UNIVERSITE PARIS DAUPHINE

2004 - 2005

Université Paris Dauphine

IUP Génie Mathématique et Informatique

INITIATION A LA PROGRAMMATION PROCEDURALE, A L'ALGORITHMIQUE ET AUX STRUCTURES DE DONNEES PAR LE LANGAGE C

Maude Manouvrier

TABLE DES MATIERES

Remerciements *: Je tiens à remercier tout particulièrement Monsieur B. Quément, professeur à l'Université Paris IX Dauphine, pour les cours qu'il m'a transmis en IUP1, et pour son autorisation de réutiliser ses TD. Je remercie également G. Jomier, professeur à Paris-Dauphine, E. Lazard, maître de conférences à Paris-Dauphine, S. Boussetta, M. Menceur et A. Schaal pour leurs conseils avisés.*

I. INTRODUCTION

Ce document d'appui va vous permettre de mieux comprendre l'algorithmique, les structures de données et la programmation en C. **Prenez néanmoins garde : ce document n'est qu'un résumé rapide. Pour bien comprendre, vous êtes invité à consulter les ouvrages de références.**

Pour commencer, familiarisons-nous avec le vocabulaire de la programmation par quelques définitions et un petit exercice simple …

A. Définitions de base de la programmation

- **Modèles de données** : Abstractions utilisées pour décrire les problèmes. Exemples : la logique, les graphes, les listes.
- **Structures de données** : Constructions du langage de programmation utilisées pour représenter les modèles de données. Exemples : les tableaux, les listes chaînées, les pointeurs.
- **Algorithmes** : Techniques utilisées pour obtenir des solutions manipulant les modèles de données et les structures de données associées. Il s'agit du cheminement qui permet, à partir de la définition d'un problème, d'en déduire une solution sous forme de programme informatique (voir Figure I.1).

Vous pouvez retrouver ces définitions dans [AU93].

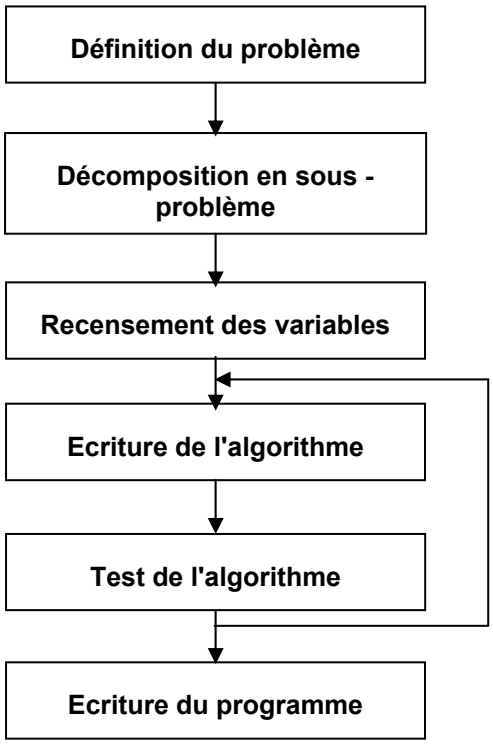

Figure I.1 - Le processus de programmation.

Un **algorithme** s'exécute de manière séquentielle. L'ordinateur exécute les instructions de la première à la dernière ligne du programme, une instruction à la fois.

Pour qu'un programme puisse s'exécuter, il faut qu'il soit correct, sinon un message d'erreur s'affiche.

B. Exemple de problème

Exercice : Créer, en suivant le processus de programmation de la Figure I.1 l'algorithme qui permette d'afficher la somme des entiers de 1 à *n*, *n* étant saisi par l'utilisateur.

La correction vous sera donnée en cours.

II. CONCEPTS GENERAUX DU LANGAGE C

Le langage C a été développé par Kernighan et Ritchie pour écrire le système Unix dans un langage portable. Il a été largement utilisé tant pour l'écriture du noyau que pour les principaux outils du système [Rif93]. Le langage est très populaire à présent, quoique ancien; il est très bien décrit dans le livre de référence de Kernighan et Ritchie [KR90].

A. Forme générale d'un programme C

Un programme source C se présente sous forme d'une **collection d'objets externes** (variables et fonctions), dont la définition est éventuellement donnée dans des fichiers séparés [Rif93].

Tout programme C est composé d'un **programme principal**, c'est-à-dire d'une fonction particulière qui porte toujours le nom *main* [Pon97].

Une **variable** est un objet manipulé par le programme, qui possède un nom et un type. Le **type** définit l'ensemble des valeurs possibles pour l'objet.

En C, tout **module** (sous-programme) porte le nom de fonction. Une **fonction** est un sous-programme. Il s'agit d'un mécanisme permettant de donner un nom à un bloc afin de pouvoir le réutiliser en différents points du programme. Une fonction permet d'enfermer certains traitements dans une "boîte noire", dont on peut ensuite se servir sans se soucier de la manière dont elle a été programmée. Toutes les fonctions, dont le programme principal, sont constituées d'un bloc. Un **bloc** est une suite d'instructions à l'intérieur d'accolades "{ }".

A.1 Exemple de programme C élémentaire

Voici un exemple de programme écrit en C. Ce programme calcule la somme des entiers de 1 à *iBorne*, la valeur de *iBorne* étant saisi par l'utilisateur. Les mots clés du langage C sont soulignés.

```
/* Inclusion des informations de la bibliothèque standard */
#include <stdio.h>

/*************************************************************/
/*                       programme principal                 */
/*            Calcul de la somme des entiers de 1 à iBorne   */
/*             iBorne étant saisi par l'utilisateur          */
/*************************************************************/

void main ()
{
/* DECLARATION DES VARIABLES     */
/* compteur de boucle            */
int iCompteur;
/* Somme initialisée à 0         */
int iSomme = 0;

/* Borne de la somme             */
int iBorne;
```

```
/* CORPS DU PROGRAMME PRINCIPAL                                    */

/* Demande de saisie de la borne à l'utilisateur                   */
printf ("Valeur de la borne de la somme :");
/* Lecture de la valeur de la borne                                */
scanf ("%d", &iBorne); /* &iBorne = adresse en mémoire*/

/* Pour tous les entiers allant de 1 à la borne                    */
for (iCompteur=1;iCompteur<=iBorne;iCompteur++)
      {
            /* Ajout de l'entier courant à la somme                */
            iSomme += iCompteur; /* ≅ iSomme=iSomme+iCompteur      */
            /* Le compteur iCompteur est incrémenté (voir corps du for)*/
      }

/*Affichage de la somme                                            */
printf("Somme = %d\n",iSomme);

} /* fin du programme principal */
```

COMMENTAIRES DU PROGRAMME :

- Un bloc commence par une accolade ouvrante **{** et se termine par une accolade fermante **}**.
- Les commentaires se mettent entre ***/**** et **/*. Les commentaires peuvent être sur plusieurs lignes
- ***void main()*** définit une fonction appelée main qui ne reçoit pas d'arguments.

  C'est le nom du **programme principal**.

  void signifie que la fonction ne retourne rien. On parle dans ce cas de **procédure**.
- En début de programme, les variables sont déclarées.

  int iCompteur; signifie que la variable ***iCompteur*** est de type entier (*integer*).
- Chaque instruction se termine par un point virgule **;**
- Il faut **initialiser[1] une variable**, sinon l'ordinateur peut lui affecter n'importe quelle valeur.

  Il n'y a pas d'initialisation par défaut.

  On peut initialiser une variable lors de sa déclaration : ***int iSomme=0*** ;
- ***printf (...)*** : le programme principal (fonction ***main***) appelle la fonction ***printf***, de la bibliothèque ***stdio.h***, pour afficher la séquence de caractères *"Valeur de la borne de la somme :".*
- ***scanf(...)*** : le programme principal (fonction ***main***) appelle la fonction ***scanf***, de la bibliothèque ***stdio.h***, qui lit la valeur tapée au clavier par l'utilisateur. Cette valeur est de type entier puisque le format de lecture est ***"%d"*** (*digit*). La valeur est mise dans la variable ***iBorne***.
- ***for (...)*** est une boucle. Le programme qui suit immédiatement le ***for*** (ici ***iSomme+=iCompteur***) va être exécuté autant de fois qu'il y a de passages dans la boucle.

[1] Initialiser une variable = action qui consiste à manipuler une variable pour la première fois, pour lui donner une valeur.

A l'intérieur des parenthèses, il y a trois parties :

1. La première partie est l'**initialisation** : *iCompteur=1*
   Elle s'effectue une seule fois, avant l'entrée dans la boucle.
2. La deuxième partie est le **test de la condition** : *iCompteur<=iBorne*
   Cette partie contrôle le déroulement de la boucle. Cette condition est évaluée :
   - → Si la condition est vraie, on exécute le corps de la boucle (*iSomme+=iCompteur*), puis on passe à la phase d'incrémentation (*iCompteur++)*.
   - → Si la condition est fausse, la boucle se termine.
3. La troisième phase est l'**incrémentation** : l'instruction *iCompteur++* est équivalent à l'instruction *iCompteur = iCompteur + 1.*

   Après cette phase, la boucle reprend en 2.

- ***printf("Somme ...")*** est l'appel à la fonction de sortie avec mise en forme.

  Son premier argument est une chaîne de caractères à afficher, dans laquelle chaque ***%*** (ici un seul) indique l'endroit où l'un des arguments suivants (le deuxième, troisième, etc.) doit se substituer, et sous quel format l'afficher.

  \n est un caractère spécial (séquence d'échappement) permettant de passer à la ligne.

Après avoir analysé cet exemple, passons aux types de base utilisés en C.

B. Types de base du C

- **Entier** : *int*

  Codé sur 2 octets en général, parfois 4, selon la plate-forme utilisée (UNIX, DOS, Windows ..). Le nombre d'octets utilisés pour coder un entier peut être un paramètre de compilation, selon la plate-forme utilisée.

  Valeurs : de -32768 à 32767 sur 2 octets et de -2147483648 à 2147483647 sur 4 octets.

  Format d'affichage : *%d*.

  Les types dérivés :

  - *unsigned int* ou *unsigned* : entier non signé. Valeur de 0 à 65535 (si sur 2 octets) - Format d'affichage : *%u*
  - *long int* ou *long* : entier sur 4 octets - Format d'affichage : *%ld*.
  - *unsigned long int* ou *unsigned long* : entier long positif- Format d'affichage : *%lu.* Le mot clé *unsigned* permet de supprimer le bit de signe.
  - *short int* ou *short* : 2 octets - Format d'affichage : *%d* (ou *%hd*)

- **Flottant ou réel** : *float* (4 octets) ou *double* (8 octets).
  Format d'affichage : *%f*.
  Les constantes numériques sont par défaut de type *double*.

- **Caractère** : *char* (1 octet)
  Valeurs : -128 à 127 ou 0 à 255 pour *unsigned char*.
  Format d'affichage : *%c* pour les caractères et *%s* pour les chaînes de caractères.

Remarques :

- Les constantes de type caractère sont notées entre apostrophes,
- Les constantes de type chaîne de caractères sont notées entre guillemets
- **Il n'existe pas en C de type chaîne de caractères**.
- On note les **caractères spéciaux (non imprimables)** en commençant par ****.

Caractères spéciaux :

Caractère spécial	Correspondance
\0	caractère nul (nul)
\a	cloche ou bip.
\b	retour arrière (*backspace*).
\f	saut de page (*form feed*)
\n	saut de ligne (*line feed*)
\r	retour chariot (*carriage return*)
\t	tabulation horizontale (*horizontal tab*)
\v	tabulation verticale (*vertical tab*)
\\	\
\'	' apostrophe
\"	" guillemets

En C, il est important de bien comprendre les règles de **conversions implicites** dans l'évaluation des expressions. Par exemple, si *f* est réel, et si *i* est entier, l'expression *f* + *i* est autorisée et s'obtient par la conversion implicite de *i* vers un *float* [CL97].

Certaines conversions sont interdites, par exemple indicer un tableau par un nombre réel. En général, on essaie de faire la plus petite conversion permettant de faire l'opération (voir Figure II.1). Ainsi un caractère n'est qu'un petit entier. Ce qui permet de faire facilement certaines fonctions comme la fonction qui convertit une chaîne de caractères ASCII en un entier (*atoi* est un raccourci pour *Ascii To Integer,* voir annexe XVIII).

C. Déclaration de variables

```
<type> <identificateur>[,<identificateur>,...];
```

C.1 Les tableaux statiques

Il s'agit de variables (caractères, entiers, réels, etc.) stockées consécutivement dans la mémoire et que l'on peut manipuler globalement ou bien élément par élément. Les tableaux statiques sont représentés par des crochets, entre lesquels est indiquée la taille du tableau.

EXEMPLE 1 : *int iTableau[10];* → tableau de 10 entiers,

iTableau est un pointeur, c'est-à-dire l'adresse de début d'une zone mémoire de taille (nombre d'octets) égale au nombre entre crochets multiplié par la taille du type du pointeur, soit 10*2 octets.

