IUT "A" Paul SABATIER

Dpt. Informatique

Bases du langage C

D.DOURS

R.FACCA

P.MAGNAUD

Table des matières

TABLE DES MATIÈRES.....I

INTRODUCTION.....1

CHAPITRE 1: LE LANGAGE DE BASE.....2

1. Éléments de base.....2
1.1. Alphabet.....2
1.2. Items syntaxiques.....2
1.2.1. Littéraux.....2
1.2.2. Mots réservés.....3
1.2.3. Identificateurs.....3
1.2.4. Opérateurs.....4
1.2.5. Séparateurs.....4
1.3. Expressions.....4
1.3.1. Expressions arithmétiques.....4
1.3.2. Expressions logiques.....4
1.3.3. Expressions de relation.....5
1.3.4. Expressions booléennes.....5
1.3.5. Priorité des opérateurs.....5

2. Déclarations de variables.....7
2.1. Variables simples.....7
2.1.1. Types prédéfinis.....7
2.1.2. Initialisation de variables simples.....8
2.2. Variables indicées (tableaux).....8
2.2.1. Types prédéfinis.....8
2.2.2. Initialisations des tableaux.....9
2.3. Définition de types.....10
2.4. Attributs de représentation.....11
2.5. Déclaration de constantes.....12

3. Instructions.....13

3.1. Instructions simples....13
3.1.1. Affectation....13
3.1.2. Instructions d'E/S....14
3.1.3. Traitement des caractères....16
3.2. Instructions de contrôle....18
3.2.1. Sélection....18
3.2.2. Choix multiple....19
3.2.3. Itérations....20
3.3. Instructions composées....21
3.3.1. Bloc d'instructions....21
3.3.2. Composition de structures de contrôle....21

4. Développement d'un programme....22
4.1. Programme source....22
4.1.1. Format du programme....22
4.1.2. Structure du programme....22
4.1.3. Directives de compilation....23
4.2. Programme objet....24
4.2.1. Compilation et édition de liens....24
4.2.2. Outils d'aide à la mise au point....25
4.2.3. Exécution d'un programme....25

Introduction

Le langage **C** est né aux laboratoires BELL, filiale de AT&T, aux USA, dans les années 70, des travaux de Brian KERNIGHAN, Denis RITCHIE et Ken THOMPSON, concepteurs du système d'exploitation UNIX.

Conçu à l'origine pour l'écriture des commandes du système UNIX, et de la majeure partie du noyau de ce dernier, il est rapidement devenu le langage privilégié de développement des applications sous UNIX.

En effet, le langage C est à la fois un langage de haut niveau qui intègre tous les concepts de la programmation structurée et un quasi langage d'assemblage qui permet de manipuler les données au niveau de leur représentation interne.

Le langage C est étroitement lié au système UNIX par le biais des appels système qui mettent à disposition du programmeur toutes les primitives du système (gestion des fichiers, des E/S, de la mémoire, des processus).

Le regain d'intérêt que connait depuis quelques années ce langage est une conséquence directe de l'essor du système UNIX.

Ce document n'est pas un manuel de référence du langage C. Il présente les éléments fondamentaux du langage et une façon de les mettre en œuvre pour élaborer des programmes modulaires, structurés, lisibles, faciles à mettre au point et à maintenir.

Chapitre 1: LE LANGAGE DE BASE

1. ELÉMENTS DE BASE

1.1. Alphabet

Il est basé sur l'ensemble des caractères codés en ASCII, il comprend :

- les lettres majuscules et minuscules
- les chiffres décimaux
- les caractères conventionnels

  + - * / < > = % ( ) [ ] { }

  | \ ! ? " ' . ; , : ^ # & _ ~

- les séparateurs

  espace <HT> <VT> <LF>

1.2. Items syntaxiques

1.2.1. Littéraux

- Entiers relatifs décimaux
- Entiers naturels octaux (nombres préfixés par 0). Ex : 033
- Entiers naturels hexadécimaux (nombres préfixés par 0x ou 0X). Ex : 0xFA05
- Réels décimaux représentés en virgule fixe ou en virgule flottante.
  Ex : +3.14, -6.03E+23
- Caractères ASCII (délimités par des apostrophes). Ex : 'A', '5'

  Les caractères, notamment ceux qui ne sont pas visualisables, peuvent aussi être définis par la valeur octale ou hexadécimale du code ASCII précédée d'un "backslash".

