Cours Langage C - Pierre Jacoboni 16 février 1998

Avertissement :

Ce polycopié sur le Langage C ne constitue qu'un support de cours de langage de programmation. Il est incomplet et nécessite de nombreux compléments donnés en cours magistral. C'est la raison pour laquelle il est donné aux étudiants sous une forme plus «aérée», permettant la prise de notes. Il s'adresse à des étudiants de Deup.

Certains concepts sont volontairement absents de ce texte, comme la compilation séparée, etc... Ces concepts sont développés en TD/TP en cours d'année.

Sommaire :

- La conception des langages impératifs à été étroitement liée à la conception des ordinateurs traditionnels qui repose sur l'architecture de Von Neumann (La mémoire possède des emplacements adressables de façon individuelle)
- Qu'est-ce qu'une variable ? on peut voir la variable des langages de programmation impératifs comme une abstraction de l'emplacement en

- L'exécution d'un programme dans un langage impératif s'effectue en exécutant séquentiellement des instructions. Les instructions conditionnelles et itératives servant à modifier l'ordre d'exécution des instructions.

1.1.3. Les Différences entre langages Impératifs et Applicatifs

- La plus importante différence provient du concept d'affectation : il est constitutif des langages impératifs alors qu'il n'existe pas dans un langage purement fonctionnel.
- L'application d'une fonction a pour effet de renvoyer une nouvelle valeur , alors que l'affectation provoque une modification de la valeur d'un objet existant.
- Le problème de l'affectation est que, quand elle est utilisée dans des sous-programmes avec des paramètres passés par référence ou avec des variables non locales, elle peut induire des effets secondaires ou effets de bords.
- Exemple : Soit la fonction Ada suivante :

```
function bizarre return  integer is
begin
        Var_Glob = Var_Glob + 1 ;
        return Var_Glob ;
end bizarre ;
```

  Des appels différents de *bizarre* renvoient des résultats différents, ainsi l'expression relationnelle *bizarre = bizarre* est fausse ! ! !
- Avec de tels effets secondaires, il devient difficile de raisonner sur les programmes pour prouver qu'ils sont corrects.
- En excluant l'affectation, il est possible de s'assurer que les fonctions ne provoquent pas d'effet secondaire et qu'elles se comportent comme des

  - L'ordre précis dans lequel les fonctions sont appelées n'a alors pas ; appeler une fonction avec un certain paramètre donnera

comment » (comment ce doit être fait). Ils se situent à un niveau plus élevé

1.2. Langage C

1.2.1. Historique

- ⇨ **Naissance** : 1970
- ⇨ **Père** : Dennis RITCHIE (Bell AT&T) sur PDP 11
- ⇨ **Objectif** : Réécriture d'UNIX (Ken THOMPSON) écrit jusqu'alors en assembleur et langage B (langage non typé).
- ⇨ **Conséquences** : Le succès d'Unix, notamment dans les universités américaine a fortement contribué à la popularité du langage C.

Aujourd'hui C est le langage de programmation sous UNIX

- ⇨ **Réputation** : C est un langage d'assemblage portable de haut niveau.
- ⇨ **Attention** : Ce qui précède dépend du programmeur et du type d'application que l'on réalise avec C.
- ⇨ **Conclusion** : C est un langage évolué permissif, capable de traiter de manière standard des informations de bas niveau très proches de la structure matérielle des ordinateurs (mots, adresses, registres, octets, bits ...)
- ⇨ **Définition du langage C :**
  - Kernighan & Ritchie : le langage C - 1978
  - Norme Ansi - 1982
- ⇨ **Bibliographie :**
  - **Le langage C norme Ansi (2ème édition)** Kernighan & Ritchie (masson)
  - **Langage C norme ansi *objet"*** Drix(masson)
  - **Programmation en C** Gottfried (Schaum)
  - **Le Langage C++ (2ième édition)** Stroustrup (Addison Wesley)

- ⇨ Cet argument constitue le format d'affichage, et peut contenir entre autre des controles comme \n : newline, \t : tabulation, \b : backspace.
- ⇨ En C, il faut gérer soi-même ses changements de lignes.

  printf("Bonjour");
  printf("tout le monde");

  donnera : ***Bonjourtout le monde*** comme résultat.

  printf("Bonjour\n");
  printf("tout le monde");

  est équivalent à

  printf("Bonjour");
  printf("\ntout le monde");

  mais aussi à : *printf("Bonjour\ntout le monde");*

  le résultat étant :

  Bonjour
  tout le monde.

1.2.3. Un exemple de programmation fonctionnelle en C

```
int max (int a, int b)
{
        if (a > b)
                return a ;
        else return b ;
}
int min (int a, int b)
{
        if (a < b)
                return a ;
        else return b ;
}
int difference (int a, int b, int c)
{
        return (max (a, max(b,c)) - min (a, min (b, c))) ;
}
int main ()
{   printf ("%d", difference (10, 4, 7)) ; }
```

1.2.4. Programme N°2

```
  /* Second Programme en C : Calcul de l'epsilon machine */

void main ()
{
 float eps;
 float min=0.0, max=1.0;
 const int N=30;
 int i=0;
 while (i<N) {
   eps=(min+max)/2.0;
   if (1.0 + eps > 1.0)
        /* eps est trop grand */
        max=eps;
   else
        /* eps est trop petit */
        min=eps;
   i=i+1;
 }
 printf ("On a trouvé %g comme valeur finale\n",eps);
}
```

- ⇨ Ligne 2 : **{** est placée avant les déclarations
- ⇨ Ligne 3 : Les variables sont déclarées par le nom du type suivi par les noms de variables. Il n'y a pas d'ordre imposé pour les déclarations.
- ⇨ Lignes 4&6 : Les variables peuvent être initialisées au moment de leur

- ⇨ Ligne 7 : La condition d'un **while** est entre ()
- ⇨ Ligne 8 : L'affectation est =

⇨ Ligne 9 : Conditionnelle :

if (exp_logique) instruction else instruction

- instruction pouvant être une instruction simple (terminée par un ;) ou un bloc d'instructions simples ({}).
- comme pour le while l'expression doit être parenthésée.

⇨ Ligne 10&13 : Commentaire placé entre /* et */

⇨ Ligne 15 : Opérateurs d'incrémentation possibles ++i (ou i++).

1.2.5. Les boucles : Programme N°3

⇨ On pourrait réécrire la boucle avec un for, ce qui donnerait :

en C

for (i=0 ; i<N ; i++) {
  eps=(min+max)/2.0 ;
  if (eps+1.0)>1.0
  max=eps ;
  else
  min=eps;
}

⇨ Le bouclage **for** en C se décompose en trois parties :

- i=0 : initialisation du contrôle
- i < N : condition de bouclage
- i++ : instruction à exécuter avant le rebouclage.

1.2.6. Affichage sur l'écran

⇨ Forme générale de printf

printf("....format....", e1, e2, ...);

⇨ Le premier argument est obligatoirement une chaîne de caractères constituée de caractères imprimables, de contrôles (\), et de codes formats (%)

⇨ Exemples de codes formats

- %d, %o, %x pour les entiers
- %f, %e, %g pour les réels

1.2.7. Lecture au clavier

⇨ Lecture par appel de la fonction de la bibliothèque standard, scanf qui fonctionne de façon analogue à printf.

scanf("format",&v1,&v2,...);

⇨ le format étant constitué ici uniquement de codes formats. Pour lire un entier on utilisera par exemple le code format %d.

⇨ le & signifie que la variable v1 doit être passée par adresse et non par valeur.

1.2.8. Les Tableaux : Programme N°4

```
        /* Quatrième programme en C : tableau */

void main()
{
  const int N=10;
  int t[N],i;
  for (i=0;i<N;i++){
      printf("Entrez t[%d]=",i);
      scanf("%d",&t[i]);
  }
}
```

- ⇨ En C il n'est pas nécessaire de définir de nouveaux types pour les tableaux.
- ⇨ Il est inutile de préciser l'intervalle de variation de l'indice ; on donne seulement le nombre N d'éléments dans le tableau, les indices varient

- ⇨ Déclaration d'un tableau d'entier : int t[N] à condition que N soit une constante.
- ⇨ Le mot clé const est une nouveauté apportée par la norme Ansi, avant on utilisait plutôt des pseudo-constantes définies par une directive du

- ⇨ Le préprocesseur est un programme travaillant en amont du compilateur qui permet d'effectuer des traitements sur le programme source.

1.2.9. Les Booleéns : Programme N°5

- ⇨ Le Programme 5 est mal écrit en C. Le programme 5bis est plus usuel et efficace.
- ⇨ Le type Booléen n'existe pas en C, on le fabrique par
  typedef enum BOOL {false, true} bool;
- ⇨ Ceci constitue un surnommage de type, il est plus lisible et plus pratique d'écrire :
  bool present ;
  que
  enum BOOL present;
- ⇨ false et true deviennent des constantes symboliques entières, valant respectivement 0 et 1.
- ⇨ Ca tombe bien justement car en C 0 c'est faux et tout le reste c'est vrai.
- ⇨ if (present) est correct car present a été affecté par false ou true (0 ou 1)
- ⇨ !fini && !present
- ⇨ ! est l'opératreur de négation, !fini vaut 1 si fini vaut 0 ou 0 si fini est non nul.
- ⇨ && est le et logique.
- ⇨ En C l'évaluation des expressions logiques s'arrête dès que la conclusion est connue, d'où le programme 5bis.
- ⇨ == est l'opérateur d'égalité, ne pas le confondre avec =
- ⇨ Ne pas confondre i++ et ++i
  - dans i++, i est utilisé puis incrémenté ;
  - dans ++i c'est le contraire.

```
 pour rechercher une information. Version grosBéta*/

void main()
{
  const int N=10;
  int t[N], i, x;
  typedef enum BOOL {false , true} bool;
  bool fini=false, present=false;
  for (i=0;i<N;i++){
       printf("Entrez t[%d]=",i);
       scanf("%d",&t[i]);
  }
  printf("Entrez une valeur entière\n");
  scanf("%d",&x);
  i=0;
  while (!fini && !present) {
      if (i<N)
        if (t[i] != x)
           i++;
        else present=true;
      else fini=true;
  }
  if (present)
    printf("Cette valeur est présente à l'indice %d\n",i);
  else
    printf("Cette valeur n'est pas dans le tableau");
}
```

```
/* Cinquième programme bis en C : Version C  du
        programme précédent */

void main()
{const int N=10;
  int t[N], i, x;
  for (i=0;i<N;i++){
       printf("Entrez t[%d]=",i);
       scanf("%d",&t[i]);}
  printf("Entrez une valeur entière\n");
  scanf("%d",&x);
  i=0;
  while ((i<N) && (t[i++] != x))    ;
  if (i<N)
    printf("Cette valeur est présente à l'indice %d\n",i);
  else printf("Cette valeur n'est pas dans le tableau");
}
```

1.2.10.Tableaux à deux dimensions : Programme N°6

```
        /* Sixième programme C : tableaux à deux
                        dimensions */

void main()
{const int N=4;
  float mat1[N][N], mat2[N][N], mat3[N][N], som;
  float m[][] = {{1., 0., 0., 0.}
                 {1., 1., 0., 0.}
                 {0., 2., 1., 0.}
                 {0., 0., 3., 1.}};
  int i,j,k;
  for (i=0; i< N; i++)
    for (j=0;j<N;j++)
      mat1[i][j]=mat2[i][j]=m[i][j];
   mat2[0][0]=4.0;
   for (i=0; i< N; i++)
      for (j=0;j<N;j++) {
         som=0.;
         for (k=0;k<N;k++)
            som += mat1[i][k] * mat2[k][j];
         mat3[i][j]=som;
      }
}
```

- ⇨ Les indices se juxtaposent entre crochets.
- ⇨ Les tableaux peuvent être initialisés au moment de leur déclaration quel que soit leur nombre de dimension.
- ⇨ Les affectations de tableaux sont impossibles en C du fait de leur représentation. Il faut les recopier case par case.