Pour accéder au $x^{ième}$ élément du tableau ***iTableau***, il faut faire : ***iTableau[x-1]***

ATTENTION : **Les éléments des tableaux sont indicés de 0 à *n-1*.**

EXEMPLE 2 : *int iTabDeuxDim[5][4];* → tableau d'entiers à deux dimensions de 20 éléments.

iTabDeuxDim est un pointeur sur une zone de 4*5*2 octets.

EXEMPLE 3 : On peut remplir un tableau lors de sa déclaration :

int iTableau[] = {4,5,8,12,-3}; → déclaration et initialisation d'un tableau (on peut mettre le nombre d'éléments à l'intérieur des crochets, mais dans ce cas, ce n'est pas obligatoire). La taille du tableau est fixée à 5 éléments.

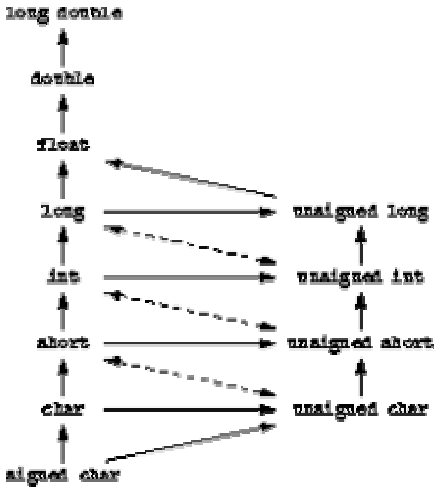

Figure II.1 - Les conversions des types en C [CL97].

C.2 Les chaînes de caractères

Les chaînes de caractères sont représentées par des tableaux statiques ou par des pointeurs.

EXEMPLES :

- *char cTabStatique[dimension];*

  cTabStatique est un identificateur dont le contenu est une adresse mémoire, à partir de laquelle on peut réserver un nombre d'octets égal à la taille en octets d'un *char* multiplié par le nombre de caractères de la chaîne, c'est-à-dire un pointeur sur une zone de caractères de taille *dimension*taille_en_octets(char)*.

  Comme pour les tableaux numériques, on peut initialiser un tableau de caractères (ou chaîne de caractères) lors de sa déclaration :

EXEMPLE 1 : *char t[] = "abcde"*
EXEMPLE 2 : *char t[] = {'a','b',...}*

- *char* cPointeur;*

  cPointeur est un pointeur vers des caractères dont la définition est dynamique. Il n'y a pas de réservation mémoire à la déclaration.

  Pour mettre des caractères dans la valeur de *cPointeur*, il faut une **allocation mémoire** :

  cPointeur = (char) malloc (x)*

  - Cette fonction retourne un pointeur, ***x*** est le nombre d'octets à allouer.
  - *(char*)* est un **cast**, c'est-à-dire un opérateur de conversion. Le pointeur retourné va être converti en un pointeur de type pointeur sur un caractère.

  Pour **libérer la zone mémoire allouée** : *free (cPointeur);*

EXEMPLE 3 :

CE QU'IL NE FAUT PAS FAIRE

```
/*************************************************************/
/*                Programme principal                        */
/* Exemple de modification d'adresse pointée par un pointeur  */
/*************************************************************/
void main ()
{
    /* Déclaration et initialisation de deux pointeurs sur des
    caractères  - NULL = pointeur vide */
    char* cPointeur1=NULL;
    char* cPointeur2=NULL;
    /* cPointeur1 pointe sur une zone mémoire de taille 15 oct */
    cPointeur1 = (char*) malloc (15);
    /* cPointeur2 pointe sur une zone mémoire de taille 10 oct */
    cPointeur2 = (char*) malloc(10);
    /* cPointeur1 pointe sur l'adresse de cPointeur2 */
    cPointeur1 = cPointeur2; /* ATTENTION: l'adresse pointée par
                                cPointeur1 est perdue. Mais la
                                mémoire est toujours allouée. */
    /* Désallocation mémoire de cPointeur1*/
    free(cPointeur1); /* équivalent à free (cPointeur2)*/
}
```

La figure, qui suit, illustre pas à pas le résultat de ce programme. Lorsque l'on désalloue la mémoire occupée par *cPointeur1* (dernière ligne du programme), *cPointeur1* et *cPointeur2* ne pointent plus sur rien. Mais, la zone de 15 octets allouée auparavant pour *cPointeur1* n'est pas libérée. L'adresse de cette zone est perdue après l'affectation *cPointeur1 = cPointeur2;*. Comme la mémoire n'a pas été désallouée, on ne peut plus rien en faire.

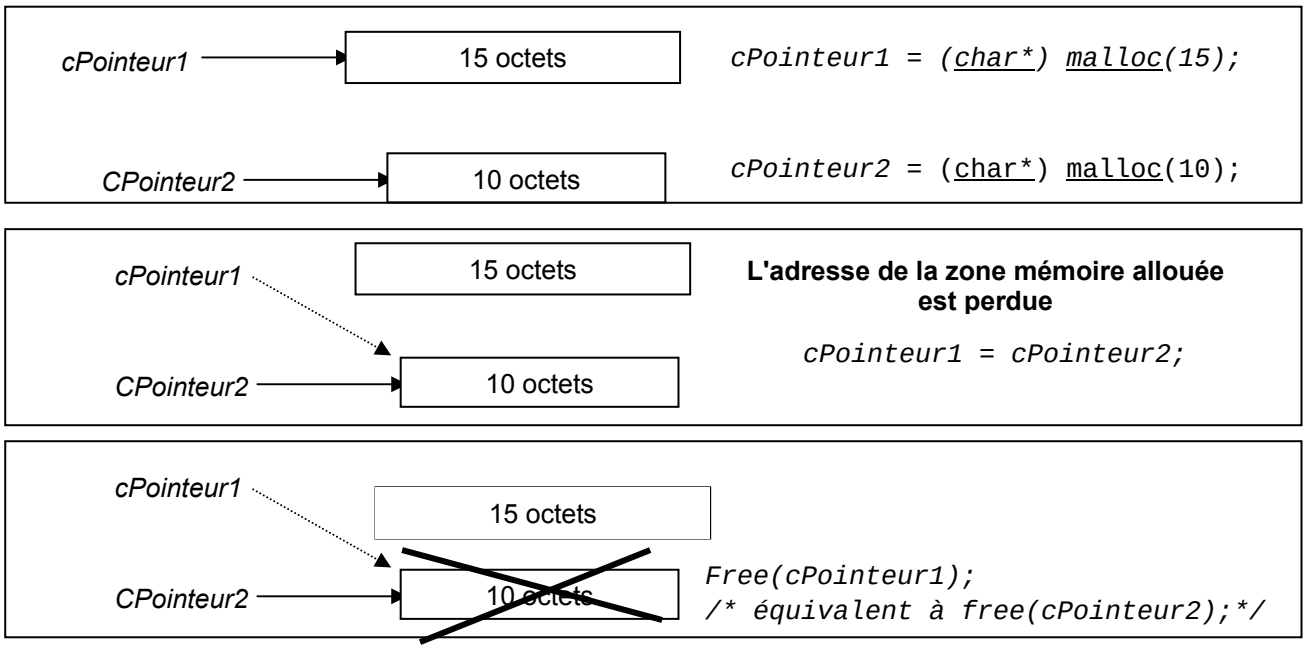

Figure II.2 - La désallocation mémoire de deux pointeurs égaux.

Conseil : Pour ne pas perdre de la mémoire inutilement, il faut toujours libérer la mémoire allouée pour un pointeur. Par convention, **initialisez toujours vos pointeurs à *NULL***, qui signifie *ne pointe sur rien*. Puis, **après chaque désallocation mémoire (appel de *free*), affectez le pointeur à *NULL***. Ainsi, dès que vous désirez changer l'adresse d'un pointeur, il vous suffit de vérifier que ce dernier est égal à *NULL*. Si ce n'est pas le cas, il faut par conséquent faire une libération de la mémoire pointée avant tout changement d'adresse. Cet exemple est valable quel que soit le type du pointeur. Pour plus de précisions sur les pointeurs, voir section VI.

D. Les opérateurs

D.1 opérateurs élémentaires

- **Arithmétiques** : `+, -, *, /, %` (modulo, i.e. reste de la division entière).
- **Relation** : == (égal), `<, >, <=, >=, !=` (différent).
- **Logique** : `!` (négation d'une condition), `&&` (et), `||` (ou).
- **Affectation** : =
- **Opérateur conditionnel** : `<exp1> ? <exp2> : <exp3>`

  Si la valeur de `exp1` est vraie (non nulle), le résultat est `exp2`, sinon le résultat est `exp3`.

EXEMPLE : *y=((x==3)?1:-1), si x=3 alors y=1, sinon y=-1*

Ordre de priorité des opérateurs unaires (par ordre décroissant) :

```
*    %    /
+    -
<    >=    <=     >
==     !=
&&
||
?:
=
```

Remarque [CL97] : L'opérateur = d'affectation étant un opérateur comme les autres dans les expressions, il subit les mêmes lois de conversion. Toutefois, il se distingue des autres opérations par le type du résultat. Pour un opérateur ordinaire, le type du résultat est le type commun obtenu par conversion des deux opérandes. Pour une affectation, le type du résultat est reconverti dans le type de l'expression à gauche de l'affectation.

D.2 Combinaison des opérateurs

1. **i++** ou **++i** ⇔ i = i + 1

Attention : *i*++ ou ++*i* ne donnent pas le même résultat lorsqu'ils interviennent au sein d'une expression. Les exemples (n°1, 2 et 4), qui suivent, le montrent. Lorsque *i*++ ou ++*i* n'agissent que sur *i* sans intervenir dans une expression, leur résultat est équivalent (voir exemple n° 3).

EXEMPLES :

Ex. 1

```
int i=5,j,k;
j=i++; /* j=5 puis i=6 */
k=++i; /* i=7 puis k=7 */
```

Ex. 2

```
int i, x, t[5];
i=3;
x=t[i++]; /* x=t[i]=t[3] et i=i+1=3+1=4 */
```

Ex. 3

```
int i = 3;
/* incrémentation de i */
i++; /* i=4 */
/* incrémentation de i */
++i; /* i =5 */
```

Ex. 4

```
int i, x, t[5];
i=3;
x=t[++i]; /* i=i+1=3+1=4 et x=t[i]=t[4]*/
```

2. **i--** ou **--i** ⇔ i = i -1

Attention : de même, *i--* et *--i* ne donnent pas le même résultat lorsqu'ils interviennent dans une expression.

EXEMPLES :

```
int i=7,l,m;
l=i--; /* l=7 puis i=6 */
m=--i; /* i=5 puis m=5 */
i=3; /* affectation de i à 3 */
while (i--) /* i est testé (tant que i ≠ 0) et après i=i-1 */
printf ("%d",i); /* affiche 2,1,0. A la sortie i=-1*/
```

3. **i+=j** ⇔ i = i + j

4. **i-=**j ⇔ i = i - j

5. **/=**; ***=** ; **%=**

Conseil [CL97] : En règle générale, il ne faut pas abuser des opérations d'incrémentation. Si c'est une commodité d'écriture, il n'y a pas de problème. Mais, si l'expression devient incompréhensible et peut avoir plusieurs résultats possibles selon un ordre d'évaluation dépendant de l'implémentation, alors il ne faut pas utiliser ces opérations et il est préférable de casser l'expression en plusieurs morceaux pour séparer la partie effet de bord. Une expression qui modifie la valeur d'une variable pendant son évaluation est dite à effet de bord.

Pour forcer la priorité des opérateurs, il est préférable d'utiliser des parenthèses.

D.3 Expressions sur les bits

Les opérations sur les bits peuvent se révéler très utiles [CL97]. On peut faire & (et logique), | (ou logique), ^ (ou exclusif), << (décalage vers la gauche), >> (décalage vers la droite), ~ (complément à un).

Attention : Il faut bien distinguer les opérations logiques && et || à résultat booléen (0 ou 1) des opérations & et | sur les bits qui donnent une valeur entière. Par exemple, si x vaut 1 (en binaire 01) et y vaut 2 (en binaire 10) *, x & y* vaut 0 (car 01 ET 10 = 00 en binaire) et *x && y* vaut 1 (1&&2 est toujours vrai).

Les opérations << et >> décalent leur opérande de gauche de la valeur indiquée par l'opérande de droite. Ainsi 3 << 2 vaut 12, et 7 >> 2 vaut 1. En effet, en binaire, 3_{10} peut s'écrire 0011_2 et si l'on

décale ce nombre de 2 bits vers la gauche, on obtient 1100_2 soit 12_{10}. De même, 7_{10} en binaire s'écrit 111_2, et s'il l'on décale ce nombre binaire de deux bits vers la droite, on obtient 001_2 soit 1_{10}.

De manière générale $y << x$ est équivalent à $y * 2^x$ et $y >> x$ est équivalent à $y / 2^x$.