  Ex : le caractère <ESC> s'écrit '\0x1b' ou '\033'

Les caractères non visualisables courants ont une notation symbolique :

'\n' saut de ligne
'\b' espacement arrière
'\r' retour chariot
'\f' saut de page
'\t' tabulation horizontale
'\v' tabulation verticale
'\a' bip

ainsi que les caractères suivants :

'\\' "backslash"
'\'' apostrophe
'\"' guillemets

- Chaines de caractères ASCII (délimitées par des guillemets). Ex : "Bonjour!"

  Il est possible de représenter dans une chaine certains caractères dans leur représentation octale ou symbolique.

  Ex : "\033[2Jcoucou!\n"

  Remarque :

  Dans la représentation interne, les chaines de caractères sont terminées par le caractère <NUL> ('\0') qui sert de marqueur de fin de chaine.

1.2.2. Mots réservés

```
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
```

1.2.3. Identificateurs

Un identificateur est formé d'une suite de caratères (lettre majuscule ou minuscule, chiffre, souligné) commençant par une lettre.

Bien que la longueur ne soit pas limitée, seuls les premiers caractères sont significatifs (voir la documentation).

Un identificateur doit être différent d'un mot réservé.

1.2.4. Opérateurs

Les opérateurs appliqués à des opérandes forment des expressions. Les différents types d'opérateurs et d'expressions seront définis au §1.3.

1.2.5. Séparateurs

Ce sont les caractères : espace <HT> <VT> <LF>.

Un commentaire, (suite de caractères commençant par /* et finissant par */) est considéré comme un séparateur et est ignoré par le compilateur.

1.3. Expressions

Une expression est une composition d'opérateurs appliqués sur des opérandes, un opérande étant une constante, une variable ou une expression.

1.3.1. Expressions arithmétiques

Les opérateurs arithmétiques applicables à des opérandes de type entier ou réel sont :

+	addition
-	soustraction
*	multiplication
/	division
%	reste de la division entière
-	opposé d'un nombre

1.3.2. Expressions logiques

Les opérateurs logiques permettant de manipuler des valeurs binaires dans des variables ou des constantes de type entier sont :

&	et
|	ou
^	ou exclusif
~	non
<<	décalage à gauche
>>	décalage à droite

Ex : E1 >> E2

E1 est décalé à droite de E2 positions si $E2 \geq 0$. Le décalage est logique si E1 est un entier non signé (cf ch 1 §2.4) et arithmétique sinon.

1.3.3. Expressions de relation

Les opérateurs relationnels permettent de comparer deux expressions arithmétiques mais sont aussi utilisables pour les pointeurs (cf ch 2 §3.3.3). Ces opérateurs sont les suivants :

==	égale (ne pas confondre avec =)
!=	différent
>	supérieur
>=	supérieur ou égal
<	inférieur
<=	inférieur ou égal

Remarque : le type booléen (vrai, faux) n'existe pas en C. Les opérateurs de relation donnent le résultat de type entier suivant :

1 si la relation est vraie
0 si la relation est fausse

1.3.4. Expressions booléennes

Comme le type booléen n'existe pas en C, les opérateurs booléens sont applicables à des expressions de relation, mais aussi à toute expression arithmétique de type entier (la valeur 0 étant considérée comme la valeur faux et toute autre valeur comme la valeur vrai). Les opérateurs sont :

!	non
&&	et
||	ou

Ces opérateurs donnent le résultat de type entier suivant :

1 si la relation est vraie
0 si la relation est fausse

Remarque : Les opérateurs && et || ne sont pas commutatifs. L'expression gauche est évaluée en premier, l'expression droite peut ne pas être évaluée.

1.3.5. Priorité des opérateurs

Lors de l'évaluation d'une expression, les règles d'évaluation suivantes sont appliquées :

- les sous-expressions parenthésées sont évaluées en premier.
- Les opérateurs les plus prioritaires sont appliqués en premier.
- Les opérateurs de même priorité sont appliqués de la gauche vers la droite sauf les opérateurs unaires (!, ~ et -) qui sont appliqués dans l'ordre inverse.

