1ére Année

IUT de Toulon
Département de Génie Electrique
et Informatique Industrielle
BP 132
83957 La Garde Cedex

Informatique

Langage C

Ce cours est une introduction à la programmation en langage C ANSI.

Nom :

Prénom :

Version 2.00
arlotto@univ-tln.fr

Introduction

Ce cours est une introduction à la programmation en langage C ANSI. Le langage C est un langage qui s'est imposé dans de nombreux domaines qui vont de la programmation de systèmes d'exploitations au développement d'applications sur microcontrôleurs embarqués. Dans cette introduction au langage, on considérera principalement la programmation sur un système pourvu des périphériques d'entrées/sorties prévus dans la librairie standard du langage (typiquement clavier, écran et gestion de fichiers). Le cas particulier de la programmation de cartes électroniques à microcontrôleur qui sont en général limitées en taille mémoire et en périphériques d'entrées/sorties ne sera pas traité dans ce cours mais dans le cours d'informatique industrielle.

Tous les programmes donnés en exemple peuvent être compilés avec n'importe quel compilateur respectant la norme C ANSI. Les fichiers exécutables obtenus pourront par exemple être exécutés sur un micro-ordinateur de type PC à partir des systèmes d'exploitation MS-DOS©, LINUX ou WINDOWS© en mode "invite de commande" en fonction du compilateur utilisé.

Tous les programmes de ce cours ont été compilés avec le compilateur C ANSI **gcc**. Ce compilateur est le compilateur standard sous LINUX et une version pour WINDOWS© est mise à disposition gratuitement pour une utilisation non commerciale par la société *cygnus* (`www.cygnus.com`).

Les fichiers sources des programmes donnés en exemple dans ce cours, une liste de liens ainsi que d'éventuels compléments et corrections sont disponibles à l'adresse :

`http://arlotto.univ-tln.fr`

Il existe de nombreux cours de C disponibles sur internet, vous trouverez des liens à partir de l'adresse ci-dessus. Pour aller plus loin , un livre est tout de même préférable. De très nombreux ouvrages ont été imprimé sur le langage C. Je vous en conseille deux :

un excellent ouvrage qui permet de bien démarrer en langage C **: Le livre du C Premier Langage** de Claude Delannoy aux *éditions Eyrolles*.

et, pour ceux qui souhaitent aller plus loin : **Méthodologie de la programmation en C** de Jean-Pierre Braquelaire aux *éditions DUNOD*.

1. Un premier programme en C !

Nous voulons écrire un programme qui calcule la surface d'un cercle de rayon donné. Nous nous rappelons tous de la formule apprise à l'école élémentaire $S = \Pi.R^2$. Mais pour écrire un programme ce n'est pas suffisant. Il faut définir une méthode, appelée *algorithme*, qui donne pas à pas et de manière détaillée, la marche à suivre pour résoudre le problème posé. Dans notre cas c'est relativement simple mais c'est là qu'est la principale difficulté de la programmation. Ensuite cette méthode sera écrite, on dit *codée*, dans un langage certes limité mais très rigoureux et sans ambiguïté appelé *langage de programmation*. Nous utiliserons le *langage C*. Le fichier obtenu qui décrit dans un langage de programmation la suite des opérations à effectuer s'appelle *fichier source.* Ce fichier n'est pas compréhensible par un ordinateur et doit être traduit par un logiciel appelé *compilateur* en *un fichier exécutable*. Ce fichier exécutable contient la suite des *instructions élémentaires* compréhensibles par le microprocesseur.

L'algorithme pourraît être :

Afficher "Entrez le rayon en mètres : "

Mettre le rayon dans la variable r

Mettre dans s la valeur de $\Pi.r^2$

Afficher "La surface vaut : "

Afficher la valeur de la variable s

Traduit en langage C cela donne :

```
#include <stdio.h>

#include <stdlib.h>

int main( void )

    {

      float s,r ;

      printf("Entrez le rayon en metres :");

      scanf("%f",&r);

      s = 3.14159 * r * r ;

      printf("La surface vaut : %f\n",s);

      system("pause");

      return 0 ;

    }
```

Le programme proprement dit démarre à la première ligne de la fonction `main()`. Avant la fonction `main()`, on peut trouver des déclarations de variables (qui sont alors dites globales), ainsi que des directives de compilations (définition de constantes, inclusions de certains fichiers,…).

Les espaces et les sauts de lignes n'ont aucune importance tant qu'ils ne coupent pas un mot clé ou un identificateur.

Ainsi, la ligne de déclaration `s = 3.14159 * r * r ;`

pourrait tout aussi bien s'écrire :

```
s =
   3.14159 * r *
             r
          ;
```

Les crochets { et } sont importants car il marque les débuts et fin de blocs. On choisit de les placer seuls sur leur ligne pour bien les repérer.

1.2 Création du programme : Edition/compilation

La méthode précise de création d'un programme C dépend en grande partie de l'environnement utilisé (système d'exploitation, environnement de développement). Néanmoins voici quelques grandes lignes.

Le fichier *source* est le fichier dans lequel vous tapez votre programme.

Le fichier *exécutable* est le fichier programme produit par le compilateur à partir du source.

Le fichier source du programme doit être un fichier au format **texte** (ascii) et pas un fichier au format particulier issu d'un traitement de texte par exemple. Le plus simple est d'utiliser le bloc-note sous Windows ou un éditeur comme Kedit sous LINUX. Le fichier doit porter obligatoirement l'extension **.c**.

Pour créer l'exécutable il faut appeler le compilateur. En mode ligne de commande et en se plaçant dans le même répertoire que le fichier source, cela donne pour gcc:

```
$ gcc -W -Wall -ofichier.exe fichier_source.c
```

Les options W et Wall permettent d'activer tous les avertissements du compilateur (warning).

S'il n'y a pas d'erreur, le programme exécutable obtenu peut être lancé par

`$ ./fichier.exe` sous LINUX (l'extension `exe` n'est pas obligatoire sous LINUX)

ou

`$ fichier` sous l'invite de commande Windows.

1.3 Les mots clés

Ce sont des mots réservés par le langage et qui ont une signification bien précise. Ils ne peuvent pas être utilisés à d'autres fins que celles prévues par le C donc pas comme nom de variable, de constante ou de fonction. Si vous utilisez un mot clé comme nom de variable par exemple, le

compilateur risque de fournir des messages d'erreurs incohérents. En voici la liste exhaustive classée par ordre alphabétique. Ils sont toujours écrits en **minuscules**.

```
auto  break  case  char  const  continue  default  do  double
else  enum  extern  float  for  goto  if int  long  register
return  short  signed  sizeof  static  struct  switch  typedef
union  unsigned  void  volatile  while
```

1.4 Les commentaires

Les commentaires permettent au compilateur d'ignorer certaines lignes. Ils sont utiles pour supprimer provisoirement certaines lignes et pour expliquer le fonctionnement d'un programme.

La partie ignorée débute par `/*` et se termine lorsque l'on rencontre `*/`. Attention, il n'est pas possible d'imbriquer des commentaires. **Il n'est donc pas possible de mettre en commentaire une partie d'un programme comportant déjà des commentaires**.

Les commentaires /* ……*/ sont les seuls définis par le C ANSI, toutefois la plupart des compilateurs peuvent accepter (avec les options appropriées) les commentaires du langage C++.

Un commentaire C++ commence par les signes `//` et se poursuit jusqu'à la fin de la ligne. C'est un des rares cas où le saut de ligne possède une signification particulière.

```
/* ceci est un commentaire C ANSI */

// ceci est un commentaire C++
```

1.5 Les blocs d'instructions

Un *bloc d'instructions* commence par le caractère **{** et se termine par le caractère **}**. Un bloc d'instructions se comporte comme une seule instruction.

1.6 Les variables globales et les variables locales

Il existe principalement deux sortes de variables en C : les *variables globales* et les *variables locales*. Pour plus d'informations voir le chapitre 10.

Une variable globale est une variable déclarée en dehors de tout bloc d'instructions (en début de fichier par exemple). Une variable globale est accessible dans toute la partie du fichier source qui suit sa déclaration. Une variable globale est initialisée à la valeur 0 avant l'appel de la fonction `main()`. Son nom est réservé pour tout le fichier source.

Une variable locale est une variable déclarée en début de bloc. Elle est accessible dans tout le bloc où elle est déclarée ainsi que dans tous les blocs contenus dans le bloc où elle est déclarée. En dehors du bloc où elle est déclarée, une variable locale n'est pas accessible et son nom peut être réutilisé. La valeur initiale d'une variable locale est indéterminée.

Le nom d'une variable peut comporter tous les caractères alphanumérique (`a..z`,`A..Z`,`0..9`), ainsi que le caractère souligné _ . Le nom d'une variable ne peut pas commencer par un chiffre.

Une déclaration de variable commence par un nom de type (éventuellement précédé de modificateurs) suivi par une liste de nom de variable (éventuellement initialisée) et se termine par un `;` .

Exemples :

```
/*Ce programme affiche 0,8 et une valeur quelconque suivie de 9 */

#include <stdio.h>

#include <stdlib.h>

int g1,g2=8; /* g1,g2 variables globales */

int main(void )

{  int l1,l2=9; /* l1,l2 variables locales à la fonction main() */

   printf("%d %d %d %d",g1,g2,l1,l2);

   return 0 ;

}
```

1.7 Les fichiers inclus

Pour utiliser les fonctions prédéfinies (fonctions de la librairie standard), comme `printf()` pour afficher un message à l'écran, il est nécessaire d'inclure dans le fichier source la déclaration de ces fonctions. Ceci se fait par l'intermédiaire de la directive `#include` suivie du nom du fichier contenant la déclaration de la fonction entre les caractères < et >.

Exemples :

`#include <stdio.h>` pour utiliser `printf()`

`#include <math.h>` pour utiliser les fonctions mathématiques `cos()`, `sin()`, etc…

Les caractères < et > signifient que le fichier est recherché dans un répertoire connu du compilateur généralement nommé `include`. On peut également inclure un fichier contenant ses propres définitions dans ce cas le nom du fichier doit être mis entre les caractères " et " .

```
#include "mon_repertoire/mon_fichier.h"
```

Le chemin à utiliser est, soit le chemin absolu, soit le chemin relatif par rapport au fichier source où est placée la directive d'inclusion. Si le fichier à inclure se trouve dans le même répertoire que le fichier source on écrira simplement :

```
#include "mon_fichier.h"
```

Remarque **: Il n'y a pas *;* à la fin des lignes commençant par #**. Ce ne sont pas des instructions mais des directives de compilation, i.e. des commandes du compilateur.

1.8 Les constantes nommées

Il existe deux types de constantes : les *constantes littérales*, qui sont les nombres, les caractères et les chaînes de caractères, et les *constantes nommées* qui sont des noms donnés aux constantes littérales. Les constantes nommées facilitent la lecture d'un programme et permettent de simplifier les modifications ultérieures. Par convention, on écrit les constantes en **MAJUSCLE** (pour les distinguer aisément des variables qui seront écrites en minuscules).

Pour nommer une constante, on utilise la directive `#define` :

```
#define NMAX 100

#define PI 3.14
```

Le compilateur va remplacer toutes les occurrences des chaînes de caractères "`NMAX`" par la chaîne de caractère "`100`" et toutes les occurrences des chaînes "`PI`" par la chaîne "`3.14`". Il s'agit d'une substitution purement textuelle.

Attention : `NMAX` et `PI` seront bien des constantes. Ainsi des instructions de la forme `NMAX=15;` ou `PI=PI+1;` n'ont pas de sens : elles reviennent à écrire `100=15; ou 3.14=3.14+1;` !

On peut aussi définir des constantes à l'aide d'autres constantes précédemment définies :

```
#define QUART_DE_TOUR   2*PI/4

#define TOUR 4*QUART_DE_TOUR
```

Attention : Il n'y a pas de ; à la fin de ces lignes.

Si on écrit

```
#define PI 3.14; (1)
```

le compilateur ne générera pas d'erreur sur cette ligne mais des expressions comme

```
float x=PI/4; (2)
```

seront alors impossible car elles se transformeront en

```
float x=3.14;/4; (3)
```

Ce type d'erreur est difficile à détecter car elle apparaît sur la ligne (2) et pas sur la ligne à modifier (1) (la ligne (3) n'est pas visible dans le source).

2. Les types de base du langage C

Les données (constantes, variables) manipulées en langage C sont typées. Cela veut dire qu'à chaque donnée, est associée une information, le *type*, qui caractérise la taille occupée en mémoire, la représentation interne, les opérations permises pour manipuler ces données ainsi que les propriétés de ces opérations. Pour déclarer une variable d'un type donné, on fait précéder le nom de la variable par le nom de son type.

2.1 Le type caractère : `char`

Les variables de ce type sont codées sur un octet (8 bits) . Elles sont destinées à recevoir des caractères généralement en code ASCII. Les caractères ne correspondent pas tous à des symboles imprimables : il existe des caractères dits caractères de contrôle qui sont associés à des actions particulières comme un changement de ligne, un retour en arrière, etc… En plus du caractère représenté, on peut s'intéresser à la valeur proprement dite d'une variable de type caractère et faire des calculs. Dans ce cas, le nombre associé n'est pas définit par la norme ANSI et dépend des compilateurs : certains considèrent la valeur comme non signée et donc fournissent un nombre entre 0 et 255 tandis que d'autres considèrent la valeur signée et donc fournissent un nombre entre -128 et +127 pour le même motif binaire. Pour s'affranchir de cette dépendance, on peut utiliser les *modificateurs* `signed` ou `unsigned` pour préciser si on veut considérer la valeur signée ou non signée lors de calculs sur des variables caractères (cf. remarque au §2.2). On aura alors les types dérivés : `signed char` et `unsigned char`.

Exemples de déclarations de variables de type caractère :

```
char c=65; /* c contient le code ASCII du caractère A */

signed char x=-12;

unsigned char t=254;

char lettre='z';
```

Notation des constantes de type caractère

Les constantes caractères se notent entre simples apostrophes (ou quotes) : `''` .

Exemple : `'b' 'A' '=' '4'` représentent respectivement les caractères b A = et 4.