E. Les structures de contrôle

E.1 Choix

```
if (<condition>) <action 1>;
  [else
<action 2>;]
```

- Toutes les conditions sont entre parenthèses.
- Le *else* est optionnel (c'est pourquoi il est mis entre crochets)
- S'il y a plusieurs instructions après le *if* ou le *else*, on les encadre par des accolades : { }.
- La valeur logique équivalente à vrai est différente de 0, et **la valeur logique équivalente à faux est 0.**

EXEMPLE 1 : `if (i)` → Si i est vrai (!=0)

EXEMPLE 2 : `if (!i)` → Si i est faux (==0)

Attention : un *else* se rapporte toujours au dernier *if* rencontré et auquel un *else* n'a pas encore été attribué (voir exemple suivant sur la droite).

EXEMPLE 3 :

```
/*     Exemple de if                */
if(x==3)
{
  y=1;
  printf("x=3 donc : y=%d",y);
}

/* Sinon, si x est différent de 3 */
else
{
  y=-1;
  printf ("x !=3 donc : y=%d",y);
}
```

```
/*   Exemple de if imbriqués   */


if(a<b)
      if(c<d)
      j=1 ;

      /* sinon (c ≥ d)          */
      else
      j=2 ;

/* sinon (a ≥ b)              */
/* on ne fait rien            */
else ;
```

E.2 Choix multiple

```
switch (<condition>)
      { case <valeur 1> : <action 1>; break;
        case <valeur 2> : <action 2>; break;
        …
        default : <action n>;
      }
```

EXEMPLES :

```
/* Exemple 1 :                    */
/* Test sur la valeur de i        */

j=0, k=0, l=0;
switch (i)
{

/* Si i a pour valeur 0       */
/* Alors j=1, et on passe à   */
/* l'instruction suivante     */
      case 0: j=1;

/* Si vaut 0 ou 1             */
/* j = 0, et on passe à       */
/* l'instruction suivante     */
      case 1: k=1;

/* Quel que soit la valeur de i  */
/* on affiche erreur             */
      default : l=1;

} /* Fin du switch*/
```

```
/* Exemple 2 :                */
/* Test sur la valeur de i    */

j=0, k=0, l=0;
switch (i)
{

/* Si i a pour valeur 0       */
/* Alors j=1, et on sort du   */
/* switch                     */
      case 0: j=1;
      break;

/* Si vaut 1, alors j=0       */
/* et on sort du switch       */
      case 1: k=1;
      break;

/*Pour i différent de 0 ou 1  */
/* on affiche erreur          */
      default : l=1;

} /* Fin du switch*/
```

Attention : Il faut mettre un *break* pour chaque *case*, afin d'éviter que toutes les actions soient exécutées en cascade (voir premier exemple précédent à gauche).

E.3 Itération

Il existe deux manière de programmer une itération :

```
while (<condition>)
<actions>;
```

```
do<actions>
while(<condition>);
```

Dans la première forme d'itération (*while* …), la condition est évaluée. Puis, tant que la condition est vraie, les actions sont exécutées.

Dans la deuxième forme d'itération (*do* ..), les actions sont exécutées au moins une fois. Puis, tant que la condition est vraie, elles sont exécutées de nouveau.

Un *break* dans un *while*, quelle que soit la forme de l'itération, permet de sortir de la boucle.

EXEMPLE 1 :

```
i=0;                    while
(i<2)
{ i++;
  printf("%d\n",i);
}
printf("fin");
```

```
-------------------
Résultat :
1
2
fin
-------------------
```

```
i=2;
while (i<2)
{ i++;
  printf("%d\n",i);
}
printf("fin");
```

```
-------------------
Résultat :
fin
-------------------
```

EXEMPLE 2 :

```
i=0;
do
{
      i++;
      printf("%d\n",i);
}
while (i<2) ;
printf("fin");
```

```
-------------------
Résultat :
1
2
fin
-------------------
```

```
i=2;
do
{
      i++;
      printf("%d\n",i);
}
while (i<2) ;
printf("fin");
```

```
-------------------
Résultat :
3
fin
-------------------
```

E.4 Boucle for

```
for ([<instructions initiales>];[<condition>];[<instructions>])
          [<corps de la boucle>];
```

```
for (exp1;exp2;exp3)
  instruction;
```

équivalent à :

```
expr1;
while (expr2)
{
      instruction;
      expr3;
}
```

L'*exp1* est exécutée à la première itération. Les *exp2* et *exp3* sont réévaluées à chaque fois que l'*instruction* (le corps de la boucle) est exécutée.

S'il y a plusieurs instructions initiales ou finales, elles sont séparées par des virgules.

<condition> ⇔ Tant que la *condition* est vraie.

Les *<instructions>* sont exécutées après le corps de la boucle mais avant de tester à nouveau la condition.

EXEMPLES :

```
for (;;);            boucle infinie.

S=0;
for (i=0;i<2;i++)    Etapes suivies :
   S+=i;             1. Instruction initiale de la boucle : i=0;
                     2. Test de la condition i<2.
                     3. Premier passage dans le corps de la boucle :
                        S=S+0 (ici S==0)
                     4. Incrémentation de i : i=1
                     5. Test de la condition : 1<2 (vrai)
                     6. Deuxième passage dans le corps de la boucle :
                        S=S+1 (ici S==1)
                     7. Incrémentation de i : i=2
                     8. Test de la condition : 2<2 (faux)
                     9. Fin


for(i=0,j=n;i<j;i++,j--)          for(c='A';c<='Z';c++)
{                                  {
     instructions                        instructions
}                                  }
```

F. Affichage écran

F.1 Syntaxe de la fonction printf

```
printf ("format",[<exp1>,<exp2>,...]);
```

format correspond aux formats de sortie de chacune des expressions *exp*.

La valeur d'un format peut être :

%d	Pour l'affichage d'un entier sous forme décimal
%f	Pour l'affichage d'un réel.
%c	Pour l'affichage d'un caractère.
%s	Pour l'affichage d'un chaîne de caractères.
%x	Pour le code hexadécimal.

EXEMPLE :

```
#include <stdio.h>

/*****************************************************************/
/*                      Programme principal                      */
/* Affichage du code ASCII et du code hexadécimal du caractère A */
/*****************************************************************/

void main ()
{
      /* Déclaration et initialisation du caractère A */
      char cA = 'A';
      /* Affichage du caractère, de son code ASCII*/
      /* et de son code hexadécimal */
      printf ("Caractère = %c, code ASCII = %d, code hexa = %x",cA,cA,cA);
}
```

F.2 En cas d'erreur de programmation

La fonction *printf* est souvent une source d'erreur de programmation, notamment lorsque [Del97]:

1. Le code de format est en désaccord avec le type de l'expression. Deux cas peuvent se présenter :
    a) Si le code de format et l'information ont la même taille (occupant le même nombre d'octets en mémoire), cela entraîne une mauvaise interprétation de l'expression. Par exemple, si vous afficher un entier avec %u (pour *unsigned*).
    b) Si le code de format et l'information n'ont pas la même taille. Il y a alors un décalage dans l'affichage.
2. Le nombre de codes de format est différent du nombre d'expression dans la liste. Il faut savoir que le langage C cherche toujours à satisfaire le code de format. S'il y a trop d'expressions par rapport aux codes, toutes les expressions ne seront pas affichées. Par exemple *printf("%d",n,p);* n'affichera que la valeur de *n*. Si, en revanche, il y a plus de codes de format que d'expressions, *printf* affichera n'importe quoi pour les codes dont les expressions manquent.

III. TRAITEMENT DES CHAINES DE CARACTERES

A. Lecture de chaîne

```
scanf ("%s", adresse de la chaîne);
```

Il n'y a pas de limite de saisie de caractères, ni de contrôle de longueur de chaîne

Les blancs (espaces) ne sont pas saisis par *scanf*. Les espaces sont considérés comme des séparateurs. La chaîne sera lue uniquement jusqu'au premier espace.

Convention pour la longueur des chaînes de caractères : le caractère terminal \0

C'est à la rencontre du **\0** que l'on sait que la chaîne est terminée.

Lorsque la chaîne est saisie caractères par caractères, il ne faut pas oublier d'ajouter ce caractère terminal. Néanmoins, *scanf* l'ajoute automatiquement.

Lorsque la chaîne de caractères est gérée par un pointeur, il faut allouer la mémoire avant d'utiliser la fonction *scanf*.

Comme pour la fonction *printf*, *scanf* peut être source d'erreurs de programmation [Del97]:

1. Si le code de format diffère avec le type de l'expression, deux cas se présentent :
   a) Si la taille du code et la taille de l'expression sont égales, il y aura introduction d'une mauvaise valeur.
   b) En revanche, si la taille du code de format et la taille de l'expression sont différentes, il y aura écrasement d'un emplacement mémoire.
2. Si le nombre de codes de format diffère du nombre d'éléments dans la liste, comme *printf*, *scanf* cherche toujours à satisfaire le code de format. S'il y a plus d'expression que de codes de format, certaines expressions ne seront pas lues. En revanche, s'il y en a moins, *scanf* cherchera à affecter des valeurs à des emplacements presque aléatoires de la mémoire.

B. Fonctions *gets* et *puts*

```
char* gets (char*);
void puts(char*);
```

- La fonction *gets* ramène l'adresse de la chaîne lue. Tous les caractères sont acceptés (même les espaces). Le *\0* est mis à la fin de la chaîne, mais il n'y a toujours pas de contrôle de longueur. ⇔ *scanf("%s", char*)* sans tenir compte des espaces
- La fonction *puts* affiche la chaîne qui est en argument, jusqu'à ce que la fonction rencontre un *\0*, et passe à la ligne. ⇔ *printf("%s",char*)*

C. Initialisation des chaînes à la déclaration

EXEMPLE :

```
char cTableau[]="abc"; /* cTableau pointe sur une zone de 4 octets       */
                       /* cTableau[0] = 'a', ….cTableau[3]='\0' */

char* cPointeur="abc";  /* l'allocation mémoire est faite directement */
                       /* à la compilation                            */
```

Attention :

- *cTableau* est un pointeur constant; le nombre d'octets pointés est constant. Néanmoins, la chaîne (le contenu) n'est pas constante, puisqu'il s'agit d'un tableau.
- *cPointeur* est modifiable car il s'agit d'une adresse. Cependant, on ne peut pas changer directement son contenu.

```
char cLigne[]="efg";
cTableau=cLigne;          /* Erreur, cTableau est constant,     */
                          /* on ne peut pas changer son adresse */
strcpy(Ctableau,cLigne);  /* Correct car changement de contenu */
                          /* cf section IV.C */
cPointeur=cLigne;         /* Correct car changement d'adresse */
strcpy(cPointeur,"efg");  /* Erreur : le contenu ne peut pas être changé */
```

D. Accès aux caractères individuels

- **Par indices** : Que l'on ait déclaré *char cTableau[10]* ou *char * cPointeur*; *cTableau[i]* désigne le i^{ème} caractère -1 de la chaîne, de même que *cPointeur[i]*.
- **Par pointeurs** : * *(cTableau+i)* est l'adresse de *cTableau* avec un décalage de *i* octets; ce qui est équivalent à *cTableau[i].* * signifie que l'on prend le contenu.

```
/* Déclaration et initialisation de la chaîne ``abcd'', */
/* pointée par cPointeur                                */
char* cPointeur="abcd";

/* Déclaration d'un pointeur sur une chaîne de caractères : cPointeur2 */
char* cPointeur2=NULL;

/* Déclaration d'un caractère cCaractere */
char cCaractere;

/* cPointeur2 pointe à la même adresse que cPointeur*/
cPointeur2=cPointeur;

/* cCaractere prend la valeur de a, et cPointeur2 pointe sur b */
cCaractere=*cPointeur2;
cPointeur2++;
```

Remarques :

- *'a'* est le caractère *a*; il occupe 1 octet.
- *"a"* est une chaîne de caractère. Elle occupe deux octets: un pour le caractère *a* et un pour le caractère de fin de chaîne.

IV. FONCTIONS

Un programme écrit d'un seul tenant est difficile à comprendre dès qu'il dépasse une ou deux pages. Une écriture modulaire permet de scinder le programme en plusieurs parties et de regrouper dans le programme principal les instructions décrivant les enchaînements [Del97]. La programmation modulaire permet d'éviter les séquences d'instructions répétitives et permet le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois. En C, il n'existe qu'une sorte de module, nommé fonction. Un programme est donc une suite linéaire de fonctions ou procédures, non emboîtées. Par convention, le début de l'exécution est donné à la procédure *main* [CL97].

A. Généralités

Convention : Les fonctions sont écrites avant le programme principal (*main*). Il est possible de les définir après, mais il faut alors les déclarer avant.

Une fonction en C est assez proche de la notion mathématique correspondante [Del97]. Elle dispose d'arguments ou peut ne pas avoir d'arguments du tout.

```
/* Déclaration d'une fonction */
type-de-retour nom-de-fonction(déclarations d'arguments);
```

On appelle cela la **signature** ou **prototype** de la fonction.

```
/* Définition du corps de la fonction */
type-de-retour nom-de-fonction(déclarations d'arguments)
{
    déclarations et instructions
}
```

Les instructions entre accolades correspondent au **corps** de la fonction.