L'ordre décroissant de priorité des opérateurs est le suivant :

```
! ~ -
* / %
+ -
<< >>
< <= >= >
== !=
&
^
|
&&
||
```

Il est conseillé de parenthéser correctement les expressions afin d'éviter des erreurs très difficiles à détecter.

2. DÉCLARATIONS DE VARIABLES

Toutes les variables d'un programme doivent être déclarées. Cette déclaration permet d'associer chaque variable à un type. La syntaxe des **déclarations de variables** est :

<*type*> <*ident*> {, <*ident*>}*;

2.1. Variables simples

2.1.1. Types prédéfinis

a) Type entier : **int**

Les variables de type int permettent de manipuler des entiers relatifs codés en Complément Vrai sur **n** bits (16 ou 32 suivant les machines). Les valeurs appartiennent à l'intervalle $-2^{n-1}..+2^{n-1}-1$.

Exemple :

```
int i,nb_elem,cpt;
```

b) Type caractère : **char**

Les variables de type char permettent de manipuler des caractères codés en ASCII sur un octet. Selon les machines le caractère est codé :

- comme un relatif en CV; les valeurs appartiennent alors à l'intervalle **-128..+127**.
- comme un naturel; les valeurs appartiennent alors à l'intervalle **0..255**.

Dans tous les cas, ces variables permettent de coder les 128 caractères ASCII de base (codes de 0 à 127).

Exemple :

```
char caract,code;
```

c) Type réel : **float**

Les variables de type float permettent de manipuler des réels codés en représentation **flottante normalisée** (en général sur 32 bits : 24 bits de mantisse et 8 bits d'exposant). Ceci donne en équivalent décimal 7 chiffres significatifs, le plus petit nombre positif étant environ $0,29.10^{-38}$ et le plus grand environ $0,17.10^{+39}$.

Exemple :

```
float somme,coeff;
```

d) Type réel : **double**

Les variables de type double permettent de manipuler des réels avec la même représentation que le type float mais codés sur un nombre de bits plus important (64 en général).

Exemple :

```
double eps,X;
```

2.1.2. Initialisation de variables simples

Une variable simple peut recevoir une valeur initiale lors de sa déclaration. Cette valeur est spécifiée par une expression affectée à la variable. La syntaxe correspondante est :

<*type*> <*ident*> [= <*expression*>]
{ , <*ident*> [= <*expression*>]}*;

Exemples :

```
int X = 0, HEURE = 60;
int JOUR = HEURE*24;
char LETTRE = 'A';
float ALFA = 0.5;
```

2.2. Variables indicées (tableaux)

2.2.1. Types prédéfinis

a) Tableaux à une dimension

Un tableau est composé de **N** éléments de même type. Chaque élément est repéré par une valeur entière appelée **indice**. L'indice du premier élément est **égal à 0**. Les indices varient donc de **0 à N-1**. Lors de la déclaration, la dimension du tableau est définie par une expression constante entre crochets. Le type des éléments est un des types prédéfinis déjà vus pour les variables simples .

Exemples :

```
int tab[10];
float TEMP[7];
char nom[20+1]; /* réserver un octet de plus pour le caractère fin de chaine '\0' */
```


b) Tableaux à plusieurs dimensions

Un tableau à plusieurs dimensions est défini comme un tableau dont les éléments sont eux mêmes des tableaux. Lors de la déclaration, on doit préciser les dimensions de chaque tableau.

Exemple :

```
int TAB2[4][3];
```

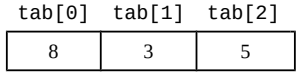
Dans ce cas, la structure du tableau est la suivante :

2.2.2. Initialisations des tableaux

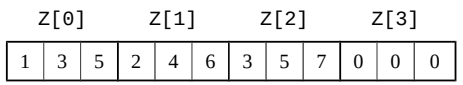
L'initialisation d'un tableau se fait en affectant des ensembles de valeurs aux différentes composantes. Le dernier indice varie en premier, puis l'avant dernier, etc...Les composantes non affectées sont initialisées à 0.