Les caractères de contrôle ainsi que certains caractères ayant une signification particulière en langage C, se notent en utilisant le caractère \ (back-slash) . Les principaux sont :

Notation C	Code ASCII (hexa)	Signification
`\n`	0A	saut de ligne
`\b`	08	retour arrière
`\r`	0D	retour chariot
`\\`	5C	\
`\'`	2C	'
`\"`	22	"
`\?`	3F	?

2.2 Le type entier : `int`

Le type `int` est utilisé pour représenter les entiers relatifs. C'est donc un type signé. La taille de ce type est la taille la plus naturelle du microprocesseur utilisé. Ainsi sur un microprocesseur 16 bits, une variable de type `int` fera 16 bits mais sur une machine 32 bits, un entier fera 32 bits.

Toutefois, la norme prévoit que la taille d'un entier ne peut être inférieure à 16 bits. Ainsi sur un microcontrôleur 8 bits un entier fera 16 bits. On est alors assuré de disposer des valeur comprises entre -2^{15} et $2^{15}-1$ soit entre –32768 et 32767 (pour un nombre de bits n entre -2^{n-1} et $2^{n-1}-1$).

Le type `int` peut être modifié au moyen de deux modificateurs de taille : `short` et `long` et d'un modificateur de signe : `unsigned` . `long` et `short` indiquent au compilateur que l'on souhaite utiliser respectivement plus ou moins de bits, tandis que `unsigned` indique que l'intervalle va de 0 à 2^n-1 (par exemple de 0 à 65535 sur 16 bits). Il arrive parfois que `int` et `short int`, représentent le même type car la norme ANSI prévoit seulement la relation :

`short int` ≤ `int` < `long int` avec `int` ≥ 16 bits et `long` ≥ 32 bits

Les compilateurs acceptent l'abréviation de `short int` en `short` et de `long int` en `long`. `unsigned` par contre ne peut pas être employé seul.

Exemples de déclarations :

```
int i=4,j;

short int s1;
```

```
unsigned long int ul1;

unsigned int x=2345;
```

Notations des constantes entières

Les constantes entières peuvent être notées de manière classique en décimal (avec ou sans signe), en hexadécimal faisant précéder la valeur par `0x` (ou `0X`) ou en octal en faisant précéder la valeur par un `0`. Exemple :

```
int i=429;
```
décimal

```
int x=0xFFFE;
```
hexadécimal

```
int y=071;
```
octal

Remarque : La notation octale est peu utilisée mais attention à l'erreur produite par une expression du type i=08 qui n'est pas toujours facile à détecter. En effet les chiffres 8 et 9 n'existant pas en octal, l'affectation précédente est impossible. Prenez donc l'habitude de ne pas mettre de zéros devant les constantes entières, car ils ne sont pas non significatifs en langage C ! Plus grave est le fait que 010 ne vaut pas 10 en décimal car aucun message d'erreur n'est produit par le compilateur.

2.3 Les types virgule flottante `float` ,`double` et `long double`

Ces types sont utilisés pour représenter de manière approchée une partie des nombres réels. La précision va croissant de `float` à `double` puis à `long double`. Un nombre réel x sera représenté dans un type à virgule flottante par deux nombres, la *mantisse* M et l'*exposant* E telles que la quantité $M.b^E$ soit la plus proche possible de x. La base b est 2 ou 16 . Par exemple dans le cas où 32 bits sont utilisés pour `float` , la mantisse est sur 24 bits et l'exposant sur 8.

Pour une constante de type flottant, la virgule utilisée en français est remplacée par un point (comme en anglais).

Exemple : `double pi=3.14, x=1E3;`

Remarque : Du fait de la représentation des variables de type flottant, les calculs sur ces variables demanderont plus de temps machine et de taille mémoire que les calculs sur les entiers car une simple addition par exemple ne se traduira pas par une seule instruction en langage machine mais par une suite plus ou moins longue d'instructions. Du moins sur un microprocesseur classique car il existe des processeurs spécialisés qui traitent les flottants avec de simples instructions machines : ce sont les DSP (Digital Signal Processor) à virgule flottante.

Attention : Du fait des arrondis de calculs, la comparaison de deux variables réelles entre elles ou d'une variable réelle à une constante a très peu de chance de donner un résultat vrai. Il faut donc vérifier que leur différence reste inférieure à une valeur arbitraire choisie pour précision.

Ainsi le programme :

```
int main(void) {
double x=4,double y;
…… /* Quelques calculs avec x et y */
……
if (x==y)
  {
    printf("Egalité");
 }
 else
{
 printf("Différent");
}  }
```

a toutes les chances de toujours afficher **`Différent`**.

Il vaudrait mieux le programmer ainsi :

```
#include <math.h>  /* Pour la fonction valeur absolue fabs() */
#define PRECISION 1E-6 /* précision choisie */
int main(void) {
double x=4,double y;
…… /* Quelques calculs avec x et y */
……
if ( fabs(x-y) < PRECISION )  /*  Si |x-y| < 10^-6 */
   {
    printf("Egalité");
 }
 else
{
 printf("Différent");
}   }
```

2.4 Choisir le type de données adapté

Dans un programme, on est souvent amené à manipuler des nombres. Très souvent un sous-ensemble des entiers naturels suffit : dénombrement d'objet, indice de tableau, nombre d'itération d'une boucle, etc… Le type **`int`** est donc le type qui s'impose dans de nombreux cas. Si on peut se limiter à une partie de l'intervalle [-32767,+32768], on peut prendre `int` sans problème. Comme le type `int` possède la représentation la plus naturelle sur le microprocesseur utilisé, le programme généré sera le plus efficace en taille et en performance car le microprocesseur manipule directement les données de ce type.

Toutefois, dans le cas particulier d'un microcontrôleur 8 bits l'utilisation du type `int` n'est pas judicieuse pour représenter des "petits" nombres (inférieurs à 255 ou compris entre –127 et +128) car ces microprocesseurs ne manipulent pas directement des variables sur 16 bits. L'utilisation du type `char` et de ses dérivés génère alors un code plus compact et plus rapide. Dans tous les autres cas le type `char` sera utilisé pour stocker des caractères (d'où son nom) ou toutes données dont la représentation doit se faire sur un octet.

Les types réels `float` ou `double` devront être réservés au cas où la dynamique des calculs (différence entre le plus grand et le plus petit nombre manipulé) est trop importante pour tenir dans un type entier.

Cette situation peut se résumer par le tableau suivant :

Intervalle utilisé compris dans	µP 8 bits	µP 16 bits ou plus
`[0,255]`	`unsigned char`	`int`
`[-128,127]`	`signed char`	`int`
`[-32768,32767]`	`int`	`int`
`[0,65535]`	`unsigned int`	`unsigned int`
`Au delà positif et négatif`*	`long int`	`long int`
`Au delà positif uniquement`*	`unsigned long int`	`unsigned long int`
`Nécessité d'utiliser les nombres réels`**	`double`	`double`
`Stockage de caractères ou d'octets sans calculs arithmétiques`	`char`	`char`

*Les limites pour chaque type entier se trouvent dans le fichier en-tête `<limits.h>`

**Les caractéristiques des types flottants sont définies dans le fichier en-tête `<float.h>`.

Sur l'intervalle `[0,127]` les types `char`, `unsigned char` et `signed char` sont équivalents.

3. Les opérateurs et les expressions

Le langage C dispose de 44 opérateurs dont voici la liste exhaustive par ordre de priorité décroissante (les opérateurs d'une même ligne ont la même priorité):

Catégorie	Opérateurs
référence	`() [] -> .`
unaire	`- ++ -- ! ~ * & (cast) sizeof`
arithmétique	`* / %`
arithmétique	`+ -`
décalage	`<< >>`
relationnel	`< <= > >=`
relationnel	`== !=`
manipulation de bits	`&`
manipulation de bits	`^`
manipulation de bits	`|`
logique	`&&`
logique	`||`
conditionnel (ternaire)	`?:`
affectation	`= += -= *= /= %= &= ^= |= <<= >>=`
séquentiel	`,`

Un opérateur est dit *unaire* lorsqu'il porte sur un opérande, *binaire* lorsqu'il porte sur deux opérandes et *ternaire* lorsqu'il porte sur trois opérandes .

Exemples :

`x= -y;` cet opérateur − est unaire.

`x= a;` l'opérateur = est binaire.

`x=(a>b)?a:b;` `?:` est le seul opérateur ternaire du C.

Remarque : Certains opérateurs ne seront pas étudiés dans ce chapitre mais figurent dans la liste pour montrer les priorités. N'hésitez pas à mettre des parenthèses lorsque vous n'êtes pas sûr des priorités des opérateurs.

3.1 Les opérateurs arithmétiques

Ils réalisent les opérations numériques classiques : addition +, soustraction −, multiplication * et division /, ainsi que l'opposé (− unaire) et sont définis pour tous les types numériques. Un opérateur supplémentaire %, calcule le reste dans la division entière. Cet opérateur ne peut porter que sur des entiers (ou le type char).

Exemple :

```
int a=14 , b=4 , q , r;

q=a/b;  /* q vaut 3 */

r=a%b; /* r vaut 2 */
```

Attention : **le quotient de deux entiers est toujours un entier**. Ainsi 4/3 vaut 1 et 3/4 vaut 0.

3.2 L'opérateur d'affectation

Cet opérateur permet de donner une valeur à une variable. Le terme variable est souligné pour bien insister sur le fait que ce qui se trouve à gauche d'un opérateur d'affectation doit être une variable et pas autre chose. Ainsi si `x` et `y` sont des variables `x=2;` est licite alors que `x+y=3;` n'a pas de sens. En effet, `x+y` n'est pas une variable; c'est une expression valant la somme de deux variables. `x+y` n'a pas de case mémoire.

L'opérateur d'affection est noté `=` comme dans `x = 2 ;` ou `x = y ;`

La forme générale est : `variable = expression ;`

Cette instruction est effectué en **deux temps** :

1/ on évalue l'expression de droite

2/ on place ensuite le résultat de l'évaluation dans la variable de gauche en appliquant une conversion si nécessaire.

L'opération d'affection force la conversion de l'expression dans le type de la variable.

3.3 Affections et expressions entre type différents

En général, on essaye de ne pas mélanger les types dans les expressions mais il existe des cas où l'on y est obliger.

Conversion non dégradante : c'est une conversion qui conserve la totalité de l'information contenue dans la variable. Par exemple la conversion d'un int en float est non dégradante.

Conversion dégradante : c'est une conversion qui altère l'information contenue dans la variable.

Par exemple la conversion d'un float en int est dégrante car on ne conserve que la partie entière de la valeur initiale.

Lors de l'évaluation d'expression entre type différent, on applique le plus possible de conversions non dégradante puis on applique des conversions dégradantes si nécessaire notamment lors de l'affection.

Exemple :

float x,y;

int a,b ;

y = 7.2 ;

x = y + 2 ; l'expression y+2 est évaluée et donne 9.2 puis le résultat est placé dans la variable x sans conversion car les types sont les mêmes.

a = y + 2 ; l'expression y+2 est évaluée et donne 9.2 puis le résultat est placé dans la variable a après avoir été converti en entier et donc a contient 9

b = (a + y)*6 ; a est converti en 9.0 , ajouté à y on obtient 16.2, qui multiplié par 6 donne 97.2. cette valeur est convertie de manière dégradante en 97 pour être placé dans b

y = a / 10 ; la valeur 10 étant entière la division donne 0 est le résultat est placé dans y.

b = a / 10.0 ; a est converti en réel la division donne 0.9 qui est convertie en 9 pour aller dans b

y = a/10.0 ; a est converti en réel la division donne 0.9 qui est placé dans y

a = a+1 ; a+1 donne la valeur 10 qui est a nouveau placé dans a

3.4 Les opérateurs relationnels

Ces opérateurs permettent de comparer des expressions. Le résultat de la comparaison est une valeur entière qui vaut 0 si le résultat de la comparaison est faux et qui vaut 1 si le résultat de la comparaison est vrai.

Opérateur	Signification
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
! =	différent de

Une erreur classique consiste à employer l'opérateur d'affection = à la place de l'opérateur relationnel ==. Ceci ne conduit généralement pas à une erreur de compilation mais le résultat n'est pas celui attendu !

Remarque : inférieur ou égal (supérieur ou égal) se notent par l'opérateur < (>) suivi de = dans le sens de la lecture (et pas =< ni =>).

3.5 Les opérateurs logiques

Il y a trois opérateurs logiques ET , OU et NON respectivement notés `&&` , `||` et `!`. Comme toute valeur non nulle est considérée comme vraie en C et que 0 est considérée comme fausse, ces opérateurs acceptent pour opérande non seulement des expressions mais également des valeurs numériques quelconques.

Opérateur	Notation C	
ET	&&	`exp1 && exp2` est vraie ssi `exp1` et `exp2` sont vraies
OU	||	`exp1 || exp2` est vraie si `exp1` ou `exp2` est vraie
NON	!	!`exp` est vraie si `exp` est fausse

Attention à ne pas confondre `&&` et `||` avec les opérateurs de manipulation de bits `&` et `|`.

3.6 L'opérateur ternaire ? :

C'est le seul opérateur à trois opérandes du C.

L'expression `exp1 ? exp2 : exp3`

vaut `exp2` si `exp1` est vraie

et `exp3` si `exp1` est fausse.

Exemple :

```
int  a=3, b=4, c;

c=(a>b) ? a : b ; /* c prend la plus grande valeur entre a et b */
```

C'est équivalent à :

```
   if (a>b)
      c=a;
   else
      c=b;
```

3.7 Les opérateurs de manipulation de bits

Le langage C dispose d'opérateurs permettant d'accéder directement au niveau du motif binaire d'une valeur de type entier ou caractère. Cette possibilité est très utilisée dans la programmation de microcontrôleurs pour remplacer le langage d'assemblage dans de nombreux cas.

Opérateur	Signification
&	ET bit à bit (AND)
|	OU bit à bit (OR)
^	OU exclusif bit à bit (XOR)
~	inversion de chaque bit (NOT)
>>	décalage à droite
<<	décalage à gauche

Exemples :

```
char c=0xF7,d;

d= c & 0x88;   /* d vaut 0x80 */

d= c | 8 ;     /* d vaut 0xFF */

d= d ^ 16 ;    /* d vaut 0xEF */

d= ~c ;        /* d vaut 0x08 */

d= c << 1 ;    /* d vaut 0xEE; */

d= c >> 1 ;    /* d vaut 0xFB ou 0x7B */
```

Lors d'un décalage à gauche les bits libérés sont toujours remplacés par des zéros (décalage logique).