1.2.11.Affectation

- ⇨ Les affectations multiples sont valides

  x=y=z ; /* z est affecté à y qui est affecté à x */
- ⇨ C'est possible parce qu'en C, toute expression syntaxiquement valide renvoie une valeur que l'on utilise ou pas. (y=z renvoie la valeur z)
- ⇨ Il existe des opérateurs où l'affectation est combinée à un opérateur, par exemple, += ; -=; *= ...
  - **x += 3;** est équivalent à **x=x+3**; mais est bien plus efficace

1.2.12.Les structures : Programme N°7

- ⇨ En C la Structure se définit en utilisant le mot clé struct.

  exemple : *struct COMPLEXE {*
  float re ;
  float im ;} ;
- ⇨ Le type ainsi définit s'appelle **struct COMPLEXE**, on préfère souvent renommer ce genre de type en utilisant **typedef**

  typedef struct COMPLEXE {
  float re ;
  float im ;} complexe ;
- ⇨ le nom du type est alors **complexe**.

⇨ En C on manipule les membres ou champs de la structure avec le .

```
/* Septième programme C :les structures */

typedef struct COMPLEXE {
            float re;
            float im;
        } complexe;

complexe add(complexe z1, complexe z2)
{ complexe z;
  z.re = z1.re + z2.re;
  z.im = z1.im + z2.im;
  return z;
}

void main()
{ complexe u,v,w;
  float a,b,c,d;
  printf("Entrez 4 valeurs réelles\n");
  scanf("%g%g%g%g",,a,b,c,d);
  u.re=a ; u.im=b;
  v.re=c ; v.im=d;
  w=add(u,v);
  printf("Somme=(%g,%g)\n",w.re,w.im);
}
```

1.2.13.Structure Générale d'un programme en C - Ordre des déclarations

⇨ Il n'existe pas d'ordre impératif de déclaration des divers éléments d'un programme en C.
⇨ Les fonctions ne peuvent pas être emboitées, tout est au même niveau.
⇨ Quand le compilateur rencontre un appel de fonction qui n'a pas été définie avant, il fait l'hypothèse qu'elle renvoie un entier.
⇨ Si c'est bien le cas, pas de problème, sinon ça ne marche pas.
⇨ Pour éviter toute surprise désagréable, il est nécessaire de **prototyper** les fonctions. Le prototype n'est pas une définition, c'est une indication pour le compilateur quant aux types des paramètres de la fonction et

Ca marche	Ca ne marche pas

void main()
{ int a,b,c;
a=add (b,c);
printf("%d",a);
}
int add(int x1,int x2)
{return x1+x2;}
	void main()
	{ float a,b,c;
	a=add (b,c);
	printf("%f",a);
	}
	float add(float x1, float x2)
	{return x1+x2;}

void main()
{ float a,b,c;
float add(float,float);
a=add (b,c);
printf("%f",a);
}
float add(float x1, float x2)
{return x1+x2;}
	=>Prototype de add

1.2.14. Une caractéristique du C : les caractères

- ⇨ La notion de caractère n'est pas conceptualisée en C. Un caractère en C est comme un caractère au niveau de la machine, c'est à dire un nombre entier. (son code ASCII)
- ⇨ Le type **char** et le type **int** sont donc très voisins, l'unique différence tient au fait que le char est généralement codé sur 1 octet tandis que l'int l'est sur 2.
- ⇨ Les littéraux caractères comme 'a', '8', '?' sont des int en C.
- ⇨ On peut donc écrire 'a'+2 qui est la somme de deux int et qui donne un

- ⇨ On peut également ajouter un int et un char,

```
int x,i ; char c;
x= i+c /* c est automatiquement converti en int */
```

1.2.15. Exemple Typique de traitement de caractères en C

- ⇨ Exemple de programme C recopiant l'entrée sur la sortie

```
#include <stdio.h>
void main()
{   int c;
    c=getchar()
    while (c != EOF) {
       putchar(c) ;
       c=getchar() ;}}
```

- ⇨ **EOF** est une pseudo-constante définie par un #define dans *stdio.h* et vaut généralement -1.
- ⇨ La fonction **getchar** renvoie un entier pris sur l'entrée standard (code ascii du caractère) et -1 si la fin de fichier est atteinte. La fonction **putchar** prend un entier en paramètre effectif et envoie le caractère correspondant sur la sortie standard.
- ⇨ Ce programme est syntaxiquement correct, mais ne sera jamais réellement écrit comme cela (sauf par un débutant). On écrira plutôt :

```
#include <stdio.h>
void main()
{   int c;
    while ((c = getchar() ) != EOF) {
       putchar(c);}}
```

- ⇨ En C une expression syntaxiquement valide renvoie un résultat, c'est le cas de l'affectation qui renvoie comme valeur la valeur affectée. donc

((c = getchar ()) != EOF) se décompose en

c = getchar(), la valeur de cette expression est ensuite comparée à *EOF*.

2. Deuxième partie : Types fondamentaux du C

⇨ En C il n'y a que 2 types fondamentaux, les entiers et les réels.

⇨ Pour les entiers : 4 variantes :

- Les caractères **char**
- Les entiers courts **short int**
- Les entiers longs **long int**
- Les entiers classiques **int**

Ces quatre variantes peuvent être signées ou non. (**unsigned**), ce qui donne au total 8 types pour les entiers.

⇨ Pour les réels : 3 variantes :

- Les réels simple précision **float**
- Les réels double précision **double**
- Les réels quadruple précision **long double**;

2.1. Les entiers

2.1.1. Les caractères

⇨ En C un caractère est un entier car il s'identifie à son code ASCII (le plus souvent).

⇨ Un caractère peut donc être codé sur un octet.

⇨ On peut donc appliquer toutes les opérations entières sur les caractères (addition, soustraction ...).

⇨ Problème, On ne sait pas si ce sont des entiers signés ou non, car cela dépend des machines. Les autres variantes du type entiers sont toujours signées par défaut (pas de unsigned).

C'est pourquoi on écrit	plûtot que
int c; if ((c=getchar())!=EOF)	**char** c; if ((c=getchar())!=EOF) ...

car **getchar ()** peut renvoyer **-1** (et donc ce test ne fonctionnera que sur les machines ou les caractères sont signés).

2.1.2. Les autres entiers

⇨ Un entier correspond généralement à un mot machine.

⇨ Les attributs short, long, unsigned peuvent qualifier un entier

- *long int x;* */* x est un entier long (>= 32 bits) */*
- *short int x;* */* x est un entier court */*
- *unsigned int x;* */* x est un entier non signé */*
- *unsigned short int x;* */* x est un entier court non signé */*

⇨ Il n'y a pas de taille fixée par le langage, mais le C garanti que

1=T_C <= T_S <= T_I <= T_L

⇨ Les valeurs de T_C, T_S, T_I et T_L dépendent des machines

⇨ Un entier unsigned est un entier pour lequel le bit de signe est significatif.

⇨ Lorsque l'on utilise les attributs **short**, **long** ou **unsigned**, on peut omettre le nom du type **int**.

- *long x;* */* x est un entier long (>= 32 bits) */*
- *short x;* */* x est un entier court */*
- *unsigned x;* */* x est un entier non signé */*

· *unsigned short x; /* x est un entier court non signé */*

2.2. Les réels

⇨ Les float sont des réels codés de manière interne sous forme de mantisse/exposant.
⇨ Les double sont des float plus longs et les long double sont des doubles plus longs.
⇨ Plus la taille est grande, plus la mantisse est grande et plus la précision est grande.
⇨ Comme pour les entiers ces tailles sont dépendantes de la machine, le C garanti juste que

T_F <= T_D <= T_LD

⇨ Tailles minimales :

char	*1 octet*
int et short int	*2 octets*
long int	*4 octets*
float	*4 octets*
double	*8 octets*

2.3. Les constantes

2.3.1. Constantes de type char

⇨ Elles n'existent pas en C, **'a'** est une constante **int** pas un **char**.

2.3.2. Constantes de type entières

⇨ Elles peuvent être écrites sous 4 formes

caractère

- *imprimable* : 'A', 'a', '+', '1', ...
- *non imprimable* : '\n', '\t', '\b', ...
- *spéciaux* : '\\', '\'', '\"', ...
- *code octal* : '\0', '\126'

· *Exemples de calculs sur les constantes caractère*

'a' + '?' vaut 160 car 'a' vaut 97 et '?' vaut 63.

```
char c1,c2 ; int i ;
c1=c2 + 'a' - 'A' ;
i=c - '0' ;
```

- *décimal* : un nombre qui ne commence pas par un 0 est en base 10

  ex : 234
- *octal* : un nombre qui commence par 0 suivi d'un chiffre est en base 8

  *ex : 0234 = 4+3*8+2*64= 156*
- *hexadécimal* : un nombre qui commence par un 0 suivi d'un x (ou X) est en base 16.

  ex : 0XABCD

⇨ Sous la forme numérique on peut dire explicitement quel type d'entier on veut en rajoutant derrière la constante :

· *u ou U* : *pour un unsigned*
· *l ou L* : *pour un long*
· *ul ou uL ou Ul ou UL* : *pour un unsigned long*
· *lu ou Lu ou lU ou LU* : *pour un unsigned long*

⇨ Une constante numérique en base 10 est normalement un int. Si le nombre est trop grand le compilateur essaiera dans l'ordre les types long int, unsigned long int.

- ⇨ Une constante numérique en base 8 ou 16 est normalement un int. Si le nombre est trop grand le compilateur essaiera dans l'ordre les types unsigned int, long int, unsigned long int.
- ⇨ Les constantes de type short int et unsigned short int n'existent pas.

