Programmation en C

Gérald MONARD

D.E.A. C.I.T. - Année Universitaire 2004-2005

Bibliographie

– http ://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html
– *Le langage C* par B.W. Kernighan et D.M. Ritchie, Masson Editions (1992).
– *Le langage C* par T. Zhang, CampusPress Editions (1997).
– *Sams Teach Yourself C in 21 Days* par B.L. Jones et P. Aitken, Sams Publishing (2003).
– *Practical C Programming* par S. Oualline, O'Reilly Editions (1997).

Contacts

Gérald MONARD
UMR 7565 - Equipe de Chimie et Biochimie Theoriques
Universite Henri Poincare - Nancy I
Faculte des Sciences - BP 239
54506 Vandoeuvre-les-Nancy Cedex, FRANCE

E-mail : Gerald.Monard@lctn.uhp-nancy.fr
tél. : 03.83.68.43.81
fax. : 03.83.68.43.71
www : http ://gerald.monard.free.fr

Table des matières

1 Le langage C

1.1 Introduction

Les ordinateurs sont des appareils électroniques permettant d'effectuer des opérations de base sur les entiers et les réels (flottants). Grosso-modo, ce sont de grosses calculatrices.

Ce qui est intéressant sur un ordinateur est qu'il est *programmable* : si on donne une suite d'instructions (*le programme*) à *exécuter* à un ordinateur, celui-ci l'exécute.

L'ordinateur est donc une sorte d'*esclave* qui fait exactement ce qu'on lui dit, qui n'a pas besoin de nourriture (autre qu'électrique) et ne fait jamais grève.

Le principal avantage et inconvénient d'un ordinateur est qu'il fait ce qu'on lui demande et seulement ce qu'on lui demande. Ainsi quand la réponse d'un ordinateur ne convient pas à l'utilisateur, il faut chercher le problème dans le programme qu'il lui a été donné et non dans l'ordinateur lui-même (*i.e.*, c'est le programmateur qui a tort, et non l'ordinateur qui a appliqué bêtement le programme qui lui a été assigné).

Un ordinateur ne reconnait que le langage *binaire* et les programmes binaires.

L'homme a d'énormes difficultés à parler un langage binaire !

Les langages de programmation permettent d'effectuer le lien entre l'homme et l'ordinateur : un programme est écrit dans un idiome (assez facilement) compréhensif par l'homme puis est traduit (par l'intermédiaire d'un interpréteur ou d'un compilateur) en une séquence d'instructions (binaires) directement reconnaissables par l'ordinateur.

Il existe deux sortes de langages : les langages *interprétés* et les langages *compilés* (= traduits).

Les *langages interprétés sont directement traduits et exécutés*. Les instructions sont converties séquentiellement au format binaire par un interpréteur puis exécutées immédiatement.

Les *langages compilés utilisent un compilateur* qui traduit le programme en langage binaire. Une fois le langage compilé, il est possible de le *stocker dans un fichier exécutable*. Le compilateur n'intervient qu'au moment de la création du code binaire. Le programme compilé peut s'exécuter sans avoir recours au compilateur.

On parle de *code source* pour désigner les instructions du programme en format texte compréhensible, et de *code executable* pour désigner les instructions du programme en langage binaire directement exécutable par l'ordinateur. Tout fichier source mis à jour doit être à nouveau compilé pour recréer le fichier exécutable correspondant.

Le langage C est l'un des tout premiers langages dits de *haut niveau* (contrairement à l'assembleur par exemple). Il a été conçu pour de nombreuses utilisations et est caractérisé par une économie d'expression, par des instructions de contrôle et des structures de données modernes, ainsi que par un ensemble très complet d'opérateurs.

A l'origine, le langage C a été inventé par Dennis Ritchie au sein des laboratoires Bell pour le système d'exploitation UNIX en 1972 (DEC PDP-11) puis a été utilisé pour *"recréer"* UNIX, le compilateur C associé et la plupart des programmes d'application sous UNIX.

Le langage C est un langage compilé.

1.2 Pourquoi C

- C est *puissant et flexible* : le langage par lui-même possède très peu de contraintes et donc il est possible d'en faire pratiquement ce que l'on veut. C est utilisé pour des projets aussi divers que des systèmes d'exploitation, des traitements de textes, du graphisme, des tableurs, et même des compilateurs pour d'autres langages
- C est l'un des langages les plus *populaires* chez les informaticiens. De ce fait, il existe de nombreux compilateurs disponibles ainsi que de nombreux utilitaires.
- C est un langage *portable*. C'est-à-dire qu'un programme écrit en C sur un ordinateur pourra être exécuté sans modification (ou presque) sur un autre ordinateur d'architecture totalement différente.
  La portabilité est facilité par la norme ANSI (American National Standard Institute).
- C est modulaire. Un code source écrit en C peut être divisé en routines (les *fonctions*). Ces routines peut alors être réutilisées par d'autres applications.

1.3 Les autres langages courants en informatique

Quelques langages courants que vous rencontrerez peut-être au cours de votre stage, de votre thèse, et/ou de votre vie professionelle :

Fortran LE langage scientifique par excellence. Existe depuis les années 1960s.
De très nombreux logiciels en chimie quantique et en mécanique moléculaire ont été bâtis en Fortran.

C++ Une extension récente de C permettant la programmation *Orientée Objets*.
Hors programme ...

Java Une sorte de C++ réécrit plus proprement, moins flexible et propriétaire (Sun Microsystems).
Avantage : une très large bibliothèques de modules (réseau, Internet, graphisme, ...).

Perl Un langage récent, populaire, permettant le traitement rapide de fichiers.
Plutôt un langage pour les systèmes d'exploitation qu'un langage *"scientifique"*.
Slogan : "There is more than one way to do it" ⟶ Langage non pédagogique !

Python Un langage récent, évolué, interprété, orienté objet, de plus en plus populaire. Il a pour particularité de pouvoir s'interfacer très facilement avec C, Fortran, Java, *etc*.

2 Premier programme en C

Voici le premier programme en C :

```c
#include <stdio.h>

int main()
{
   printf("Hello a tous\n");
   return(0);
}
```

Compilation et exécution

```
$ cc -O -Wall hello1.c -o hello1
$ ./hello1
```

cc désigne le compilateur C

-O -Wall sont des options du compilateur C qui permettent (`-O`) d'optimiser le code executable (*i.e.*, le rendre plus rapide à l'exécution) et (`-Wall`) d'indiquer à l'écran tout problème possible dans le code source (*Warning all*).

-o hello1 sauvegarde le résultat de la compilation (l'exécutable) dans le fichier `hello1`.

Les fichiers sources en langage C se terminent généralement par .c.

Les commentaires commencent par /* et se terminent par */.

```c
#include <stdio.h>

/* ceci est un commentaire */
int main()
{
   /* un commentaire
      peut etre present
      sur plusieurs lignes */
   printf("Hello a tous\n");
   return(0);
}
```

`#include <stdio.h>` correspond à l'inclusion des définitions de la bibliothèque standard de fonctions d'entrées/sorties : On veut afficher quelque chose à l'écran, ceci peut être assez compliqué pour l'ordinateur (où dans l'écran, quelles fontes, couleurs, etc.), on fait alors appel à des fonctions pré-programmées dans l'ordinateur pour gérer cela à notre place ; parmi toutes les fonctions préprogrammées existantes, celles concernant les entrées et les sorties standards (*i.e.*, à l'écran) sont regroupées dans une *bibliothèque* de fonctions appelées `stdio` (*standard input/output*). On inclut les définitions de ces fonctions pour pouvoir les appeler plus loin.

`int main()` correspond à la définition du programme principal.

Tout programme en langage C contient un `main` et un seul. C'est par cette fonction que démarre toujours l'exécution d'un programme en langage C. Le programme se termine lorsque l'exécution de cette fonction est terminée.