Exemples :

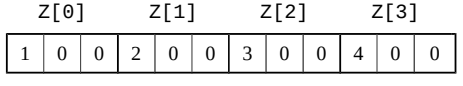
```
int tab[3] = {8,3,5};
```




```
int Z[4][3] = { {1,3,5},{2,4,6},{3,5,7} };
```




```
int Z[4][3] = { {1},{2},{3},{4} };
```




```
char mesg2[] = "****Erreur\n";
```

0	1	2	3	4	5	6	7	8	9	10	11
*	*	*	*	E	r	r	e	u	r	\n	\0

Remarque : si la taille des composantes n'est pas spécifiée le compilateur la déduit de la liste des valeurs d'initialisation.

```
float coeff[] = { 1.5, 0.58, 12., -78, +2.5 }; /* réserve et initialise un tableau de 5 éléments */
```

2.3. Définition de types

L'opérateur **typedef** permet d'associer un nom symbolique à un type existant ou construit. L'utilisation de constructeur est la suivante :

typedef <*spec_type*> <*type*>;

<*spec_type*> peut être un type déjà défini ou une définition de type
<*type*> est le nom symbolique associé à <*spec_type*>

L'utilisation de cet opérateur permet d'améliorer la lisibilité du programme.

Exemples :

```
typedef char t_ch20[20+1]; /* définition d'un type chaine de 20 caractères */
t_ch20 NOM, PRENOM;    /* déclaration de 2 variables chaine de 20 caractères */
typedef t_ch20 t_tab_ch[100];    /* définition d'un type tableau de chaines */
t_tab_ch TAB_NOM;          /* déclaration d'une variabl tableau de chaines */
```

Nous verrons des applications de ce type de déclaration dans le chapitre 2 du cours :

- pour la déclaration des paramètres formels de type tableau dans les fonctions,
- pour la déclaration de variables structurées.

2.4. Attributs de représentation

Des attributs supplémentaires permettent de préciser la taille de la représentation interne des variables de type int. Ces attributs, qui préfixent la déclaration de type int, sont :

long qui précise un entier long

short qui précise un entier court

La taille exacte dépend de la machine cible (consulter le manuel de référence du Langage C spécifique à la machine). La seule certitude est que :

taille(short int) ≤ taille(int) ≤ taille(long int)

L'attribut **unsigned** permet de spécifier un entier naturel (entier non signé)

Si **n** est la taille de la représentation, les valeurs appartiennent à l'intervalle **0..2^n-1**.

Remarques :

- Le mot clé **int** peut être supprimé dans les déclarations d'entiers **unsigned**, **short** ou **long**.
- L'attribut **unsigned** peut aussi préfixer une déclaration de type char, auquel cas les valeurs des caractéres codés en ASCII appartiennent à l'intervalle **0..255**.

Exemples :

```
long int VAL;
short cpt;
unsigned char CARACT;
ushort ind; /* contraction de unsigned short int */
```

L'opérateur **sizeof**, appliqué à une variable ou a un type, retourne un entier égal à la taille en octets de la réprésentation interne de cette variable ou de ce type.

Exemples :

```
sizeof( float )
sizeof( VAL )
```

2.5. Déclaration de constantes

Une déclaration de variable initialisée et préfixée par l'attribut **const** indique que le contenu de cette variable ne pourra plus être modifiée pendant l'exécution du programme.

Exemple :

```
const float coef = 3.5;
```

Toute affectation ultérieure de cette variable génèrera une erreur de compilation. Toutefois , lors de l'exécution, un accés indirect à la variable par le biais d'un pointeur peut altérer son contenu sans provoquer d'erreur!

3. INSTRUCTIONS

3.1. Instructions simples

3.1.1. Affectation

<*variable*> = <*expression*> ;

Exemple :

```
x = 12;
tab[i][j] = coeff*21.5409;
eol = '\n';
```

L'expression est évaluée et fournit une valeur qui est affectée à la variable. Le type de la valeur et de la variable doivent être compatibles. Des conversions de types peuvent intervenir implicitement dans l'évaluation de l'expression et dans l'affectation. Le programmeur peut aussi demander explicitement des conversions dans l'évaluation de l'expression.

a) Conversions implicites

La plupart des opérateurs imposent un type donné pour leurs opérandes ce qui définit aussi le type du résultat. Si le programmeur utilise des opérandes d'un type compatible mais différent du type attendu, des conversions de valeurs seront effectuées vers le type attendu. Ces conversions implicites peuvent avoir lieu :