Lors d'un décalage à droite d'une valeur non signée les bits libérés sont remplacés par des zéros (décalage logique).

Lors d'un décalage à droite d'une valeur signée les bits libérés peuvent être soit remplacés par des zéros (décalage logique), soit remplacés par le bit de signe (décalage dit arithmétique) : cela dépend des compilateurs. Il faut s'efforcer d'écrire des programmes dont les résultats sont indépendants du choix du compilateur : programmes portables. Il faut donc forcer les bits libérés à la valeur voulue par un masque approprié.

Attention à ne pas confondre les opérateurs `&` , `|` et `~` avec les opérateurs `||`, `&&` et `!`. L'emploi de l'un pour l'autre ne génère en général pas d'erreur mais le résultat n'est pas celui attendu.

3.8 Les opérateurs d'incrémentation ++ et --

Ces opérateurs incrémentent ou décrémentent une expression entière (ou un pointeur).

Exemple simple :
```
int i=9;

i++;  /* est équivalent à i=i+1; */
```

```
i--;   /* est équivalent à i=i-1; */
```

Attention, il faut parfois distinguer l'effet proprement dit (incrémentation ou décrementation) de la valeur de l'expression. Il y alors une différence entre la notation postfixée (`i++`) et la notation préfixée (`++i`) . Dans l'expression `++i`, `i` est incrémenté avant l'évaluation de l'expression qui vaut alors `i+1` alors que dans l'expression `i++`, l'expression est évaluée avant l'incrémentation et vaut alors `i`. La tableau ci-dessous résume la situation :

Expression	Effet	Valeur
`i++`	i←i+1	i
`++i`	i←i+1	i+1
`i--`	i←i-1	i
`--i`	i←i-1	i-1

Exemple :

```
int n,i=2;

n= i++ + 5; /* n vaut 7 , i vaut 3 */

n= --i ;    /* n vaut 2 , i vaut 2*/

n= i-- ;    /* n vaut 2 , i vaut 1 */
```

3.9 L'opérateur `sizeof()`

L'opérateur `sizeof()` permet de connaître la taille en octets d'un type. Par définition `sizeof(char)` vaut toujours `1` puisque le type `char` occupe toujours un octet en mémoire. Par contre pour tous les autres types les résultats dépendent de la machine et du compilateur utilisés. L'opérateur `sizeof()` doit systématiquement être utilisé dans tous les cas où la taille allouée à un type a une importance (allocation dynamique de la mémoire par exemple) afin de rendre le programme portable (indépendant de l'implémentation). Il est aussi très pratique pour connaître la taille d'un type complexe (structure).

Exemple : l'instruction :

```
printf("la taille du type int vaut %d octets", sizeof(int));
```

permet d'afficher la taille allouée au type `int`.

3.10 L'opérateur de conversion explicite (cast)

Ils permettent de forcer la conversion d'une expression dans un type donné (conversion explicite). Beaucoup de conversions sont possibles mais toutes ne donnent pas un résultat satisfaisant. Pour effectuer une conversion on fait précéder une expression par le nom du type dans lequel on désire la convertir écrit entre parenthèses.

Exemple :

```
char c=12;

float x;

x=(float)c;
```

Dans l'exemple ci-dessus, l'utilisation de l'opérateur de cast n'est pas absolument nécessaire car l'opérateur d'affectation force la conversion dans le type d'arrivée. Cet opérateur sera utile lors de l'emploi des pointeurs (cf. remarques au §6.2).

L'opérateur de cast peut être utilisé pour forcer le résultat d'un calcul dans un type donné. Par exemple si `nb_notes` et `somme_notes` sont des variables de type entier et moyenne une variable de type flottant. Le calcul de la moyenne par l'instruction

```
moyenne = somme_notes / nb_notes ;
```

donnera une valeur eronnée pour certaines valeurs des variables entières (cf. §4.1). C'est le résultat de la division des deux entiers, qui est donc un entier, qui est converti ensuite dans le type flottant de la variable `moyenne` lors de l'affectation. Il faut donc convertir chacune des deux variables avant la division pour effectuer une division en flottants :

```
moyenne = (float) somme_notes / (float) nb_notes ;
```

4. Les instructions usuelles

Dans la suite `instruction(s);` désigne une ou plusieurs instructions.

Lorsque `instruction(s);` représente une seule instruction, les { } ne sont pas obligatoires. Toutefois, pour des raisons de lisibilité et de maintenabilité, il ne vaut mieux pas utiliser cette possibilité.

4.1 L'instruction `if`

`instruction(s);`

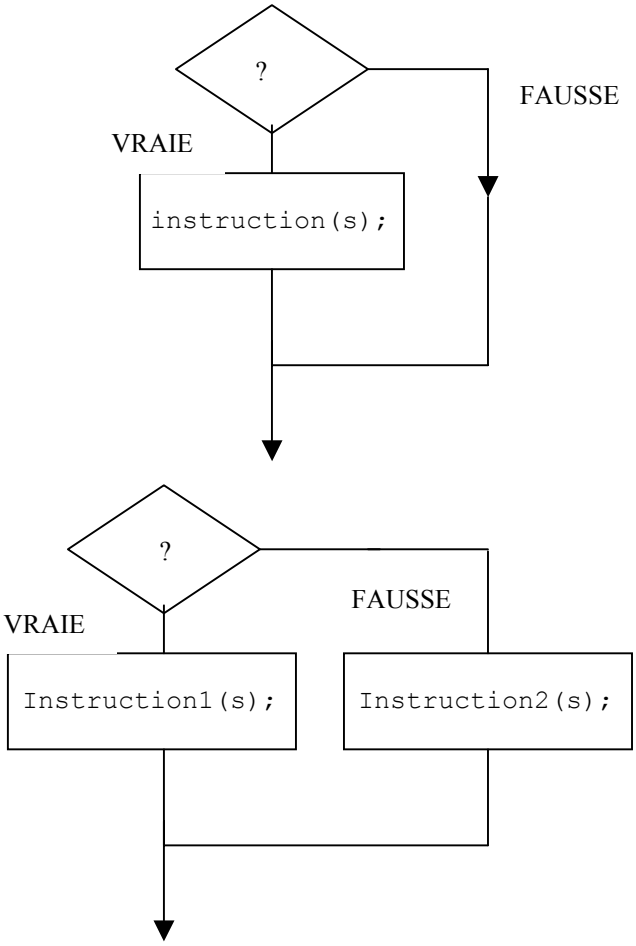
Elle correspond aux instructions algorithmiques SI…FINSI et SI…SINON…FINSI.

Syntaxe :

```
if  (expression)
    {
    instruction(s);
    }
```

ou

```
if  (expression)
    {
    instruction1(s);
    }
else
    {
    instruction2(s);
    }
```



4.2 L'instruction `while()`

Elle correspond à l'instruction algorithmique TANTQUE…FINTANTQUE.

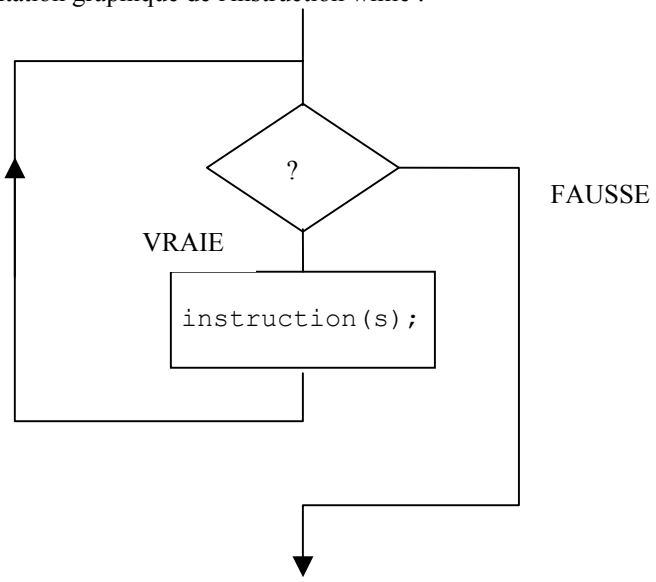
Syntaxe :

```
while (expression)
   {
      instruction(s);
   }
```

Attention, il n'y a pas de `;` ici !!

Remarque : Parfois `instruction(s);` peut être vide, c'est-à-dire un simple `;` . Dans ce cas, le programme boucle tant que l'expression est vraie. (les `{ }` sont alors superflus).

Représentation graphique de l'instruction while :



4.3 L'instruction `do...while()`

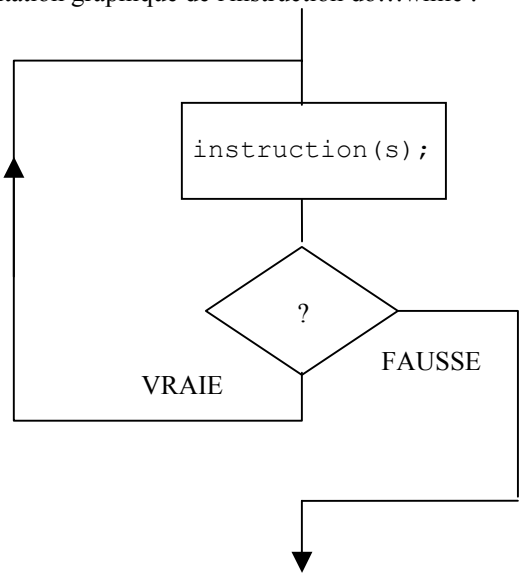
Elle correspond à l'instruction algorithmique FAIRE….TANTQUE.

Syntaxe :

```
do
   {
      instruction(s);
   }
while (expression);
```

Attention à ne pas oublier le point virgule à la fin de cette instruction.

Représentation graphique de l'instruction do…while :



4.4 L'instruction for

Cette instruction permet d'écrire des boucles sous forme condensée.

Syntaxe :

```
for ( expression1 ; expression2 ; expression3 )
        {
           instruction(s);
        }
```

L'expression 1 est évaluée une seule fois au début. Puis si l'expression 2 est vraie, les instruction(s) sont exécutées. Puis l'expression 3 est évaluée. Le cycle recommence tant que l'expression 2 est vraie.

Une instruction `for` est l'équivalent sous forme condensée de :

```
expression1 ;
while (expression2)
   {
      instruction(s);
      expression3;
   }
```

Exemple :

```
int i;
for (i=0; i<4 ; i++ )
{
  printf("tour n° %d", i);
}
```

Une, deux ou trois expressions peuvent être vides :

```
int i=0;
for ( ; i<4; )
{
printf("tour n° %d",i);
i++;
}
```

Les deux instructions for ci-dessus sont équivalentes à

```
int i;
i=0;
while (i<4)
{
   printf("tour n° %d",i);
   i++;
}
```

4.5 L'instruction `break`

L'instruction `break` permet de passer directement à la fin d'un bloc sans exécuter les instructions qui suivent. Elle permet aussi de sortir d'une boucle.

Exemple : modification du programme du §2.1 pour prendre la valeur 5 au bout de 10 essais

```
int n_essai=10; /* en début de programme */
do
   {
      printf("Entrez un nombre entre %d et %d : ",NMIN,NMAX);
      scanf("%d",&n);
      if ((n_essai--)==0)
         { n=5 ;
          break ; /* sortie de la boucle */
         }
   } while (n<NMIN || n>NMAX);/* on recommence si réponse fausse*/
```

Remarque : On peut se passer de l'instruction `break` si on modifie l'expression dans l'instruction `while` par `(n<NMIN || n>NMAX) && ((n_essai--)!=0)`. Mais dans ce cas le programme avec l'instruction `break` peut sembler plus clair. (il faudra de plus affecter la valeur 5 à `n` si `n_essai` vaut -1 après la boucle).

4.6 L'instruction `continue`

L'instruction `continue` permet de passer directement au tour de boucle suivant sans exécuter les instructions qui suivent dans le bloc.

Exemple : Impression des entiers de 1à 40 en sautant les multiples de 3

```
int i;

for ( i=1;  i<=40; i++)

{

   if (i%3==0)    /* si le reste dans la division par 3 est */

      {           /* nul alors le nombre est multiple de 3 */

      continue;

      }

   printf("%d ",i);

}
```

L'instruction `continue` effectue un branchement avant l'évaluation de l'expression `i<=40` (et pas après).

4.7 L'instruction `switch`

L'instruction `switch` permet de choisir les instructions à exécuter en fonction de la valeur d'une expression de type `int`. Elle remplace avantageusement une longue série d'instructions `if…else`.

Syntaxe : Les crochets `[ ]` indiquent que les instructions sont optionnelles. Ils ne doivent pas figurer dans le programme.

```
switch (expression)

   {

      case CONSTANTE_1   : [ instruction1(s); ]

      case CONSTANTE_2   : [ instruction2(s); ]

      ……

      case CONSTANTE_N   : [ instructionN(s); ]

      [default           : [ instruction_defaut(s); ] ]

     }
```

L'expression est évaluée puis un branchement est effectué à la ligne dont la constante est égale à la valeur de l'expression. Les instructions sont alors exécutées à partir de cette ligne et jusqu'à la fin ou jusqu'à rencontrer une instruction `break`. Si aucune constante n'est égale à la valeur de l'expression , le branchement est effectué à la ligne `default` si elle existe.

Exemple :

```
int c;
/* Acquisition de la variable c */
……..
switch (c )
         {
      case 0 :
      case 1 : printf(" c < 2");
               break;
      case 2 : printf(" c vaut 2");
               break;
      case 3 :
      case 4 : printf(" c > 2");
               break;
      default : printf(" c hors limite");
   }
```

5. Les tableaux et les pointeurs

5.1 Les tableaux

Un tableau est un ensemble de variables de même type auxquelles on peut accéder par un indice.

Déclaration :

```
char tab[5]; /*tableau de nom tab de 5 élément de type char */

int vect[6] /*tableau de nom vect de 6 éléments de type int */
```

Le nombre d'élément est aussi appelé dimension du tableau. La dimension d'un tableau est obligatoirement une constante (ou une expression constante).