Il existe deux types principaux de fonctions :

- Les fonctions qui retournent quelque chose (si on ne met rien, une fonction retourne un entier (*int*) par défaut). Pour cela, il faut utiliser l'instruction ***return*** : *return expression*;
- Les fonctions qui ne retournent rien (*void*) dont la signature est :
  void nom_fonction(déclarations_d'arguments)

  Ces fonctions sont également appelées **procédures** (fonctions qui ne retournent rien).

EXEMPLES :

```
/* fonction calculant le produit de deux entiers                    */
/*                                                                  */
/* Paramètres d'entrée : deux entiers                               */
/* Type retour : entier                                             */

int calcule_produit (int iExp1, int iExp2)
{ int iProduit;
  iProduit = iExp1 * iExp2;
  return iProduit;
}
```

```
/* Procédure affichant le produit de deux entiers */
/*                                                */
/* Paramètres d'entrée : deux entiers             */
/* Type retour : rien                             */

void affiche_produit (int iExp1, int iExp2)
{ int iProduit;
  iProduit = iExp1 * iExp2;
  printf ("Le produit de %d et %d est égal à %d", iExp1, iExp2,
iProduit);
}
```

B. Transmission des arguments en C

En C, la transmission des arguments se fait par valeur. Les exemples qui suivent montrent ce qu'il faut et ce qu'il ne faut pas faire, lors de la transmission des arguments à une fonction.

Remarque : les **variables** déclarée à l'intérieur des fonctions sont dites **locales** à la fonctions. Leur durée de vie est limitée à celle d'une exécution de la fonction. Leur **portée** (ou **espace de validité**) est donc limitée à cette fonction.

EXEMPLE 1 :

Correct

```
/* Définition de la fonction saisie :                    */
/* saisie d'une chaîne au clavier                        */
/*                                                       */
/*    Paramètre : une chaîne de caractères               */
/*    Type retour : rien                                 */

void saisie (char* cChaine)
{
      /* Appel de la fonction de la bibliothèque stdio.h */
      /* Le contenu de l'argument cChaine est modifié     */
      gets (cChaine);
}

/* Programme principal*/
void main ()
{
      /* Déclaration d'une chaîne de caractères */
      char* cCh;

      /* Allocation mémoire pour une chaîne */
      /* contenant 50 caractères de 0 à 49  */
      cCh = (char*)malloc(50);

      /* Saisie de la chaîne au clavier   */
      /* par appel de la fonction saisie  */
      saisie(cCh);
}
```

EXEMPLE 2 : **Faux**

```
/* Définition de la fonction saisie :                          */
/* saisie d'une chaîne au clavier                              */
/*    Paramètre : une chaîne de caractères                     */
/*    Type retour : rien                                       */

void saisie (char* cChaine)
{
      /* Déclaration d'une chaîne possédant 80 caractères */
      /* La chaîne est locale à la fonction saisie        */
      char cCh[80];

      /* Appel de la fonction de la bibliothèque stdio.h */
      cChaine=gets(cCh); /* FAUX */
}
/* même programme principal que dans l'exemple 1 */
```

Dans l'exemple 2, *cCh* est **allouée localement et statiquement dans la fonction** *saisie*. A la sortie de la fonction, *cCh* est perdue. A la sortie de saisie, *cChaine* n'a donc aucune valeur.

EXEMPLE 3 : **Faux**

```
/* Définition de la fonction saisie :                          */
/* saisie d'une chaîne au clavier                              */
/*    Paramètre : une chaîne de caractères                     */
/*    Type retour : rien                                       */

void saisie (char* cChaine)
{
      /* Allocation mémoire de l'argument */
      cChaine= (char*)malloc (80);

      /* Appel de la fonction de la bibliothèque stdio.h */
      gets (cChaine);
}

/* Programme principal*/
void main ()
{
      /* Déclaration des variables */
      char* cCh;

      /* Saisie de la cChaine au clavier */
      /* par appel de la fonction saisie */
      saisie(cCh); /* cCh n'a pas d'adresse! */
}
```

Il y a, dans l'exemple 3, un **problème de transmission d'adresse**. *cChaine* est allouée localement à la fonction. La saisie est faite dans *cChaine*, mais il n'y a rien dans *cCh*.

Attention : En C, on ne peut faire que des **transmissions par valeur**. Pour **faire une transmission par référence,** il faut explicitement transmettre l'adresse contenant la variable.

EXEMPLE 4 :

Transmission par référence

```
/* Définition de la fonction saisie :                          */
/* saisie d'une chaîne au clavier                              */
/*   Paramètre : Un pointeur sur une chaîne de caractères      */
/*   Type retour : rien                                        */
void saisie (char** cChaine)
{     /* Allocation mémoire de l'argument */
      *cChaine= (char*)malloc(80);
      /* Appel de la fonction de la bibliothèque stdio.h */
      gets(*cChaine);
}
/* Programme principal*/
void main()
{   /* Déclaration des variables */
      char* cCh;
      /* Saisie de la chaîne au clavier  */
      /* par appel de la fonction saisie */
      /*  On  transmet  l'adresse  de  cCh  (utilisation  du
      caractère de déférencement &) en argument de saisie */
      saisie(&cCh);
}
```

Dans l'exemple 4, *cChaine* pointe sur une zone qui pointe sur une zone de caractères . A la sortie de la fonction *saisie*, *cChaine* est perdue, mais *cCh* pointe bien sur une zone de caractères de 80 octets. Par conséquent, il s'agit bien d'un passage **par valeur**, puisqu'il y a eu transmission d'adresse, mais cela revient à un passage par référence.

On peut également utiliser une fonction de la sorte :

EXEMPLE 5 :

Fonction retournant une chaîne de caractères

```
/* Définition de la fonction saisie :                          */
/* saisie d'une chaîne au clavier                              */
/*    Paramètre : Aucun                                        */
/*    Type retour : Une chaîne de caractères                   */
char* saisie ()
{     /* Définition d'une variable locale de */
      /* type chaîne de caractères */
      char* cChaine;
      /* Allocation mémoire de la chaîne de caractères */
      cChaine=(char*)malloc(80);
      /* Saisie de la chaîne au clavier */
      gets (cChaine);
      /* La chaîne saisie est retournée par la fonction */
      return cChaine;
}
/* Programme principal*/
void main ()
{     /* Déclaration des variables */
      char* cCh;
      /* Saisie de la chaîne au clavier  */
      /* par appel de la fonction saisie */
      cCh=saisie();
}
```

EXEMPLE 6 :

Faux

```
/* Définition de la fonction saisie :                    */
/* saisie d'une chaîne au clavier                        */
/*     Paramètre : Aucun                                 */
/*     Type retour : Une chaîne de caractères            */

char* saisie ()
{
      /* Définition d'une variable locale */
      /* de type chaîne de caractères     */
      char cChaine[80]; /* variable locale,                  */
                        /* la chaîne du programme principal */
                        /* ne sera pas allouée              */

      /* Saisie de la chaîne au clavier */
      gets (cChaine);

      /* La chaîne saisie est retournée par la fonction */
      return cChaine;
}
/* même programme principal que dans l'exemple précédent */
```

Dans l'exemple 6, l'allocation est statique et locale à la fonction. *cCh* dans le programme principal n'aura donc aucune valeur.

C. Fonctions de traitement de chaînes de caractères

Bibliothèque : # *include<string.h>*

- *int* `strlen` *(char*)* : retourne le nombre de caractères d'une chaîne.
- *int strcmp (char* s, char* t)* : compare deux chaînes *s* et *t*, et retourne une valeur positive si *s* est alphanumériquement supérieure à *t*, une valeur négative si *s* est alphanumériquement inférieure à *t*, ou 0 si *s* et *t* sont égales.
- *char* strcpy (char* dest,char* src)* : copie la chaîne *src* dans la chaîne *dest*. La fonction retourne *dest*.

Exercice : Programmez la fonction strcpy sous forme de boucle (vous utiliserez une boucle *while*) de deux manières : (1) avec un accès aux caractères par indice, (2) avec un accès au caractères par pointeurs (voir section III.D).

La correction vous sera donnée en cours.

D. Fonctions récursives

Le langage C autorise la récursivité des fonctions. Toute fonction récursive possède une condition d'arrêt et comporte dans sa définition au moins un appel à elle-même. Il existe un théorème qui dit :

Toute fonction récursive peut être écrite sous forme itérative (l'inverse n'est pas vrai).

Un exemple de fonction récursive est donné dans l'annexe XVIII.B.

V. CONSTANTES

```
const type nom = valeur;
```

Le mot clé *const* permet de qualifier une variable dont on s'interdit de modifier le contenu.

EXEMPLE :

```
/* Constante de tabulation verticale en ASCII */
const char TABV = '\013';
```

VI. COMPLEMENT SUR LES POINTEURS

A. Définition des pointeurs

Un pointeur est une référence sur une donnée. Il s'agit d'une adresse en mémoire où sont stockées les données référencées. On peut définir des pointeurs sur n'importe quoi :

```
/* pointeur sur un entier */
int* p;
```

Cette déclaration signifie que **p* , c'est-à-dire l'objet d'adresse *p*, est de type entier; ce qui signifie que *p* est l'adresse d'un entier.

```
/* Tableau de 100 pointeurs sur entier */
int* t[100];

/* pointeur sur un pointeur d'entier */
int** p;
```

La constante ***NULL***, définie dans le fichier <*stdio.h*> est la valeur standard pour un pointeur vide (ne pointant sur rien). Cette constante est très utile pour l'initialisation des pointeurs et les tests (vérification qu'un pointeur pointe sur une zone mémoire alloué).

B. Opérateur * et &

L'opérateur ***** désigne le contenu de l'adresse pointée ; par exemple, **iPointeur* désigne le contenu de la zone mémoire pointée par *iPointeur*. L'opérateur **&** (que nous avons déjà utilisé dans *scanf*) est un opérateur unaire qui fournit comme résultat l'adresse de son opérande.

EXEMPLE :

```
/* Déclaration d'un pointeur sur un entier */
/* et initialisation du pointeur à NULL    */
int* iPointeur=NULL;

/* Déclaration et initialisation d'un entier */
int iEntier =20;
/* Placement de l'adresse de l'entier dans le pointeur */
iPointeur = &iEntier;
/* Modification de l'entier pointé */
*iPointeur = 30; /* équivalent à n=30; */
```

C. Allocation mémoire et initialisation

L'initialisation d'un pointeur peut se faire **en transmettant l'adresse d'un autre pointeur**, ou en **allouant de la mémoire** :

```
/* Déclaration d'un entier et de deux pointeurs sur un entier et
initialisation des pointeurs à NULL */
int iEntier;
int* iPointeur1=NULL;
int* iPointeur2=NULL;

/* Initialisation de iPointeur1 :              */
/* iPointeur1 pointe sur l'adresse de iEntier  */
iPointeur1=&iEntier;
```

Les fonctions ***malloc*** et ***free*** de la librairie C permettent respectivement d'allouer de la mémoire à un pointeur ou de désallouer la mémoire pointée par le pointeur. Ces fonctions sont définies dans la librairie <*stdlib.h*>.

```
/*Allocation mémoire de iPointeur2 sur une zone mémoire */
/* de la taille d'un entier                             */
iPointeur2=(int*)malloc(sizeof(int));
```

La fonction ***sizeof***, de la librairie C, permet de calculer la taille occupée par un type de données. En effet, cette taille peut varier en fonction du type de la machine utilisée. Cette fonction offre ainsi à l'utilisateur une transparence parfaite. L'utilisateur n'a pas besoin de retenir, pour chaque machine, le nombre d'octets occupés par un type donné. Cette fonction permet également, dans le cas d'un type complexe, de ne pas calculer la taille en octets occupée par le type.

Les opérateurs de base sur les pointeurs sont **&** pour référencer un objet et ***** pour le déréférencer (pour désigner la valeur d'un pointeur), écrits en préfixe :

```
/* iPointeur pointe sur une zone mémoire dont on modifie la valeur */
*iPointeur = 3;
/* iValeur est égale à la valeur de la zone mémoire */
/* où pointe iPointeur                              */
iValeur=*iPointeur;
```

D. Opérations sur les pointeurs

D.1 Comparaison

Condition : les pointeurs doivent être de même type.