- dans l'évaluation de l'expression
- dans l'affectation du résultat à la variable

Pour plus de précisions se reporter au manuel de référence du compilateur.

b) Conversions explicites

Le programmeur peut explicitement modifier le type du résultat d'une expression en un type différent mais compatible, de la façon suivante:

(<*type*>) <*expression*>

Exemples :

```
int X=5, Y=2, Q;
float RES;
...
Q = X/Y;/* Q prend la valeur 2, quotient de la division entière */
RES = X/Y;      /* RES prend la valeur 2.0, quotient de la division entière converti en réel */
RES=((float)X)/Y;  /* RES prend la valeur 2.5, résultat de la division réelle */
```

Remarque :

Dans la syntaxe du langage C, le symbole = de l'affectation est considéré comme un opérateur qui attribue à l'opérande gauche <*variable*> la valeur de l'opérande droit <*expression*> et qui retourne cette valeur. L'affectation est donc considérée comme une expression et peut donc apparaître partout où une expression est attendue.

Cette possibilité, abondamment utilisée par les "professionnels du C", est à éviter absolument car elle induit des erreurs (effets de bords) très difficiles à détecter, et complique la maintenance du programme.

Exemple :

```
D = (C = A+D) + C;  /* ??? */
```

- L'opérateur d'affectation peut être combiné avec d'autres opérateurs pour abréger l'écriture de certaines formes d'affectations.

Exemple :

```
x += y;   /* équivalent à x = x+y */
i++;      /* équivalent à i = i+1 */
j--;      /* équivalent à j = j-1 */
```

3.1.2. Instructions d'E/S

Il n'y a pas d'instructions générales d'E/S comme par exemple dans les langages COBOL ou FORTRAN. Les opérations d'E/S sur des fichiers sont réalisées par des fonctions spécifiques qui seront décrites en détail dans la 3ème partie du cours (Bibliothèque standard C). Nous présenterons pour l'instant une forme simplifiée des fonctions d'E/S formatées sur les fichiers standards (écran et clavier du Terminal). Pour utiliser ces fonctions il faut inclure dans le programme le fichier de déclarations **stdio.h** en utilisant la directive :

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


a) Sorties formatées sur l'écran

La fonction **printf** permet d'afficher sur l'écran du Terminal les données passées en paramètre. La forme générale de l'appel de cette fonction est la suivante :

printf(<*format*>[, <*expression*> {,<*expression*>}*]);

- <*expression*> définit la valeur qui sera affichée,

- <*format*> est une constante chaine de caractères contenant :

  - des spécificateurs de format associés à chaque expression passée en paramètre,

  - du texte à afficher.

Les principaux spécificateurs de format sont :

%d pour éditer des entiers
%f pour éditer des réels en virgule fixe
%e pour éditer des réels en notation exponentielle
%s pour éditer des chaines de caractères

Exemples :

```
int valeur;
char nom[20+1];
...
printf("Résultat= %d\n",valeur);
printf("\nNOM: %s\n",nom);
printf("Entrer une valeur?:");
```

b) Entrées formatées au clavier

La fonction **scanf** permet de saisir au clavier du Terminal des données et de les affecter aux variables passées en paramètre. La forme générale de l'appel de cette fonction est la suivante :

scanf(<*format*>, <*variable*> {,<*variable*>}*);

- <*variable*> est un nom de variable à laquelle sera affectée la donnée saisie. Si c'est une variable simple le nom doit être préfixé par le symbole & (une explication sera donnée dans le chapitre 2 du cours au paragraphe relatif au passage des paramètres dans les fonctions).

- <*format*> est une constante chaine de caractères contenant des spécificateurs de format associés à chacune des variables passées en paramètre.

Les principaux spécificateurs de format sont :

%d pour saisir des entiers
%f pour saisir des réels
%s pour saisir des chaines de caractères

Exemples :

```
int ind;
float tab[30];
char libelle[40+1];
...
scanf("%d%f", &ind, &tab[ind]);
scanf("%s", libelle);
```

Remarques :

- Plusieurs données peuvent être saisies à la suite en les séparant par des espaces.

- La saisie doit être validée par la touche RETURN ou ENTER.