On peut initialiser un tableau lors de sa déclaration en notant les valeurs entre crochets `{}` :

```
int t[3]={4,2,3};

char n[4]={'a',65,'b','x'};
```

Accès aux éléments : Un élément est une variable du type définit par la déclaration du tableau. L'indice est une expression de type `int`. **Attention : le premier élément a pour indice 0. Le dernier élément a pour indice (dimension-1).**

Avec les déclarations ci-dessus on peut écrire :

```
int i=1,j=2,k=2;

tab[0]=3;

tab[i]='A';

vect[i+j]=t[0];
```

Attention l'accès a des éléments inexistants (`tab[5]`, `vect[-2]`,`n[4]`,…) n'est pas contrôlé par le compilateur et provoque souvent le "plantage" du programme.

En résumé, on peut dire que lorsqu'on déclare un tableau ce qui il y a entre les crochets représente le nombre d'éléments du tableau alors que par la suite, lorsque l'on accède à un élément de ce tableau, ce qui il y a entre les crochets représente l'indice de l'élément.

Exemple :

```
#include <stdio.h>

int main (void)

  {

    int tab[5],i=0,n;

    while (i<5)

      { printf ("entrez un nombre :");
```

```
        scanf("%d",&n);
        tab[i]=n;
        i++;
      }
     i=0;
    printf("les nombres du tableau sont :");
     while (i<5)
      {
       printf("\ntab[%d]=%d",i,tab[i]);
       i++;
      }
  }
```

5.2 Les pointeurs

Un pointeur est une variable destinée à contenir l'adresse d'une variable. Il existe autant de types de pointeurs que de types de variables. Le nom d'un type de pointeur est formé par le nom du type de variable sur lequel il pointe suivi du caractère `*`.

Exemples de déclaration:

```
int * adi;        /* adi est de type int *  */
char* adc ;       /* adc est de type char *  */
unsigned int * adu; /* adu est de type unsigned int *  */
char ** ppc ;  /* ppc est de type char** c'est donc un pointeur*/
               /* sur une variable de type char * */
```

Considérons les instructions suivantes :

```
int * ad ;
int i;
ad = &i;
*ad = 512 ;
```

L'instruction `ad = &i;` affecte au pointeur ad l'adresse de la variable `i`. On alors dit que `ad` *pointe sur* `i`. On dispose ainsi de deux manières d'accéder au contenu de `i` : la manière classique en faisant référence à `i` directement, comme dans `i=512;`, et la manière indirecte en faisant référence au contenu de l'entier ayant pour adresse `ad`, comme dans `*ad =512;`.

L'opérateur `&` placé devant une variable donne l'adresse de celle-ci.

L'opérateur `*` placé devant un pointeur donne accès au contenu de l'adresse pointée.

Pour affecter une valeur à un pointeur, on a le choix entre trois méthodes :

1/ Affecter l'adresse d'une variable déjà déclarée à l'aide de l'opérateur `&` comme dans l'exemple ci-dessus : `ad = &i;`

2/ Affecter une valeur "en dur" : c'est-à-dire que le programmeur choisit lui-même l'adresse qui sera modifiée lors de l'accès au contenu du pointeur.

Exemple :

```
char * porta ;

porta = 0x1000 ; /* attention ! voir remarque */

*porta = 0xF0 ; /* Le contenu de l'adresse 0x1000 prend 0xF0 */
```

Remarque : Cette méthode ne doit être employée que pour accéder à un objet ayant adresse fixe et immuable comme un port d'entrée/sortie, un registre d'un microcontrôleur ou d'une carte électronique. Utiliser l'adressage en dur pour tout autre utilisation conduit généralement à un plantage du système ou pire à un fonctionnement erratique car on n'est jamais certain que la zone mémoire choisie n'est pas utilisée par ailleurs. D'ailleurs la plupart des compilateurs génère un avertissement (warning) ou une erreur sur des instructions du type `porta = 0x1000 ;` et impose l'utilisation de la conversion explicite. Il faut donc écrire:

```
porta = (char *)0x1000;
```

3/ Affecter une adresse obtenue du système d'exploitation par l'appel d'une fonction d'allocation dynamique. Cette adresse est généralement située dans une zone mémoire spéciale appelée le tas (heap en anglais).

Un constante particulière `NULL` est définie pour indiquer qu'un pointeur ne pointe pas sur une adresse mémoire valide .

Exemple :

```
int * p ;

.....

if (p != NULL )

   {            /*  p pointe une adresse valide */

   .....}  /*  on peut donc accéder à *p */

else

   {          /* p ne pointe sur rien de valide */

   .....}
```

5.3 Opérations sur les pointeurs

Il est possible d'ajouter un pointeur de type quelconque et une expression entière. Dans ce cas la valeur du pointeur augmente de la valeur de l'expression multipliée par la taille du type pointé.

Exemple :

```
int i=4; double x;

double * adx;

adx = &x;

adx = adx + i ;
```

`adx` augmente de 4 fois la taille d'une variable de type `double` sur la machine considérée.

Ceci explique pourquoi il est nécessaire d'indiquer le type lors de la déclaration d'un pointeur. En effet l'instruction `adx=adx+i;` ne donnera pas le même résultat selon que `adx` est de type `char *` ou `double *` par exemple. En fait, on a rarement besoin de connaître effectivement la taille du type pointé car ces opérations sont le plus souvent réalisées lorsque `adx` pointe sur un élément d'un tableau. Dans ce cas l'ajout de `i` fait pointer `adx` i élément plus loin dans le tableau (cf. §6.4).

Les opérateurs d'incrémentation et de décrémentation `++` et `--` s'appliquent également aux pointeurs. De la même manière que pour l'addition, le pointeur augmente ou diminue de la taille du type pointé. Pour un tableau, le pointeur pointe donc sur l'élément suivant ou sur l'élément précédent.

5.4 Tableaux et pointeurs

Considérons les instructions suivantes :

```
int tab[10];

int i = 2;

*(tab + i) = 15 ;
```

Par convention**, le nom d'un tableau , `tab`, est l'adresse du premier élément**. Donc le nom augmenté de i, `(tab+i)`, représente l'adresse du $i^{ème}$ élément et `*(tab+i)` est équivalent à `tab[i]`. Ainsi, l'instruction `*(tab + i)=15;` place la valeur 15 dans le $3^{ème}$ éléments du tableau `tab`.

Retenons l'équivalence **: `tab[i]` ⇔ `*(tab +i)`**

Comme le nom du tableau `tab` est l'adresse du premier élément c'est donc une constante de type `int *`.

Dans le programme du paragraphe 6.1 les lignes

```
scanf("%d",&n);

tab[i]=n;
```

peuvent être remplacées par `scanf("%d",tab+i);`

et la ligne

```
printf("\ntab[%d]=%d",i,tab[i]);
```

peut se remplacer par

```
printf("\ntab[%d]=%d",i,*(tab+i));
```

5.5 Les chaînes de caractères

Les chaînes de caractères sont des tableaux de caractères se terminant par le caractère de valeur 0 (noté `'\0'`). (Une chaîne de n caractères imprimables occupe donc un espace de n+1 octets en mémoire). Les chaînes de caractères constantes sont notées entre " et ".

Exemple : `char texte[]="bonjour";`

La ligne ci-dessus déclare une chaîne de caractères nommée texte initialisée avec bonjour. Il n'est pas obligatoire de préciser la taille, le compilateur va alors réserver la place pour 7+1=8 octets.

Si on écrit : `char texte[21]="bonjour";` on déclare une chaîne de 20 caractères dont les 8 premiers sont initialisés (le 8ème octet étant la fin de chaîne).

Pour imprimer ou lire des chaînes de caractères avec les fonctions `printf` et `scanf` le format est `%s` (string).

Il existe de nombreuses fonctions dans la librairie standard du C pour traiter les chaînes de caractères. La plupart des prototypes sont dans **`string.h`**. En voici quelques-unes :

Remarques préalables :

- `size_t` est un type que l'on peut considérer équivalent à `int`.
- `const` devant un nom de type en paramètre d'une fonction interdit que le paramètre soit modifié par la fonction (bien que passé par adresse).
- Dans un prototype, on peut rajouter des noms pour les paramètres pour améliorer la lisibilité. Ce nom est alors considéré comme un commentaire par le compilateur.

`size_t strlen (const char * chaine);` Cette fonction renvoie la longueur de la chaîne passée en paramètre. (sans tenir compte du caractère de fin `'\0'`)

`char * strcpy (char * destination, const char * source);` Cette fonction recopie la chaîne `source` à l'adresse `destination`. Elle retourne l'adresse de début de la chaîne `destination`. Une fonction similaire, **`strcat`**, effectue la copie à la fin de la chaîne `destination`.

`int strcmp (const char * chaine1, const char * chaine2);` Cette fonction compare les chaînes `chaine1` et `chaine2` et fournit en retour :

une valeur négative si `chaine1<chaine2`

une valeur positive si `chaine1>chaine2`

zéro si `chaine1=chaine2`

`char * strchr (const char * chaine, char c);` Cette fonction recherche la première occurrence du caractère `c` dans la chaîne `chaine` et fournit son adresse en retour. Elle fournit la valeur NULL si le caractère n'est pas trouvé.

Voici un programme qui montre un exemple d'utilisation de ces fonctions :

```
#include <stdio.h>

#include <string.h>

int main (void)

  {

    char texte[21]="BXnjour ";

    char nom[11];

    char * adc;

    printf("\nEntrez votre nom (10 lettres maxi):");

    scanf("%10s",nom);

    printf("votre nom fait %d lettres\n",strlen(nom));

    strcat(texte,nom);

    strcat(texte,"!!\n");

    texte[1]='o';

    printf("%s",texte);

    if (strcmp(nom,"ordinateur")==0)

        printf("\nNous avons le meme nom!!\n");

    adc=strchr(nom,'i');

    if (adc != NULL)

         {

          printf("il y a au moins un i dans votre nom,");

          printf("il est en position %d\n",adc-nom);

         }

    else

          printf("il n'y a pas de i dans votre nom");

   }
```

Une autre fonction intéressante, dont la déclaration se trouve dans `stdlib.h` , est la fonction **`int atoi(const char * chaine);`** qui convertit une chaîne de caractères en entier (cf. §7.6).

6. Les fonctions

6.1 Généralités

Le langage C permet de découper un programme en plusieurs parties appelées *fonctions*. Ceci permet d'améliorer la lisibilité de gros programmes et de tester indépendamment chaque partie d'un programme. La fonction `main` est la fonction exécutée en premier dans un programme. Voici un exemple d'école d'un programme qui utilise une fonction qui détermine la plus grande de deux valeurs.

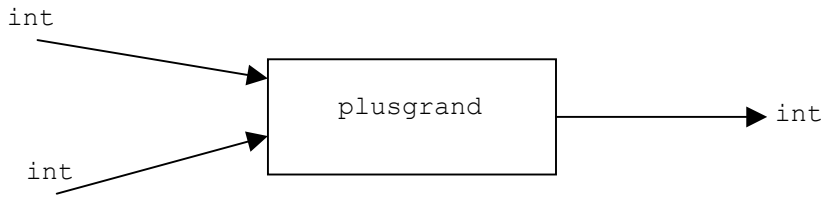
```
#include <stdio.h>

int plusgrand(int , int );

int main(void)
     {
      int n1,n2,g;

      printf("Entrez deux nombres : ");

      scanf("%d %d",&n1,&n2);

      g = plusgrand (n1, n2);

      printf("Le plus grand des deux nombres est : %d \n",g);
     }

int plusgrand(int p1, int p2)
    {

     if (p1>p2)
          return p1;
     else
          return p2;
     }
```

prototype de la fonction `plusgrand` (déclaration de la fonction)

appel de la fonction `plusgrand`

définition de la fonction `plusgrand`

La fonction `plusgrand` peut être considérée comme un programme indépendant qui prend deux variables entières et retourne une variable entière. Ceci peut se représenter par le schéma ci-dessous :



A gauche, on représente les données entrantes, à droite la *valeur retournée*. Ce schéma est décrit au compilateur par la ligne `int plusgrand(int , int );` placée en début de fichier. Cette ligne, appelée *prototype* de la fonction `plusgrand`, constitue la *déclaration de la fonction*. Elle indique au compilateur que par la suite, il pourra rencontrer une fonction nommée `plusgrand`, qui prendra deux entiers et qui en fournira un troisième. Les valeurs qui entrent dans la fonction sont appelées *paramètres* (ou *arguments*) de la fonction. La déclaration par le prototype

permettra au compilateur de vérifier que *l'appel* de la fonction est correctement effectué. La vérification porte sur le nombre et le type des paramètres ainsi que sur le type de la valeur retournée. Attention dans le cas général l'ordre intervient.

Le traitement particulier effectué par la fonction, "le programme", est décrit dans la *définition* de la fonction. Pour écrire la définition de la fonction, on recopie la ligne de prototype (sans le point virgule) et on donne un nom à chaque paramètre. Ici `p1` et `p2`.

```
int plusgrand(int p1, int p2)
    {
     ……..

     }
```

Le programme entre les crochets est alors un programme identique à un programme écrit dans un bloc `main` sauf qu'il y deux variables supplémentaires "prédéclarées" : ce sont les paramètres `p1` et `p2`. Les valeurs de ces variables seront reçues de "l'extérieur" de la fonction lors de l'appel de celle-ci. Les paramètres p1 et p2 sont appelés *paramètres formels* car ils ne servent qu'à décrire le comportement de la fonction. On peut faire l'analogie avec la définition d'une fonction en mathématiques. Dans $f(x)=x^2+x+1$, la lettre x ne sert qu'à décrire la fonction f. La valeur n'intervient qu'au moment du calcul d'une valeur particulière de f comme f(3) qui correspond à l'appel en C.

On peut si c'est nécessaire déclarer d'autres variables comme dans une fonction `main`. Des variables déclarées à l'intérieur de la fonction sont dites *locales* à la fonction.

On remarque une nouvelle instruction, l'instruction `return`. La syntaxe générale est

```
return expression ;
```

Lorsque qu'une instruction `return` est rencontrée, l'expression est évaluée, puis l'exécution de la fonction s'arrête et la valeur de l'expression est prise comme valeur de retour de la fonction. L'exécution se poursuit alors dans le programme appelant (`main` ou autre fonction).