2.3.3. Constantes de type logique

- ⇨ Les constantes de type logique n'existent pas en tant que telle en C. On utilise la convention suivante sur les entiers :
  0 ⇔ faux et **tout entier positif ⇔ vrai**

2.3.4. Constantes de type réel

- ⇨ Les constantes *double* sont de la forme
  <partie entière>.<partie fractionnaire> <e ou E><partie exposant>
  On peut omettre
  soit <partie entière> : .475e-12
  soit <partie fractionnaire> : 482.
  mais pas les deux
  On peut omettre
  soit . : 12e7
  soit <partie exposant> : 12.489
  mais pas les deux.
- ⇨ Les constantes *long double* sont des constantes *double* suivies de l ou L
- ⇨ Les constantes *float* sont des constantes *double* suivie de f ou F

2.3.5. Constantes chaine de caractères

- ⇨ Elles sont constituées de caractères quelconques encadrés par des guillemets. ex : "abc".
- ⇨ Elles sont stockées en mémoire statique sous la forme d'un tableau de caractères terminé par la valeur '\0' (valeur entière 0).

a	b	c	\0

- ⇨ Dans le cas de 2 opérandes entiers, le résultat de la division est entier, dans tous les autres cas, il est réel.
- ⇨ Dans le cas de la division entière,le C garantit que (si b <> 0) on a : *(a/b)*b + a%b = a*
- ⇨ En C, on dispose aussi du - unaire

3.1.2. Les opérateurs relationnels

- ⇨ Les principaux opérateurs relationnels sont :
  - · *==égalité (<> de l'affectation)*
  - · *!=* *différence*
  - · *>* *supérieur*
  - · *>=* *Supérieur ou égal*
  - · *<* *Inférieur*
  - · *<=* *Inférieur ou égal*
- ⇨ Les 2 opérandes doivent avoir le même type arithmétique. Si ce n'est pas le cas, des conversions sont effectuées automatiquement selon les

- ⇨ Le type BOOLEEN n'existe pas explicitement en C : les opérateurs de relation fournissent les valeurs 0 ou 1 (respectivement FAUX et VRAI) du type int.
- ⇨ En C, on peut écrire A < B < C car cette expression correspond à (A < B) < C ce qui n'est probablement pas le résultat escompté par le programmeur : En effet si A < B, l'expression équivaut à 1 < C et sinon

3.1.3. Les opérateurs logiques

- ⇨ Les principaux opérateurs logiques sont :
  - · *!* *Négation unaire d'une valeur logique*
  - · *&&* *ET de 2 valeurs logiques*

- || *OU de 2 valeurs logiques*

⇨ Ces opérateurs interviennent sur des valeurs de type int : la valeur 0 est considérée comme la valeur FAUX; toutes les autres valeurs comme la valeur VRAI.

⇨ Les valeurs produites sont 0 (FAUX) ou 1 (VRAI) du type int.

⇨ Les opérateurs && et || impliquent l'évaluation de l'opérande gauche avant celle de droite.

⇨ L'opérande de droite peut ne pas être évalué si la valeur obtenue à partir de l'opérande de gauche suffit à déterminer le résultat :

- *0 pour l'opérande gauche de l'opérateur && implique FAUX*
- *1 pour l'opérande gauche de l'opérateur || implique VRAI*

int Tab[10];
Soit le test : (k < 10) && (Tab[k] != v)
Si k est supérieur ou égal à 10 ,l'expression (Tab[k] != v) ne sera pas évaluée et il vaut mieux car pour k supérieur ou égal à 10 Tab[k] est

3.1.4. Priorités des opérateurs

⇨ Lors de l'évaluation des expressions, certaines règles sont appliquées systématiquement par le compilateur. Le programmeur doit donc en tenir compte lors de l'écriture d'un programme.

- 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 qui sont appliqués de la droite vers la gauche)
- Si des sous-expressions parenthèsées sont présentes, les opérateurs présents dans celles-ci sont appliqués en premier

⇨ L'ordre des priorités des opérateurs déjà vus est le suivant (priorités

- ! - (opérateurs unaires)
- * / %
- + -
- < <= >= >
- == !=
- &&
- ||

⇨ Ainsi A + B * C est évalué comme A + (B * C)

⇨ **Attention**, tout programme qui dépend de l'ordre d'évaluation des opérandes doit être considéré comme incorrect, car pour la plupart des opérateurs il n'existe aucune règle et cela dépend du compilateur.

3.1.5. Priorité des opérateurs : exemples

a=1 ; *x= (a=2) + a ;*	*a=1 ;* *x = f(a=2) + g(a) ;*

/* 2 exemples incorrects,
on ne sait pas si le second a vaut 1 ou 2 */

```
printf("%d %d",++n, f(n))
```

/* Incorrect car on ne peut pas savoir
lequel de ++n et f(n) sera évalué en premier */

a[i] = i++ ;	*for (i=0; s1[i]; s2[i]=s1[i++]);*

/* Incorrect mais plus subtile */

a[i++] = i;	*for (i=0 ; s1[i] ; s2[i]=s1[++i]);*
a[i] = ++i ;	*for (i=0 ; s1[i] ; s2[i++]=s1[i]);*

/* Correct */

3.2. Autres opérateurs

3.2.1. Opérateurs binaires

- Les opérateurs binaires ont des opérandes généralement de type int, mais ils peuvent également s'appliquer aux autres types entiers (char, short ou long, signés ou non).Ces opérateurs sont au ras de la machine, ils servent à manipuler des mots bit à bit pour faire des masques par exemple.
- **a & b** : et binaire (bit a bit) => mettre des bits à 0

  unsigned n,c ;
  c=n & 0177 / c sera constitué des 7 bits de poids faible de n et complété à gauche par des 0 */*
- **a | b** : ou binaire => mettre des bits à 1

  unsigned n,c ;
  c = n | 0177 / c sera constitué des 7 bits de poids faible de n et complété à gauche par des 1 */*
- **a ^ b** : ou exclusif binaire => inverser des bits

  unsigned n,c ;
  c = n ^ 0177 / c sera constitué des bits de n, les 7 bits de poids faible étant inversés */*
- **a << b** : décalage à gauche

  Les bits de a sont décalés de b positions vers la gauche, les bits de poids fort sont perdus, des 0 arrivent sur la gauche.
- **a >> b** : décalage à droite

  Les bits de a sont décalés de b positions vers la droite, les bits de poids faible sont perdus, des bits X arrivent sur la gauche.

  si a est non signé ou signé positif : X=0;
  sinon (a négatif) : pas de norme.
- **~a**: complément à 1 => les bits de a sont inversés.

3.2.2. Opérateur d'affectation

- = est le symbole d'affectation
- l'affectation est une expression => **elle renvoie une valeur égale à la valeur de l'objet affecté**.
- On peut combiner l'affectation avec l'un des dix opérateurs suivant : + - * / % << >> & | ^
- Cela peut permettre d'évaluer une expression couteuse

  ex : Tab[indice(i)] = Tab[indice(i)] + n%k;
  Tab [indice(i)] += n % k ;
- Exemple : opérateur d'affectation et opérateurs binaires :

```
int bitcount (unsigned int n)
{ /* compter les bits à 1 dans n */
    int b ;
    for (b=0 ; n ; n >>=1 )
            if (n & 01) b++ ;
    return b ;
```

}

3.2.3. Opérateur conditionnel et opérateur de séquence

- ⇨ **e1 ? e2:e3** est une expression qui vaut e2 si e1 est vraie et e3 sinon.
- ⇨ **e1, e2** est une expression qui vaut e2 mais dont l'évaluation entraine au préalable celle de e1 (utilisation dans for).

Exemple en C	*Equivalent Pascal*
i=0 ; *while (i == N ? 0 :k=i, a[i++] != x) ;*	*continue := true ; i:=0;* *while continue do begin* *if i=N then continue := false ;* *else begin* *k:=1; i:=i+1;* *if a[i-1] = x* *then continue:=false* *end* *end*

3.2.4. Opérateur d'appel de fonctions ()

- ⇨ Les arguments effectifs sont convertis lors de l'appel de la fonction. Le compilateur vérifie que le nombre d'arguments effectifs est égal au nombre d'arguments formels.

3.2.5. Opérateur sizeof

- ⇨ **sizeof (x)** donne la taille de x.
- ⇨ Si x n'est pas un nom de type les parenthèses sont facultatives.
- ⇨ ex : *int x, y, z;*
  y=sizeof x;
  z=sizeof (int) ;
- ⇨ **Attention** si x est une expression comme *sizeof(i++)* l'expression elle-même n'est pas calculée.

3.2.6. Opérateur de cast (t) a

- ⇨ Le **cast** sert à forcer le type d'un objet, c'est à dire convertir un objet d'un type vers un autre.
- ⇨ ex : *(long int) x*y / (1. +sin(x))*
  int f (float x)
  *{ return (int) x*x ;}*

3.3. Conversions numériques

On peut se trouver face à trois types de conversions

- ⇨ **La conversion intégrale** où l'intégrité du nombre est préservée
  - Un entier signé converti en un entier signé plus long
  - un float converti en double
- ⇨ **La conversion conforme** qui ne concerne que les entiers. L'entier change de valeur, mais la connaissance du type initial et de la valeur après conversion permet de reconstituer la valeur d'origine.
  - Un entier court négatif converti en entier non signé plus long
- ⇨ **La conversion dégradante** où il y a perte définitive d'informations utiles
  - Un entier est converti en entier plus court avec perte de bits significatifs.
  - Un double est converti en float avec perte de bits significatifs dans la mantisse.

3.4. Conversions implicites

- ⇨ Promotion entière :

- Signés ou non, les *char*, *short*, *champ de bits*, *énumération* sont implicitement convertis en *int* ou *unsigned int*.

⇨ Conversions d'arguments effectifs
- Lors de l'appel d'une fonction les arguments effectifs sont convertis dans le type des arguments formels déclarés dans le prototype de la fonction.

⇨ Conversions arithmétiques usuelles
- C'est se rapprocher du type "le plus grand"

⇨ Exemple de conversion implicite

```
int atoi (char s[])  /*a(scii) to i(nteger) */
{    int n = 0, i ;
     for (i=0; s[i] >= '0' && s[i]<= '9'; i++)
          /* s[i] est converti implicitement en int */
          n = 10*n + s[i] - '0';
     return n;
}
```

4. Quatrième Partie : Les Principales instructions

⇨ Une instruction simple est une expression suivie d'un ; qui fait partie de l'instruction (ce n'est pas un séparateur comme en Pascal).

⇨ Une instruction composée est une suite d'instructions encadrée par {et}. Une instruction composée peut se mettre partout où l'on peut mettre une instruction.

4.1. if

⇨ syntaxe : *if (expression) instruction*
- si expression est vraie (<> 0) instruction est exécutée

⇨ Attention au piège de l'affectation, ne pas confondre ces deux programmes :

int x,y; *...* *y=f(...);* *if (x == y)* *.....* *.....*	*int x,y;* *...* *y=f(...);* *if (x = y)* *.....* *.....*

4.2. if ... else

⇨ Syntaxe : *if (expression) instruction1 else instruction2*
- *instruction1* et *instruction2* sont des instructions simples (penser au ; (<> Pascal)), ou des instructions composées.