- Dans le cas de la saisie d'une chaine de caractères, les caractéres sont rangés dans le tableau à partir de l'élément d'indice 0. La saisie de la chaine s'arrête dès qu'un espace ou une fin de ligne (touche RETURN) est rencontré. Le caractère '\0' (fin de chaine) est rajouté dans le tableau après le dernier caractère de la chaine.

Attention il n'y a pas de contrôle de débordement !!!

3.1.3. Traitement des caractères

On a vu au §2.1.1 que le type char est en fait un type entier. Donc les opérateurs arithmétiques, logiques et de relation sont applicables à des opérandes de type char.

Exemple :

```
( rep == 'N' )  /* relation d'égalité */
```

Il existe des fonctions spécifiques de traitement des caractères et des chaines de caractères qui seront décrites en détail dans le chapitre 3 du cours (Bibliothèque standard C). Nous présenterons pour l'instant quelques fonctions de base de traitement des chaines de caractères. Pour utiliser ces fonctions, il faut inclure dans le programme le fichier de déclarations string.h en utilisant la directive :

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


a) Concaténation de chaines

`strcat``(chaine1,chaine2)`

Cette fonction concatène `chaine2` à `chaine1`

b) Copie de chaine

`strcpy``(chaine1,chaine2)`

Cette fonction recopie `chaine2` dans `chaine1`

c) Comparaison de chaines

`strcmp``(chaine1,chaine2)`

Cette fonction compare `chaine1` avec `chaine2`. La comparaison est basée sur la relation d'ordre liée au codage des caractéres en ASCII. La fonction retourne une valeur entière qui définit le résultat de la comparaison de la façon suivante :

0	si `chaine1 = chaine2`
>0	si `chaine1 > chaine2`
<0	si `chaine1 < chaine2`

Remarques :

Les paramètres de ces fonctions doivent être des tableaux de type **char** ou des constantes chaines de caractères, excepté dans **strcpy** et **strcat** où `chaine1` est obligatoirement une variable tableau.

Exemples :

```
char jour[8+1], TAMPON[256];
...
strcpy(jour, "DIMANCHE");
strcpy(TAMPON, "");      /* chaine vide */
if (strcmp(jour,"LUNDI") != 0)  /* relation d'inégalité */
```

3.2. Instructions de contrôle

3.2.1. Sélection

a) Sélection simple

if (<*expression*>)
  <*instruction*>;

Si <*expression*> est VRAI <*instruction*> est exécutée, dans le cas contraire <*instruction*> n'est pas exécutée. L'exécution continue ensuite en séquence.

Exemple :

```
if (indic == 0)
   cpt=20;
```

Attention, ne pas confondre = = et = : `if (indic = 0)`
`cpt=20;`

Cette construction est syntaxiquement correcte puisqu'une affectation est considérée comme une expression (cf §3.1.1). La variable indic sera mise à 0, et comme 0 est interprété comme FAUX l'instruction cpt=20; ne sera jamais exécutée !!!

b) Alternative

if (<*expression*>)
  <*instruction1*>;
else
  <*instruction2*>;

Si <*expression*> est VRAI <*instruction1*> sera exécutée, sinon <*instruction2*> sera exécutée. Dans les deux cas l'exécution continue en séquence.

Exemple :

```
if (VAL >= 0.0)
   ABS_VAL = -VAL;
else
   ABS_VAL = VAL;
```


3.2.2. Choix multiple

switch (<*expression*>)
{
 case <*valeur1*>:
  [<*instruction1*>;
   break;]
 case <*valeur2*>:
  [<*instruction2*>;
   break;]
 ...
 case <*valeurN*>:
  [<*instructionN*>;
   break;]
 [**default**:
   <*instructionX*>]
}

Si <*expression*> prend une des valeurs énumérée dans la liste des choix, l'instruction correspondante est exécutée, dans le cas contraire c'est <*instructionX*> qui est exécutée (si la clause default est présente). Dans tous les cas l'exécution continue en séquence à l'instruction qui suit la structure switch.