Le `int` correspond au type (entier) du code de retour d'erreur du programme exécuté (cf. UNIX).

`printf ("Hello a tousn");` affiche `Hello a tous` à l'écran (sortie standard).

Le `\n` permet un retour chariot à l'écran.

`printf` est une fonction définie dans `stdio` qui permet l'affiche de chaîne de caractères (`"..."`) à l'écran.

`return (0);` indique la fin du programme principal. Tout s'est en principe bien déroulé, on retourne donc le code d'erreur `0` pour indiquer qu'il n'y a pas de problème. Ce code d'erreur est lu par le système d'exploitation.

Toute instruction en C se terminent par `;` .

3 Concepts fondamentaux

Un programme C se compose d'éléments de base comme :
– les expressions
– les instructions
– les blocs d'instructions
– les blocs de fonctions
– *etc.*

3.1 Constantes et variables

Une constante est un valeur non modifiable. Une variable peut contenir des valeurs différentes

```
1 i = 1;
2 i = 10;
```

`1` et `10` sont des constantes. `i` est une variable qui prend ici successivement les valeurs 1 et 10.

`=` est l'opérateur d'*affectation*.

3.2 Expressions

Une expression est une combinaison de constantes, de variables et d'opérateurs utilisés dans des calculs mathématiques.

```
1 (2+3)*10
```

autres exemples :

```
1 6
2 i
3 6+i
```

3.3 Opérateurs arithmétiques

Une expression peut contenir des symboles appelés *opérateurs arithmétiques* en langage C.

Liste des opérateurs :

+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo

Les opérateurs de multiplication et de division sont prioritaires par rapport à l'addition et à la multiplication.

Pour modifier l'ordre de priorité, on peut utiliser les parenthèses ( ).

```
2+3*10
```

est différent de

```
(2+3)*10
```

3.4 Instructions

En langage C, une instruction est une ligne exécutable, terminée par un point-virgule.

Exemple :

```
i = 1;
printf("Hello a tous\n");
return(0);
```

sont des instructions.

3.5 Blocs d'instructions

Les instructions peuvent être regroupées en blocs compris entre deux accolades : l'une dite d'ouverture ({), l'autre dite de fermeture (}). *Un bloc d'instructions est traité comme une instruction simple par le compilateur C.*

```
{
    i = 2;
    j = i*10;
}
```

3.6 Déclarations

Il faut déclarer toutes les variables avant de s'en servir. Une déclaration précise un type et comporte une liste de une ou plusieurs variables de ce type. Par exemple :

```
int mini, maxi, intervalle;
char c;     /* on peut mettre un commentaire pour indiquer
               à quoi sert la variable */
```

On peut également initialiser les variables au moment où on les déclare. Si le nom de variable est suivi du signe = et d'une expression, cette expression donne la valeur initiale de la variable.

```
int i = 0;
float eps = 1.0e-5;
char a = 'a';
```

4 Noms et types de données

Le langage C comporte très peu de types de base : `char`, `int`, `float`, `double`.

On peut également appliquer un certain nombre de qualificatifs à ces types de base, comme par exemple `short` ou `long` pour les entiers.

4.1 Type `char`

Un objet de type `char` correspond à un élément du jeu de caractères utilisé par l'ordinateur.

Un caractère compris entre deux guillemets simples (`'`) s'appelle une *constante caractères*. Par exemple, `'a'`, `'A'`, `'b'`, `'7'` sont des constantes caractères du jeu de caractères ASCII.

Il existe un correspondance entre chaque caractère et un code numérique unique (compris entre 0 et 255). Par exemple, les caractères `'A'`, `'a'`, `'B'`, et `'b'` ont respectivement comme valeur numérique unique 65, 97, 66 et 98. Ainsi les deux instructions d'affectation suivantes sont équivalentes :

```
1 char x;
2 x = 'A';
3 x = 65;
```

En langage C, le caractère d'échappement (`\`) est toujours suivi d'un caractère spécial. Par exemple `\n` correspond au retour chariot et au passage à une nouvelle ligne.

Exemple :

Caractère	Description
`\n`	Retour chariot : passage à une nouvelle ligne
`\t`	Tabulation : place le curseur au niveau de la tabulation suivante
`\r`	Caractère de début de ligne : replace le curseur en début de ligne
`\\`	insère le caractère `\`
`\"`	insère le caractère `"`

etc.

4.2 Type `int`

Le mot-clé `int` correspond au type entier applicable aux variables. Un nombre entier est une valeur sans partie décimale. En conséquence, une division entière a comme effet de tronquer le résultat par rapport à la valeur décimale.

La longueur d'un entier dépend directement du système d'exploitation et du compilateur C utilisés.

Codage	valeur mini.	valeur maxi.
32 bits (4 octets)	-2147483648	2147483647
16 bits (2 octets)	-32768	32767

4.3 Type `float`

A la différence d'un nombre entier, un nombre à virgule flottante comprend une partie décimale dissociée de la partie entière par une virgule. En C, le type `float` définit les nombres décimaux, aussi appelés *nombres réels*. En général, la précision du type `float` est de sept chiffres après la virgule.

4.4 Type `double`

En C, un nombre à virgule flottante peut aussi être de type `double`. En général, la précision du type `double` est de quinze chiffres après la virgule.

4.5 Notation scientifique

Les valeurs décimales peuvent être exprimées à l'aide de la notation scientifique.

Ainsi un nombre se composera d'une mantisse et d'un exposant, séparés indifféremment par `e` ou `E`.

Ainsi, 5000 peut s'écrire `5e3`, -300 est égal à `-3e2`, 0.0025 à `2.5e-3`.

4.6 La fonction `printf`

La fonction printf de la bibliothèque standard <stdio.h> prend comme arguments une chaîne de caractère pouvant contenir des *indicateurs de formats*.

Par exemple, dans :

```
1    printf(" Voici l'entier deux : %d\n",2);
```

L'indicateur de format %d permet d'afficher l'entier 2.

De manière générale, printf s'occupent de mettre en forme les données en sortie.

```
1 printf(const char *format, ...);
```

L'argument const char *format contient deux types d'objets : des caractères ordinaires et des indicateurs de format.

Les indicateurs de format commencent par % et se terminent par un caractère de conversion.

Entre les deux, on peut placer dans l'ordre :
- des drapeaux
  - `-` cadre l'argument à gauche dans son champ d'impression
  - `+` imprime systématiquement le signe du nombre
  - `espace` si le premier caractère n'est pas un signe, place un espace au début
  - `0` pour les conversions numériques, complète le début du champ par des zéros
- un nombre, qui précisent la largeur minimum du champ d'impression
- un point, qui sépare la largeur du champ de la précision désirée
- un nombre qui donne la *précision* (voir exemple) que ce soit pour une chaîne, un entier, un flottant (`f`, `e`, `E`, `g`, ou `G`)

Indicateurs	Description
%d %i	Format entier
%c	Format caractère
%f	Format décimal (virgule flottante)
%e %E	Format de notation scientifique
%g	utilise %f ou %e selon la longueur de la valeur décimale
%G	utilise %f ou %E selon la longueur de la valeur décimale
%o	Format octal non signé
%s	Format chaîne de caractère
%u	Format entier non signé
%x %X	Format hexadécimal non signé
%p	Affichage du pointeur d'argument
%n	Nombre de caractères écrits
%%	%

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("%d\n", 1);
6     printf("%f\n",0.10);
7     printf("%e\n",0.10);
```