4.3. while

⇨ Syntaxe : *while (expression) instruction*
- *instruction* est une instruction simple ou composée

⇨ Il est fréquent en C de voir une partie du travail ou la totalité reportée dans l'expression :
- *while ((c=getchar())!= EOF)*
  putchar(c) ;
- *while (s[i++]) ;*

4.4. for

⇨ Syntaxe : *for (expression1 ; expression2 ; expression3)*
instruction
- *instruction* est une instruction simple ou composée.
- expression1 sert à initialiser
- expression2 est la condition de rebouclage
- expression3 est l'expression d'incrémentation

⇨ Le for du C est un **tantque** traditionnel des autres langages. il peut dans la plupart[1] des cas être réécrit de la façon suivante :

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

⇨ Les expressions peuvent comporter plusieurs instructions

```
char s[]="Coucou c\'est nous";
int c,i,j;
for ( i=0, j=strlen(s)-1 ; i<j ; i++ , j++) {
    c=s[i] ; s[i]=s[j] ; s[j]=c ;}
```

⇨ Rien n'oblige en C la présence des trois expressions
- *for (;;) ...* est valide et équivalent à *while (1) ...*

4.5. do ... while

[1] sauf quand on utilise continue

```
switch (i) {
    case 1 : a=s[j];
    case 2 : b++;
    case 3 : a=s[j-1];}
/* lorsque i vaut 1 les trois instructions sont exécutées, quand i vaut 2 ce sont les deux dernières (b++ et a=s[j-1]). */
```

⇨ Pour éviter cela, il faut utiliser la fonction **break**

```
switch (i) {
    case 1 : a=s[j];break;
    case 2 : b++;break;
    case 3 : a=s[j-1]; /* le dernier break est inutile */}
```

⇨ Il n'y a pas de possibilité de donner d'énumérations de valeurs ou d'intervalles

5. Cinquième Partie : La programmation en C

5.1. Les Fonctions

5.1.1. Présentation générale

- Une fonction est une portion de programme formant un tout homogène, destiné à remplir une certaine tâche bien délimitée.
- Lorsqu'un programme contient plusieurs fonctions, l'ordre dans lequel elles sont écrites est indifférent, mais elles doivent être indépendantes.
- En règle générale, une fonction appelée a pour rôle de traiter les informations qui lui sont passées depuis le point d'appel et de retourner une valeur. La transmission de ces informations se fait au moyen d'identificateurs spécifiques appelés arguments et la remontée du résultat par l'instruction return.
- Certaines fonctions reçoivent des informations mais ne retournent rien (par exemple printf), certaines autres peuvent retourner un ensemble de valeurs (par exemple scanf).

5.1.2. Définition de fonction

- La définition d'une fonction repose sur trois éléments
  - ♦ Son type de retour
  - ♦ La déclaration de ses arguments formels
  - ♦ Son corps
- De façon générale, la définition d'une fonction commence donc par :
  type_retour nom (type-arg-f1 arg-f1, type-arg-f2 arg-f2 ...)
- Les arguments formels permettent le transfert d'informations entre la partie appelante du programme et la fonctions. Ils sont locaux à la fonction, et lors de l'appel ils seront mis en correspondance avec les arguments effectifs.
- Contrairement à d'autres langages, le C ne permet pas la modification **Le seul mode de passage des paramètres est le mode par valeur**. Cela signifie que « les valeurs des paramètres effectifs sont copiées dans les paramètres formels ».
- L'information retournée par une fonction au programme appelant est transmise au moyen de l'instruction return, dont le rôle est également de rendre le contrôle de l'exécution du programme au point où a été
- La syntaxe générale de la fonction return est la suivante :
  return expression ;

5.1.3. Déclaration de fonction

- Lorsque l'appel d'une fonction figure avant sa définition, la fonction appelante doit contenir une **déclaration** de la fonction appelée. On appelle cela **prototype** de la fonction.
- **Attention a ne pas confondre définition et déclaration de fonctions**. La **déclaration est une indication** pour le compilateur quant au type de résultat renvoyé par la fonction et éventuellement au type des arguments, et rien de plus.
- Dans sa forme la plus simple la déclaration d'une fonction peut : *type_de_retour nom () ;*
- Les prototypes de fonctions sont souvent regroupées par thèmes dans des fichiers dont l'extension est généralement h (ex stdio.h). Ces fichiers sont inclus dans le source du programme par une directive du

: Soit en obtenant un total de 2,3, ou 12 au premier essai, soit en obtenant d'abord un total de 4,5,6,8,9 ou 10, suivi d'un ou plusieurs lancers jusqu'à obtenir un total de 7 avant d'avoir retrouver le total du premier lancer.

- Le jeu simulé doit être interactif, de sorte que chaque lancer soit déclenché par le joueur en pressant une touche du clavier. Le résultat de chaque lancer est affiché par un message. A la fin de la partie, l'ordinateur demande à l'utilisateur s'il souhaite rejouer.

5.1.5. Récursivité

- La récursivité est la caractéristique des fonctions capables de s'appeler elles-mêmes de façon répétitive, jusqu'à ce que soit vérifiée une condition d'arrêt. Le C permet d'écrire des fonctions récursives.
- Exemple Affichage renversé d'un texte.

```
#include <stdio.h>
void main() {
        void renverse(void) ;
        printf (" Saisir une ligne de texte ci-dessous \n " ) ;
        renverse() ;
}
/* Lecture d'une ligne et affichage en sens inverse */
void renverse (void) {
        char c ;
        if ((c = getchar()) != '\n' )
                renverse() ;
        putchar(c) ;
        return ;
}
```

⇨ Exemples :

- ***auto int a, b, c ;*** Var automatiques entières
- extern float racine1, racine2 ; Var externes de type réelles
- static int compteur = 0 ; Var entière statique initialisée à 0
- extern char etoile ; Var externe caractère

5.2.2. Variables de classe automatique

⇨ Les variables de classe automatique sont toujours déclarées au sein d'une fonction, pour laquelle elle joue le rôle de variable locale. Leur portée est limitée à cette seule fonction. C'est le mode par défaut, donc **auto** est facultatif.

⇨ Ces variables peuvent être initialisées en incluant les expressions adéquates dans leur déclaration. La valeur d'une variable de classe auto n'est pas conservée d'un appel à l'autre de la fonction.

⇨ Il est possible de réduire la portée d'une variable automatique à une étendue inférieure à celle de la fonction. Il faut la définir pour cela dans une instruction composée, sa durée de vie est alors celle de

5.2.3. Variables de classe externe

⇨ Les variables de classe externe au contraire des variables de classe automatique ne sont pas limitées à la seule étendue d'une fonction. Leur portée s'étend de l'endroit où elles sont définies à la fin du programme. Elles constituent en C ce qu'on appelle une variable

- ⇨ Une variable externe peut par conséquent se voir affecter une valeur dans une fonction, valeur qui pourra être accédée depuis une autre fonction. (Attention aux dangers !!)
- ⇨ Lorsque l'on utilise des variables externes il est important de différencier leur définition et leur déclaration. La définition d'une variable externe s'écrit comme la définition de n'importe qu'elle autre variable ordinaire. Elle doit se situer en dehors et en amont des fonctions qui utiliseront la variable. **Elle ne doit pas contenir le mot clé extern** (compiler dependent).
- ⇨ Toute fonction utilisant une variable externe doit comporter une déclaration explicite de cette variable, (obligatoire si la fonction est définie avant la variable, facultatif si la fonction est définie après, mais obligatoire si c'est Jacoboni qui corrige la copie !). Cette déclaration se traduit par la répétition de la définition précédée du mot clé **extern**.
- ⇨ Aucune allocation de mémoire n'est faite à la suite d'une déclaration puisque cette allocation est faite une fois pour toute lors de la définition de la variable. Ceci explique qu'il n'est pas possible d'initialiser une variable externe au moment de sa déclaration, mais uniquement lors

- ⇨ Les variables externes peuvent être initialisées dans leur définition, mais leur valeur initiale doit être une constante. (En aucun cas une expression). En l'absence de toute initialisation explicite, les variables externes sont automatiquement initialisées à 0. Il est cependant préférable d'expliciter dans la définition, l'initialisation à 0 lorsqu'elle

- ⇨ Les Tableaux peuvent être également de classe automatique ou externe, mais il n'est pas possible d'initialiser un tableau de classe automatique.
- ⇨ Il est important de souligner les risques inhérents à l'usage des variables externes, en particulier les effets de bords. Ces variables sont souvent la source d'erreurs difficiles à identifier. Le choix par le programmeur des classes de mémorisation doit donc être effectué de façon réfléchie, pour répondre à des situations bien déterminées.

5.2.4. Variables de classe Statique des programmes monofichiers

- ⇨ Les variables statiques sont définies dans les fonctions comme en automatique et ont donc la même portée. (Locale à la fonction).
- ⇨ Par contre, ces variables conservent leur valeur durant tout le cycle du programme. Ce sont des variables rémanentes.
- ⇨ On les définit de la même façon que les variables **automatique** sauf que leur déclaration doit commencer par le spécificateur de classe **static**.
- ⇨ Les définitions de variables statiques peuvent contenir des affectations de valeurs initiales qui doivent respecter les points suivants :
  a) Les valeurs initiales doivent être des constantes et pas des expressions,
  b) En l'absence de valeur initiale, une variable statique est toujours initialisée à 0.

5.3. Communication entre modules

⇨ Une application peut être conçue comme une collection de modules, chaque module :

- ♦ Une collection de sous-programmes,
- ♦ Une collection de données partagées,
- ♦ Un objet abstrait possédant une représentation et offrant des opérateurs de manipulation.

⇨ De tels modules peuvent constituer des unités de programme en C. Mémorisée dans un fichier, une unité de programme constitue une unité de compilation.

⇨ Après compilation de toutes les unités constituant l'application, l'éditeur de lien a pour tâche de rassembler les divers modules résultats des compilations (modules objets) afin de constituer l'aplication totale.

⇨ Il est donc nécessaire d'établir un protocole de communication entre les divers modules. Il permet de spécifier dans le programme source, les liens à établir entre

5.3.1. Règles de communication

⇨ On dit qu'une variable ou une fonction est partagée entre deux modules lorsqu'elle est utilisée dans ces deux modules. En C seules les variables globales peuvent être partagées entre deux modules. Par contre, toute fonction peut être partagée entre deux modules.

⇨ Dans le module où la variable est définie, une définition introduit l'identificateur de la variable, son type et éventuellement une valeur initiale. La variable est visible dans tout le module, à partir du point de

⇨ Dans un autre module où la variable est utilisée, une déclaration introduit l'identificateur de type de la variable. Cette déclaration doit être **extern**. La variable est alors visible dans tout le module à partir du point de cette déclaration.

⇨ Exemple

Module 1	**Module 2**
int v= 23 ; */* Variable val de module 2 */* ***extern*** *int val ;* */* fonction fonct de module 2 */* ***extern*** *int fonct()* *main() {* *....* *}*	*int val = 100 ;* *int fonct (int p) {* */* variable v de module 1 */* ***extern*** *int v ;* *.....* *}*

⇨ Une fonction définie dans un module peut être manipulée dans un autre module, a condition d'avoir effectué une déclaration d'importation dans le second module.