Voyons maintenant le programme principal, i.e. la fonction `main`.

On y trouve un appel de notre fonction

```
g = plusgrand (n1, n2);
```

Tout ce passe comme si la valeur de `n1` était affectée au paramètre formel `p1` et la valeur de `n2` au paramètre `p2`. `n1` et `n2` sont les *paramètres effectifs*. Le contrôle est alors passé à la fonction `plusgrand` et après une instruction `return expression;`, la valeur de `expression` est affectée à la variable `g`.

Les lignes `printf(……;` et `scanf(……;` sont aussi des appels de fonctions. Ce sont des fonctions prédéfinies dont la définition ne se trouve pas dans le fichier source mais dont le code exécutable est placé dans une *librairie*, la *librairie standard*. Les prototypes doivent alors se trouver dans le fichier . Ils se trouvent dans le fichier `stdio.h` et sont inclus dans le programme par la directive `#include <stdio.h>`.

Une fonction n'a d'intérêt que si on l'utilise plusieurs fois dans un même programme ou dans des programmes différents. Comme `plusgrand` prend des paramètres de type entier, toute expression de type entier peut devenir un paramètre de `plusgrand`. De même la valeur de retour de `plusgrand` est de type `int` et à ce titre elle peut être utilisée partout où une expression de type `int` peut être utilisée. Voici quelques possibilités:

```
g=plusgrand(n2+n1,4);
```

```
n1 = g + plusgrand(8,9);

n2=plusgrand(plusgrand(3,n2),n2*n1);
```

On n'est pas obligé d'affecter la valeur de retour à une variable :

```
if (plusgrand(n1,n2) < 0)

{ printf ("Les deux nombres sont négatifs");

}
```

On n'est même pas obligé d'utiliser la valeur de retour d'une fonction. Ce dernier cas n'a d'intérêt que si la fonction a un autre effet que de renvoyer une valeur comme par exemple un affichage (`printf`) ou une acquisition (`scanf`).

6.2 Les différents types de fonctions

Pour la suite les éléments entre < et > sont obligatoires et les éléments entre [et] sont optionnels. < et > ainsi que [et] ne doivent pas figurer dans le programme.

6.2.1 Fonctions avec paramètre(s) et valeur de retour

C'est le cas de la fonction `plusgrand` du §7.1.

Le prototype est de la forme :

```
<type de la valeur retournée> <nom de la fonction>(<type du premier paramètre>[,<type du 2ème paramètre>[,…]]);
```

Exemple : `int fonction1 (float, int, int );`

Un appel possible est : `int a,b,c; float x;`

```
a=fonction1(x,b,c);
```

6.2.2 Fonctions avec paramètre(s) et sans valeur de retour

C'est par exemple une fonction qui affiche une donnée. Il n'y a pas d'instruction `return` dans sa définition. Le type de retour est alors indéfini, on emploie le mot clé `void`.

Le prototype est de la forme :

```
void  <nom de la fonction>(<type du premier paramètre>[,<type du 2ème paramètre>[,…]]);
```

Exemple :

```
#include <stdio.h>

void aff_som(double , double );
```

```
int main(void)
     {
        double a=9,b=4;
        aff_som(a,b);
        aff_som(b,6);
     }
void aff_som(double px, double py)
      {
         printf( "la somme de mes paramètres est : %d", px+py);
       }
```

6.2.3 Fonctions sans paramètre et avec valeur de retour

C'est par exemple le cas d'une fonction qui retourne une variable aléatoire comme la fonction `rand` de la librairie standard ou la fonction `getchar` qui lit un caractère au clavier.

Le prototype est de la forme :

```
<type de la valeur retournée> <nom de la fonction>(void);
```

Exemple : dans `stdio`.h , on trouve le prototype : `int getchar(void);`

```
#include <stdio.h>
int main (void)
        { char touche;
          printf("Appuyez sur une touche ");
          c=getchar();
          printf("Vous avez tapé sur %c\n", touche);
        }
```

6.2.4 Fonctions sans paramètre ni valeur de retour

Ce type de fonction peut être utilisé pour dupliquer un code répétitif sans dupliquer toutes les instructions.

Le prototype est de la forme :

```
void <nom de la fonction>(void);
```

Exemple:

```
#include <stdio.h>

void att_touche (void) ;

int main(void)

   {  att_touche();

      printf("Merci\n");

      att_touche();

      printf("Au revoir\n");

   }

void att_touche (void)

   { char c;

    printf("Tapez sur une touche");

    c=getchar();

    putchar('\n');

   }
```

6.3 Simulation d'une transmission "par adresse"

Lors de l'appel d'une fonction, le programme appelant transmet seulement une copie de la valeur des paramètres. Il est donc impossible pour une fonction de modifier la valeur d'un de ces paramètres et que cette modification se retrouve dans le programme appelant. Considérons la fonction suivante :

```
#include <stdio.h>

void aff_A(int );

int main(void)

   {    int n=3;  printf(" %d",n);

        aff_A(n); printf(" %d",n);

   }

void aff_A(int n)

   { while (n--)

        { printf("A "); }

     printf(" %d",n);

   }
```

L'affichage produit par ce programme est : 3 `A A A 0 3`

En effet, bien que l'on ait employé le même nom `n` dans la définition de la fonction et dans le programme principal (ce qui est déconseillé au programmeur débutant!), les deux `n` n'ont rien de commun si ce n'est qu'ils ont la même valeur lors de l'appel de la fonction. Dans la fonction le paramètre formel `n` est décrémenté jusqu'à zéro. Ainsi l'affichage de la valeur de `n` à la fin de la fonction `aff_A` donne `0` mais dans le programme principal la variable `n` n'est pas modifiée et l'affichage donne toujours `3`. On obtiendrait un programme identique avec la définition de `aff_A` ci-dessous :

```
void aff_A(int x)

   { while (x--)

         { printf("A "); }

     printf(" %d",x);

   }
```

Le nom du paramètre formel n'a donc aucune importance.

Si on veut que la fonction puisse accéder réellement à la variable `n` du programme principal pour pouvoir la modifier, il faut que la fonction ait connaissance de l'adresse de `n` (et pas seulement de sa valeur). L'appel de la fonction modifiée sera alors `aff_Av2(&n);`. Le paramètre est alors l'adresse d'un `int`, c'est donc une variable de type `int *` . Le nouveau prototype sera alors : `void aff_Av2(int * );` et le programme modifié deviendra :

```
#include <stdio.h>

void aff_Av2(int * );

int main(void)

   {    int n=3;  printf(" %d",n);

        aff_Av2(&n); printf(" %d",n);

   }

void aff_Av2(int  * adn)

   { while (*adn--)

         printf("A ");

     printf(" %d",*adn);

   }
```

L'affichage produit sera alors : 3 `A A A 0` **`0`** . La fonction accède maintenant au contenu de l'adresse qui lui est passé en paramètre. En conclusion :

La modification de la valeur d'un paramètre dans une fonction ne se répercute jamais dans le programme appelant.

Si on veut que la fonction accède directement à une variable du programme appelant pour la modifier, il faut lui passer l'adresse de cette variable.

6.4 Tableaux en paramètre de fonction

Pour passer un tableau à une fonction, il faut lui passer l'adresse du premier élément. Si on veut des fonctions qui puissent traiter des tableaux de tailles différentes, il faut également passer le nombre d'éléments du tableau.

Voici l'exemple d'une fonction qui imprime les éléments d'un tableau d'entier :

```
/* Prototype */

void imprime_tab(int *,int );

/* Definition */

 void imprime_tab(int * tableau, int taille)

      {  int i=0;

         while (i<taille)

            {

             printf("\ntableau[%d]=%d",i,tableau[i]);

            i++;

            }

      }

/* Utilisation dans un programme */

int main(void)

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

       imprime_tab(tab,4);

   }
```

Comme on passe l'adresse de début du tableau à la fonction, celle-ci peut alors modifier la valeur des éléments du tableau. Dans de nombreux cas, une telle modification n'est pas désirée. On peut éviter le risque de modification (du à une erreur de programmation) en employant le mot clé `const` devant le type du paramètre qui ne doit pas être modifié. Le prototype deviendrait alors :

```
void imprime_tab(const int *,int );
```

et seule la première ligne de la définition serait modifiée pour être conforme au nouveau prototype.

```
 void imprime_tab(const int * tableau, int taille)

   { ..... }
```

Avec ce nouveau prototype, toute tentative de modification d'un élément du tableau, par une instruction du type `tableau[i]=3;` par exemple, conduira à une erreur de compilation.

On peut également utiliser la notation du type tableau d'entiers, `int []`, pour le premier paramètre de la fonction `imprime_tab()`. Dans ce cas le prototype deviendrait :

```
void imprime_tab(int [],int );
```

et la première ligne de la définition deviendrait :

```
void imprime_tab(int tableau[], int taille)

   { ..... }
```

6.5 La compilation séparée

Le langage C permet de découper un programme en plusieurs fichiers afin d'améliorer la lisibilité de gros programmes (en regroupant dans un même fichier les parties d'un programme participant à une même fonctionnalité de l'application), de permettre le travail en équipe (chaque membre travaillant sur un fichier ou groupe de fichiers), et la réutilisation de fonctions déjà écrites pour d'autres applications (librairies de fonctions) ou faisant parties du langage (librairie standard). Une application courante comprend un bonne dizaine de fichiers.
Nous allons découper le programme du paragraphe 7.1 sur trois fichiers.

Un premier fichier nommé *fichier en-tête* contiendra la déclaration de la fonction `plusgrand`. Traditionnellement, on lui donne l'extension `.h` (header). Un deuxième fichier contient la définition de la fonction `plusgrand`. Il doit porter l'extension `.c` puisqu'il est destiné à être compilé. On peut l'appeler `fonction.c` par exemple. Le troisième fichier contiendra le reste du programme avec une simple modification : la déclaration de la fonction sera remplacée par l'inclusion du fichier en-tête. Comme il contient la fonction `main`, on peut l'appeler `main.c`.

Fichier `fplusg.h`

```
   int plusgrand(int, int );
```

Fichier `fplusg.c`

```
/*******************************************************/
/* Fonction : plusgrand                                */
/* But : retourne le plus grand de ces deux paramètres */
/* Paramètres :                                        */
/*  int p1 : première valeur à comparer                */
/*  int p2 : deuxième valeur à comparer                */
/* Retour :                                           */
/*    int : plus grand entre p1 et p2                  */
/*******************************************************/
int plusgrand(int p1, int p2)
    {

     if (p1>p2)
          return p1;
     else
          return p2;
     }
```

Fichier `main.c`

Inclusion du prototype de la fonction `plusgrand`

```
#include <stdio.h>

#include "plusgrand.h"

int main(void)
```

```
{
 int n1,n2,g;

 printf("Entrez deux nombres : ");

 scanf("%d %d",&n1,&n2);

 g = plusgrand (n1, n2);

 printf("Le plus grand des deux nombres est : %d \n",g);
}
```

La méthode pour générer un programme exécutable dépend du compilateur utilisé. Dans certains environnements on doit créer un projet. Un projet est un fichier qui contient la liste des fichiers nécessaires à la génération du programme exécutable ainsi que les liens de dépendances entre ces différents fichiers. En mode ligne de commande, il suffit de donner la liste des fichiers à compiler. Pour le compilateur gcc par exemple, on tapera : `gcc –o pgm plusgrand.c main.c` où `pgm` est le nom du fichier exécutable.

6.6 Les arguments de la fonction `main`

La fonction main est une fonction différente des autres . Elle est obligatoire dans un programme et c'est la fonction qui est appelée en premier juste après l'initialisation des variables statiques (globales). L'environnement extérieur au programme, le système d'exploitation, transmet à la fonction main la ligne de commande qui a été invoquée lors de l'appel du programme. Par exemple, si on a tapé en ligne de commande : `prog bonjour monsieur 4` pour exécuter un programme nommé prog. Les mots `prog`, `bonjour` et `monsieur` et la valeur `4` seront transmis à la fonction main sous forme de chaîne de caractères. La fonction main pourra alors les exploiter pour modifier le déroulement du programme. Plus précisément, le prototype complet de la fonction main est :

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

Le premier paramètre `argc`, est le nombre de paramètres. Ce sera au minium 1 car le nom du programme est le premier paramètre.

Le second paramètre `argv`, est de type `char **` , c'est donc un pointeur sur des éléments de type `char *`. C'est donc un pointeur sur des chaînes de caractères. Ces différentes chaînes de caractères sont les paramètres tapés en ligne de commande au lancement du programme.