```c
    printf("%E\n",0.10);
    printf("%g\n",0.10);
    printf("%G\n",0.0000010);

    printf("12345 12345 12345\n");
    printf("%5d %-5d %d\n",1,1,1);
    printf("%5d %-5d %d\n",10,10,10);
    printf("%5d %-5d % 5d\n",-10,-10,-10);
    printf("%+5d %-5d % 5d\n",10,10,10);
    printf("%5d %-5d %d\n",1000,1000,1000);
    printf("%5d %-5d %d\n",1000000,1000000,1000000);

    printf("%10.4f\n",123.456789);

    printf("%s\n","toto");
    printf("%10s\n","toto");
    printf("%-10s\n","toto");
    return(0);
}
```

5 Opérateurs

En plus de opérateurs standards vus précedemment (`+`, `-`, `*`, `/`, `%`) le langage C possède d'autres opérateurs plus évolués :
– des opérateurs arithmétiques d'affectation
– des opérateurs unaires
– des opérateurs d'incrémentation et de décrémentation
– des opérateurs relationnels
– des opérateurs de conversion de type

5.1 Opérateurs arithmétiques d'affectation

C'est l'opérateur `=`, à ne pas confondre avec le '=' mathématique.

Par exemple, `a=5` signifie que l'on affecte à la variable `a` la valeur 5. On ne peut pas écrire en langage C `5=a` .

Il existe en langage C des opérateurs d'opérations-affectations qui regroupe les deux propriétés. Ce sont les opérateurs `+=`, `-=`, `*=`, `/=`, et `%=`.

Ainsi `x+=y` équivant à `x=x+y` , *etc.*.

5.2 Opérateurs unaires

Un opérateur unaire en C est l'opérateur `-` : *moins*, qui transforme une valeur en son opposé.

```c
x = -1,234;
y = -x;
z = x - -y; /* équivalent à z = x - (-y);
               équivalent à z = x + y; */
```

5.3 Opérateurs d'incrémentation et de décrémentation

Le langage C propose une solution simple au cas où l'on veut incrémenter ou décrémenter de 1 une valeur entière : l'opérateur `++` ou l'opérateur `--`.

Il existe deux façons en C d'incrémenter (respectivement de décrémenter) 1 à une variable `x` : `x++` (resp. `x--`) ou `++x` (resp. `--x`).

Les instructions suivantes sont équivalentes :

```
x = x + 1;
x += 1;
x++;
++x;
```

Dans le cas de l'association des opérateurs `++` ou `--` et une affection, on distingue `--x` qui correspond à une pré-incrémentation (l'incrémentation a lieu avant l'affectation), de `x++` qui correspond à une post-incrémentation (l'incrémentation a lieu après l'affectation)..

Exemple :

```
#include <stdio.h>

int main()
{
   int a,b,c,d,result;
   a = 1;
   b = 1;
   c = 1;
   d = 1;
   result = a++;
   print(" Resultat de a++ : %d\n",result);
   result = ++b;
   print(" Resultat de ++b : %d\n",result);
   result = c--;
   print(" Resultat de c-- : %d\n",result);
   result = --d;
   print(" Resultat de --d : %d\n",result);
   return(0);
}
```

5.4 Opérateurs de conversion de types

Il est possible de convertir en C un type en un autre de type en faisant précéder l'opérande de l'opérateur de conversion.

La syntaxe générale est

```
(type) x
```

où `type` correspond au type de données cibles et x est une variable (ou une expression) contenant la valeur du type en cours.

Par exemple (float) 5 convertit l'entier `5` en nombre décimal de type float .

Exemple :

```
#include <stdio.h>

int main()
{
   int x, y;
   x = 7;
   y = 5;
   printf("x = %d et y = %d\n",x,y);
   printf("x / y = %d\n", x /y);
   printf("(float) x/y = %f\n", (float)x/y);
   return(0);
}
```

5.5 La fonction `sizeof`

Le langage C fournit en standard une fonction permettant de déterminer la taille d'un type de données : la fonction sizeof .

```c
#include <stdio.h>

int main()
{
    printf("Taille d'un int : %d\n", sizeof(int));
    printf("Taille d'un char : %d\n", sizeof(char));
    printf("Taille d'un float : %d\n", sizeof(float));
    printf("Taille d'un double : %d\n", sizeof(double));
    return(0);
}
```

6 Fonctions

Les fonctions constituent la structure des programmes C. Outre les fonctions de bibliothèque, il est possible de définir et d'utiliser des fonctions personnalisées.

```c
int add_entiers(int x, int y)
{
    int resultat;
    resultat = x+y;
    return(resultat);
}
```

6.1 Type d'une fonction

Le type d'une fonction correspond au type de la valeur renvoyée par la fonction (dans l'exemple, add_entiers est de type int).

Si une fonction ne renvoie aucune donnée, alors elle est de type void .

```c
void affiche(int a, float y)
{
    printf("Voici un entier : %d\n", a);
    printf("Voici un flottant : %f\n", y);
}
```

6.2 Nom d'une fonction

Le nom d'une fonction doit être le plus explicite possible afin d'éclaircir l'utilisateur sur le traitement réalisé par la fonction.

6.3 Arguments d'une fonction

Il est parfois nécessaire de transmettre des informations que la fonction traite lors de son exécution. Ces informations s'appellent les *arguments*. L'argument figure entre parenthèses juste après le nom de la fonction qui va l'exploiter.

Le nombre d'arguments varie selon le traitement réalisé. Les arguments transmis à une même fonction sont séparés par des virgules.

Si une fonction ne requiert pas d'arguments, les parenthèses sont alors vides.

6.4 Début et fin d'une fonction

Les accolades d'ouverture et de fermeture marquent le début et la fin d'une fonction.

Elles permettent aussi de délimiter un bloc d'instructions. En fait, ces marques indiquent que la séquence contient plusieurs instructions.

6.5 Corps d'une fonction

Il s'agit de l'emplacement compris entre les deux accolades et contenant déclarations de variables et instructions qui s'exécutent séquentiellement lors de l'appel de la fonction.

6.6 Appel d'une fonction

Les fonctions ne représentent que des définitions informatiques d'instructions à traiter, elles ne sont exécutées que si on les appelle.

Une fonction doit toujours être déclarée avant d'être utilisée.

```c
#include <stdio.h>

void affiche(int a, float y)
{
    printf("Voici un entier : %d\n", a);
    printf("Voici un flottant : %f\n", y);
}

int add_entiers(int x, int y)
{
    int resultat;
    resultat = x+y;
    return(resultat);
}

int main()
{
    int a;
    int b;
    int c;

    a = 5;
    b = 7;
    c = add_entiers(a, b);
    affiche(c,3.56);
    a = add_entiers(c, a);
    return(0);
}
```

7 Pointeurs

Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable.

Un ordinateur classique possède un tableau de cases mémoire consécutives numérotées, ou adressées, qui l'on peut manipuler individuellement ou par groupes de cases contiguës. L'instruction sizeof vu précédemment permet par exemple de connaître combien d'octets contiguës dans la mémoire prend une variable.

Chaque variable stockée en mémoire possède une adresse : l'endroit où elle est stockée. Cette adresse peut être connue en employant *l'opérateur adresse* &.

```c
/* ne pas confondre adresse et contenu */
int n;
int m;
n = 5;
m = 2;
printf(" n vaut %d, son adresse est %p\n", n, &n);
printf(" m vaut %d, son adresse est %p\n", m, &m);
```

```
n = 10;
printf(" n vaut %d, son adresse est %p\n", n, &n);
```

Une variable pointeur ne possède pas de type particulier. En réalité on ajoute le caractère * à un type défini pour déclarer une variable pointeur ainsi que le type de variable quelle adresse.

Ainsi on ne déclare pas un pointeur, mais ce qu'il contient.