⇨ Il est des cas où on ne désire pas que les symboles définis dans un module soient accessibles dans un autre module. Pour cela on utilise le préfixe **static**.

⇨ Exemple

Module 1 *static int V = 0 ;* /* Variable locale au module 1 */
Module 2 *int V = 0 ;* /* Variable exportable */
Module 3 *extern int V ;* /* Variable V définie dans le module 2 */

5.4. Utilisation de make

⇨ L'utilitaire make permet de maintenir une application multi-fichiers, en s'appuyant sur les dates des dernières modifications des fichiers. Il dispense l'utilisateur d'avoir à se souvenir :

- ♦ Des relations existant entre les fichiers,
- ♦ Des fichiers récemment modifiés,
- ♦ Des fichiers à recompiler,
- ♦ Des opérations à effectuer pour construire l'application.

⇨ En effet, lorsqu'une composante est modifiée, un appel à make permet de réactualiser automatiquement les fichiers qui ont besoin de l'être. Pour cela, make s'appuie sur un fichier (le makefile) qui décrit une fois pour toutes les relations entre les fichiers et les actions à entreprendre pour les mises à jour.

⇨ Le fichier en cours de réactualisation est appelé cible. Le traitement effectué par make , pour le fichier cible courant comporte 3 étapes :

- ♦ Trouver dans le fichier description les lignes qui concernent la cible,
- ♦ s'assurer que tous les fichiers dont dépend la cible existent et sont à jour,
- ♦ créer si nécessaire la cible.

5.4.1. La syntaxe du makefile

⇨ Le fichier description comporte une séquence d'enregistrements spécifiant les relations entre fichiers. Un enregistrement occupe une ou plusieurs lignes, selon le schéma suivant (<tab> désigne le caractère de tabulation).

```
Cibles : FICHIERS_PREREQUIS ; COMMANDE
<tab> COMMANDE
.................
<tab> COMMANDE
```

⇨ Une commande qui occupe plusieurs lignes peut être poursuivie sur la ligne suivante en terminant la ligne par le caractère \

⇨ Le plus délicat dans l'écriture du makefile consiste a déterminer les relations entre les fichiers.

⇨ Exemple : considérons une application dont le programme executable est contenu dans le fichier **applic.out**. Les sources C correspondant sont dans les fichiers **sleep.c** et **test_sleep.c**. Le fichier de description suivant (makefile1) permet de maintenir automatiquement une telle application.

```
applic.out : test_sleep.o sleep.o
        cc test_sleep.o sleep.o -o applic.out
test_sleep.o : test_sleep.c
        cc -c  test_sleep.c
sleep.o : sleep.c
        cc -c sleep.c
```

⇨ Utilisation :

```
$ ls sleep.o test_sleep.o
    sleep.o not found
    test_sleep.o not found
$ make -f makefile1
    cc -c test_sleep.c
    cc -c sleep.c
    cc test_sleep.o sleep.o -o applic.out
$ ls sleep.o test_sleep.o applic.out
```

Les fichiers n'existent pas

applic.out sleep.o test_sleep.o Ces fichiers ont été créés

$ make -f makefile1

'applic.out' is up to date.

⇨ Supposons maintenant qu'une modification est faite dans le fichier sleep.c. il est alors nécessaire de reconstruire l'application

$ make -f makefile1

cc -c sleep.c *test_sleep.c n'est pas recompilé*

cc test_sleep.o sleep.o -o applic.out

6. Sixième partie : Les Pointeurs et les Tableaux

⇨ Un pointeur, en C comme en PASCAL, est une adresse permettant de désigner un objet (une variable ou une fonction) en mémoire centrale.

⇨ Par extension, on appelle pointeur la variable qui contient cette adresse.

6.1. Pointeurs et adresses

⇨ Un pointeur définissant l'adresse d'un objet, l'accès à cet objet peut alors être réalisé par une indirection sur le pointeur.

- L'opérateur unaire **&** fournit l'adresse de l'objet opérande (qui doit donc être une lvalue[2]). On dit aussi que cet opérateur est l'opérateur de _____.
- L'opérateur unaire ***** considère son opérande comme un pointeur et retourne l'objet pointé par celui-ci. On dit aussi que cet opérateur est l'opérateur **d'indirection**.

Exemples :

```
int X, Y;
/* Soit PX un pointeur sur des int */
PX = &X;      /* PX <- l'adresse de X */
Y = *PX;      /* Y <- l'objet pointé par PX (ie X) */
```

Ceci équivaut donc à : Y = X;

Remarques :

1- La déclaration d'un pointeur doit spécifier le type de l'objet pointé. On dit alors que **le pointeur est typé.**
Ainsi la déclaration : int *PX; signifie que PX pointera des objets de type int. (Littéralement : *PX sera un int)

2- La définition d'un pointeur ne fait rien d'autre que ce qu'elle doit faire, c'est à dire qu'elle réserve en mémoire la place nécessaire pour mémoriser un pointeur, et en aucune façon de la place pour mémoriser un objet du type pointé

3- L'objet atteint par l'intermédiaire du pointeur possède toutes les propriétés du type correspondant.

Exemples :

```
PX = &X;
Y = *PX + 1;      /* Y = X + 1   */
*PX = 0;          /* X = 0       */
*PX += 10;        /* X = X + 10 */
(*PX)++;          /* X = X + 1   */
```

Dans le dernier exemple, les parenthèses sont nécessaires du fait de la priorité supérieure de l'opérateur ++ sur l'opérateur * : Si elles n'y étaient pas, cela serait interprété comme : Incrémente le pointeur PX et fournit ensuite la valeur pointée, ce qui n'est pas ce que l'on voulait...

4- Un pointeur pouvant repérer un type quelconque, il est donc possible d'avoir un pointeur de pointeur.

*Exemple : int **PPint; PPint pointe un pointeur d'entiers*

5- L'opérateur & n'est pas autorisé sur les variables ayant l'attribut **register**, ce qui est normal car celles-ci peuvent avoir été placées dans un registre interne du CPU et ceux-ci n'ont pas d'adresse...

6.2. Opérations sur les pointeurs

[2] LVALUE= Tout ce qui peut être à gauche d'un symbole d'affectation.

⇨ La valeur NULL, est une valeur de pointeur, constante et prédéfinie dans stddef.h. Elle vaut 0 et signifie "Aucun objet". Cette valeur peut être affectée à tout pointeur, quel que soit son type.

- En ce cas, ce pointeur ne pointe sur rien...
- Bien entendu, l'utilisation de cette valeur dans une indirection provoquera une erreur d'exécution.

⇨ L'affectation d'un pointeur à un autre n'est autorisée que si les 2 pointeurs pointent le même type d'objet (ie ont le même type)

Exemple :

```
P = NULL;
P = Q;     /* P et Q sont de pointeurs sur le même type */
P = 0177300;          /* illégal */
P = (int *) 0177300;  /* légal */
```

⇨ L'incrémentation d'un pointeur par un entier n est autorisée. Elle ne signifie surtout pas que l'adresse contenue dans le pointeur est incrémentée de n car alors cette adresse pourrait désigner une information non cohérente : être à cheval sur 2 mots par exemple...

⇨ L'incrémentation d'un pointeur tient compte du type des objets pointés par celui-ci : elle signifie "passe à l'objet du type pointé qui suit immédiatement en mémoire". Ceci revient donc à augmenter l'adresse contenue dans le pointeur par la taille des objets

⇨ Dans le cas d'une valeur négative, une décrémentation a lieu.

⇨ Les opérateurs combinés avec l'affectation sont autorisés avec les pointeurs. Il en va de même pour les incrémentations / décrémentations explicites.

Exemples :

```
int *Ptr, k;     /* Ptr est un pointeur , k est un int */

Ptr++;
Ptr += k;
Ptr--;
Ptr -= k;
```

6.3. Comparaison de pointeurs

⇨ Il est possible de comparer des pointeurs à l'aide des relations habituelles : < <= > >= == !=

⇨ Cette opération n'a de sens que si le programmeur a une idée de l'implantation des

⇨ Un pointeur peut être comparé à la valeur NULL.

Exemples :

```
int *Ptr1, *Ptr2;
......if (Ptr1 <= Ptr2)
......if (Ptr1 == 0177300)          /* illégal */
......if (Ptr1 == (int *)0177300)   /* légal */
......if (Ptr1 != NULL)    ......
```

6.4. Soustraction de pointeurs

⇨ La différence de 2 pointeurs est possible, pourvu qu'ils pointent sur le même type d'objets. Cette différence fournira le nombre d'unités de type pointé, placées entre les adresses définies par ces 2 pointeurs.

⇨ Autrement dit, la différence entre 2 pointeurs fournit la valeur entière qu'il faut ajouter au deuxième pointeur (au sens de l'incrémentation de pointeurs vue plus haut) pour obtenir le premier pointeur

6.5. Affectation de chaînes de caractères

⇨ L'utilisation d'un littéral chaîne de caractères se traduit par la réservation en mémoire de la place nécessaire pour contenir ce littéral (complété par le caractère '\0') et la production d'un pointeur sur le premier caractère de la chaîne ainsi mémorisée.

⇨ Il est alors possible d'utiliser cette valeur dans une affectation de pointeurs.
Exemple :

```
char *Message;
Message = "Langage C";  /* est autorisé */
/* La chaîne est mémorisée et complétée par '\0'. La variable Message reçoit l'adresse
```

6.6. Pointeurs et tableaux

⇨ On rappelle que la déclaration d'un tableau dans la langage C est de la forme : int Tab[10];

⇨ Cette ligne déclare un tableau de valeurs entières dont les indices varieront de la

⇨ En fait, cette déclaration est du "sucre syntaxique" donné au programmeur. En effet, de façon interne, elle entraine :

- La définition d'une valeur de pointeur sur le type des éléments du tableau, cette valeur est désignée par le nom même du tableau (Ici Tab)
- La réservation de la place mémoire nécessaire au 10 éléments du tableau, alloués consécutivement. L'adresse du premier élément (Tab[0]) définit la valeur du pointeur Tab.