Voici un programme qui imprime la liste de ces paramètres :

```
#include <stdio.h>

int main(int argc , char ** argv )

   {

     while (argc--)

       {

         printf("%s ",*argv++);

       }

   }
```

Nous allons modifier le programme du paragraphe 7.1 pour que les deux nombres soient rentrés sur la ligne de commande au lancement du programme. En supposant que `pgm` soit le nom du programme exécutable, l'utilisateur va entrer une commande de la forme `pgm 144 432`. Les deux nombres seront transmis à la fonction `main` sous forme de chaîne de caractères (ex '1','4','4',0 pour 144) et il faudra les convertir en entier pour que la fonction `plusgrand()` puisse les exploiter. Pour cela on peut utiliser la fonction `int atoi(char *);` qui convertit une chaîne de caractères en un nombre entier.

```
#include <stdlib.h>

#include <stdio.h>

int plusgrand(int , int );

int main(void)
{
   int n1,n2,g;

   argv++; /* argv pointe la deuxième chaîne qui représente n1 */

   /* *argv est l'@ de début de la 2eme chaîne de caractère */

   n1=atoi(*argv);

   argv++; /* argv pointe la troisième chaîne qui représente n2 */

   n2=atoi(*argv);

   g = plusgrand (n1, n2);

   printf("Le plus grand des deux nombres est : %d \n",g);
 }

int plusgrand(int p1, int p2)
{

     if (p1>p2)
          return p1;
     else              /* Nb: le else n'est pas nécessaire */
          return p2;   /* car l'exécution de la fonction   */
 }                     /* s'arrête après un return .       */
```

6.7 La valeur de retour de la fonction `main`

Le programme peut retourner une valeur en fin d'exécution. Cette valeur peut être exploitée par le système d'exploitation ou par le programme qui l'a appelé dans le cas d'un système multitâche. C'est pourquoi on trouve `int` en type de retour de la fonction `main`. La valeur de retour peut être passée par la fonction `void exit(int);` de `stdlib.h` qui met fin au programme ou par une instruction `return`. Deux constantes prédéfinies `EXIT_SUCCESS` et `EXIT_FAILURE` peuvent être utilisées pour indiquer une terminaison respectivement normale ou anormale du programme. Si il n'y a pas de système d'exploitation, comme dans certiains systèmes à microcontrôleur, le type de retour de `main` peut être simplement `void`. Les prototypes de `main` peuvent donc être :

```
void main(void);  ou int main (void);

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

7. Les fonctions `printf` et `scanf`

La fonction **`printf`** permet d'afficher des caractères sur la sortie standard (généralement l'écran). Les caractères affichés sont soit ceux présents dans la chaîne de caractères fournie en premier paramètre, soit le résultat d'une conversion des paramètres suivants selon une règle définie par le code qui suit le caractère `%` dans la chaîne de caractères. Exemple :

```
int i=3;

double x=20.6;

printf("La valeur de i est %d et la valeur de x est %f.",i,x);
```

affiche : `La valeur de i est 3 et la valeur de x est 20.60000.`

Le deuxième paramètre, `i`, est converti au format `d` c'est-à-dire décimal. Le troisième est converti au format `f` c'est-à-dire réel en notation décimale.

Liste des principaux formats (voir les nombreuses autres possibilités dans un ouvrage de référence):

`%c` : caractère "en clair". Affiche le symbole correspondant à la valeur de l'expression.

`%d` : décimal. Affiche la valeur en décimal d'expression de type `int` (ou `char`).

%x : hexadécimal. Ecrit la valeur hexadécimal des expression de type `int` ou `char` (`%X` hexa en majuscule).

`%u: unsigned int` (ou `unsigned char`)

`%f` : pour les types `double` ou `float`, écrit la valeur en notation décimale.

`%e` : pour les types `double` ou `float`, écrit la valeur en notation scientifique.

`%s` : affiche la chaîne de caractères dont l'expression représente l'adresse de début.

`%%` : affiche le caractère `%`.

`%03d` : affiche en décimal sur 3 caractères en complétant par des zéro à gauche si nécessaire.

La fonction **`scanf`** permet de lire des données sur l'entrée standard (généralement un clavier) et de les convertir avec des codes de formats similaires à la fonction `printf`. Les formats `%c`, `%d`, `%f` et `%s` sont interprétés de la même manière que pour `printf`. L'interprétation des formats numériques s'arrête dès la rencontre d'un caractère non numérique ou d'un séparateur comme un espace ou un saut de ligne.

Exemple : `int n; float x;`

```
scanf("%f %d",&x,&n);
```

Les valeurs tapées au clavier seront converties en `float` dans `x` et en `int` dans `n`.

Noter bien qu'il est nécessaire de passer l'adresse des variables pour que la modification puisse se répercuter dans la suite du programme.

8. Les structures

Une structure est un type de données qui regroupe plusieurs variables de type différents.

Exemple :

```
struct s_etudiant

      {  char nom[21];

         char prenom[21];

         double moyenne;

      } ;
```

Les instructions ci-dessus définissent une structure nommée `s_etudiant` composée de trois *champs*. Un champ est un élément d'une structure. Ici un champ est destiné à contenir le nom d'un étudiant, un autre son prénom et un autre sa moyenne. Les instructions ci-dessus ne définissent qu'un type, elles ne réservent pas de variable de type `s_etudiant`. La réservation d'une variable se fait comme pour un type ordinaire en ajoutant le mot clé `struct` comme si le nom du type était `struct s_etudiant` :

```
struct s_etudiant etudiant1 ;
```

ou pour un tableau dont les éléments sont des structures :

```
struct s_etudiant classe[24] ;
```

On accède alors à un champ par l'opérateur **.** (point).

Exemple :

```
strcpy(etudiant1.nom,"EINSTEIN");  /* copie de chaîne */

strcpy(etudiant1.prenom,"Albert");

etudiant1.moyenne=20;

classe[17].moyenne=15.3;
```

Pour passer une structure à une fonction, il suffit de mentionner le nom `struct s_etudiant` dans le prototype comme pour une variable ordinaire. Exemple : `int f( struct s_etudiant, int);`.

La fonction `f` n'aura alors accès qu'à la valeur des champs de la structure. Si on veut passer l'adresse, le prototype deviendra par exemple `int f1(struct s_etudiant *, int);` . Elle pourra être appelée par `f1(&etudiant1,5);` . On va donc manipuler un pointeur sur une structure :

```
struct s_etudiant * ad_etudiant ;
```

Dans ce cas on accédera à un champ par l'opérateur -> au lieu de l'opérateur point.

```
ad_etudiant = &etudiant1;

ad_etudiant->moyenne=18; /* ou bien (*ad_etudiant).moyenne=18; */
```

9. Classes d'allocation et portées des variables

Il existe trois classes d'allocations de variables :

- *La classe statique* : Les variables de cette classe possèdent une adresse définie lors de la compilation et qui ne change pas au cours de l'exécution du programme. La zone mémoire occupée par ces variables ne peut donc pas être réutilisée au cours de l'exécution du programme (même si leur valeur n'est plus utilisée). La zone mémoire occupée par ces variables est initialisée à zéro avant l'appel de la fonction main. La valeur des variables statiques est donc nulle si on ne précise aucune valeur d'initialisation au moment de leur déclaration.

- *La classe automatique* : Les variables automatiques sont des variables locales dont l'emplacement mémoire est alloué au moment de l'entrée dans la fonction ou dans le bloc où elles sont déclarées. Leur valeur initiale est donc quelconque si on ne précise aucune valeur au moment de leur déclaration. L'emplacement alloué à une variable automatique est libéré au moment de la sortie du bloc ou de la fonction. Cet emplacement peut être réutilisé pour d'autres variables automatiques dans la suite du programme.

- *La classe register* : C'est la classe des variables qui sont stockées dans un registre du microprocesseur. Une variable automatique de type scalaire – pas une structure ni un tableau – peut être de la classe register si sa déclaration est précédée du mot clé `register` et si suffisamment de registres sont disponibles. L'utilisation de cette classe peut réduire le temps d'exécution d'une partie critique d'un programme.

Type de variables	Déclaration	Portée*	Classe d'allocation
Globale	en dehors de toute fonction	la partie du fichier source suivant sa déclaration	statique
Locale à une fonction	en début de fonction	la fonction	automatique
Locale à un bloc	en début de bloc	le bloc	automatique
Locale "rémanente"	en début de fonction, avec l'attribut `static`	la fonction	statique

d'après "Programmer en langage C" de C. Delannoy Editions Eyrolles.

*La porté d'une variable est la zone du programme dans laquelle cette variable est accessible.

Le mot clé **extern** placé devant la déclaration d'une variable globale permet l'utilisation d'une variable globale déclarée dans un autre fichier.

Exemple :

Fichier 1 :

```
int g = 6; /* déclaration "réelle" de la variable globale g */

int main(void)

{ ……

  g=9;

  ……

}
```

Fichier 2 :

```
extern int g ; /* répétition de la "déclaration" précédée de extern */

               /* pour pouvoir utiliser g dans ce fichier         */

               /* aucune initialisation n'est permise             */

int f1(int) ;

int f1(int i)

   {

     g=i ;

    ……

   }
```

Le mot clé **`static`** possède deux significations :

- utilisé devant la déclaration d'une variable locale, il place cette variable dans la classe statique.
- utilisé devant la déclaration d'une fonction, il limite la portée du nom de la fonction au fichier source. La fonction n'est alors utilisable que dans le fichier source où elle est définie.

Exemple 2 :

```
#include <stdio.h>

int reserver_place(void);

int main(void)

      { int i =20;

        while(i--)

            reserver_place();

      }

int reserver_place(void)

      {

       static int place = 15 ; /* La valeur de cette variable se conserve*/

                               /* entre chaque appel                     */

      if (place)

        {

         place = place - 1;

         printf("Reservation effectuee. ");

        }

      if (place)

         printf("Il reste %d place(s).\n",place);

     else

        printf("Il ne reste plus de place !\n");


    }
```

10. Les fichiers

Le langage C dispose de fonctions permettant de lire et d'écrire dans des fichiers stockés sur un disque dur ou sur autre périphérique (disquette, cd-rom, zip, etc..). Toutes ces fonctions utilisent un "pointeur" sur le fichier pour accéder aux informations. Le terme "pointeur" est mis entre guillemets car il ne désigne pas vraiment une adresse mais plutôt un index sur le prochain caractère (ou bloc d'informations) qui sera lu ou écrit dans le fichier. Ce "pointeur" est de type `FILE *` (`FILE` en majuscule). Sa valeur réelle a peu d'importance (sauf si c'est `NULL`) car il existe des fonctions spéciales pour l'initialiser et pour le modifier.

Pour utiliser un fichier il faut préalablement "l'ouvrir" en lecture ou en écriture. L'ouverture est utilisée pour initialiser le pointeur sur le fichier. Ensuite on peut lire ou écrire dans le fichier (selon le mode d'ouverture réalisé). Il faut ensuite "fermer" le fichier.

10.1 Ouvertures/ Fermeture

Déclarons un pointeur sur fichier : **`FILE * f;`**

mode d'ouverture

Ouvrons le fichier en lecture : **`f=fopen("toto.txt","r");`**

`toto.txt` est le nom du fichier à ouvrir. Le fichier est recherché dans le répertoire courant. `"r"` indique une ouverture en lecture.

Il existe d'autres modes d'ouverture, en voici quelques uns :

- `"w"` : ouverture pour écriture. Attention le fichier est détruit s'il existe déjà.

- `"a"` : ouverture pour écriture après la fin du fichier (append). Si le fichier n'existe pas il sera créé et on se ramène au cas `"w"`

- `"rb"/"wb"` lecture/écriture de fichier en mode binaire : il n'y aucune interprétation des caractères lus/écrits. Dans le mode `"r"` et `"w"` certaines séquences de caractères peuvent avoir une interprétation particulière (cr suivi de lf par exemple). Cette distinction n'a de sens que pour les systèmes d'exploitations qui font un cas particulier des fichiers textes (MS-DOS par exemple).

Après l'ouverture le pointeur (`f` ici) indique le premier caractère du fichier.

Si le pointeur `f` est différent de la valeur `NULL`, on peut alors lire ou écrire dans le fichier en utilisant des fonctions spécifiques. La lecture ou l'écriture déplacent automatiquement le pointeur après le dernier caractère lu ou écrit. On peut donc lire ou écrire les informations les unes après les autres sans se soucier de la gestion du pointeur (cette méthode est parfois appelée *accès séquentiel*). On peut également déplacer le pointeur à des positions déterminées dans le fichier à l'aide de fonctions spéciales (cette méthode est parfois appelé *accès direct*). On peut utiliser en même temps les deux méthodes d'accès à un même fichier.

Si le pointeur `f` vaut `NULL`, c'est qui il y a eu un problème. On ne peut ni lire ni écrire et toute tentative se soldera par un "plantage" du programme (terminaison anormale).

Après l'utilisation du fichier (lecture ou écriture), il faut le refermer par :

```
fclose(f);
```

10.2 Ecritures dans un fichier

Il existe deux méthodes d'écriture dans un fichier. L'écriture *formatée* et l'écriture *brute*. Il est possible de panacher les deux méthodes pour un même fichier.

L'écriture formaté est la seule méthode portable.

10.2.1 Ecriture formatée

Par cette méthode, on crée un fichier au format "texte", qui peut être lu par n'importe quel éditeur de texte. Les fonctions sont très simples à utiliser car ce sont les mêmes que celles employées pour afficher un message à l'écran, avec un paramètre supplémentaire qui est un pointeur de fichier. Le pointeur doit avoir été préalablement initialisé par un appel à la fonction `fopen()` avec `"w"` ou `"a"` comme mode d'ouverture.

Les deux principales fonctions sont :

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

Ecrit le caractère `c` (après conversion en `unsigned char`) à la position courante sur le fichier `f`. Retourne la valeur écrite ou la valeur `EOF` si la fin du fichier est rencontrée ou en cas d'erreur.

`int fprintf (FILE * f, const char * format,...);`

Cette fonction est identique à la fonction `printf` pour le fichier indexé par `f`. La valeur retournée est le nombre de valeurs effectivement écrites ou la valeur `EOF` si la fin de fichier est rencontrée.

Exemple :

```
/* Ce programme crée un fichier sinus.txt */
/* et y écrit les valeurs de la fonction y(t)=17sin(wt)
/* avec w=2pi/T pour 256 points entre 0 et T */
#include <stdio.h>
#include <math.h>
#define PI 3.14159265359
#define N_MAX 255
#define AMPLITUDE 17
   int main(void)
   {
      FILE * fw ;       /* Déclaration d'un pointeur de fichier */
      int n=0;                    /* indice du point courant */
      double y;
```

```
/* Ouverture en écriture de sinus.txt dans le répertoire courant*/
      fw= fopen("sinus.txt","w");
      if ( fw != NULL )
      {                    /* Ecriture possible */
         fprintf(fw,"Valeurs de 17sinus(wt) sur une periode :\n");
         while ( n <= N_MAX)
         {
            y=AMPLITUDE*sin((2*PI*n)/N_MAX);
            /* Ecriture dans le fichier comme à l'écran */
            fprintf(fw,"%d %lf\n",n,y);
            n++;
         }
         fclose(fw);                /* Fermeture du ficher */
      }
      else                        /* Ecriture impossible */

      {
       printf("\nImpossible de créer le fichier \"sinus.txt\"\n");
      }
   }
```

10.2.2 Ecriture brute (binaire)

Par cette méthode, on écrit directement les valeurs binaires se trouvant en mémoire dans un fichier. Cette méthode d'écriture n'est pas portable car les représentations internes des variables varient d'un environnement à un autre. Par exemple les entiers peuvent être codés par 2 ou 4 octets ou plus. Néanmoins cette méthode d'écriture génère des fichiers plus compact. Les fichiers générés par des écriture brute, ne pourront pas être lus par des éditeurs de texte classiques. On devra utiliser un programme spécifique ou un éditeur hexadécimal.

La fonction d'écriture brute est `fwrite()` :

`size_t fwrite(void * adr, size_t taille, size_t nblocs, FILE * f);`

Ecrit à la position f sur le fichier les `nblocs` de `taille` octets à partir de l'adresse `adr`.

Retourne le nombre de blocs réellement écrits. Une valeur inférieure à `nblocs` indique une erreur.

Exemple : Ecriture brute d'un réel, puis d'un tableau de 5 entiers.