EXEMPLE :

```
/* Déclaration de deux pointeurs sur des entiers */
int* iPointeur1=NULL;
int* iPointeur2=NULL;
...
/* Comparaison des adresses de iPointeur1 et iPointeur2, */
/* mais pas de leur contenu                              */
if(iPointeur1==iPointeur2) ...
```

D.2 Addition et soustraction d'un entier

EXEMPLE :
```
/* Pointeur pointe sur l'élément suivant de la structure */
/* sur laquelle il pointe                                */
Pointeur++;
/* Décalage du pointeur de 2 éléments */
/* dans la structure pointée          */
Pointeur+=2;
```

Il est possible de soustraire deux pointeurs (mais pas de les additionner)

EXEMPLE :
```
/* Calcul du nombres d'octets entre les deux adresses */
/* de Pointeur1 et Pointeur2                          */
/* il faut que p>q en adresse                         */
iDifference_Adresse = Pointeur1-Pointeur2;
```

E. Gestion dynamique des tableaux

EXEMPLE :
```
/* Déclaration d'un pointeur sur des entiers iPointeur */
/* et de la taille du tableau iTaille                  */
int* iPointeur=NULL;
int iTaille;
/* Saisie de la taille du tableau par l'utilisateur */
scanf ("%d",&iTaille);

/* Allocation dynamique de la zone mémoire occupée */
/* par le tableau                                  */
iPointeur = (int*)malloc (iTaille*sizeof(int));
```

VII. TYPE STRUCTURES

A. Définition

```
struct [<nom de la structure>]
  { <type> <nom ou identifiant>; /* champ */
    ...
  } [<nom d'une variable>];   /* <nom d'une variable> est     */
                              /* une variable de type         */
                              /* struct <nom de la structure> */
```

EXEMPLE :
```
/* Structure personne, composée de deux champs : cNom et iAge */
struct personne
{
  char* cNom;
  int iAge;
};
/* Déclaration d'un élément individu de type struct personne */
struct personne individu;
```

```
/* Déclaration d'un élément humain de type struct pers */
struct pers
{
  char* cNom;
  int iAge;
} humain;
```

B. Accès à un élément d'une structure

Pour accéder aux champs d'une structure on utilise un point. Et pour accéder à ces mêmes champs à l'aide d'un pointeur sur une structure on utilise une combinaison du point ou de l'étoile (voir exemple) ou une flèche.

EXEMPLE :

```
/* Structure personne, composée de deux champs : cNom et iAge */
struct personne
{
 char* cNom;
 int iAge;
};
int main()
{
/* Déclaration d'un élément individu de type struct personne */
/* et d'un pointeur pindividu sur la structure                */
struct personne individu;
struct personne *pindividu=NULL;

…

/* Accès au nom de individu (variable de type struct personne)*/
printf ("%s", individu.cNom);
/* Accès à l'âge de individu */
printf("%d",individu.iAge);

/* Deux méthode d'accès au nom de pindividu*/
/* (pointeur sur une structure)            */
/* 1 : les parenthèses sont importantes car . a priorité sur * */
printf("%s",(*pindividu).cNom);
/* 2 : Il est préférable d'utiliser cette méthode */
printf("%s", pindividu->cNom);
}
```

C. Allocation mémoire des pointeurs sur les structures

Il faut toujours faire attention à bien allouer le pointeur sur la structure, ainsi que les champs de la structure, s'il s'agit de pointeurs.

```
/* Allocation mémoire du pointeur pindividu */
/* sur la structure personne                */
pindividu = (struct personne*)malloc(sizeof(struct personne));

/* Allocation mémoire des champs de la structure */
(*pindividu).cNom = (char*)malloc(100);
```

VIII. DEFINITION DE TYPE

En C, on peut déclarer les types en utilisant le mot clé ***typedef.***

A. Exemples de type

```
/* Type d'un pointeur sur un entier */
typedef int *pointentier;
/* Déclaration d'une variable de type pointeur sur un entier */
pointentier a = NULL; /* a est un pointeur sur un entier */

/* Déclaration d'un type tableau de 10 entiers */
typedef int tableauentiers[10];
/* Déclaration d'un tableau de 10 entiers */
tableauentiers Tableau;

/* Déclaration d'un type pointeur sur un tableau de 10 entiers */
typedef int (*tableauIpointeurs)[10];
/* Déclaration d'un pointeur sur un tableau de entiers */
tableauIpointeurs TabPointeurs;
```

Ci-dessous un exemple de programme manipulant les pointeurs sur tableau d'entiers et les tableaux de pointeurs sur des entiers :

```
#include <stdio.h>

/* définition du type pointeur sur un tableau de 10 entiers (la définition
de ce type est utile pour le malloc) */
typedef int (*type_pointeur_sur_tableau)[10];

int main()
{
  /* déclaration d'un tableau de 10 pointeurs sur entier */
  int *tableau_de_pointeurs[10];
  /* déclaration d'un pointeur sur un tableau de 10 entiers */
  type_pointeur_sur_tableau pointeur_sur_tableau;

  int iCompteur=0;

  /* déclaration et initialisation d'un tableau de 10 entiers */
  int iTab[10]={1,5,8,49,5,6,7,3,-1,5};

  /* tableau_de_pointeurs est un tableau statique donc inutile d'allouer de
la memoire pour le tableau (c'est déjà fait!) mais il faut allouer de la
mémoire pour chaque pointeur du tableau */
  for(iCompteur=0; iCompteur<10;iCompteur++)
    tableau_de_pointeurs[iCompteur]=(int*)malloc(sizeof(int));

  /* Affectation d'une valeur à chaque élément du tableau */
   for(iCompteur=0; iCompteur<10;iCompteur++)
    *(tableau_de_pointeurs[iCompteur])=iCompteur;

   printf("Affichage des éléments du tableau de pointeurs\n");
   /* Affichage des éléments du tableau */
  for(iCompteur=0; iCompteur<10;iCompteur++)
    printf("le %d element du tableau a pour valeur : %d
\n",iCompteur,*(tableau_de_pointeurs[iCompteur]));
```

```
  /* Au lieu d'appeler malloc on peut donner à chaque pointeur l'adresse
d'un élément du tableau */
   for(iCompteur=0; iCompteur<10;iCompteur++)
     tableau_de_pointeurs[iCompteur]=&iTab[iCompteur];

  printf("\nAutre affichage des éléments du tableau de pointeurs\n");
   /* Affichage des éléments du tableau */
  for(iCompteur=0; iCompteur<10;iCompteur++)
    printf("le   %d   element   du   tableau   a   pour   valeur   :   %d
\n",iCompteur,*(tableau_de_pointeurs[iCompteur]));


  /* Pour allouer de la memoire au pointeur sur tableau */
  pointeur_sur_tableau=(type_pointeur_sur_tableau)malloc(sizeof(int[10]));

  /* Affectation de valeurs aux élements du tableau */
  for(iCompteur=0; iCompteur<10;iCompteur++)
    (*pointeur_sur_tableau)[iCompteur]=iCompteur;

  printf("\nAffichage des éléments du tableau d'entiers pointé\n");
  /* Affichage des éléments du tableau */
  for(iCompteur=0; iCompteur<10;iCompteur++)
    printf("le   %d   element   du   tableau   a   pour   valeur   :   %d
\n",iCompteur,(*pointeur_sur_tableau)[iCompteur]);


  /* Pour affecter au pointeur l'adresse d'un tableau de 10 entiers */
  pointeur_sur_tableau=&iTab;

   printf("\n Autre Affichage des éléments du tableau d'entiers pointé\n");
  /* Affichage des éléments du tableau */
  for(iCompteur=0; iCompteur<10;iCompteur++)
    printf("le   %d   element   du   tableau   a   pour   valeur   :   %d
\n",iCompteur,(*pointeur_sur_tableau)[iCompteur]);
}
```

B. Exemples de type structurés

```
/* Déclaration d'un type personne :   */
/* structure à deux champs nom et âge */
typedef struct pers /* pers est optionnel */
{
char* cNom;
int iAge;
} personne; /* personne est le nom du type */
/* ce n'est plus une variable comme précédemment */

/* Déclaration de variables de type personne */
personne individu;
personne* pindividu=NULL;
/* Déclaration de variables de même type en utilisant struct pers */
struct pers individu2;
```

Attention : une fonction peut retourner une structure, mais il s'agira d'une copie, ce qui peut entraîner des problèmes lorsqu'il y a des pointeurs dans la structure; une même adresse peut alors être pointées par différents pointeurs. Il est donc préférable d'utiliser des pointeurs.

Allocation mémoire :

```
/* Allocation mémoire de pindividu */
pindividu = (personne*)malloc(sizeof((struct pers));
```

Attention : on veut la taille de *struct pers*, et pas du pointeur.

La définition d'une structure peut être récursive :

```
/* Exemple de structure récursive */
typedef struct liste
{
 int iElement;
 struct liste *suivant;
} *liste;
/*les noms de la structure et du pointeur peuvent être les mêmes */
```

IX. MODELES DE DONNEES LISTE

A. Définitions et concepts généraux

Une **liste** est une séquence finie de zéro, un, ou plusieurs éléments d'un type donné [AU93]. Si les éléments sont de type *E*, on dit que le type de la liste est "liste de *E*". Il existe ainsi des listes d'entiers, des listes de nombres réels, des listes de structures, etc.

Une liste est souvent écrite comme une séquence d'éléments séparés par des virgules et est entourée par des parenthèses : $(a_1,a_2, ...a_n)$ où les a_i sont les éléments de la liste.

A.1 Longueur d'une liste

La longueur d'une liste est le nombre d'éléments (même les doublons) dans la liste. Si le nombre d'occurrences est zéro, on dit que la liste est *vide*.

EXEMPLES : $L = (a,b,t,y,a,u)$ de longueur 6.

$L = (1,2,3,6)$ de longueur 4

A.2 Parties d'une liste

Si la liste n'est pas vide, alors elle comprend un premier élément, appelé la **tête**.

Le reste de la liste est appelé **queue**.

Si $L=(a_1,a_2, ...a_n)$ est une liste, alors pour tous *i* et *j* tels que $1 \leq i \leq j \leq n$, $(a_i,a_{i+1}, ...a_j)$ est une **sous-liste** de *L*.

A.3 Position d'un élément

A chaque élément de la liste est associé une **position**. Si $(a_1,a_2, ...a_n)$ est une liste et $n \geq 1$, alors on dit que a_1 est le premier élément de la liste, a_2 le second, et ainsi de suite, a_n étant le dernier. On dit que a_i est de position *i*. De plus, on dit que a_i suit a_{i-1} et précède a_{i+1}.

Une position contenant l'élément a est une **occurrence de** a.

Le nombre de positions dans une liste est égal à sa longueur. Il est possible qu'un même élément apparaisse à plusieurs positions. **Il ne faut donc pas confondre position et élément de la liste**.

A.4 Opérations sur les listes

a) Insertion

Les éléments d'une liste sont ordonnés. On peut insérer un élément *x* à la position *i* d'une liste *L*. Cette action consiste à placer *x* dans la liste *L* à la $i^{ème}$ place, et de décaler les éléments de la liste à partir de la position *i* d'une position dans la liste (le $i^{ème}$ élément devient le $i^{ème}+1$, etc.).

EXEMPLE :

L=(2,3,4,2,6,7)

1 - Insertion de l'élément 7 à la position 3.
⇒ *L=(2,3,7,4,2,6,7)*

2 - Insertion de l'élément 1 à la position 1.
⇒ *L=(1,2,3,7,4,2,6,7)*

3 - Insertion de l'élément 7 à la fin de la liste.
⇒ *L=(1,2,3,7,4,2,6,7,7)*

b) Suppression

Il est possible de supprimer l'occurrence de position *i* d'une liste, qui signifie que l'élément de position *i* dans la liste va être enlevé de la liste. Il est également possible de supprimer un élément *x* de la liste. Cette opération consiste à supprimer toutes les occurrences *x* apparaissant dans la liste. Si *x* n'est pas dans la liste, la suppression est sans effet.

EXEMPLE :

L=(1,2,3,7,4,2,6,7,7)

1 - Suppression de l'occurrence de position 2.
⇒ *L=(1,3,7,4,2,6,7,7)*

2 - Suppression de l'élément 7.
⇒ *L=(1,3,4,2,6)*

3 - Suppression de l'élément 5.
⇒ *L=(1,3,4,2,6)*

c) Recherche

La recherche d'un élément *x* dans une liste est une opération qui retourne *VRAI* ou *FAUX* en fonction de la présence ou non de l'élément *x* dans la liste. Cette opération peut également retourner la position du premier élément d'occurrence *x* rencontré dans la liste, et 0 si l'élément n'existe pas dans la liste.

d) Concaténation

On concatène deux liste *L* et *M* en formant une liste commençant par les éléments de *L* et se poursuivant avec les éléments de *M*.

EXEMPLE :

L=(5,6,7,89) *M=(3,5,6,7,8,9,0,23,4)*

La liste résultat de la concaténation de L et M est :

(5,6,7,89,3,5,6,7,8,9,0,23,4)

On peut concaténer plus de deux listes.

e) Autres opérations[2]

- L'opération ***first*** retourne le premier élément de la liste. Le type retour de *first* est un élément de la liste.

  EXEMPLE : *L=(5,6,7,89)* *first(L)=5*

- L'opération ***last*** retourne la queue de la liste. Le type retour de *last* est une liste.

  EXEMPLE : *L=(5,6,7,89)* *last(L)=(6,7,89)*

[2] **Attention**: il ne s'agit pas de fonctions C, mais de conventions.

Ces deux opérations provoquent une erreur si la liste est vide.

- Il est possible de les combiner.