⇨ La désignation d'un élément (Tab[5] par exemple) est automatiquement traduite, par le compilateur, en un chemin d'accès utilisant le nom du tableau (Dans notre exemple ce serait donc *(Tab + 5))

Exemples :

```
int *Ptab, Tab[10];
```

On peut écrire :

```
Ptab = &Tab[0];
Ptab = Tab;          /* Equivalent au précédent */
Tab[1] = 1;
*(Tab + 1) = 1;      /* Equivalent au précédent */
```

Donc, pour résumer, on a les équivalences suivantes :

```
Tab + 1   est équivalent à    &(Tab[1])
*(Tab + 1)    est équivalent à    Tab[1]
*(Tab + k)    est équivalent à    Tab[k]
```

- **Aucun contrôle n'est fait** pour s'assurer que l'élément du tableau existe effectivement :
- Si Pint est égal à &Tab[0], alors (Pint - k) est légal et repère un élément hors des bornes du tableau Tab lorsque k est strictement positif.
- Le nom d'un tableau n'est pas une lvalue, donc certaines opérations sont impossibles (exemple : Tab++)

Application :

```
#include <stdio.h>
main()
{
 char Tab[32], *Ptr;
        /* Initialisation des données */
Tab[0] = 'Q'; Tab[1] = 'W'; Tab[2] = '\0';
Ptr = "ASDFGHJKL";
        /* Edition des chaines */
printf("Contenu des chaines : \n");
printf("  Tab : %s\n  Ptr : %s\n", Tab, Ptr);
        /* Utilisation des tableaux */
printf("Edition de l'élément de rang 1 des tableaux\n");
printf("  Tab : %c\n  Ptr : %c\n", Tab[1], Ptr[1]);
        /* Utilisation des pointeurs */
printf("Edition du caractère pointé\n");
printf("  Tab : %c\n  Ptr : %c\n",*Tab, *Ptr);
        /* Utilisation des pointeurs incrémentés */
printf("Edition du caractère suivant le caractère pointé\n");
printf("  Tab : %c\n  Ptr : %c\n", *(Tab + 1), *(Ptr + 1));
        /* Règle des priorités */
printf ("Edition identique non parenthésée \n");
printf("  Tab : %c\n  Ptr : %c\n", *Tab + 1, *Ptr + 1);
}
```

En sortie, on aura :

```
Contenu des chaines :
Tab : QW
Ptr : ASDFGHJKL
Edition de l'élément de rang 1 des tableaux
Tab : W
Ptr : S
Edition du caractère pointé :
Tab : Q
Ptr : A
Edition du caractère suivant le caractère pointé
Tab : W
Ptr : S
Edition identique non parenthésée :
Tab : R
Ptr : B
```

6.7. Passage des paramètres

⇨ le seul mode de passage des paramètres à une fonction est le passage par valeur.

⇨ Le problème est que lorsqu'un sous-programme doit fournir une réponse par un de ses paramètres, ou modifier ceux-ci, le mode de passage par valeur ne peut plus convenir.

⇨ De plus, dans le cas de paramètres structurés de taille importante, la copie de ceux-ci entraine une consommation mémoire parfois superflue.

⇨ C'est à la charge du programmeur de gérer tous ces problèmes en fournissant un pointeur sur l'objet en question, ce qui non seulement permet au sous-programme de travailler directement sur l'objet réel et donc de le modifier, mais aussi réduit la taille des échanges, puisqu'il n'y a copie que d'une adresse.

Exemple classique :

```
#include <stdio.h>
void Echange_1 (A,B)
int A,B;
{
 int C = A;
 A = B;
 B = C;
}
```

```
void Echange_2 (A,B)
int *A,*B;
{
 int C = *A;
 *A = *B;
 *B = C;
}
```

```
main()
{
 int X, Y;
 printf("Valeur de X : ");
 scanf ("%d",&X);
 printf("\nValeur de Y : ");
 scanf ("%d",&Y);
 Echange_1 (X,Y)
 printf("\nAprès appel de Echange_1 :\n X = %d\n Y = %d\n", X, Y);
 Echange_2 (&X,&Y)
 printf("Après appel de Echange_1 :\n X = %d\n Y = %d\n", X, Y);
}
```

Ce qui produit à l'exécution :

Valeur de X : ***12***
Valeur de Y : ***34***
Après appel de Echange_1 :
X = 12
Y = 34
Après appel de Echange_2 :
X = 34
Y = 12

⇨ **Quand un tableau est transmis comme paramètre effectif à un sous-programme, c'est en fait l'adresse de base de ce tableau qui est transmise par valeur**.

⇨ **Ceci a pour effet de transmettre le tableau par référence sans intervention du programmeur**.

⇨ De plus, à l'intérieur de la fonction appelée, **le paramètre tableau devient une lvalue,** certaines opérations sont donc possibles.

Exemple :

```
#include <stdio.h>
void Test_Tab (T)
int T[5];
{
 printf(" Editions dans la fonction \n");
```

```
 printf(" Valeur du paramètre tableau : %o \n",T);
 printf(" Valeur du contenu : %d\n",T[0]);
 printf(" Test d'incrémentation paramètre \n");
 T++; /* T est devenue une lvalue */
 printf(" Valeur du paramètre tableau : %o \n",T);
}

main()
{
 int Tab[5];
 Tab[0] = 10;
 printf("Editions avant appel de fonction\n");
 printf(" Valeur du paramètre tableau : %o\n",Tab);
 printf(" Valeur du contenu : %d\n",Tab[0]);
 Test_Tab(Tab);
}
```

Exemple de sortie :

```
Editions avant appel de fonction
 Valeur du paramètre tableau : 177514
 Valeur du contenu : 10
Editions dans la fonction
 Valeur du paramètre tableau : 177514
 Valeur du contenu : 10
 Test d'incrémentation paramètre
 Valeur du paramètre tableau : 177516
```

6.8. POINTEURS ET TABLEAUX MULTIDIMENSIONNELS

⇨ Un tableau unidimensionnel peut se représenter grâce à un pointeur (le nom du tableau) et un décalage (l'indice).

⇨ Un tableau à plusieurs dimensions peut se représenter à l'aide d'une notation similaire, construite avec des pointeurs.

⇨ Par exemple, un tableau de dimension 2, est en fait un ensemble de deux tableaux à une seule dimension. Il est ainsi possible de considérer ce tableau à deux dimensions comme un pointeur vers un groupe de tableaux unidimensionnels consécutifs. La déclaration correspondante pourra s'écrire de la manière suivante:

*type-donnée (*varpt)[expression 2];*

au lieu de la déclaration classique:

type-donnée tableau [expression l][expression 2];

⇨ Ce style de déclaration peut se généraliser à un tableau de dimension n de la façon suivante:

*type-donnée (*varpt)[expression 2][expression 3] ... [expression n];*

qui remplace la déclaration équivalente:

type-donnée tableau[expression 1][expression 2] ... [expression n];

⇨ Dans ces déclarations, type-donnée désigne le type du tableau, varpt est le nom du pointeur vers le tableau, tableau est le nom du tableau correspondant, et expression 1, expression 2,...expression n sont des expressions entières positives qui spécifient le nombre maximum d'éléments associés à chacune des dimensions.

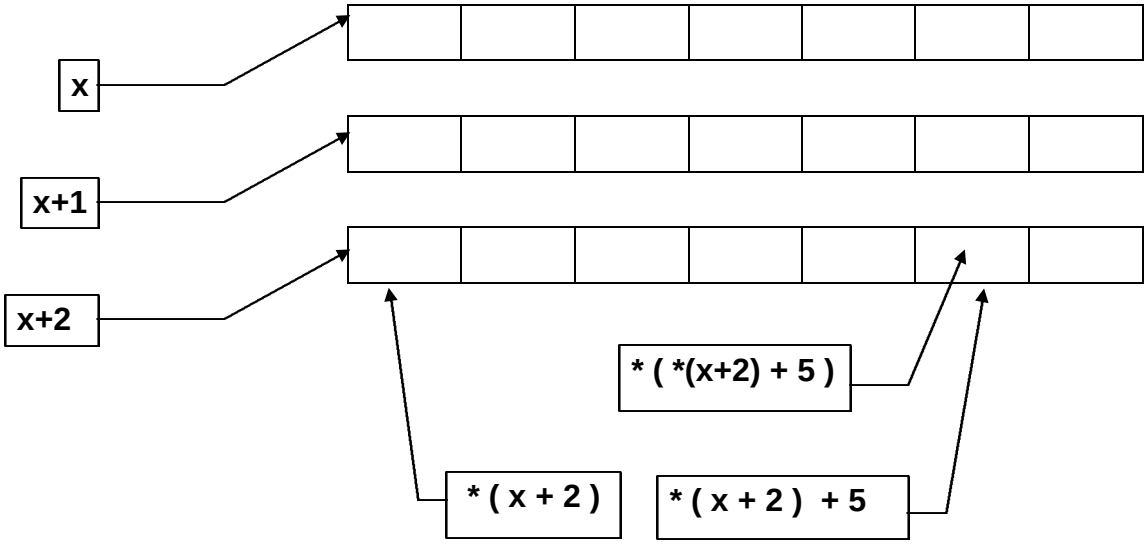
⇨ On remarquera les parenthèses qui encadrent le nom du tableau, ainsi que l'astérisque qui introduit le pointeur. Ces parenthèses sont indispensables, car sans

leur présence, on définirait un tableau de pointeurs, au lieu d'un pointeur vers un ensemble de tableaux, puisque les symboles [ ] et * s'évaluent normalement de droite

⇨ Exemple : On suppose que x est un tableau d'entiers de dimension 2, ayant 10 lignes et 20 colonnes. On peut le déclarer de la façon suivante:

*int (*x) [20];* au lieu de *int x[10] [20];*

⇨ Dans la première déclaration, on dit que x est un pointeur vers un groupe de tableaux d'entiers contigus, possédant chacun une dimension et 20 éléments. Ainsi, x pointe le premier tableau de 20 éléments, qui constitue en fait la première ligne (ligne 0) du tableau d'origine de dimension 2. De manière analogue. (x+1) pointe vers le deuxième tableau de 20 éléments, qui représente la seconde ligne (ligne 1) du tableau d'origine, et ainsi de suite



⇨ Si l'on considère à présent un tableau tridimensionnel de réels, t, on peut le définir par:

*float (*t) [20] [30];* au lieu de *float t[10] [20] [30];*

⇨ La première forme définit t comme un pointeur vers un groupe contigu de tableaux de réels, de dimension 20 x 30. Dans ce cas t pointe le premier tableau 20 x 30, (t+1) pointe le second tableau 20x30, etc.

⇨ Pour accéder à un élément donné d'un tableau à plusieurs dimensions, on applique plusieurs fois l'opérateur d'indirection. Mais cette méthode est souvent plus pénible d'emploi que la méthode classique.

⇨ Exemple : x est un tableau d'entiers de dimension 2, à 10 lignes et 20 colonnes. L'élément situé en ligne 2 et colonne 5 peut se désigner aussi bien par :

x[2] [5] que par ** (* (x+2) +5)*

⇨ Explications :

- **(x+2)** est un pointeur vers la ligne 2, donc *** (x+2)** représente la ligne 2 toute entière.
- Puisque la ligne 2 est elle-même un tableau, *** (x+2)** est aussi un pointeur vers son premier élément.
- On ajoute ensuite 5 à ce pointeur. L'expression **(* (x+2) +5)** donne donc un pointeur vers l'élément 5 (le sixième) de la ligne 2.
- L'objet désigné par ce dernier pointeur, *** (* (x+2) +5)** est donc bien l'élément cherché, en colonne 5 de la ligne 2, ou bien encore x [2] [5] .

⇨ Les programmes mettant en jeu des tableaux multidimensionnels peuvent s'écrire de plusieurs manières différentes. Choisir entre ces formes relève surtout des goûts personnels du programmeur.

6.9. Reprise des premiers exercices de TD.

Exercice 4

⇨ Ecrire une fonction Reverse (s) qui inverse une chaine s passée en paramètre. Ecrire un programme utilisant cette fonction.