Remarques :

- <*expression*> doit être une expression entière.
- <*valeur1*> … <*valeurN*> doivent être des constantes entières toutes différentes.
- la clause **default** si elle apparait doit être la dernière de la liste
- l'instruction **break** qui suit l'instruction associée à un choix peut être supprimée, dans ce cas l' exécution du programme continue sur le choix suivant

Exemples :

```
switch (code_sexe)
{
   case 1:  SEXE='M'; break;
   case 2:  SEXE='F'; break;
   default:  SEXE='X';
}
```

```
switch (reponse)
{
   case 'o':
   case 'O':
   case 'y':
   case 'Y':  OK=1; break;
   case 'n':
   case 'N':  OK=0; break;
   default:  OK=-1;
}
```

3.2.3. Itérations

a) Tant que

`while` (<*expression*>)
  <*instruction*>;

Cette structure de contrôle permet de répéter l'exécution de <*instruction*> tant que <*expression*> est VRAI. L'évaluation de <*expression*> est faite avant d'exécuter <*instruction*>. L'itération se termine dès que <*expression*> est FAUX.

b) Faire … tant que

`do`
  <*instruction*>
`while` (<*expression*>);

Cette structure de contrôle permet de répéter l'exécution de <*instruction*> tant que <*expression*> est VRAI. A la différence du while l'évaluation de <*expression*> est faite aprés l'exécution de <*instruction*>. L'itération se termine dès que <*expression*> est FAUX.

c) Pour

`for` (<*inst_I*> ; <*expression*> ; <*inst_F*>)
  <*instruction*>;

<*inst_I*> est exécutée initialement puis <*expression*> est évaluée. Si <*expression*> est VRAI <*instruction*> est exécutée puis <*inst_F*>. <*expression*> est évaluée à nouveau, etc… L'itération se termine dès que <*expression*> est FAUX. Il s'agit d'une forme simple de l'instruction for, il existe des formes plus complexes qu'il est déconseillé d'utiliser.

3.3. Instructions composées

3.3.1. Bloc d'instructions

Un bloc est composé d'une séquence d'instructions englobées dans une paire d'accolades. La syntaxe correspondante est :

{ { <*instruction*>; }* }

3.3.2. Composition de structures de contrôle

Un bloc est une instruction composée qui est syntaxiquement équivalente à une instruction simple. Un bloc peut donc être utilisé partout où une instruction simple est utilisée.

Il en est de même pour les instructions de contrôle. Ces notions permettent de composer entre elles toutes les structures de contrôle (imbrication).

Exemple :

```
int i=0,fini=0, tab[30],val;
while (! fini)
{
    printf("Entrer une valeur ou 999 pour arreter: ");
    scanf("%d",&val);
    if (val != 999)
    {
        tab[i]=val;
        i++;
    }
    else
        fini=1;
 }
```

4. DÉVELOPPEMENT D'UN PROGRAMME

4.1. Programme source

4.1.1. Format du programme

Un programme source en langage C se présente comme un texte structuré en lignes. Chaque ligne est composée d'une suite de caractères ASCII appartenant à l'alphabet défini au §1.1 et est terminée par un caractère fin de ligne (<LF> sous UNIX).

La mise en page des diverses composantes du programme (instructions, déclarations, etc...) est libre. On peut donc coder plusieurs instructions sur une ligne, ou coder une instruction sur plusieurs lignes. Des espaces sont utilisés pour séparer les éléments de base du langage. Ces possibilités doivent être utilisées au maximum pour améliorer la lisibilité du programme. Les commentaires sont des suites de caractères délimitées par /* et */

Exemple :

```
/* Programme principal */
```

Le programme source sera saisi sous un éditeur de texte et archivé dans un fichier. Le compilateur impose que les noms de fichiers sources soient postfixés par le suffixe **.c**.

4.1.2. Structure du programme

De façon générale un programme source est constitué du programme principal et de sous-programmes correspondants à la décomposition fonctionnelle du traitement à réaliser.

Le programme source comporte en général un entête constitué de commentaire généraux, de directives de compilation, de définitions de types, de déclarations globales, etc…

Dans le Langage C, les sous-programmes seront traduits par des fonctions (les procédures n'existent pas). Le programme principal sera lui même traduit par une fonction appelée **main** (fonction principale). La structure générale d'un programme est donc :

```
   <déclarations globales>
{<définition de fonction>}*
   int main()
   {
     <déclarations locales>
     <instructions>
   }
```

Les fonctions et leur utilisation seront vues en détail dans le chapitre 2.