```
#include <stdlib.h>

#include <stdio.h>

int main (void)

   {

      FILE * fw;

      int tab[5] = { 256, 65534, 0 , 15, 2 };

      double x=6.28;

      fw=fopen("toto.bin","w");

      if (fw!=NULL)

      {

        fwrite(&x,sizeof(double),1,fw); /* Ecriture d'un double */

        fwrite(tab,sizeof(int),5,fw);  /* Ecriture de 5 entiers */

        fclose(fw);

       return (EXIT_SUCCESS);

      }

      return (EXIT_FAILURE);

   }
```

Voici, en hexadécimal, le contenu du fichier `toto.bin` obtenu par l'exécution du programme ci-dessous, compilé avec le compilateur `gcc` sous `windows98`© :

```
1F 85 EB 51 B8 1E 19 40 00 01 00 00 FE FF 00 00

00 00 00 00 0F 00 00 00 02 00 00 00
```

La valeur `1F85EB51B81E1940` représente la valeur `6.28` au format double (ici sur 8 octets). Ensuite , on trouve les cinq entiers qui sont codés ici sur 4 octets (32 bits). Ainsi la valeur `256` qui est `0x00000100` sur 32 bits puis la valeur `65534` qui est `0x0000FFFE`, etc… Remarquez que selon le format propre aux microprocesseurs Intel© les octets de poids faibles sont placés avant les octets de poids forts. (c'est l'inverse sur des microprocesseurs Motorola©).

Remarque :

- Il faut toujours employer l'opérateur `sizeof()` pour obtenir la taille d'un type.

10.3 Lectures dans un fichier

Pour lire dans un fichier, il est nécessaire de connaître la structure du fichier pour en déduire les formats attendus. Ainsi le choix de la méthode de lecture, des types des variables dans lesquelles seront stockées les valeurs lues et de l'ordre de la lecture n'est pas indépendant du contenu supposé du fichier. Il est possible de panacher les deux méthodes d'accès à un même fichier.

10.3.1 Lecture d'un fichier contenant des données formatées (texte)

Les principales fonctions sont :

`int fgetc (FILE * f);`

Lit le caractère courant du fichier `f` et fournit la conversion en `int` du caractère lu si on n'est pas en fin de fichier ou la valeur EOF si on est en fin de fichier. (Voir également §11.4)

Exemple :

```
int c;

FILE * f;

f=fopen("toto.txt","r");

c=fgetc(f);  /* c contient le premier caractère de toto.txt */
```

`char * fgets(char * s, int max, FILE * f);` Cette fonction lit des caractères dans le fichier `f` jusqu'à rencontrer le caractère de saut de ligne `\n` ou que `max-1` caractères soient lus. Les caractères lus sont placés dans la chaîne `s`. Cette fonction est très utile pour lire une ligne dans un fichier. Elle retourne la valeur `NULL` si erreur ou une fin de fichier a été rencontrée. Elle retourne l'adresse `s` de début de chaîne si la lecture est correcte.

```
Exemple :

char   chaine[161]; /*Pour stocker un chaîne de 160 caractères
maximum */

fgets(chaine,161,f);  /* Place un ligne du fichier dans le tableau
chaine */
```

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

Cette fonction est identique à la fonction `scanf` pour le fichier indexé par `f`. La valeur retournée est le nombre de valeurs effectivement lues ou la valeur `EOF` si la fin de fichier est rencontrée.

Exemple :

```
int i,j;

fscanf(f,"%d %d",&i,&j); /*Lit deux entiers séparés par au moins
un espace et les place dans les variables i et j */
```

10.3.2 Lecture d'un fichier contenant des données brutes (binaire)

`size_t fread (void * adr, size_t taille, size_t nblocs, FILE * f);`

Lit à la position courante sur le fichier `f`, `nblocs` de taille octets et les range à partir de l'adresse `adr`. Retourne le nombre de blocs réellement lus.

Exemples :

```
double vecteur[12];

fread(vecteur,sizeof(double),12,f); /* Place 12 doubles
consécutifs lus dans f dans le tableau vecteur */

char c;

fread(&c,sizeof(char),1,f); /* Lit 1 octet à la position courante
dans le fichier f et le place dans la variable c */
```

10.4 La détection de la fin d'un fichier

La notion de fin de fichier n'a de sens qu'en lecture. Elle permet de savoir si le pointeur de fichier pointe au delà du dernier octet du fichier.

La détection de la fin d'un fichier peut se faire de deux manières :

- Par appel à la fonction `feof()` :

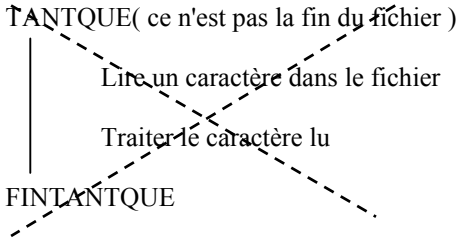
  `int feof(FILe * f);`

  Cette fonction renvoie une valeur non nulle si la fin de fichier `f` est atteinte et une valeur nulle dans le cas contraire.

- Par la valeur retournée par certaines fonctions de lecture comme `fgetc()`**:**

  On obtient la valeur constante **`EOF`** lorsque la fin de fichier est atteinte. Cette constante doit être différente de tous les caractères possibles pour pouvoir être distinguée des caractères normaux. Ce n'est donc pas une variable de type `char`. En général , on lui attribue une valeur négative du type `int`. **Un caractère lu dans un fichier doit donc toujours être déclaré de type `int`**.

Il est important de remarquer que la fin d'un fichier n'est pas atteinte lorsqu'on lit le dernier caractère mais lorsque l'on a tenté de lire au delà. Ainsi pour lire un fichier caractère par caractère jusqu'à la fin, un algorithme naturel du type :



ne peut pas convenir car il conduira à un traitement lorsque la lecture aura retourné `EOF`.

Il faut donc lire un caractère, puis tester si la fin du fichier est atteinte avant de continuer :

```
TANTQUE ( toujours)
|   Lire un caractère dans le fichier
|   SI (c'est la fin du fichier)
|   |   SORTIR de la boucle
|   FINSI
|   Traiter le caractère lu
FINTANTQUE
```

On retrouve le même problème avec la lecture ligne par ligne et la lecture formatée.

Cet algorithme est implémenté dans les programmes des paragraphes suivants.

10.4.1 Lecture caractère par caractère jusqu'à la fin

Cet exemple montre la lecture caractère par caractère d'un fichier nommé `data.txt` dans le répertoire courant. Le traitement réalisé ici consiste simplement à afficher les caractères lus à l'écran et à les compter. Remarquez que le caractère lu dans le fichier est déclaré en type `int` pour éviter la confusion entre la constante `EOF` (`-1` ici) et le caractère de code `0xFF` qui correspond à ÿ en ISO-LATIN-1.

```
/* Ce programme lit le fichier data.txt caractère par caractère */
/* Il compte le nombre de caractères et les affiche à l'écran */
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
FILE * fr ; /* Déclaration d'un pointeur de fichier */
int c ;     /* Le "caractère" c est déclaré en int */
            /* pour permettre un traitement correct de EOF */
unsigned long int nb=0;    /* Le nombre de caractères peut être très grand */
/* Ouverture en lecture de data.txt dans le répertoire courant*/
fr= fopen("data.txt","r");
```

```
if ( fr != NULL )
      {    /* Lecture possible */
        while (1)
            {
             /* Lecture d'un caractère à la position courante */
              c=fgetc(fr);
               if (c==EOF)
                   {
                   /* sortie de la boucle si fin de fichier */
                    break ;
                   }
         /* placer ici le traitement du caractère lu */
             putchar(c);
             nb++;
              }
         fclose(fr);        /* Fermeture du fichier */
         printf("\n data.txt comporte %lu caracteres\n",nb);
        }
       else     /* Lecture impossible */
       {
         printf("\nImpossible de lire le fichier \"data.txt\"\n");
         return EXIT_FAILURE ;
       }
return EXIT_SUCCESS ;
}
```

Une variante souvent utilisée de codage de cet algorithme est présentée en page suivante. On utilise l'opérateur `,` (virgule) qui permet d'effectuer une opération (ici la lecture d'un caractère) avant l'évaluation de l'expression qui sera prise en compte pour continuer la boucle `while()` (ici le test de fin de fichier).

```
/* Ce programme lit le fichier data.txt caractère par caractère */
/* Il compte le nombre de caractères et les affiche à l'écran */
#include <stdio.h>
#include <stdlib.h>
int main(void)
  {
   FILE * fr ;   /* Déclaration d'un pointeur de fichier */
   int c ;       /* Le "caractère" c est déclaré en int */
                 /* pour permettre un traitement correct de EOF */
  unsigned long int nb=0;/* Le nombre de caractères peut être très grand */
  fr= fopen("data.txt","r"); /* Ouverture en lecture de data.txt dans le
répertoire courant*/
  if ( fr != NULL )
    {                            /* Lecture possible */
     while (c=fgetc(fr),c!=EOF)   /*lecture puis test de fin de fichier */
         {
           /* placer ici le traitement du caractère lu */
            putchar(c);
            nb++;
         }
      fclose(fr);                       /* Fermeture du fichier */
      printf("\n data.txt comporte %lu caracteres\n",nb);
      }
   else                        /* Lecture impossible */
      {
       printf("\nImpossible de lire le fichier \"data.txt\"\n");
       return EXIT_FAILURE ;
      }
   return EXIT_SUCCESS ;
   }
```

10.4.2 Lecture ligne par ligne jusqu'à la fin

```
/* Ce programme lit le fichier data.txt ligne par ligne */
/* Il compte le nombre de ligne et affiche celles commençant par un #*/
#include <stdio.h>
#include <stdlib.h>
#define MAX_LIGNE 80 /*Nombre maximum de caractères dans une ligne */
int main(void)
   {
    FILE * fr ;              /* Déclaration d'un pointeur de fichier */
    unsigned long int nbl=0; /* Le nombre de lignes peut être très grand */
    char ligne[MAX_LIGNE+1];   /* Pour stocker une ligne */
   /* Ouverture en lecture de data.txt dans le répertoire courant */
    fr= fopen("data.txt","r");
    if ( fr != NULL )
      {                        /* Lecture possible */
        /* Lecture puis test de fin de fichier */
        while (fgets(ligne,MAX_LIGNE+1,fr), feof(fr)==0)
        {
           /* Placer ici le traitement de la ligne lue */
            nbl++;
            if (ligne[0]=='#')
            {
               printf("%s",ligne);
            }
        }
        printf("\n data.txt comporte %lu lignes\n",nbl);
        fclose(fr);                /* Fermeture du fichier */
      }
      else                      /* Lecture impossible */
      {
        printf("\nImpossible de lire le fichier \"data.txt\"\n");
        return EXIT_FAILURE ;
      }
      return EXIT_SUCCESS ;
   }
```

10.5 Les "fichiers" stdin , stdout et stderr

Lorsqu'un programme démarre trois "fichiers" sont ouverts automatiquement. Ce sont en fait des flots d'entrées/sorties associées à des périphériques de l'ordinateur mais on peut les traiter en C comme s'il s'agissait de fichiers (sauf qu'il n'est pas possible de déplacer le pointeur). Comme ils sont ouverts automatiquement , on peut lire ou écrire sans appel préalable à `fopen()`.

`stdin` : (standard input). Ce flot est ouvert en lecture et est généralement associé au clavier.

`stdout` : (standard output). Ce flot est ouvert en écriture et est généralement associé à un terminal ou à la fenêtre depuis laquelle est lancé le programme dans le cas d'un environnement multifenêtré.

`stderr` : (standard error). Ce flot est ouvert en écriture et est généralement associé au même terminal ou à la même fenêtre que `stdout`. Toutefois certains environnements associent un terminal ou une fenêtre différents pour y afficher les messages d'erreur.

Ainsi :

```
printf("bonjour");
```

est équivalent à

```
fprintf(stdout,"bonjour");
```

et

```
scanf("%d",&i);
```

est équivalent à

```
fscanf(stdin,"%d",&i);
```

10.6 Déplacement du pointeur dans un fichier

Pour l'instant , nous avons vu que lorsque on lit ou on écrit dans un fichier, la position s'incrémente automatiquement du nombre d'octet lus ou écrits. Ainsi, la lecture ou l'écriture suivante se fait à la suite dans le fichier. Cette méthode est appelée *accès séquentiel* au fichier. Pour des fichiers qui ne sont pas intrinsèquement séquentiels (comme un terminal par exemple), il est possible d'utiliser conjointement à l'accès séquentiel, la méthode dite d'*accès direct* par laquelle on fixe la position dans le fichier avant une opération de lecture ou d'écriture en utilisant :

```
fseek(FILE * f, long noct, int org);
```

Cette fonction place le pointeur du fichier `f` à `noct` octets de l'origine `org` qui peut prendre trois valeurs :

Si `org` vaut `SEEK_SET` on se déplace de `noct` par rapport au début de fichier.

Si `org` vaut `SEEK_CUR` on se déplace de `noct` par rapport à la position courante.

Si `org` vaut `SEEK_END` on se déplace de `noct` par rapport à la fin du fichier.

Remarque : Rappelons que cette fonction n'est pas utilisable sur un écran et donc sur `stdout`.

Exemple d'accès direct :

Dans le fichier d'une image au format Bitmap© (.bmp) sous Windows©, on trouve la largeur en pixels de l'image codée sur 4 octets à 0x12 octets de début du fichier et la hauteur en pixels codée sur 4 octets à 0x16 octets du début. Le nombre de couleurs se trouve à 0x2E octets du début codé également sur 4 octets. Le programme `info_bmp.c` donné en exemple extrait ces informations d'une image au format bmp.

10.7 Précautions pour lecture et écriture

Précaution pour la lecture :

L'ouverture en lecture n'est pas "dangereuse" pour un fichier. Par contre avant de lire dans un fichier ouvert en lecture il faut s'assurer que le pointeur n'a pas la valeur `NULL` qui indique un problème d'accès. C'est pourquoi les programmes proposés en exemples contiennent :

```
if (f ! = NULL)
   {
      /* Lecture */
   }
```

Précautions pour l'écriture :

L'ouverture en écriture par le mode `"w"` détruit le contenu du fichier s'il existait préalablement. Si on ne veut pas détruire par mégarde un fichier important, il faut tester l'existence avant l'ouverture en écriture. Pour cela, il suffit de l'ouvrir en lecture. Si le pointeur obtenu a la valeur `NULL`, c'est que le fichier n'existe pas. On peut alors l'ouvrir en écriture sans crainte. Avant d'écrire, il faudra comme dans le cas de la lecture tester que le pointeur n'a pas la valeur `NULL`.

Le programme `precaution.c` illustre ces différentes précautions.

10.8 Exemples de lecture et d'écriture

Voir les programmes aux pages suivantes :

`info_bmp.c` affiche des informations issues de l'en-tête des fichiers image au format bmp. Il utilise la lecture de données brutes et l'accès direct.

`precaution.c` est un programme qui recopie un fichier en un autre en s'assurant de la validité des opérations de lecture/écriture.