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

void main()
{
    void Reverse (char*) ;
    char *chaine="Je suis content d\'être en Td de C";

    printf ("\n%s",chaine) ; /* avant */
    Reverse (chaine) ;   ;
    printf("\n%s",chaine) ;  /* après */
}

void Reverse (char *s)
{
    int c,i,j ;
    for (i=0, j=strlen(s) -1; i<j ; i++, j--) {
        c = *(s +i) ;
        *(s+i) = *(s+j) ;
        *(s+j) = c ;
    }
}
```

Exercice 6

⇨ Ecrire une fonction StrCat (t,s) qui ajoute une chaîne t à la fin d'une chaîne s. (s doit être suffisament grande). Ecrire un programme utilisant cette fonction.

```
#include <stdio.h>
void StrCat ( char * t , char* s )
{
    while ( *s ) s++ ;
    while (*s++ = *t++) ;
}
void main()
{   char source[80] ;
    char complement[10];
    printf("\nEntrez La source : ");
    scanf("%s",source) ;
    printf("\nEntrer la chaîne : ");
    scanf("%s",complement);
    StrCat (complement, source) ;
    printf("\nRésultat  : %s",source) ;}
```

6.10. Tableaux de pointeurs

⇨ Les pointeurs étant des informations comme les autres, ils peuvent être mis dans des tableaux.

Exemple : `char *Ptr_Char[7];`

⇨ Cette déclaration se lit comme : Ptr_Char est un tableau dont les éléments sont du type char*. Les 7 éléments de Ptr_Char sont donc des pointeurs vers des caractères

⇨ D'après ce qui a été vu précédemment cela signifie que chaque élément du tableau peut pointer vers une chaîne de caractères.

⇨ On conçoit qu'avec une telle représentation, une permutation de chaîne est très facile puisqu'il suffit de permuter les pointeurs correspondants.

⇨ De plus, les différentes chaînes peuvent avoir des tailles différentes, ce qui ne serait pas possible si l'on déclarait un tableau de caractères à 2 dimensions comme :

```
char Tab_Char[7][10]
```

⇨ On peut initialiser un tableau de pointeur de la façon suivante :

```
char *Ptr_Char[7] = {"Lundi", "Mardi", "Mercredi", "Jeudi","Vendredi", "Dimanche"};
```

et on pourrait l'utiliser ainsi :

```
for (i = 0; i<7; i++)
    printf(" Jour %d = %s\n",i, Ptr_Char[i]);
```

6.11. Pointeurs de fonctions

⇨ Une fonction ne peut être considérée comme une variable; cependant elle est implantée en mémoire et son adresse peut être définie comme l'adresse de la première instruction de son code.

⇨ Lorsque le traducteur reconnait, dans une expression, l'identificateur d'une fonction qui ne correspond pas à un appel de fonction (absence de parenthèses), il engendre alors l'adresse de cette fonction.

⇨ Il est donc possible de définir un pointeur sur une fonction, ce qui sera particulièrement utile lorsqu'on désirera passer une fonction en paramètre.

Exemple :

```
int (*f)();       /* pointeur de fonction retournant un entier */
```

- Ici (*f) signifie : f est un pointeur; *f définit l'objet pointé
- (*f)() signifie : l'objet pointé est une fonction
- int (*f)() signifie : l'objet pointé est une fonction qui retourne un entier

⇨ **ATTENTION !!!** à ne pas confondre :

`int (*f)()` avec : **`int *f ()`**

6.12. Allocation dynamique

⇨ La bibliothèque standard C dispose de 3 fonctions d'allocation : **malloc**, **calloc** et **realloc** qui ressemblent au new pascal, et une fonction de désallocation : free

⇨ Elles ont leur prototype dans stdlib.h :

- `void *malloc (size_t tob) ;`
- `void *calloc (size_t nob, size_t tob) ;`
- `void *realloc (void *ptr, size_t tob) ;`
- `void free (void *) ;`

⇨ Le type size_t est défini dans stdlib.h et est équivalent à unsigned int ou unsigned long int (ça dépend des systèmes).

⇨ tob représente une taille d'objet en octets et nob un nombre d'objets.

- ⇨ malloc renvoie un pointeur universel (void *) vers le début de la zone allouée de tob octets.
- ⇨ calloc renvoie un pointeur universel (void *) vers le début de la zone allouée de nob objets de taille tob octets (tableau dynamique).
- ⇨ realloc permet de modifier la taille d'une zone précédemment allouée.
- ⇨ Ces trois fonctions renvoient NULL si l'allocation a échoué.
- ⇨ free rend au système (désalloue) la zone pointée par le pointeur paramètre. Si ce pointeur a déjà été désalloué ou s'il ne correspond pas à une valeur allouée par malloc, calloc ou realloc, il y a une erreur.
- ⇨ Le comportement du système est alors imprévisible.
- ⇨ Exemple : Une fonction qui reçoit en argument un pointeur à initialiser par malloc et qui renvoie un booléen indiquant si l'allocation s'est bien passée.

```
int alloc (int **p) /* p est un pointeur de pointeur d'entier */
{
    *p=(int *) malloc (sizeof (int)) ;
    return *p != NULL ;
}
main ()
{...
    int *pp ;
    p=NULL ;
    if (alloc (&pp))   /* adresse du pointeur                    */
       *p=5 ; /* pp a été modifié par alloc                */
.....                  /*  il n'est plus à NULL                  */
}
```

7. Septième partie : Structures et Unions

⇨ On a déjà vu que le constructeur tableau permettait de regrouper plusieurs éléments. Toutefois, ceux-ci devaient être tous du même type.

⇨ Le constructeur structure permet de regrouper de éléments de types différents.

⇨ De plus, ce constructeur permet de définir des modèles d'objets alors que dans le cas d'un tableau les objets sont anonymes.

7.1. Déclaration de structure

Syntaxe :
```
struct identificateur
{
        liste de champs
};
```

Exemple :
```
struct Quidam
{
        char *Nom;
        int Age;
        float Taille;
};
```

⇨ Les composants d'une structure sont appelés champs de la structure.

⇨ Une telle déclaration définit un modèle d'objet. Elle n'engendre pas de réservation mémoire. Le modèle ainsi défini peut être utilisé dans une déclaration de variable.

⇨ Exemple : struct Quidam Michel, Anne; /* Déclare 2 objets */

⇨ Remarque : Dans une structure, tous les noms de champs doivent être distincts. Par contre rien n'empêche d'avoir 2 structures avec des noms de champs en commun : l'ambiguité sera levée par la présence du nom de la structure concernée.

7.2. Initialisation de structure

⇨ Une déclaration d'objet structure peut contenir une initialisation. Mais alors, la déclaration doit être globale ou statique.

⇨ La valeur initiale est constituée d'une liste éléments initiaux (1 par champ) placée entre accolades.

⇨ Exemple : *struct Quidam Jean = {"Dupont" , 35, 1.80};*

7.3. Accès à un champ

Syntaxe : *<ident_objet_struct>.<ident_champ>*

⇨ L'opérateur d'accès est le symbole "." (point) placé entre l'identificateur de la structure et l'identificateur du champ désigné.

⇨ Exemples :
```
Michel.Age = 45;...
T_19_25 = (Michel.Age >= 19) && (Michel.Age <= 25);
```

7.4. Utilisation des structures

⇨ Composition de structures

- Puisqu'une structure définit un type, ce type peut être utilisé dans une déclaration de variable comme on l'a vu en V.1, mais aussi dans la déclaration d'une autre structure comme type d'un de ses champs.
- Exemple :
```
struct Date
        { int Jour,Mois,Annee; };
```

```
struct Quidam
        {
          char *Nom,
                *Adresse;
          struct Date Naissance;
        };
struct Quidam Michel = {   "Dupont", "Rue Lamarck", {21,05,60}   };
```

On peut alors écrire :

```
if (Michel.Naissance.Jour == 20) ...
```

⇨ Sur les compilateurs C respectant la norme ansi, il est possible d'affecter l'intégralité des valeurs d'une structure à une seconde structure ayant impérativement la même composition.

⇨ Exemple :

⇨ struct bidon a , b ;

a=b est une instruction valide, qui recopie tous les champs de b dans les champs de a.

⇨ Attention la recopie est superficielle.

7.5. Tableaux de structures

⇨ Un modèle structure peut aussi apparaître comme type des éléments d'un tableau.

⇨ Exemple :

```
struct Entree
        {
                char *Symbole;
                int  Idef;
                char *Info;
        };
...
struct Entree Tds[20];       /* tableau de 20 structures */
Tds[12].Symbole = "Jean";
```

7.6. Structures et pointeurs

⇨ Le type associé à un pointeur peut être une structure.

⇨ Exemple :

```
struct Date *Ptr_Date;       /* Ptr_Date pointe sur des objets de type Date */
```

Il est alors possible d'utiliser ce pointeur de la façon suivante :

```
Ptr_Date->Jour = 12;         /* équivaut à (*Ptr_Date).Jour */
```

7.7. Passage de structures comme arguments de fonctions

⇨ La norme ansi a introduit le transfert direct d'une structure entière comme argument d'une fonction, ainsi que la remontée de cette structure depuis une fonction appelée via une instruction return.

⇨ Le passage de paramètres de type structure se fait par valeur.

⇨ Sur certains compilateur ancien on doit passer et retourner des pointeurs sur les structures.

7.8. Structures récursives

⇨ Ce genre de structures est fondamental en programmation car il permet d'implémenter la plupart des structures de données employées en informatique (files d'attente, arborescences, etc...)

⇨ Exemple :

```
struct Individu
        {
        char *Nom;
        int Age;
        struct Individu Pere, Mere;
        };
/* Cette définition est incorrecte : la taille de Pere et Mere n'est pas connue */

struct Individu
        {
        char *Nom;
        int Age;
        struct Individu *Pere, *Mere;
        };
```

7.9. Les Unions

⇨ Une déclaration d'union se présente comme une déclaration de structure. Mais, alors qu'un objet structuré est composé de la totalité de ses champs, un objet union est constitué d'un seul champ choisi parmi tous les champs définis.

⇨ Exemple :

```
union Int_Float
        {
        int i;
        float f;
        };

union Int_Float X;
```

⇨ A un instant donné, l'objet X contiendra un entier ou un flottant, mais pas les deux ! C'est au programmeur de connaître à tout instant le type de l'information contenue dans X.

⇨ L'accès à cette information se fait de façon analogue à l'accès à un champ de structure.

⇨ Le programmeur utilisera X.i s'il désire manipuler l'infomation de type int, sinon il utilisera X.f

⇨ Un objet de type union occupe une place égale à celle du plus grand de son champ. Ainsi si, par exemple, un int occupe 2 octets et un float 5 octets, alors l'objet X occupera 5 octets.

⇨ **ATTENTION !!!**

- ⇨ Ce constructeur peut être la source d'erreurs très difficiles à détecter.
- ⇨ Donc, à manier avec précaution et seulement en cas de besoin .

7.10. Schémas d'implantation

⇨ Le C permet de préciser l'implantation mémoire d'une structure. Ceci est particulièrement intéressant pour représenter des informations proches du niveau matériel comme par exemple le registre d'état du CPU.

⇨ Exemple :