```
int entier;         /* variable entiere */
int *pointeur;      /* pointeur sur un entier */

entier = 5;

printf(" entier = %d, son adresse est %p\n", entier, &entier);
printf(" pointeur = %p, son adresse est %p, il pointe sur %d\n",
       pointeur, &pointeur, *pointeur);

pointeur = &entier /* pointeur contient l'adresse de entier */

printf(" pointeur = %p, son adresse est %p, il pointe sur %d\n",
       pointeur, &pointeur, *pointeur);

*pointeur = 10;     /* on change la valeur que pointe pointeur,
                       donc on change la valeur de entier */
printf(" entier = %d, son adresse est %p\n", entier, &entier);
printf(" pointeur = %p, son adresse est %p, il pointe sur %d\n",
       pointeur, &pointeur, *pointeur);
```

Remarques : Il est possible d'employer plusieurs pointeurs qui pointent sur la même variable. On peut même définir des pointeurs de pointeurs ...

```
int a;
int *prt_a, *prt_b;  /* pointeur sur des entiers */
int **prt_prt;       /* pointeur sur un pointeur */
a = 5;
prt_a = &a;
prt_b = &a;
prt_prt = &prt_a;
printf("Toujours le meme entier !\n");
printf(" a = %d\n", a);
printf(" prt_a = %d\n", *prt_a);
printf(" prt_b = %d\n", *prt_b);
printf(" prt_prt = %d\n", **prt_prt);
```

7.1 Pointeur NULL

Un pointeur peut prendre une valeur particulière signifiant qu'il ne porte à aucune adresse définie dans la mémoire : c'est l'adresse NULL définie dans la librairie <stdlib.h>. Ainsi, il est possible d'initialiser un pointeur "à zéro" avant de l'utiliser.

```
int a = 5;
int *prt_a = NULL;
prt_a = &a;
```

8 Pointeurs et fonctions

Si on déclare une fonction de la manière suivante : void fonction(type var), cette fonction utilise un argument de type type que l'on nomme tout le long de la définition var. Lors de l'appel de la fonction (f(b)), l'ordinateur

recopie le contenu de la variable b (de type type) dans une case mémoire représentant l'argument var tel qu'il est défini dans fonction .

Si lors de l'exécution de fonction la valeur de var est modifié, ce n'est que var qui est modifiée (*i.e.*, la copie locale de b) et non b elle-même.

Exemple :

```c
#include <stdio.h>

void fonction(int var)
{
    printf("Fonction, avant modification, var = %d\n", var);
    var = 3;
    printf("Fonction, apres modification, var = %d\n", var);
}

int main()
{
    int b;
    b = 2;
    printf("Main, avant appel a fonction, b = %d\n", b);
    fonction(b);
    printf("Main, apres appel a fonction, b = %d\n", b);
    return(0);
}
```

Pour pouvoir modifier la valeur de b **dans** `fonction` **, il faut utiliser des pointeurs.** Au lieu transmettre b, on transmet l'adresse de b comme argument de la fonction `fonction`. Ainsi, la copie locale `var` dans `fonction` contient une adresse, l'adresse de b. On peut alors modifier le contenu pointé grâce à l'opérateur `*`.

```c
#include <stdio.h>

void fonction(int *ptr_var)
{
    printf("Fonction, avant modification, *ptr_var = %d\n", *ptr_var);
    *ptr_var = 3;
    printf("Fonction, apres modification, *ptr_var = %d\n", *ptr_var);
}

int main()
{
    int b;
    b = 2;
    printf("Main, avant appel a fonction, b = %d\n", b);
    fonction(&b);
    printf("Main, apres appel a fonction, b = %d\n", b);
    return(0);
}
```

9 Booléens et opérations sur les booléens

En langage C, il n'existe pas vraiment de type booléen. Pour représenter un booléen, on utilise un entier (type int) : si celui-ci vaut `0` alors il est équivalent à `faux`, sinon, quelque soit sa valeur $\neq 0$, il est considéré comme `vrai`.

On associe au pseudo-type booléen en C des opérateurs :

Opérateur C	Description
||	ou logique
&&	et logique
!	non logique

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

```c

int main()
{
    int v = 1;
    int f = 0;
    printf("vrai ou faux : v || f : %d\n", v || f);
    printf("vrai et faux : v &&  f : %d\n", v && f);
    printf("non vrai     :   !v   : %d\n", !v);
    printf("non faux     :   !f   : %d\n", !f);
    return(0);
}
```

De plus il existe des opérateurs de comparaison qui permettent de comparer deux expressions à la fois :

Opérateur C	Description
==	égal à
!=	non égal à
>	supérieur à
<	inférieur à
>=	supérieur ou égal à
<=	inférieur ou égal à

```c
#include <stdio.h>

int main()
{
    int x, y;
    double z;
    x = 7;
    y = 25;
    z = 24.46;
    printf(" x = %d, y = %d, z = %f\n");
    printf(" x >= y : %d\n", x >= y);
    printf(" x == y : %d\n", x == y);
    printf(" x <  z : %d\n", x <  z);
    printf(" y >  z : %d\n", y >  z);
    printf(" x != y - 18 : %d\n", x != y - 18);
    printf(" x + y != z : %d\n", x + y != z);
    return(0);
}
```

10 Instructions conditionnelles

10.1 Instructions `if` et consorts

Il est possible en C (comme dans tous les langages évolués) d'exécuter des instructions sous conditions.

Par exemple, la racine carrée d'un nombre x ne peut se calculer que si x est positif. Avant tout calcul d'une racine carrée, il est donc nécessaire de vérifier la positivité de la variable.

Cela se fait en C en utilisant

```c
if (expression_conditionnelle)
    instruction;
```

Exemple :

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

int main()
{
    float x;
    x = 5;
```

```
    if (x >= 0)
        printf("La racine carrée de %f vaut %f\n", x sqrt(x));
}
```

Souvent, une condition peut impliquer une action ou une autre : *si ... alors ... sinon ...*. Cela se traduit en C par :

```
if (expression_conditionnelle)
    instruction1
else
    instruction2;
```

Attention : si il y a plusieurs instructions après le `if`, il faut penser à introduire des `{ }`.

Si `expression_conditionnelle` vaut `vrai` alors on exécute `instruction1`, sinon (dans le cas ou `expression_conditionnelle` vaut `faux`) on exécute `instruction2`.

Exemple :

```
/* calcul d'une valeur absolue */

float valeur_absolue(float x)
{
    float fabs;

    /* si x est positif, la valeur absolue de x est x  */
    /* si x est négatif, la valeur absolue de x est -x */
    if (x >= 0)
    {
        fabs = x;
    }
    else
    {
        fabs = -x;
    }
    return(fabs);
}
```

L'ensemble `if (...) inst1 else inst2` est perçue comme une seule instruction. Il peut donc être inséré dans un ensemble `if (...) inst1 else inst2` de manière à former des *if imbriqués*.

Exemple : la suite de Fibonnacci ($U_n = U_{n-1} + U_{n-2}$ avec $U_0 = 0$ et $U_1 = 1$) :

```
int fibo(int n)
{
    int u_n;
    if (n == 0)
    {
        u_n = 0;
    }
    else if (n == 1)
    {
        u_n = 1;
    }
    else
    {
        u_n = fibo(n-1) + fibo(n-2);
    }
    return(u_n);
}
```

10.2 Opérateur conditionnel : `a ?b :c`

Si nous reprenons l'exemple du calcul de la valeur absolue précédente, il est possible de la simplifier comme suit :

```c
float valeur_absolue(float x)
{
   float fabs;
   fabs = (x>0)?x:-x;
   return(fabs)
}
```

L'expression a?b:c teste la condition a, si elle est vrai, alors on renvoie b, sinon on renvoie c. C'est un opérateur conditionnel qui traite trois opérandes.

a?b:c équivaut à écrire une "fonction" :

```c
   if (a) { return(b) } else { return(c) }
```

10.3 Instruction `switch`

L'instrucion switch est une instruction qui permet d'éviter dans beaucoup de cas l'emploi d'instructions if imbriquées.