EXEMPLE : *L=(5,6,7,89)* *first(last(L))=6*

B. Implémentation de liste par des tableaux

Une manière courante d'implanter une liste en C, ou en tout autre langage, est d'utiliser un tableau pour stocker les éléments. Il est possible de maintenir un compteur du nombre d'élément de la liste dans une variable séparée, et de stocker les éléments dans des emplacements contigus du tableau. Attention, ce type d'implantation n'est pas pratique à utiliser pour insérer ou supprimer des éléments dans la liste. En effet, il faut à chaque fois décaler tous les éléments du tableau.

C. Liste chaînée

Une liste chaînée est une manière d'implanter une liste de telle sorte que l'on obtienne la structure suivante :

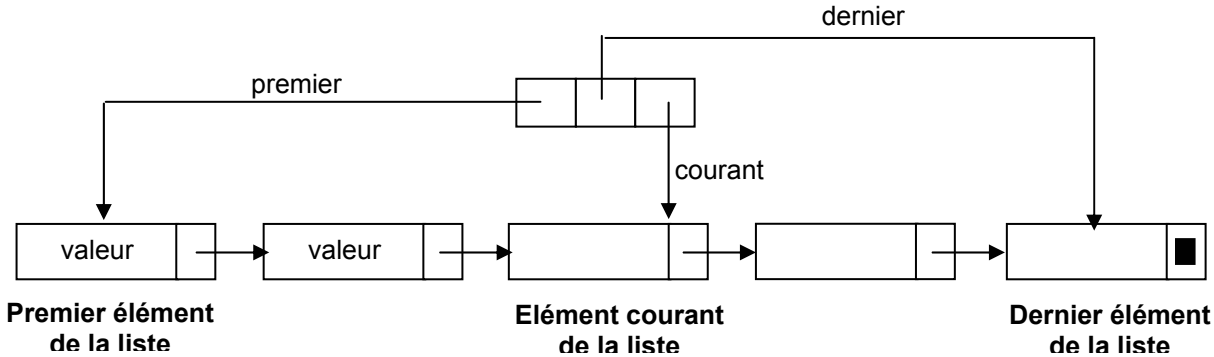

Figure IX.1 - Une liste chaînée.

L'**insertion d'un élément dans une liste chaînée**, peut se faire après l'élément courant de la liste.

Trois cas se présentent alors :

1. **La liste est vide**. Dans ce cas, les pointeurs *premier*, *courant* et *dernier* pointent, après insertion de l'élément *e,* sur *e* (voir Figure IX.2).

2. **La liste est non vide, et l'élément courant est un élément interne à la liste** (*courant* ne pointe pas sur le dernier élément de la liste). Après insertion de l'élément *e*, l'élément suivant de *e* sera le *suivant* de l'élément courant (pointé par *courant)*, l'élément suivant de l'élément courant sera *e,* et l'élément courant sera *e* (voir Figure IX.3).

3. **La liste est non vide, et l'élément courant est le dernier élément**. Après insertion de l'élément *e*, le *suivant* du dernier élément sera *e*, l'élément suivant de *e* sera nul, et le dernier élément, ainsi que l'élément courant seront *e* (voir Figure IX.4).

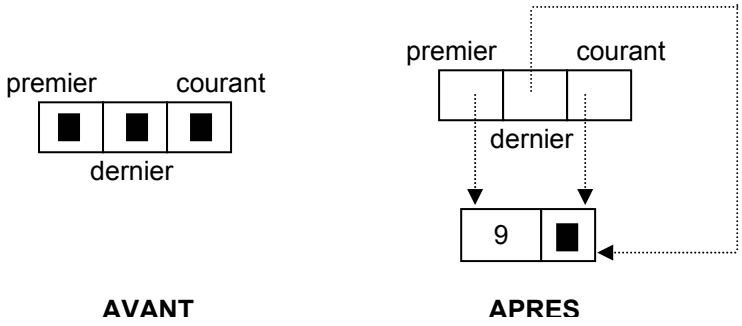

Figure IX.2 - Insertion d'un élément en position courante dans une liste vide.

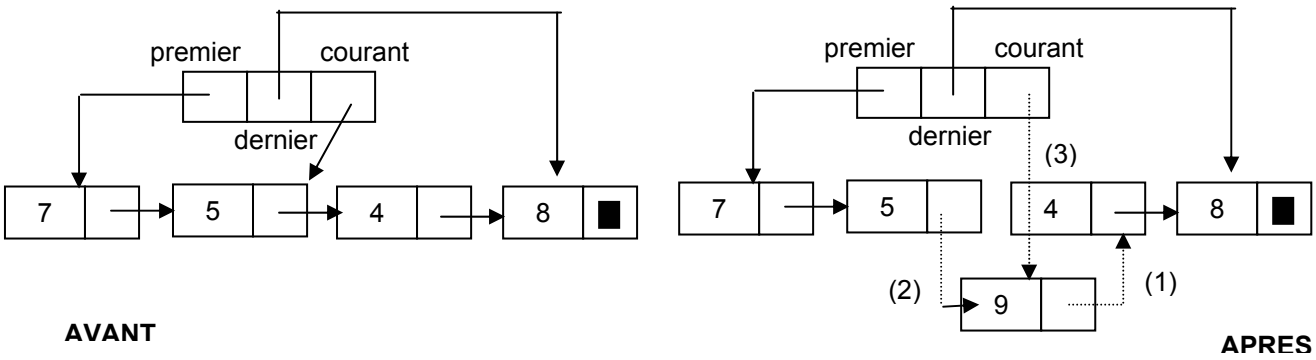

Figure IX.3 - Insertion d'un élément en position courante dans une liste non vide.

(1) Le pointeur *suivant* de l'élément ajouté pointe sur l'élément suivant de l'élément *courant*.

(2) Le *suivant* de l'élément courant pointe sur l'élément ajouté.

(3) L'élément courant devient l'élément ajouté.

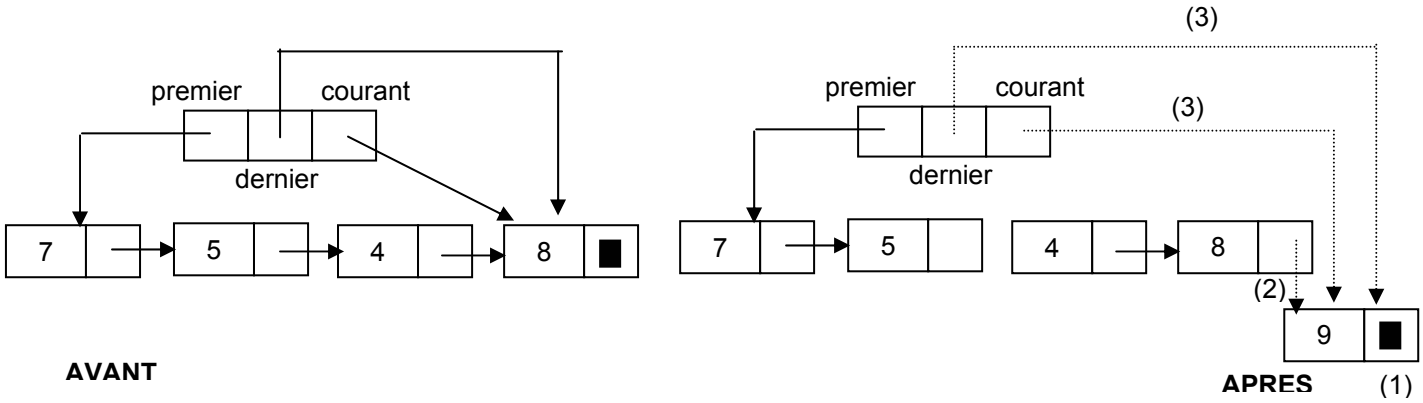

Figure IX.4 - Insertion d'un élément en dernière position dans une liste non vide.

(1) Le pointeur *suivant* de l'élément ajouté est nul.

(2) Le pointeur *suivant* du dernier élément de la liste pointe sur l'élément ajouté.

(3) Les pointeurs *dernier* et *courant* pointent sur l'élément ajouté.

NB : L'ordre des opérations ci-dessus a son importance. Par exemple, dans le cas de la Figure IX.3, si on change le pointeur *courant* dés le début, on ne peut plus insérer l'élément.

A voir : un exemple d'implantation de liste chaînée est donné dans le polycopié des exercices corrigés.

X. PILE

A. Définition générale

La notion de pile intervient couramment en programmation, son rôle principal consiste à implanter les appels de procédures [CL97]. Une **pile** est un type de données abstrait basé sur le modèle de données liste, dans lequel les opérations sont réalisées à une extrémité de la liste, appelé **sommet de la pile** [AU93]. Le terme **LIFO** (*Last-In-First-out*) est synonyme de pile.

Une pile peut être représentée sous forme d'un tableau :

3 ← sommet de la pile
4
5
6
7
8

Figure X.1 - Un exemple de pile.

B. Opérations sur les piles

Les opérations sur les piles sont les suivantes :

- **sommet** : retourne le sommet de la pile.
- **empiler** : ajoute un élément au sommet de la pile.

Tableau X-1 - résultat de l'opération *empiler 9*

pile :	empiler 9 :
3	9
4	3
5	4
6	5
7	6
8	7
	8

- **dépiler** : enlève le sommet de la pile.

Tableau X-2 - Résultats successifs de l'opération dépiler

pile :														
3														
4	4													
5	5			5										
6	6			6			6							
7	7			7			7			7				
8	8	3		8	3 4		8	3 4 5		8	3 4 5 6		8	3 4 5 6 7	3 4 5 6 7 8

C. Implémentation d'une pile par une liste chaînée

La figure qui suit représente une implémentation de pile en liste chaînée.

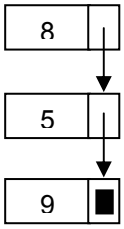

Figure X.2 - Un exemple de pile sous forme de liste chaînée.

D. Pile d'exécution de la mémoire des ordinateurs

L'organisation de la mémoire de l'ordinateur à l'exécution est représentée par la Figure X.3. Lorsqu'un programme est exécuté, les informations telles que les arguments des fonctions, leur valeur de retour, et leurs variables locales, sont stockées dans la pile d'exécution.

Par exemple, supposons qu'un programme principal fasse appel à la fonction *int multiplication(int arg1, int arg2).* Dés l'appel de la fonction *multiplication* dans le programme principal, les deux arguments *arg1* et *arg2* de la fonction sont stockés dans la pile d'exécution. Puis ***l'adresse de retour***

vers le programme principal, c'est-à-dire l'adresse de l'instruction suivant l'appel de la fonction *multiplication* dans le programme principal, est à son tour stockée dans la pile. Cette ***adresse de retour*** permet la reprise de l'exécution du programme principal après l'exécution de la fonction *multiplication*. Finalement, dès l'exécution de *multiplication*, les variables locales de la fonction sont stockées dans la pile.

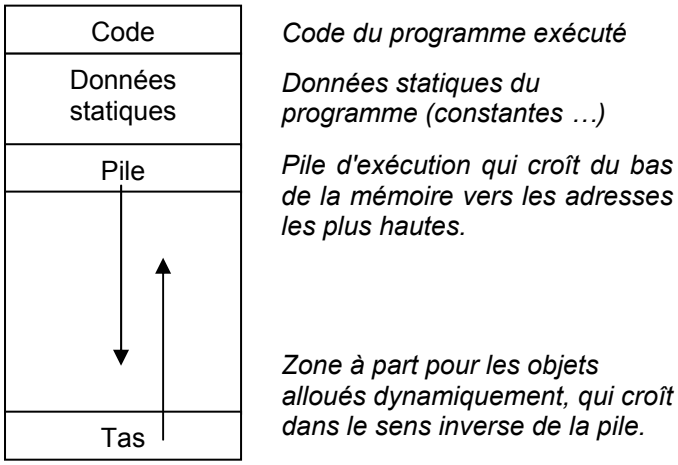

Figure X.3 - L'organisation de la mémoire d'un ordinateur à l'exécution d'un programme [AU93].

Prenons l'exemple d'un programme :

```
/* Fonction Q */
void Q()
{ /* Déclaration des variables locales à Q */
  int p,q,r;
  .....
}
void P(int u,int v)
{ /* Déclaration des variables locales à P */
  int a,b;
...
  /* Appel de la fonction Q */
  Q();
}
void main()
{ /* Déclaration des variables locales au programme principal  */
  int x,y,z;
/* Appel de la fonction P */
  P(2,8);
...
}
```

Pile d'exécution lorsque la fonction Q est en cours d'exécution :

```
x | y |z |adresse_retour | u=2 | v=8 | a| b | adresse_retour | p |q |r
```

Pour plus de détails sur la pile d'exécution, vous êtes invité à consulter le [Tan91].

Exercice : Donnez la pile d'exécution de la fonction *fact* qui calcul le factoriel de l'argument (voir annexe XVIII.B).

La correction vous sera donnée en cours.

A voir *:* un exemple d'implantation de pile par une liste chaînée est donné dans le polycopié des exercices corrigés.

