier court
i	Donnée de type entier décimal. octal ou hexadécimal
o	Donnée de type entier octal
s	Donnée de type chaîne suivie d'un espace (ajout du caractère nul, '\0', en bout de chaîne)
u	Donnée de type entier décimal non signé
x	Donnée de type entier hexadécimal
[ ]	Donnée de type chaîne, pouvant contenir des espaces

Un *préfixe* peut être placé avant certains caractères de contrôle:

Préfixes

Préfixe	Signification
h	Donnée de type court (entier court ou entier court non signé)
l	Donnée de type long (entier long, entier long non signé ou réel en double précision)
L	Donnée de type long (entier long)

D'après "Programmation en C ". B.S. Gottfried

Caractères de contrôle de `printf`

Caractère de contrôle	Signification
c	Donnée restituée sous forme de caractère simple
d	Donnée restituée sous forme d'entier décimal signé
e	Donnée restituée sous forme de réel avec un exposant
f	Donnée restituée sous forme de réel sans exposant
g	Donnée restituée sous forme de réel respectant les spécifications de type e ou f selon leur valeur ; les zéros et points décimaux non significatifs ne sont pas affichés
i	Donnée restituée sous forme d'entier décimal signé
o	Donnée restituée sous forme d'entier octal non précédé de 0
s	Donnée restituée sous forme de chaîne
u	Donnée restituée sous forme d'entier décimal non signé
x	Donnée restituée sous forme d'entier hexadécimal non précédées de 0x

L'interprétation de certains de ces caractères est différente de celle de la fonction scanf.
Un *préfixe* peut être placé avant certains caractères de contrôle:

Indicateurs

Indicateur	Signification
-	Donnée justifiée à gauche dans le champ (les espaces nécessaires au remplissage du champ sont ajoutés *après* la donnée et non *avant).*
+	Toute donnée numérique est précédée d'un signe (+ ou -) : sans cet indicateur, seules les données négatives sont précédées de leur signe.
0	Bourrage du début du champ par des zéros au lieu d'espaces: porte uniquement sur des données justifiées à droite dans un champ dont la largeur minimum est supérieure à la longueur de la donnée. *(Remarque:* certains compilateurs considèrent que l'indicateur fait partie intégrante du spécificateur du champ: cette convention permet d'appliquer l'effet du zéro en dernier lorsque figurent plusieurs indicateurs).
' ' (espace)	Permet de faire explicitement précéder une valeur positive d'un espace: l'effet de cet indicateur est annulé par l'indicateur + lorsque celui-ci est également présent.
# (utilisé avec les spécificateurs `o` et `x`)	Affichage de 0 et 0x respectivement avant toute valeur octale ou hexadécimale.
# (utilisé avec les spécificateurs `e, f` et `g`)	Affichage du point décimal de toutes les valeurs réelles. même lorsque celles-ci correspondent à un nombre entier ; annule par ailleurs la suppression des zéros non significatifs dans une spécification de type g.

D'après "Programmation en C ", B.S. Gottfried