Elle prend la forme de :

```c
switch (expression)
{
   case expression1 :
      instruction1;
   case expression2 :
      instruction2;
   .
   .
   .
   default :
      instruction_par_defaut;
}
```

L'expression conditionnelle `expression` est évaluée en premier. Si la valeur de renvoi est égale à l'expression constante `expression1`, l'instruction `instruction1` est exécutée. Si elle est égale à la valeur `expression2`, `instruction2` s'exécute, *etc.*. Si la valeur de renvoi n'est égale à aucune des expressions, l'instruction figurant immédiatement après le mot-clé default s'exécute.

Le mot-clé case régit chaque cas.

Exemple :

```c
include <stdio.h>

int main()
{
   int jour;

   jour = 3;
   switch (jour)
   {
      case 1:
         printf("%d : lundi\n",jour);
      case 2:
         printf("%d : mardi\n",jour);
      case 3:
         printf("%d : mercredi\n",jour);
      default:
         ;    /* instruction vide */
   }
   return(0);
}
```

En C, on exécute toutes les instructions figurant après le cas traité. Pour éviter d'exécuter une suite d'instructions, il faut faire "sortir" l'exécution du switch , cela s'effectue en utilisant l'instruction break .

```c
include <stdio.h>

int main()
{
   int jour;

   jour = 3;
   switch (jour)
   {
      case 1:
         printf("%d : lundi\n",jour);
         break;
      case 2:
         printf("%d : mardi\n",jour);
         break;
      case 3:
         printf("%d : mercredi\n",jour);
         break;
      default:
         printf("%d : repos !\n",jour);
   }
   return(0);
}
```

```c
#include <stdio.h>

int main()
{
   int n;
   n = 23;

   switch (n%10)
   {
      case 1 :
      case 3 :
      case 5 :
      case 7 :
      case 9 :
         printf("%d est impair\n",n);
         break;
      case 0 :
      case 2 :
      case 4 :
      case 6 :
      case 8 :
         printf("%d est pair\n",n);
         break;
      default :
           ;
   }
   return(0);
}
```

11 Boucles

En langage C, il est possible de répéter des instructions à l'aide de *boucles*.

Une boucle peut toujours être divisée en trois parties :

1. l'initialisation
2. la boucle par elle-même
3. la terminaison

11.1 Boucle `while ...`

Le premier type de boucle en langage C est la boucle while (”tant que”).

Syntaxe :

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

Tant que `condition` est vrai, `instruction` est exéctuée.

Exemple :

```
int n;
printf("Compte a rebours :\n");
n = 10;                 /* initialisation */
while (n >= 0)          /* condition : tant que n est positif */
{
   printf(" %d \n", n);
   n = n-1;             /* decrementation de n
                           -> cela permet de rendre fausse la
                              condition a un certain moment */
}
printf("Partez !\n");   /* terminaison */
```

Attention : Il faut toujours penser dans `instruction` à ajouter une instruction permet de rendre fausse la condition à un certain moment. Dans le cas contraire, la boucle continuera indéfiniment ...

L’instruction break vu précédemment peut aussi être utilisée dans les boucles afin d’en sortir.

```
n = 10;
while (1)
{
   printf(" %d\n",n);
   if (n <= 0)
   {
      break;
   }
   n--;
}
printf("Partez !\n");
```

Moins fort que break , l’instruction continue permet de ”sauter” les instructions en cours tout en restant dans la boucle. On passe alors à l’itération suivante.

```
/* affichage des nombres pairs */
n = 10;
while (n >= 0)
{
   if ( n%2 == 1 )
      continue;
   printf(" %d\n",n);
}
printf("Partez !\n");
```

11.2 Boucle `do ... while ...`

Une autre boucle possible en C est la boucle do ... while ... pour laquelle les instructions sont exécutées **avant** le test sur la condition.

Syntaxe :

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

Les instructions sont exécutées tant que la condition est vrai.

```
n = 10;
do {
    printf(" %d\n",n);
    n--;
}
while (n >= 0)
printf("Partez !\n");
```

La boucle do ... while ... peut être intéressante dans le cas où une(des) opération(s) doi(ven)t être exécutée(s) avant le test.

Exemple :

```
int n;
/* la boucle se repete tant que n n'est pas positif */
do {
    printf("Donnez un nombre positif : ");
    scanf("%d", &n);
}
while (n < 0)
```

11.3 Boucle `for (...) ...`

Comme on peut le constater dans les exemples précédents, on utilise souvent les boucles avec des *compteur*. Le compteur est initialisé, puis, tant que le compteur vérifie une certaine condition, la boucle est exécutée. De plus, le compteur est modifiée en fin de boucle.

```
compteur = ...                  /* initialisation */
while ( condition(compteur) )   /* condition */
{
    instructions;
    compteur = compteur + ...;  /* modification du compteur */
}
```

Le langage C fournit un raccourci pour ce genre d'écriture :

```
for (expression1 ; expression2 ; expression3)
    instructions;
```

`expression1`, `expression2`, et `expression3` sont séparées par des `;`.

`expression1` constitue une instruction exécutée **avant** la boucle principale constitué de `instructions`. Cela concerne le plus souvent l'initialisation d'une ou plusieurs variables.

`expression2` constitue une condition qui est testé **avant** l'exécution de la boucle principale. Tant que `expression2` est vrai, la boucle est exécutée.

`expression3` est une instruction qui est exécuté en fin de boucle principale.

Exemple :

```
int n;
for ( n = 10; n <= 0 ; n = n-1)
{
    printf("%d\n",n);
}
```

Ce qui équivaut à :

```
int n;
n = 10;                 /* expression1 */
while (n <= 0)          /* expression2 */
{
    printf("%d\n",n);   /* instructions */
```

```
    n = n-1;            /* expression3 */
}
```

`instruction1` et `instruction3` peuvent contenir plusieurs instructions séparées par des virgules.

```
for (i = 8, j = 0 ; i == 0 ; i--, j++)
{
    printf(" i = %d, j = %d, i+j = %d\n",i,j,i+j);
}
```

Boucle infinie On peut représenter une boucle infinie par `for (;;)` .

11.4 Boucles imbriquées

Il est possible d'imbriquer des boucles les unes dans les autres. Dans ce cas, l'ordinateur termine le traitement itératif en cours avant de revenir au traitement itératif juste au-dessus.

```
for (i = 0; i < 4 ; i++)
{
    printf("Iteration %d de la boucle externe\n", i);
    for (j = 0; j < 4; j++)
    {
        printf("   Iteration %d de la boucle interne\n", j);
    }
    printf("Fin iteration boucle externe %d\n",i);
}
```

12 Tableaux

Lorsque l'on veut stocker plusieurs éléments de même type, il est possible d'utiliser en C la notion de *tableau*. Un tableau est un ensemble de variables de même type. Chaque variable de ce tableau est appelée *élément* du tableau. Tous les éléments sont référencés à l'aide du nom du tableau et sont stockés dans des emplacements contigus en mémoire.

12.1 Déclaration d'un tableau

Syntaxe :

```
type Tableau[Taille];

/* exemples */
int tableau_entier[8];
double tableau_reel[100];
```

`Taille` définit le nombre d'éléments du tableau. Il doit être un entier positif (!).

12.2 Indexation d'un tableau

Les éléments d'un tableau sont repérés par leur ordre de placement dans ce tableau : leur *index*.

Les éléments d'un tableau à n éléments sont indexés de 0 à $n-1$. Par exemple, pour le tableau `int tab[7];` de 7 éléments, les éléments le constituant vont de `tab[0]` , à `tab[6]` .

Si dans `int tab[7];` , `tab` fait référence à un tableau à 7 éléments, `tab[0]` , ..., `tab[6]` font références aux éléments du tableau (de type `int`).