XI. FILES

Un autre type de données abstrait basé sur le modèle de donnée liste est la **file** [AU93]. Il s'agit d'une forme restreinte de liste dans laquelle les éléments sont insérés à une extrémité, la **queue**, et retirés à une autre extrémité, la **tête**. Le terme ***FIFO*** (*First-in-first-out*) est un synonyme de file. Une file d'attente à la caisse d'un magasin illustre bien cette notion de file. Les files sont utilisées en programmation pour gérer des objets qui sont en attente d'un traitement ultérieur, par exemple des processus en attente d'une ressource du système, des sommets d'un graphe, des nombres entiers en cours d'examen de certaines de leurs propriétés, etc. Dans une file, les éléments sont systématiquement ajoutés en queue et supprimés en tête. La valeur d'une file est par convention celle de l'élément de tête [CL97].

On peut représenter une file par une zone continue en anneau (voir Figure XI.1). Si on ajoute un élément, on déplace la queue de 1. Si on enlève un élément, on déplace la tête de 1.

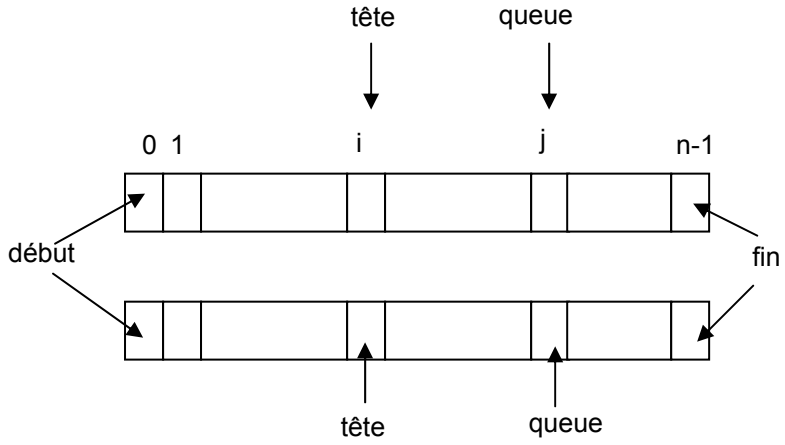

Figure XI.1 - Une file représentée en anneau continu.

A voir *:* un exemple d'implantation de file est donné dans le polycopié des exercices corrigés.

XII. GESTION DES FICHIERS

On peut manipuler des fichiers, grâce aux pointeurs vers les structures fichiers, définies dans <*stdio.h*> : ***FILE****

```
/* DECLARATION D'UN FICHIER */
FILE* fd; /* fd est un descripteur de fichier */
         /*(pointeur sur un fichier)        */
```

Il existe plusieurs fonctions permettant de manipuler les fichiers :

A. Ouverture de fichier

```
FILE* fopen (char* nom, char* mode)
```

Attention : *fopen* retourne *NULL* si la tentative d'ouverture a échoué. Il faut également que l'utilisateur courant ait les droits sur ce fichier (cf. polycopié d'architecture).

mode =

- "**r**" Ouverture du fichier en lecture seule.
- "**w**" Ouverture du fichier en écriture, écrase le contenu précédent si le fichier existait.
- "**a**" Ajout, ouvre un fichier texte et se positionne en écriture à la fin du fichier.
- "**r+**" Ouvre un fichier en mode mise à jour (lecture - écriture)

- "**w+**" Crée un fichier en mode mise à jour, écrase le contenu précédent si le fichier existait.
- "**a+**" Ajout, ouvre un fichier en mode mise à jour, et se positionne à la fin du fichier.

En réalité, *FILE* est un type de structure défini dans le fichier *<stdio.h>.* La déclaration *FILE* fichier;* déclare un pointeur sur un fichier. Mais elle ne réserve qu'un emplacement pour un pointeur [Del97]. C'est la fonction *fopen* qui créera effectivement une telle structure et qui en fournira l'adresse en résultat.

EXEMPLE DE PROGRAMME :

```
/* Inclusion de la biblio standard */
#include <stdio.h>

int main()
{
/* Déclaration d'un fichier */
FILE* fichier;

/* Ouverture du fichier /home/users/manouvrier/toto en lecture */
fichier = fopen("/home/users/manouvrier/toto","r");
/* Test de l'ouverture pour vérifier les éventuels problèmes */
if (fichier== NULL) /* Il y a eu un problème */
  {
    /* Affichage d'un message d'erreur */
    printf("Erreur ouverture fichier ");

  }

/* Si l'ouverture a pu se faire */
else ...
}
```

B. Fermeture de fichier

```
int fclose (FILE* stream)
```

EXEMPLE :

```
/* fermeture du fichier /home/users/manouvrier/toto */
fclose(fd);
```

C. Lecture du fichier

- `int fgetc (FILE* stream)`
  `/* Cette fonction lit un caractère (unsigned char converti en int). Le caractère est retourné ou EOF (fin de fichier) est retourné si une erreur survient. */`

EXEMPLE :

```
#include<stdio.h>
FILE *fichier;
int ch;
....
while(ch = (fgetc(fichier)!=EOF)) ...
```

- `char *fgets(char* s, int n, FILE* stream)`
  `/* Cette fonction lit au plus n-1 caractères et les place dans s. s est retournée ou NULL si la tentative a échoué. */`

EXEMPLE :
```
#include<stdio.h>
FILE *fichier;
char ligne[80], *zone;
zone = fgets(ligne,80,fichier);
```

D. Ecriture dans un fichier

- `int fputc (int c, FILE* stream)`
  `/* cette fonction écrit le caractère c (converti en unsigned char). le caractère écrit est retourné, EOF sinon */`
- `int fputs (const char* s, FILE* stream)`
  `/* cette fonction écrit la chaîne s dans le fichier. Elle retourne une valeur positive ou nulle, EOF s'il y a eu une erreur */`

E. Positionnement du curseur dans un fichier

`int fseek (FILE* stream, long offset, int origin)`
`/* positionne le pointeur de fichier. Une lecture ou écriture de fichier commencera à la nouvelle position. La position est fixée à offset caractères de origin.*/`

Valeurs possibles de *origin* :

SEEK_SET (début de fichier)

SEEK_CUR (position courante)

SEEK_END (fin de fichier).

EXEMPLE :
```
#include <stdio.h>
FILE* fichier;
int rc;
...
/* positionnement du pointeur au début du fichier */
rc = fseek(fichier,0,SEEK_SET);
```

F. Récupération de la position du curseur dans le fichier

```
int ftell (FILE *stream)
/* Récupération de la position du pointeur dans le fichier */
```

EXEMPLE :
```
#include <stdio.h>
FILE* fichier;
int pos;
...
pos=ftell(fichier);
```

G. Ecriture et lecture de zones dans un fichier

- *size_t fwrite(void* tab, size_t taille, size_t nb, FILE *fd)* : cette fonction écrit *nb* blocs de *taille* octets dans le fichier de descripteur *fd*, les bocs étant stockés dans le tableau *tab*. La fonction retourne le nombre de blocs effectivement écrits (si tout a fonctionné *nb*) ou 0 en cas d'erreur [CH00]
- *size_t fread(void* tab, size_t taille, size_t nb, FILE *fd)* : cette fonction lit jusqu'à *nb* blocs de *taille* octets dans le fichier de descripteur *fd* et les place dans le tableau *tab*. Elle retourne le nombre de blocs effectivement lus (inférieur à *nb* si la fin de fichier est atteinte) ou 0 en cas d'erreur [CH00].

EXEMPLE :

```
#include <stdio.h>
FILE* monfichier;
char* ptr;
int nbrez;
int nbrelu;
long zoneslu[50];
...
/* Ecriture de 10 éléments de taille sizeof(ptr), dont le premier  */
/* est pointé par ptr, dans le fichier nommé monfichier.           */
/* Le nombre d'élément écrits est renvoyé.                         */
nbrez = fwrite(ptr,sizeof(ptr),10,monfichier);
/* Lecture dans monfichier de 50 éléments de taille sizeof(long)   */
/* dont le premier élément est pointé par ptr                      */
/* Le nombre d'éléments lus est renvoyé par la fonction.           */
nbrelu = fread(zoneslu,sizeof(long),50,monfichier);
```

H. Fichiers prédéfinis

Il existe en C un certain nombre de fichiers prédéfinis qu'il n'est pas nécessaire d'ouvrir ni de fermer : ***stdin*** (unité d'entrée, par défaut le clavier), ***stdout*** (unité de sortie, par défaut l'écran), ***stderr*** (unité d'affichage des message d'erreurs, par défaut l'écran).

XIII. ARGUMENTS DE LA LIGNE DE COMMANDE

Il existe un moyen pour transmettre des arguments de la ligne de commande ou des paramètres à un programme, au début de son exécution [KR90].

Quand on appelle *main*, deux arguments lui sont passés. Le premier (baptisé conventionnellement *argc* signifiant nombre d'arguments - *argument count*) représente le nombre d'arguments de la ligne de commande qui a appelé le programme. Le second (*argv*, signifiant vecteur d'arguments - *argument vector*) est un pointeur sur un tableau de chaînes de caractères qui contiennent les arguments, à raison de un par chaîne.

Par convention *argv[0]* est le nom par lequel le programme a été appelé, et par conséquent, *argc* vaut au moins 1 [KR90]. Si *argc* est supérieur strictement à 1, cela signifie que le programme à *argc-1* arguments. La norme spécifie également que *argv[argc]* doit être un pointeur nul.

EXEMPLE :

```
#include <stdio.h>
/*************************************************************/
/* Programme écho : affiche les caractères saisis au clavier   */
/* Paramètres d'entrée :                                       */
/*                       int argc : nombre d'arguments         */
/*                       char *argv[]: tableau d'arguments    */
/*************************************************************/

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

{
  /* Déclaration d'un compteur */

  int iCompteur;

  /* Pour tous les arguments saisis après le nom du programme       */
  /* Le compteur commence à 1, argv[0] contenant le nom du programme */

  for(iCompteur=1;iCompteur<argc;iCompteur++)

     /* Affichage de l'argument s'il y a encore des arguments après */

     /* on affiche un espace à la suite, sinon on affiche rien      */

     printf("%s%s", argv[iCompteur], (iCompteur<argc-1)? " " : "");

  printf("\n");

}
```

XIV. CREATION D'UN PROGRAMME EN LANGAGE C

La manière de développer et d'utiliser un programme en langage C dépend naturellement de l'environnement de programmation dans lequel vous travaillez. De manière générale, un programme C est composé de différents modules (ensembles de fonctions) cohérents, c'est-à-dire programmés pour une même action. Par exemple, si votre programme doit montrer un exemple d'utilisation de liste chaînée, vous pouvez le diviser en deux modules : l'un comportant les types et fonctions classiques des listes chaînées, l'autre contenant le programme principal et éventuellement d'autres fonctions donnant un exemple de manipulation de liste chaînée.

Un certain nombre de fichiers d'extension .h correspondent à des classes de fonctions. Par exemple le fichier *stdio.h* pour les fonctions standards, *math.h* pour les fonctions mathématiques, etc. Ces fichiers, dits fichiers "en-tête", contiennent les déclarations relatives aux fonctions prédéfinies, ainsi que la définition des macros. Ainsi, pour pouvoir utiliser des fonctions que vous avez définies dans plusieurs programmes, vous devez créer un fichier .h contenant la signature de ces fonctions, et l'inclure (par *#include* voir section XVI.A.1) dans les différents programmes où vous utilisez ces fonctions.

XV. REGLES DE LA PROGRAMMATION

Voici quelques règles indispensables de programmation [AU93], à suivre impérativement :

- Découper le programme en **modules** (groupes de fonctions répondant à un même problème) cohérents.
- Organiser le programme de manière que sa structure soit claire.
- Ecrire des **commentaires** intelligents pour expliquer le programme.

  Décrire clairement les modèles de données sous-jacents, les structures de données sélectionnées pour les représenter, et l'opération accomplie par chaque procédure. Lors de l'écriture d'une procédure, établir les hypothèses faites sur ses entrées, dire comment sa sortie se rapporte à l'entrée, etc.

- Utiliser des noms ayant un sens pour les procédures et les variables.

  Par exemple, au lieu d'écrire *int i* (la variable *i* est de type entier), écrire *int iCompteur* qui signifie que la variable *iCompteur* est de type entier (car elle est préfixée de *i* pour *integer*) et qu'il s'agit d'un compteur.

- Lorsque c'est possible, éviter les constantes explicites.

  Par exemple, ne pas utiliser 7 pour le nombre des nains. A la place, il est préférable de **définir une constante** comme *NombredeNains*, de façon à ce que l'on puisse facilement modifier toutes les utilisations de cette constante (ex. modification de la constante en 8).

- Eviter les **variables globales** (variables définies pour tout le programme), sauf si les données de ces variables sont effectivement utilisées par la plupart des procédures du programme. De préférence, utilisez des variables d'environnements.