4.1.3. Directives de compilation

Ce sont des commandes pour un préprocesseur qui va mettre en forme le texte du programme source avant la compilation.

Ces directives permettent :

- l'inclusion de fichiers dans le fichier source
- la définition de symboles
- la définition de macro-opérations
- la compilation conditionnelle

Nous ne présenterons que les directives les plus courantes.

a) Inclusion de fichier

Cette directive permet d'inclure dans le programme source un texte contenu dans un autre fichier :

- inclusion de fichiers de la Bibliothèque standard C : Sous UNIX ces fichiers sont catalogués dans le répertoire **/usr/include**

```
#include <fichier>
```

Exemples :

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

- inclusion de fichiers quelconques :

```
#include "fichier"
```

Cette possibilité permet au programmeur de créer ses propres bibliothèques, de découper son programme en plusieurs fichiers, etc… La syntaxe du nom de fichier dépend du système d'exploitation de la machine.

b) Définition de symboles

```
#define <ident> <chaine>
```

Cette directive ordonne au préprocesseur de remplacer toute apparition ultérieure de <*ident*> dans le programme source par <*chaine*>. Il est fortement conseillé d'utiliser cette possibilité qui améliore la lisibilité du programme.

Attention :

Le symbole <*ident*> doit impérativement être différent de tout nom de variable ou de fonction utilisé par ailleurs dans le programme.

Exemples :

```
#define VRAI 1
#define FAUX 0
#define nbmax 300
```

4.2. Programme objet

4.2.1. Compilation et édition de liens

Sous UNIX la compilaton, l'édition de liens et la génération du code éxécutable (fichier objet) sont réalisées par un seul utilitaire cc en 4 phases principales:

- mise en forme du source par le préprocesseur
- compilation et production d'un source Assembleur
- assemblage et production d'un objet relogeable
- édition de lien et production d'un objet exécutable

L'appel de l'utilitaire **cc** est le suivant :

cc <*source*> { <*option*> }*

<*source*> est le nom du fichier source avec le suffixe .c

<*option*> est une directive donnée au compilateur. Les principales directives sont :

-o <*objet*> le fichier objet exécutable aura pour nom <*objet*>, si cette option est omise le nom par défaut est **a.out**

-c la phase d'édition de lien est supprimée, le code objet exécutable n'est pas généré

-g permet l'utilisation ultérieure d'un outil de mise au point symbolique du programme objet ("debugger").

Sous Linux l'utilitaire s'appelle **gcc**.

Exemple :

```
cc prog4.c -g -o prog4
```

Par défaut les messages d'erreurs à la compilation sont affichés sur l'écran. On peut les rediriger dans un fichier de la façon suivante:

Exemple :

```
cc prog4.c -o prog4 2>prog4.err
```

Ce fichier peut ensuite être imprimé ou affiché à l'écran avec les commandes UNIX : lp, pg, more, etc…

4.2.2. Outils d'aide à la mise au point

Sous UNIX les utilitaires de base sont:

a) L'analyseur syntaxique **lint**

Cet utilitaire effectue des contrôles syntaxiques et sémantiques plus stricts que le compilateur cc. Il signale sous forme de "warnings" les constructions ambiguës susceptibles de produire des erreurs à l'exécution.

b) Le "debugger" symbolique

Le debugger est un outil d'aide à la mise au point du programme objet qui permet:

- de tracer l'exécution du programme,
- de visualiser les variables, de les modifier,
- d'analyser l'image mémoire du programme aprés une exécution interrompue (analyse post-mortem).

Sous UNIX il n'y a pas d'utilitaire standard, et chaque constructeur fournit le sien. Par exemple **dbx** pour SunOS, **xdb** pour HP-UX, **tbx** pour Motorola System VR4. Toutefois la Free Sotware Foundation, dans le cadre de GNU, fournit le source des utilitaires **gdb** et **xgdb** qui peuvent ainsi être porté sur différentes plateformes.

4.2.3. Exécution d'un programme

Sous UNIX le progamme objet est catalogué dans le fichier **a.out** ou <*objet*> si l'option **-o** <*objet*> a été utilisée à la compilation. Pour exécuter le programme, il suffit de taper le nom du fichier lorsque l'on est sous le contrôle de l'interpréteur de commandes.