12.3 Initialisation d'un tableau

Un tableau peut s'initialiser soit éléments par éléments en donnant pour chaque élément sa valeur, soit globalement par l'emploi d'accolades.

```
char jour[7] = {'l', 'm', 'm', 'j', 'v', 's', 'd'};
int tab[3];
int i;

tab[0] = 1;
tab[1] = 2;
tab[2] = 3;
for(i = 0; i < 3; i++)
   printf("tab[%d] = %d\n",i,tab[i]);
```

12.4 Taille d'un tableau

La taille d'un tableau peut être retrouvée grâce à la fonction sizeof .

```
int tab[100];

printf("taille du tableau \t= %d\n",sizeof(tab));
printf("taille d'un element \t= %d\n", sizeof(int));
printf("nombre d'element \t= %d\n",sizeof(tab)/sizeof(int));
```

12.5 Tableaux à plusieurs dimensions

Il est possible de déclarer en C des tableaux à plusieurs dimensions.

```
type Nom[Taille1][Taille2]...[TailleN];
```

Les tableaux à plusieurs dimensions peuvent être initialisés comme précédemment.

```
int array_int[2][3];
array_int[0][0] = 1;
array_int[0][1] = 2;
array_int[0][2] = 3;
array_int[1][0] = 4;
array_int[1][1] = 5;
array_int[1][2] = 6;
```

Ce qui est équivalent à

```
int array_int[2][3] = {1, 2, 3, 4, 5, 6};
```

Ou encore

```
int array_int[2][3] = { {1, 2, 3}, {4, 5, 6}};
```

12.6 Chaînes de caractères

Une chaîne de caractères en C est représentée sous la forme d'un tableau de caractères. Ainsi, les deux définitions sont équivalentes :

```
char chaine[8] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
/* et */
char chaine[8] = "Bonjour";
```

Le caractère `'\0'` correspond au caractère "nulle" qui définit la fin d'une chaîne de caractères.

Exemple :

```c
char chaine[8] = "Bonjour";
printf("chaine vaut : %s\n",chaine);
   /* affiche : chaine vaut : Bonjour */

chaine[3] = '\0';
printf("chaine vaut : %s\n",chaine);
   /* affiche : chaine vaut : Bon */
```

12.7 Taille implicite

Parfois, lorsqu'on initialise un tableau, on préfère que ce soit le compilateur qui compte le nombre d'éléments dans le tableau. Cela peut se faire en utilisant une déclaration de taille implicite [] .

```c
char chaine[] = "Bonjour";
printf("chaine vaut : %s \n",chaine);
printf("taille de chaine = %d\n" , sizeof(chaine));
```

12.8 Tableaux et pointeurs

```c
int list_int[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
printf("list_int = %p\n", list_int);
printf("&list_int[0] = %p\n", &list_int[0]);
printf("&list_int[9] = %p\n", &list_int[9]);
printf("sizeof(list_int) = %d\n", sizeof(list_int));
printf("&list_int[9]-&list_int[0] = %d\n",&list_int[9]-&list_int[0]);
```

Un tableau en C est stocké en mémoire de manière linéaire (contigüe). Le nom du tableau correspond à un pointeur sur l'adresse de la première variable stockée.

12.9 Pointeurs, mémoires et tableaux

Parfois, il n'est pas possible de connaître à l'avance (*i.e.*, à la compilation) la taille d'un tableau. Le langage C offre ce qu'on appelle *l'allocation dynamique de la mémoire* grâce à la bibliothèque `<stdlib.h>`. C'est-à-dire que l'on peut à travers quelques fonctions standards demander à l'ordinateur de la mémoire contigüe afin de stocker des données, et notamment des tableaux.

La fonction `malloc` permet d'allouer de la place mémoire.

Syntaxe : void *malloc(size_t taille) où `taille` est la quantité de mémoire désirée en d'octets. malloc renvoie le pointeur NULL s'il n'a pu alloué la mémoire demandée.

La fonction `free` permet de libérer la place mémoire allouée et dont on n'a plus l'usage.

Syntaxe : void free (void *p)

```c
int n = 7;
int *tableau;
tableau = (int *) malloc(n*sizeof(int));
if (tableau == NULL)
{
   printf("Probleme d'allocation memoire\n");
   exit(1);
}
else
{
   printf("Adresse de tableau = %p\n", tableau);
   printf("Taille de tableau  = %d\n", sizeof(tableau));
```

```
}
free(tableau);
```

Il faut toujours libérer la place mémoire allouée avant de quitter un programme.

La fonction `calloc` permet non seulement d'allouer de la place mémoire comme pour la fonction malloc, mais elle initialise aussi toute la mémoire allouée à zéro.

Syntaxe : void * calloc (size_t nelements, size_t taille)

Cette fonction alloue une place mémoire équivalente à nelements de taille taille (en octets).

```
int n = 7;
int *tableau;
tableau = (int *) calloc(n, sizeof(int));
if (tableau == NULL)
{
   printf("Probleme d'allocation memoire\n");
   exit(1);
}
else
{
   printf("Adresse de tableau = %p\n", tableau);
   printf("Taille de tableau  = %d\n", sizeof(tableau));
   for (i = 0; i < n; i++)
      printf("tableau[%d] = %d\n", i, tableau[i]);
}
free(tableau);
```

12.10 Arithmétique sur les pointeurs

Il est possible dans un tableau de "déplacer" un pointeur : pointeur+n déplace le pointeur dans la mémoire de n*sizeof(*pointeur) octets.

```
int list_int[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
printf("list_int = %p\n", list_int);
printf("list_int+3 = %p\n", list_int+3);
printf("*(list_int+3) = %d\n", *(list_int+3));
```

On peut aussi (vu dans un exemple précédent) soustraire deux pointeurs de même type. Cela permet de connaître l'écart entre ces deux pointeurs.

13 La bibliothèque d'entrées-sorties `<stdio.h>`

Le langage C offre en standard un certain nombre de bibliothèques de fonctions, de *macros*, de définition de types, ... qui ne font pas partie du langage C proprement dit, mais qui aident beaucoup à son utilisation courante. Parmi celles-ci, la bibliothèque standard d'entrées-sorties `<stdio.h>` est très importante.

Une fonction de cette bibliothèque a déjà été rencontré : `printf` qui permet d'écrire du texte sur la sortie standard.

scanf permet de lire l'entrée-standard.
Attention il est nécessaire d'utiliser un passage de variable par adresse et non par valeur.

```
int entier;
printf("Donner un nombre entier : ");
scanf("%d", &entier);
printf("Vous avez rentre la valeur : %d\n", entier);
```

sprintf, sscanf sont similaires à `printf` et `scanf` mis à part que les résultats sont écrits ou lues dans une chaîne de caractères.

```c
char chaine[100];
char chaine2[] = "3141592" ;
int entier;

sscanf(chaine2,"%d", &entier);
sprintf(chaine,"Vous avez rentre la valeur : %d\n",entier);
printf("%s", chaine);
```

fopen permet d'ouvrir un fichier en lecture ou en écriture.