XVI. COMPILATION ET EXECUTION D'UN PROGRAMME C

Un programme en C est constitué par [Pon97] :

- des directives pour le **préprocesseur**,
- l'entête du programme principal,
- le bloc du programme principal
- et pour chaque fonction son entête et son bloc.

La **compilation** d'un programme C consiste à traduire le code source en langage machine. Cette opération comporte deux étapes [Del97] :

1. Le **traitement du préprocesseur** : qui exécute les directives qui le concernent (voir section XVI.A). Il reconnaît ces directives par le caractère #.
2. Puis la **compilation** proprement dite, qui traduit le texte en langage C fourni par le préprocesseur en langage machine.

Le résultat de la compilation porte le nom de **module objet**. Ce module n'est pas directement exécutable [Del97]. Il lui manque notamment les différents modules objet correspondant aux fonctions prédéfinies (ou standards) utilisées par le programme. C'est le rôle de **l'éditeur de liens** que d'aller rechercher dans la bibliothèque standard les modules objet nécessaires. La bibliothèque est en effet une collection de modules objets organisée, suivant l'implémentation concernée, en un ou plusieurs fichiers. L'éditeur de liens vérifie que chaque objet utilisé est défini une et une seule fois dans l'ensemble des modules et, fait la liaison entre cet objet et les objets utilisés dans d'autres modules [Rif93]. Le résultat de l'édition des liens est le **programme exécutable**. Il s'agit d'un ensemble autonome d'instructions en langage machine.

A. Directives du préprocesseur.

Le préprocesseur est un programme qui est exécuté avant la compilation proprement dite. Il prépare la compilation en tenant compte de directives, ces directives sont écrites à raison de une par ligne, elles commencent toutes par #.

A.1 Directive #include*.*

Elle permet d'incorporer, avant compilation, le texte figurant dans un fichier quelconque. Il y a deux syntaxes possibles :

```
#include <Nom_de_Fichier> ou  #include "Nom_de_Fichier"
```

La première forme recherche le fichier dans un répertoire défini par l'implémentation (variable d'environnement). La deuxième recherche le fichier mentionné dans le même répertoire que celui où se trouve le fichier source.

En général, la première forme est utilisé pour faire appel aux bibliothèques standards. La deuxième forme est plutôt utilisée pour les fichiers que vous créez vous-mêmes.

Attention : Un fichier incorporé par *#include* peut à son tour comporter des directives *#include*, c'est le cas dans certains fichiers en-têtes relatifs à la bibliothèque standard. De plus lorsque vous créez vous-mêmes vos fichiers en-têtes, il faut prendre garde à ne pas introduire plusieurs fois des déclarations identiques [Del97]. Ce genre de problème se résout par l'emploi de directives conditionnelles associé à la définition de symboles particuliers (voir section A.4)

A.2 Directive #define.

```
#define [<classe>] <type> <identificateur>
```

La *classe* représente la durée de vie et la visibilité de la constante :

- *auto* : de durée de vie et de visibilité dans le bloc où elle a été déclarée uniquement.
- *static* : durée de vie dans le programme, mais visibilité dans le bloc de sa déclaration.
- *extern* : constante définie à l'extérieur (dans une librairie par exemple).

La directive *#define* permet soit la définition de **symboles**, soit la définition de **macros**.

Un symbole est en fait un texte qui sera remplacé dans le code par sa définition.

EXEMPLE : `#define toto 10`

Un symbole est une macro sans paramètre.

Attention : un *#define toto 10* ne déclare pas une constante. Le préprocesseur remplace les occurrences de *toto* dans le programme par *10*, avant la compilation.

EXEMPLE DE DEFINITION DE MACRO :

```
#define carre(x) x * x
```

Si le code contient "carre(5)" cette partie est remplacée par "5 * 5".
Si le code contient "carre(a)" cette partie est remplacée par "a * a".
Si le code contient "carre(y)" cette partie est remplacée par "y * y".
Si le code contient "carre(2 + b)" cette partie est remplacée par "2 + b * 2 + b".

Dans le dernier cas on peut remarquer que ce genre de substitution peut conduire à des erreurs [Pon97]. Il faudrait donc mieux définir *#define carre(x) (x)*(x)*.

A.3 Directives #ifdef, #else et #endif.

Ces directives permettent une compilation conditionnée par l'existence d'un symbole [Pon97].

```
#ifdef toto
  instruction_toto1;
  ....
  instruction_toto2;
#else
  instruction_non_toto1;
  ....
  instruction_non_toto_n;
#endif
```

La directive *#else* peut être absente.

EXEMPLE [DEL97]:

#define MISEAUPOINT

```
      …
#ifdef MISEAUPOINT
      instructions 1
#else
      instructions 2
#endif
```

Ici, les instructions 1 seront incorporées par le préprocesseur, tandis que les instructions 2 ne le seront pas.

A.4 Directives #ifndef, #else et #endif.

Ces directives permettent une compilation conditionner par l'absence d'un symbole.

```
#ifndef toto
    instruction_non_toto;
    ....
    instruction_non_toto;
#else
    instruction_toto;
    ....
    instruction_toto;
#endif
```

La directive *#else* peut être absente.

EXEMPLE :directive permettant de ne pas inclure plusieurs fois le même fichier en-tête.

```
#ifndef _MONFICHIER_EnTETE
      #define _MONFICHIER_EnTETE
#endif
```

Conseil : Ce type de symboles permet d'inclure dans un fichier en-tête un autre fichier en-tête, en s'assurant que ce dernier n'a pas déjà été inclus (afin d'éviter la duplication de certaines instructions risquant de conduire à des erreurs de compilation). Vous pouvez donc utiliser cette technique dans vos propres fichiers en-tête.

A.5 Directives #if, #elif, #else et #endif.

Ces directives permettent une compilation conditionnée par la valeur d'une expression (théoriquement booléenne) ne contenant que des symboles et des constantes. En effet, le préprocesseur ne connaît pas les variables puisqu'il agit avant la compilation.

La directive *#else* peut être absente. Dans le cas de structures de choix on peut remplacer le *#else* par *#elif condition*. Dans les expressions derrière un *#if* ou un *#elif* on peut utiliser l'opérateur *defined*. L'opérateur *defined* a un paramètre qui est un symbole, il rend vrai si le symbole est défini et faux sinon [Pon97]. Voir [KR90] pour plus de détails.

EXEMPLE :

```
#define max 10
....
#if max > 5
    instruction_grand;
    ....
    instruction_grand;
#else
    instruction_petit;
    ....
    instruction_petit;
#endif
```

B. Commandes de compilation et d'exécution sous UNIX

Un programme écrit en C doit être sauvegardé dans un fichier à l'extension *.c* (Ex. *programme.c*) [KR90]. Il se présente sous la forme d'une collection d'objets externes (fonctions et variables) dont la définition est éventuellement donnée dans des fichiers séparés [Rif93].

1. Un programme est compilé à l'aide de la commande ***cc*** (petit cc).

   `cc programme.c`

2. Si le fichier n'a pas d'erreur, le compilateur va créer un programme exécutable **a.out**.

3. Pour exécuter le programme, il suffit alors de taper **a.out**.

4. Si on veut donner un nom au programme exécutable : *cc -o programme programme.c*

5. Pour exécuter le programme, il suffit alors de lancer *programme*

La commande de compilation peut être plus complexe. Par exemple, `cc -o prog prog.c f1.c prog1.o` qui correspond à la demande de construction d'un binaire exécutable de nom *prog* à partir des fichiers sources de noms *prog.c* et *f1.c* et d'un module objet (et donc déjà compilé) de nom *prog1.o*.

Lorsque votre programme est composé de plusieurs fichiers, le mieux est d'utiliser **un utilitaire de gestion de dépendances** : ***make***. Cet utilitaire permet de maintenir un programme obtenu à partir d'un ensemble de modules distincts en prenant en compte les dates de modification du programme à maintenir et des différents modules entrant dans la construction de ce programme [Rif93]. Il faut pour cela utiliser un fichier particulier : ***makefile***. Ce fichier contient la description des liens entre les différents modules, et les règles de construction de ce programme.

EXEMPLE : Les lignes qui suivent sont un exemple de fichier ***makefile***.

```
obj = p1.o p2.o p3.o
appli : $(obj)
cc -o appli &(obj)
p1.o: p1.c p.h q.h
cc -c p1.c
p2.o: p2.c
cc -c p2.c
p3.o: p3.c p.h q.h r.h
cc -c p3.c
```

La première ligne est la macro-définition de l'identificateur *obj.* La seconde ligne indique que le fichier *appli* dépend des fichiers *p1.o, p2.o* et *p3.o*. Par conséquent, si un des trois fichiers est plus récent que lui (a été modifié), la commande *cc -c appli p1.o p2.o p3.o* doit être exécutée.

Les lignes qui suivent sont construites sur le même schéma. Elles expliquent comment créer les fichiers *p1.o, p2.o* et *p3.o*. La commande *cc -c* compile sans faire appel à l'éditeur de liens, ce dernier effaçant le fichier à extension *.o* pour construire l'exécutable.

Pour lancer la gestion des dépendances, la commande est : *make* ref_make_fic où *ref_make_fic* est le fichier de description des dépendances. Si ce fichier n'est pas désigné , par défaut un fichier de nom *makefile* ou *Makefile* dans le répertoire courant sera recherché.

Pour plus de précision sur les outils de développement sous UNIX et les différents compilateur existants (cc, gcc, etc.), vous pouvez consulter le chapitre 9 de [Rif93]. Vous pouvez également utiliser le *man* sous UNIX ou l'aide en ligne de votre compilateur.

XVII. REFERENCES

[AU93] Alfred AHO et Jeffrey ULLMAN *Concepts fondamentaux de l'informatique* Dunod, 1993. ISBN : 2-10-00116830

[CH00] **Jean-Marc Champarnaud et Georges Hansel. *PASSEPORT POUR UNIX ET C*, Vuibert Informatique, 2000.** ISBN : 2-7117-**8663-3**

[CL97] Robert CORI et Jean-Jacques LEVY *Algorithmes et Programmation* Ecole Polytechnique

[Del97] Claude Delannoy Programmer en langage C - Eyrolles - 1997

ISBN :2-212-08985-6

[KR90] B.W. KERNIGHAN et D.M. RICHIE *Le langage C* Masson - Prentice Hall, 1990.

ISBN : 2-225-82070-8

[Pon97] Philippe PON *Résumé du cours "Introduction au langage C". 1997*

[Rif93] JM. Rifflet *La programmation sous UNIX* - EdiScience - 1993
ISBN : 2-84074-013-3

[Tan91] Andrew Tanenbaum *Architecture de l'ordinateur* InterEditions 1991.

En plus des ouvrages précédemment cités, vous pouvez consulter deux ouvrages très bien faits :

- *Maîtrise des algorithmes en C*, de Kyle Loudon, O'Reilly (Classique) ; ISBN : 2841770966, 2000

  Il s'agit d'un ouvrage très complet sur l'algorithmique, comportant beaucoup d'exemples en C

- *C: How to Program*, de Harvey M. Deitel, P.J. Deitel, Prentice Hall ; ISBN : 0130895725, 2000

  Il s'agit d'un ouvrage qui vous servira tout votre IUP puisqu'il décrit trois langages, le C, Java et C++

XVIII. ANNEXE

A. Fonction *atoi*

Le code de la fonction *atoi* donné ici provient de [CL97]. Cette fonction permet de convertir une chaîne de caractères en entier. Par exemple, *atoi("12345")* retournera l'entier 12345

```
int atoi (char s[])
{
      int i, n;
      n = 0;
      for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
            n = 10 * n + (s[i] - '0');
      return n;
}
```

Commentaires de la fonction :

Cette fonction (voir section IV pour plus de détails sur les fonctions) prend en paramètre la chaîne de caractère à convertir (*s*) et retourne sa valeur en entier (*int*). Puis, deux variables locales *i* et *n* sont déclarées. La variable *n* est ensuite initialisée à 0. Une boucle *for* (voir section II.E.4) permet pour chaque caractère de *s*, si ce caractère est un entier (test *s[i] >= '0' && s[i] <= '9'*, qui retourne vrai si le caractère est '0', '1', ou .. '9', et faux sinon), de calculer sa valeur entière correspondante. *s[i] - '0'* permet de calculer l'entier qui représente la différence dans le code ASCII entre *s[i]* et '0'.

Remarque : cette fonction existe dans la bibliothèque standard du C.

B. Fonction factorielle récursive

```
/* Fonction récursive factorielle                                    */
/* Paramètres d'entrée :                                             */
/*                     int n : valeur dont on va calculer la factorielle */
/* Paramètres d'entrée : factorielle de l'argument
*/
int fact(int n)
{
    /* Si n est plus petit ou égal à 1 */
    if(n<=1)
    { /* On retourne 1 */
       return 1;
    }
    /* Si n strictement plus grand que 1 */
    else
    { /* On retourne n * le factoriel de n-1 */
       return (n * fact(n-1));
    }
}
```

Index

#

A

B

C

D

E

F

G

I

L

M

N

O

P