```
#include <stdio.h>    /* info_bmp.c */
#include <stdlib.h>
#define MAX_NOM 160
int main (void)
{
FILE * f ;
char nom_bmp[MAX_NOM+1];
int largeur,hauteur,nb_couleurs ;
fprintf(stdout,"Entrez le nom de l\'image au format bmp :");
/* Lecture du nom du fichier image */
fscanf(stdin,"%s",nom_bmp);
/* Ouverture en lecture du fichier image */
f = fopen(nom_bmp,"r");
if ( f != NULL )
 {                  /* Lecture possible */
  fseek(f,0x12,SEEK_SET); /* Déplace le pointeur sur la largeur */
  fread(&largeur,4,1,f); /*Lecture d'un bloc de 4 octets */
            /* Le pointeur est alors à la position 0x16 */
            /* (4 octets lus à partir de 0x12)          */
            /* Un nouvel appel à fseek n'est pas nécessaire */
  fread(&hauteur,4,1,f);   /* Lecture de la hauteur */
  fseek(f,0x2E,SEEK_SET);  /* Déplace le pointeur sur nb couleur*/
  fread(&nb_couleurs,4,1,f); /* lecture du nombre de couleur */

  fprintf(stdout,"\nL'image %s a :\n",nom_bmp);
  fprintf(stdout," - une largeur de %d pixels\n",largeur);
  fprintf(stdout," - une hauteur de %d pixels\n",hauteur);
  fprintf(stdout," - %d couleurs\n",nb_couleurs);

  fclose(f);
  }
  else
  {                  /* Lecture impossible */
    fprintf(stderr,"Impossible de lire %s\n",nom_bmp);
    return EXIT_FAILURE ;
  }
 return EXIT_SUCCESS ;   }
```

```
/************************* precaution.c ****************************/
/* Ce programme demande à l'utilisateur un nom de fichier pour lire */
/* et un nom de fichier pour écrire.                                */
/* Le programme recopie le premier fichier dans le second           */
/* Avant de faire la copie, il vérifie que le fichier source        */
/* existe et que le fichier destination n'existe pas                */
/* version  date     auteur      commentaire                        */
/* 1.0      30/03/01 arlotto     création                           */
/*                                                                  */
/********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>   /* Pour la fonction tolower() (passage en minuscule)*/
#define MAX_LIGNE 80
int main(void)
{
FILE * fw, * fr ;
/* Pour stocker les noms des fichier à lire et à écrire */
char nom_r[MAX_LIGNE+1],nom_w[MAX_LIGNE+1];
char c ;   /* Pour stocker un octet lu */
long int nb_octets ; /* Nombre d'octets lus*/
fprintf(stdout,"Entrer le nom du fichier a recopier : ");
fscanf(stdin,"%s",nom_r);  /* Lecture du nom de fichier */
/* Certains systèmes (MS-DOS par exemple) font un distinction entre des
fichiers textes et des fichiers binaires. Dans ce cas il vaut mieux ouvrir
le fichier en mode binaire "rb" pour pouvoir copier n'importe quel type de
fichier   */
fr = fopen(nom_r,"rb");
if ( fr == NULL ) /* Cas où il est impossible de lire le fichier source */
{
   fprintf(stderr,"\nImpossible de lire %s\n",nom_r);
   exit(EXIT_FAILURE);
}

fprintf(stdout,"Entrer le nom du fichier destination: ");
fscanf(stdin,"%s",nom_w);  /* Lecture du nom de fichier */
```

```
/* On commence par ouvrir le fichier destination en lecture */
fw = fopen(nom_w,"rb");
/*et on doit obtenir la valeur NULL car le fichier ne devrait pas exister*/
if ( fw != NULL )         /* Cas où le fichier destination existe */
   {
      char reponse ;
      fprintf(stdout,"Le fichier %s existe, si vous choisissez de continuer\nson contenu sera perdu.\nVoulez vous continuer (o/n) ? ",nom_w);
      /* Lecture de la réponse en éliminant le saut de ligne précédent */
      fscanf(stdin,"\n%c", &reponse);
      reponse = tolower(reponse);        /* Forcée en minuscule */
      fclose(fw);         /* On ferme le fichier destination car :*/
                          /* Soit on quitte le programme */
                          /* Soit on le réouvrira en écriture */
      if (reponse != 'o')
            {
               fprintf(stdout,"\nAucun fichier copie.\n");
               fclose(fr);
               exit(EXIT_SUCCESS);
             }
      }

      /* A ce niveau : soit le fichier destination n'existe pas */
      /*               soit l'utilisateur veut bien perdre son contenu */
      /* On peut donc l'ouvrir en écriture sans risque */
   fw = fopen(nom_w,"wb");
   if ( fw == NULL )
     {  /* Cas où il est impossible de créer le fichier destination */
        fprintf(stderr,"\nImpossible de creer %s\n",nom_w);
        fclose(fr);
        exit(EXIT_FAILURE);
     }
```

```
 /*Le fichier de destination est bien ouvert en écriture */
 /* La copie peut alors se faire */
nb_octets = 0 ;
while ( c=fgetc(fr) , c!=EOF )
   {
      fputc(c,fw);
      nb_octets++;
   }
fclose(fr);
fclose(fw);

fprintf(stdout,"\n %ld octets de %s copie dans %s\n"
,nb_octets,nom_r,nom_w);
return (EXIT_SUCCESS);
         }
```

10.9 Ouvrir un fichier dans un autre répertoire

Il est possible d'ouvrir en lecture ou en écriture, un fichier situé n'importe où dans l'arborescence de la machine (autre repétoire, disquettes, CD-ROM, lecteur réseau, etc..) en précisant son nom absolu lors de l'ouverture dans la fonction `fopen()` :

```
FILE * f = fopen("/home/phil/mon_fichier","w");
```

```
FILE * f = fopen("a:/mon_fichier","r");
```

```
FILE * f = fopen("d:/mon_fichier","r");
```

La donnée du nom se fait de préférence en séparant les répertoires par une barre de fraction / (slash) comme sous UNIX plutôt que qu'une barre inversée \ (back-slash) comme sous MS-DOS.

On peut également utiliser le nom relatif. Le répertoire de référence est alors le répertoire courant, c'est à dire le répertoire contenant le fichier exécutable .

Ainsi :

```
FILE * f = fopen("../mon_fichier","r");
```

ouvre `mon_fichier` dans le répertoire parent du répertoire courant et

```
FILE * f = fopen("mon_fichier","r");
```

ouvre mon_fichier dans le répertoire courant.

11. La librairie standard

La bibliothèque standard est un ensemble de fonctions dont certaines ont déjà été décrites. La place manque ici pour les décrire toutes et l'étudiant est invité à consulter un ouvrage de référence sur le C ou à consulter le site `arlotto.univ-tln.fr` pour une liste de sites où on peut trouver une description exhaustive de la librairie. La documentation fournie avec les compilateurs commerciaux décrit en général les fonctions de la librairie standard. Attention au fait que les compilateurs commerciaux proposent également une bibliothèque plus étendue que la libraire standard. Si ces fonctions supplémentaires peuvent s'avérer utiles, il faut être conscient du fait que leur emploi rend le programme non portable. C'est-à-dire qu'il ne pourra pas être compilé avec un autre compilateur sans modifications.

Pour utiliser une fonction de la librairie standard, il faut :

- consulter la documentation pour savoir ce que fait la fonction.
- connaître le nom du fichier en-tête contenant son prototype pour l'inclure dans le fichier source par une directive `#include <xxxx.h>` .
- connaître le prototype pour identifier l'ordre et le type des paramètres lors de l'appel de la fonction

Exemple :

Documentation :

Fonction mathématique en-tête `math.h` :

La fonction `pow` renvoie le valeur de x^y avec `x` et `y` de type double.

Prototype : `double pow(double x, double y);`

Programme utilisant `pow` :

```
 #include <math.h> /* pour le prototype de pow */
#include <stdio.h> /* pour le prototype de printf */
int main(void)
   {
      double a=16, b=3, r;
      r=pow(a,b) ; /* on calcule 16 puissance 3 */
      printf("16^3=%f",r);
   }
```

12. Quelques erreurs classiques

Ce chapitre traite quelques erreurs de programmation et non d'erreurs syntaxique. Les erreurs de syntaxe se repèrent facilement car elles génèrent un message du compilateur et bloquent le processus de génération du programme exécutable. L'analyse minutieuse du ou des messages du compilateur permet de corriger rapidement l'erreur dans la plupart des cas. Certaines erreurs syntaxique comme l'emploi d'un mot clé pour un nom de variable, l'oubli d'une fermeture de bloc `}`, etc…., déroutent l'algorithme d'analyse du compilateur et produisent des messages d'erreurs qui semblent incohérents. Un examen minutieux du fichier source et quelques tentatives de compilation en mettant en commentaire les parties suspectes du programme permettent la correction de ces erreurs.

Les erreurs de programmation, dites erreurs sémantiques, sont beaucoup plus gênantes car elles ne bloquent pas la génération d'un exécutable mais le programme n'a alors pas le comportement attendu. Il faut alors utiliser un *debugger* pour analyser le déroulement du programme en suivant la valeur de certaines variables, en plaçant des points d'arrêts à des endroits judicieusement choisis, et en déroulant le programme instruction par instruction , i.e. en *pas à pas*.

Il est impossible de dresser la liste des erreurs de programmation possibles, chaque promotion me permettant d'en découvrir de nouvelles toujours plus inattendues. Néanmoins, certaines erreurs sont tellement fréquentes que l'étudiant est invité à en prendre connaissance pour pouvoir réagir efficacement quand il les commettra.

12.1 Confusion entre l'affectation = et la comparaison ==

Le programmeur a écrit :
```
if (a=b)
   {
      printf("egalite entre a et b");
   }
```
alors qu'il pensait
```
if (a==b)
   {
      printf("egalite entre a et b");
   }
```

L'impression est réalisée dans tous les cas ou `b` est non nul quelque soit la valeur de `a`. En effet, dans le premier cas on a une affectation de la valeur `b` à la variable `a` et comme toute valeur non nulle est vraie, l'impression se produit si le résultat est non nul.

De même, si le programmeur écrit `a==b;` au lieu de `a=b;` , la valeur de `a` ne change pas. En fait, il y a simplement comparaison de `a` à `b` et pas d'utilisation du résultat de cette comparaison. Certains compilateurs produisent un *avertissement* sur ce type d'instruction.

12.2 Erreurs avec l'instruction `if`

Le programmeur a écrit
```
if (a > b) ;
   {
      a=7;
   }
```
au lieu de
```
if (a > b)
   {
      a=7;
   }
```

il obtient l'affectation de `a` à 7 dans tous les cas car le point virgule est une instruction nulle et c'est elle qui est exécutée si `a>b` . L'instruction `a=7;` est exécutée dans tous les cas.

Si le programmeur écrit

```
if (a>b)
     if (x>y)
         printf("ordre a,b et x,y");
else
       printf("ordre b,a");
```

il obtiendra un programme équivalent à

```
if (a>b)
   {  if (x>y)
         printf("ordre a,b et x,y");
      else
         printf("ordre b,a");
   }
```

Un `else` se rapporte toujours au dernier `if` rencontré si aucun `else` ne lui a pas déjà été attribué.

Il faut donc se méfier de la présentation et se rappeler qu'elle ne joue aucun rôle en langage C.

L'emploi systématique des ouvertures et fermetures de bloc permet d'éviter ces erreurs.

12.3 Erreur avec les commentaires

L'oubli d'une séquence de fermeture de commentaire `*/` entraîne la mise en commentaire de toutes les lignes du programme jusqu'à la prochaine séquence de fermeture de commentaire.

L'oubli de `*/` après dans le premier commentaire fait que le programme ci-dessous affiche seulement `AE`.

```
#include <stdio.h>

int main(void)

   {

      printf("A");  /* affiche A

      printf("B");

      printf("C");

      prinft("D");  /* affiche D */

      printf("E");

     }
```

12.4 Erreur avec `#define`

Voir le paragraphe 1.8.

12.5 Une variable locale masque une variable globale

Lorsque le même nom est déclaré pour plusieurs variables, il n'y pas d'erreur si ces variables sont dans des blocs différents ou imbriqués ou bien s'il s'agit d'une variable globale et de une ou plusieurs variables locales. Par contre lorsqu'on fait référence à une variable dans un bloc, on fait toujours référence à celle dont la déclaration est la plus "proche": celle déclarée dans le bloc courant, ou celle déclarée dans le bloc qui contient le bloc courant, et en dernier lieu à la variable globale.

Par exemple, le programme ci-dessus affiche `42464` :

```
#include <stdio.h>

#include <stdlib.h>

void fonction(int);

int i=6;

int main(void)

   {

      int i=4;

      printf("%d",i);

      if (i>3)

      {

         int i = 2 ;

         printf("%d",i);

      }

      fonction(i);

      printf("%d",i);

      return EXIT_SUCCESS ;

   }

   void fonction(int p)

   {

      printf("%d%d",p,i);

   }
```