Syntaxe : `FILE *fopen(const char *filename, const char *mode)`

Le premier argument est une chaine de caractere contenant le nom du fichier à lire ou à écrire, le second argument est une chaîne de caractère décrivant le type de lecture/écriture :

`"r"`	ouvre un fichier en lecture
`"w"`	ouvre un fichier en écriture, et écrase le contenu précédent si le fichier existait
`"a"`	ajoute : ouvre et crée un fichier texte et se positionne en écriture à la fin du fichier
`"r+"`	ouvre un fichier en mode mise à jour (lecture + écriture)
`"w+"`	crée un fichier texte en mode mise à jour
`"a+"`	ajoute : ouvre et crée un fichier texte en mode mise à jour et se positionne en écriture à la fin du fichier

Si on ajoute `"b"` au mode, comme dans `"rb"`, cela indique un fichier binaire et non un fichier texte.

fclose ferme le fichier en lecture ou en écriture.

fprintf, fscanf sont similaires à `printf` et `scanf` mis à part que les résultats sont écrits ou lues dans un fichier.

```c
#include <stdio.h>

int main()
{
   FILE *pointeur_fichier = NULL;
   char nom_fichier[] = "hello1.c";
   char ligne;

   /* on ouvre le fichier */
   pointeur_fichier = (FILE *)fopen(nom_fichier, "r");
   /* on verifie qu'il est bien ouvert */
   if (pointeur_fichier == NULL)
   {
      printf("Erreur d'ouverture du fichier %s\n", nom_fichier);
      exit(1);
   }
   printf("fichier %s : \n", nom_fichier);
   /* lecture des caractères du fichier */
   while ( fscanf(pointeur_fichier,"%c",&ligne) != EOF)
   {
      printf("%c",ligne);
   }
   /* fermeture du fichier */
   fclose(pointeur_fichier);
   return(0);
}
```

`EOF` décrit la fin d'un fichier.

14 Portée des variables

Un programme complexe est généralement divisé en problèmes de moindre importance, traités chacun à l'aide d'une ou de deux fonctions (ou routines). Toutes ces fonctions sont regroupées dans un programme capable de résoudre le problème général posé.

Certaines variables doivent être accessibles à l'ensemble des fonctions du programme. D'autres doivent avoir un accès limité. Elles sont alors *masquées* (ou cachées) de la plupart des fonctions.

Restreindre la portée des variables est très utile lorsque plusieurs développeurs travaillent sur des parties différentes d'un même programme. Ainsi plusieurs fonctions peuvent porter le même nom sans générer de conflit.

En C, la portée d'une variable est définie dans la déclaration. Les variables ayant une portée locale ne sont visibles et exploitables que par le bloc dans lequel elles résident.

14.1 Portée locale

Un bloc est un ensemble d'instructions compris entre deux accolades. Une variable déclarée dans un bloc d'instructions a une portée de bloc. Elle n'est active et accessible qu'entre les accolades du bloc. La portée de la variable est donc *locale*.

14.2 Portée d'un bloc imbriqué

Il est possible de déclarer des variables dans un bloc imbriqué. Une variable déclarée dans le bloc extérieur et partageant le même nom que des variables du bloc intérieur est cachée par celles-ci.

```c
#include <stdio.h>

int main()
{
   int i = 32;
   printf("Bloc externe : i = %d\n", i);
   {
      int i, j;
      printf("Bloc interne :\n");
      for (i = 0, j = 5; i <=5; i++, j--)
      {
         printf("i = %d, j = %d\n", i, j);
      }
   }
   printf("Bloc externe : i = %d\n", i);
   return(0);
}
```

14.3 Portée de fonction

La portée de fonction indique qu'une variable est active et visible uniquement dans le cadre de la fonction où elle réside.

14.4 Portée globale

Une variable a une portée globale (ou portée de programme) lorsqu'elle est déclarée hors de toute fonction.

```c
#include <stdio.h>

int i = 1;
double j = 2.5;

void fonction1()
{
   printf("fonction1 : i = %d, j = %f\n", i, j);
}

int main()
{
   int i = 5;
```

```c
    fonction1();
    printf("Bloc principal : i = %d, j = %f\n", i, j);
    {
        double j = -1.5;
        fonction1();
        printf("Bloc imbrique : i = %d, j = %f\n", i, j);
    }
    return(0);
}
```

15 Autres types et fonctions

15.1 Type `enum`

Le type `enum` ou *énuméré* permet de déclarer des constantes entières nommées. Les programmes contenant de telles déclarations sont plus lisibles et plus faciles à mettre à jour.

Syntaxe : `enum tag {liste_enumeree} liste_variables;`

`tag` est le nom de l'énumération. `liste_variables` désigne les noms de variables de type `enum`. `liste_enumeree` contient une liste de noms énumérées représentant des constantes entières. Seul le paramètre du milieu est obligatoire.

```c
enum jour_semaine {lundi, mardi, mercredi, jeudi,
                   vendredi, samedi, dimanche} jour1, jour2;
```

Comme dans les tableaux les éléments de listes énumérées sont incrémentés de 0 à n. Il est possible de changer ses valeurs en les affectant :

```c
#include <stdio.h>

int main()
{
    enum jour_semaine {lundi,
                       mardi=12,
                       mercredi,
                       jeudi,
                       vendredi=20,
                       samedi,
                       dimanche} jour1, jour2;
    printf(" lundi = %d\n", lundi);
    printf(" mardi = %d\n", mardi);
    printf(" mercredi = %d\n", mercredi);
    printf(" jeudi = %d\n", jeudi);
    printf(" vendredi = %d\n", vendredi);
    printf(" samedi = %d\n", samedi);
    printf(" dimanche = %d\n", dimanche);
    return(0);
}
```

15.2 Définitions `typedef`

Le mot-clé `typedef` permet de créer des types personnalisés et des synonymes de types existants. Les synonymes se substituent aux noms de types qu'ils désignent dans les programmes. Ils rendent souvent le code source plus compréhensible.

Syntaxe : `typedef type nouveau_type;`

```c
typedef int ENTIER;
ENTIER i, j;

typedef double REEL, *ADRESSE_REEL;
```

```
REEL x, y;
ADRESSE_REEL ptr_x, ptr_y;
```

15.3 `main` et ses arguments

La fonction `main` peut recevoir des arguments : les arguments de la ligne de commande.

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

`argc` représente le nombre d'arguments. `argv` représente un tableau de chaîne de caractères donc chaque éléments représentent un arguments de la ligne de commande.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i = 0;
   printf("Nombre d'arguments = %d\n", argc);
   printf("Nom de la fonction = %s\n", argv[0]);
   for (i = 1; i < argc; i++)
   {
      printf("argument %d : %s\n", i, argv[i]);
   }
   return(0);
}
```

16 Structures

Une structure permet en C de regrouper au sein d'une même entité des données de types différents (de la même manière qu'un tableau permet de stocker des éléments de même type).

Alors que les éléments d'un tableau sont référencés à l'aide d'une valeur d'indice, les éléments d'une structure portent chacun un nom spécifique. Les éléments d'une structure s'appellent les *champs* ou membres.

16.1 Déclaration de structures

Syntaxe générale :

```c
struct nom_structure {
   type1 variable1;
   type2 variable2;
   type3 variable3;
      .
      .
      .
};
```

Le mot-clé `struct` signale la déclaration d'une structure, `nom_structure` correspond à l'étiquette de la structure, `variable1`, `variable2`, `variable2`, etc sont les membres de cette structure, de types `type1`, `type2`, `type3`, etc.

```c
struct etudiant
{
   char nom[100];
   char prenom[100];
   int jour_naissance;
   int mois_naissance;
   int annee_naissance;
   float note;
};
```

Après avoir déclaré la structure, on peut définir les variables correspondantes :

```
struct etudiant alphonse, louis;
```

16.2 Lecture de membres de structure

A partir d'une variable de type structure, on peut utiliser l'opérateur '.' pour accéder aux champs de la structure.

```
alphonse.mois_naissance = 10;
```

```
/* lecture des membres d'une structure */
#include <stdio.h>

int main()
{
   struct etudiant
   {
      char nom[100];
      char prenom[100];
      int jour_naissance;
      int mois_naissance;
      int annee_naissance;
      float note;
   } eleve;

   printf("Nom de l'eleve : ");
   gets(eleve.nom);
   printf("Prénom de l'eleve : ");
   gets(eleve.prenom);
   printf("Jour de naissance : ");
   scanf("%d",&(eleve.jour_naissance));
   printf("Mois de naissance : ");
   scanf("%d",&(eleve.mois_naissance));
   printf("Annee de naissance : ");
   scanf("%d",&(eleve.annee_naissance));
   printf("Note (sur 20) : ");
   scanf("%f",&(eleve.note));

   printf("Information saisies : \n");
   printf(" Nom et Prenom : %s, %s\n", eleve.nom, eleve.prenom);
   printf(" Ne(e) le %d-%d-%d\n",
      eleve.jour_naissance,
      eleve.mois_naissance,
      eleve.annee_naissance);
   printf(" Note recue : %.1f/20.0\n", eleve.node);
   return(0);
}
```

16.3 Initialisation de structures

Une structure peut être initialisée à l'aide d'accolades.

```
   struct etudiant
   {
      char nom[100];
      char prenom[100];
      int jour_naissance;
      int mois_naissance;
      int annee_naissance;
      float note;
   } eleve;

   eleve = { 'Dupond', 'Francis', 15, 2, 1980, 12.0 };
```

16.4 Structures et appels de fonctions

Le langage C permet de passer une structure entière à une fonction. De même, une fonction peut renvoyer une structure après traitement.

En pratique, il est intéressant de définir grâce à `typedef` un nouveau type se référant à la structure employée.

```c
#include <stdio.h>

typedef struct etudiant
   {
      char nom[100];
      char prenom[100];
      int jour_naissance;
      int mois_naissance;
      int annee_naissance;
      float note;
   } etudiant;

etudiant get_info(etudiant eleve)
{
   printf("Nom de l'eleve : ");
   gets(eleve.nom);
   printf("Prénom de l'eleve : ");
   gets(eleve.prenom);
   printf("Jour de naissance : ");
   scanf("%d",&(eleve.jour_naissance));
   printf("Mois de naissance : ");
   scanf("%d",&(eleve.mois_naissance));
   printf("Annee de naissance : ");
   scanf("%d",&(eleve.annee_naissance));
   printf("Note (sur 20) : ");
   scanf("%f",&(eleve.note));
   return(eleve);
}

int main()
{
   etudiant eleve;
   eleve = get_info(eleve);
   printf("Information saisies : \n");
   printf(" Nom et Prenom : %s, %s\n", eleve.nom, eleve.prenom);
   printf(" Ne(e) le %d-%d-%d\n",
      eleve.jour_naissance,
      eleve.mois_naissance,
      eleve.annee_naissance);
   printf(" Note recue : %.1f/20.0\n", eleve.node);
   return(0);
}
```

16.5 Pointage vers des structures

Il est possible en C d'utiliser un pointeur adressant une structure définie. Dans ce cas, l'opérateur `->` peut s'avérer utile : si `ptr` est un pointeur sur une structure, `ptr->champ` équivaut à utiliser `(*ptr).champ`.

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

typedef struct ETUDIANT
   {
      char nom[100];
      char prenom[100];
      int jour_naissance;
      int mois_naissance;
      int annee_naissance;
      float note;
   } ETUDIANT, *ETUDPTR;
```

```c

void get_info(ETUDPTR eleve)
{
    printf("Nom de l'eleve : ");
    gets(eleve->nom);
    printf("Prénom de l'eleve : ");
    gets(eleve->prenom);
    printf("Jour de naissance : ");
    scanf("%d",&(eleve->jour_naissance));
    printf("Mois de naissance : ");
    scanf("%d",&(eleve->mois_naissance));
    printf("Annee de naissance : ");
    scanf("%d",&(eleve->annee_naissance));
    printf("Note (sur 20) : ");
    scanf("%f",&(eleve->note));
}

int main()
{
   ETUDPTR eleve=NULL;
    eleve = (ETUDPTR)malloc(sizeof(ETUDIANT));
    if (eleve==NULL) exit(1);
    get_info(eleve);
    printf("Information saisies : \n");
    printf(" Nom et Prenom : %s, %s\n", eleve->nom, eleve->prenom);
    printf(" Ne(e) le %d-%d-%d\n",
       eleve->jour_naissance,
       eleve->mois_naissance,
       eleve->annee_naissance);
    printf(" Note recue : %.1f/20.0\n", eleve->note);
    return(0);
}
```

17 Préprocesseur

Le préprocesseur est un outil distinct du compilateur C. Il permet de traiter un fichier source avant sa compilation. Il apporte différentes fonctionnalités :
– la définition de macro
– la compilation conditionnelle
– l'ajout d'instructions de compilation
– ...
Lors de l'appel du compilateur, le préprocesseur agit automatiquement en amont afin de "préparer" le code source. Le processeur lit ligne par ligne le fichier source et interprète les lignes commençant par #.

Cela a déjà été vu avec l'instruction `#include <stdio.h>`, qui demande au préprocesseur d'inclure le fichier `stdio.h` dans le code source. C'est ce fichier qui définit les fonctions `printf`, `scanf`, ..., qui définit le type `FILE`, ...

En utilisant le compilateur `gcc`, il est possible d'arrêter le processus de compilateur juste après le passage du préprocesseur grâce à l'option `-E` (`gcc -E source.c`).

17.1 `#include`

La directive `#include` permet d'inclure un fichier *d'entête*. Si ce fichier est un fichier système, on applique la directive `#include <fichier>`, si ce fichier est un fichier utilisateur, on applique la directive `#include "fichier"`.

17.2 `#define`

La directive `#define` ordonne au préprocesseur de remplacer chaque occurence d'une chaîne (le nom de la macro) par la valeur correspondante (le corps de la macro). Ce remplacement s'effectue dans tout le corps du texte

après la rencontre de la directive `#define` et ce jusqu'à la fin du fichier, ou la rencontre d'une directive `#undef` correspondante.

Syntaxe : `#define nom_macro corps_macro`

Il est fortement conseiller de toujours entourer `corps_macro` par des parenthèses dans le cas où celui-ci représente une expression.

```c
#include <stdio.h>

#define DIX         10
#define SOMME1    (12+8)
#define SOMME2     12+8

int main()
{
    int a;
    printf("dix = DIX = %d\n",DIX);
    a = SOMME1*5;
    printf("a = SOMME1*5 = %d\n", a);
    a = SOMME2*5;
    printf("a = SOMME2*5 = %d\n", a);
    return(0);
}
```

Un nom de macro peut être suivi d'un ou plusieurs arguments, ce qui permet de l'utiliser comme une fonction ordinaire. De plus, une macro peut utiliser dans son corps une autre macro.

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

#define CARRE(a)     ((a)*(a))
#define FABS(a)      ((a>0)?(a):(-a))
#define SCAL(x,y)    (((x)*(y))/sqrt(CARRE(x)*CARRE(y)))

int main()
{
    int n = 10;
    double x = -2.3;
    double y = 2.5;
    int i;

    for (i=0; i<n; i++)
       printf("Le carre de %d vaut %d\n", i, CARRE(i));
    printf("valeur absolue de %f = %f\n",x,FABS(x));
    printf("valeur absolue de %f = %f\n",-x,FABS(-x));
    printf("scal(%f,%f) = %f\n",x,y,SCAL(x,y));
    return(0);
}
```

17.3 `#ifdef` et consorts

Il est possible de définir les conditions de compilation de certaines séquences, spécialement lors du débogage et de la mise au point d'un programme.

`#ifdef ... #endif` permet d'inclure ou d'exclude un groupe d'instructions du programme.

```c
#ifdef nom_macro
    instruction1
    instruction2
    instruction3
    ...
#endif
```

`#ifndef ... #endif` permet de définir le code à exécuter lorsqu'un nom de macro n'est pas défini.

`#if ... #elif ... #else ... #endif` permet d'inclure ou d'exclure certaines instructions selon que la valeur représentée par l'expression conditionnelle est différente ou égale à zéro.

```
#if expression
    instruction1
    instruction2
    instruction3
    ...
#endif
```