```
struct Status_Register
        {
         unsigned E : 1;
         unsigned F : 1;
         unsigned H : 1;
         unsigned I : 1;  /* Mot d'état du 6809 */
         unsigned N : 1;
         unsigned Z : 1;
         unsigned V : 1;
         unsigned C : 1;
        };
```

⇨ Seuls des champs de type unsigned sont autorisés

⇨ Chaque nom de champ est suivi du symbole ":" et d'un entier représentant la taille en bits de ce champ. L'accès aux champs suit les mêmes règles que pour une structure classique. Cependant l'opérateur & (Référence à) ne peut être appliqué à un tel champ (il peut l'être sur la structure globale)

8. Huitième partie : Entrées / sorties

⇨ En C les E/S relèvent de la bibliothèque standard et pas du langage lui-même qui est

⇨ Le traitement des E/S repose sur 3 éléments :
- le type **FILE** défini dans stdio.h
- Trois variables de type **FILE *** :
  stdin : entrée standard,
  stdout : sortie standard,
  stderr : sortie standard d'erreur.
- Des fonctions précompilées : **getchar**, **putchar**, **scanf**, **printf**, ...

⇨ *FILE* est l'alias d'une structure dont l'utilisateur n'a pas besoin de connaître l'organisation.

⇨ Ce type n'est jamais utilisé en tant que tel, on utilise toujours le type *FILE* * que l'on appelle un **flux**.

⇨ Les troix flux standard *stdin*, *stdout*, *stderr* sont automatiquement ouverts au lancement de tout programme. Ils sont connectés par défaut à l'écran pour les sorties, au clavier

⇨ Ils peuvent être explicitement redirigés vers un fichier ou un périphérique.

8.1. Les E/S formatées : printf

⇨ prototype : *int printf(const char * format, identificateurs ....)*

⇨ *printf* convertit ses arguments d'après les spécifications de son format et écrit le résultat dans le flot de sortie standard *stdout*.

⇨ Elle retourne le nombre de caractères écrits ou une valeur négative en cas d'erreur.

⇨ La chaîne de caractères format contient deux types d'informations : des caractères ordinaires qui sont recopiés sur le flot de sortie, et des spécifications de conversions dont chacune provoque la conversion et l'impression des arguments suivants de printf.

⇨ Chaque spécification de conversion commence par % et se termine par un caractère de conversion. Entre les deux on peut placer dans l'ordre :
- **des drapeaux** qui modifient la spécification :
  - cadre l'argument à gauche
  + imprime systématiquement le signe du nombre
  espace : si le premier caractère n'est pas un signe, place un espace au

  0 : pour les conversion numériques, complète le début du champ par des 0.
- **Un nombre** qui précise la largeur minimum du champ d'impression.
- **Un point** qui sépare la largeur du nombre de décimales.
- **Un nombre** indiquant la précision
- **Une lettre** h (short ou unsigned short), l (long ou unsigned long) ou L (long double) qui modifie la largeur du champ

⇨ La largeur et la précision peuvent être données en paramètre à printf en mettant * à leur place. Les paramètres suivant le format seront alors convertis en entiers et

⇨ Les spécifications de conversions sont :

	Type de l'argument
d,i	int ; notation décimale signée

o	int ; notation octale non signée
x, X	int ; notation hexadécimale non signée
u	int ; notation décimale non signée
c	int ; un seul caractère après conversion en unsigned char
s	char * ; les caractères sont imprimés jusqu'à \0
f	double ; notation décimale de la forme [-]mmm.ddd la précision par
e, E	double ; notation exponentielle
g, G	double ; l'impression se fait suivant le format %e si l'exposant est inférieur a -4 ou supérieur a la précision, sinon %f
p	void * ; écrit l'argument sous forme de pointeurs
n	int * ; le nombre de caractères écrits jusqu'à présent par cet appel à printf est écrit dans l'argument.
%	affiche un %

8.2. Les E/S formatées : scanf

⇨ prototype : int scanf (const char * format, adresses ...)

⇨ le format contient :

- des espaces ou des caractères de tabulations qui sont ignorés
- des caractères ordinaires (différents de %), dont chacun est supp s'identifier au caractère suivant du flot d'entrée.
- des spécifications de conversion du même type que pour printf.

8.3. Les E/S : les fichiers

⇨ Il existe deux types de fonctions de manipulation de fichiers en C.

- Les fonctions de niveaux 1
- Les fonctions de niveaux 2.

⇨ Les fonctions de niveaux 1 sont des fonctions proches du système d'exploitation qui permettent un accès direct aux informations car elles ne sont pas bufferisées. Elles manipulent les informations sous forme binaire sans possibilités de formatage, et le fichier est identifié par un numéro (de type entier) fixé lors de l'ouverture du fichier.

⇨ Les fonctions de niveaux 2 sont basées sur les fonctions de niveaux 1, elles effectuent des E/S bufferisées, permettent une manipulation binaire ou formatée des informations. Le fichier est identifié par un flux (FILE *) qui contient des informations élaborées relatives au fichier : adresse du buffer, pointeur sur le buffer, numéro du fichier pour les fonctions de niveaux 1, etc....

8.4. E/S fichiers :Les fonctions de niveaux 1

⇨ Ouverture d'un fichier : prototype
*int open (char * nomfichier , int mode, int permissions)*

⇨ mode est une combinaison de

- O_CREAT : Le fichier est créé s'il n'existe pas
- O_RDONLY : Ouverture en lecture seule
- O_WRONLY : Ouverture en écriture seule
- O_BINARY : Mode non translaté (binaire)
- O_TEXT : Mode translaté (ascii)
- O_RDWR : Ouverture en lecture/écriture
- O_TRUNC : Si le fichier existe son contenu est détruit.

⇨ open renvoie un entier qui sera utilisé pour accéder au fichier par la suite. open renvoie -1 en cas d'erreur.

⇨ On considèrera que permissions vaut toujours 0 (pour nous)

⇨ Fermeture du fichier : close (int numérofichier)

⇨ Ecriture dans le fichier
*int write (int num_fic, char *adre, int nombre_octets)*

⇨ lecture dans le fichier
*int read (int num_fic, char * adre_stock, int nombre_octets)*
read renvoie le nombre d'octets transférés

⇨ Fin de fichier
eof (numéro_fichier)
  ⇨ renvoie -1 dès que la fin de fichier a été atteinte, 0 sinon.

⇨ Accès direct
lseek (numéro_fichier, déplacement, mode)
  ⇨ mode peut prendre les valeurs 0, 1, 2 selon que le déplacement doit être :
    0 : par rapport au début du fichier
    1 : par rapport à la position courante
    2 : par rapport à la fin du fichier.

⇨ Exemple : recopier l'entrée sur la sortie.

```
#include <stdio.h>
void main()
{   char tamp[BUFSIZ] ;
    int n;
    while ((n=read(stdin, tamp, BUFSIZ))>0)
       write(stdout, tamp, n);
}
```

8.5. E/S fichiers :Les fonctions de niveaux 2

⇨ Ouverture d'un fichier : prototype
*flux = FILE * **fopen** (char * nom_fic, char *mode)*
  ⇨ avec mode = *"r", "w", "a", "a+", "r+", "w+"*

⇨ Fermeture d'un fichier : fclose (FILE * flux)

⇨ Ecriture "brutale" :
*size_t **fwrite** (const void *ptr, size-t taille, size_t nobj, FILE * flux) ;*

⇨ Lecture "brutale" :
*size_t **fread** (void * ptr, size-t taille, size_t nobj, FILE * flux);*

⇨ Accès direct :
*int **fseek** (FILE *flux, long offset , int mode)*
  ⇨ *mode* peut prendre les valeurs 0, 1, 2 selon que le déplacement doit être :
    0 : par rapport au début du fichier
    1 : par rapport à la position courante
    2 : par rapport à la fin du fichier.

8.6. Les E/S formatées

⇨ ***fscanf** (FILE * flux, const char * format , liste_adresses)*

scanf est construit avec fscanf

⇨ ***fprintf**(FILE * flux, const char * format , liste_expressions)*

printf est construit avec fprintf

⇨ *int* ***fgetc*** *(FILE * flux)* => saisir un caractère dans le flux*

en fait c=getchar() <=> c=fgetc (stdin)

⇨ *int* ***fputc*** *(int c, FILE * flux)* => écrire un caractère dans le flux

en fait putchar(c) <=> fputc (stdin,c)

⇨ *char * **fgets** (char * chaîne, int lgmax, FILE * flux)* : lire une chaine dans le flux.

Cas particulier ***gets*** *(char * chaîne)*

⇨ *int* ***fputs*** *(char * chaîne, FILE *flux)* : écrire une chaine dans le flux.

Cas particulier puts *(char * chaîne)*

⇨ Autres fonctions d'E/S :

- ⇨ *int* ***ungetc****(int c, FILE * flux)* remet c dans le flot
- ⇨ *int* ***ftell*** *(FILE * flux)* donne la position courante dans le fichier ou -1 en cas d'erreur

9. Neuvième partie : Compléments

9.1. Compléments 1 : Le préprocesseur

⇨ Le préprocesseur est un programme standard qui effectue des modifications sur un texte source. Ce texte source peut être absolument n'importe quoi (du Pascal, du Fortran, une recette de cuisine, ...), car c'est un outil standard Unix. C'est essentiellement en C qu'il est utilisé.

⇨ Le préprocesseur modifie le source d'après les directives données par le programmeur, et introduites par le caractère #. Ces directives peuvent apparaître n'importe où dans le fichier, pas nécessairement au début.

⇨ La directive du préprocesseur commence par un # et se termine par la fin de ligne. Si la directive ne tient pas sur une seule ligne, on peut l'écrire sur plusieurs en terminant les premières lignes par \ qui annule le retour chariot.

9.1.1. Première utilisation : substitution de symboles

⇨ Syntaxe: *#define symbole chaîne*

⇨ Généralement et pour plus de clarté les symboles #-définis sont mis en majuscule (ce n'est pas obligatoire).

⇨ Le symbole est remplacé par la chaîne à chaque fois qu'il est rencontré dans le source.

⇨ La chaine considérée commence au premier caractère non blanc après le symbole, et se termine sur le premier caractère non blanc en allant vers la gauche à partir de la fin de ligne.

⇨ Avant la norme ansi le #define était la seule façon de définir des constantes

```
#define MAXTAB 100...
int tab[MAXTAB];...
if (i>MAXTAB)...
```

⇨ La substitution de symbole permet aussi de modifier l'aspect d'un programme source :

```
#define Loop for(;;){
#define EndLoop }
```

Ce qui peut donner :

```
Loop
    i++
    if (...)
            break;
    ...
EndLoop
```

⇨ **Danger** : Lisibilité du programme par les autres ???

9.1.2. Deuxième utilisation : Macro-instructions

⇨ Une définition de macro_instruction permet une substitution de texte paramétrée par des arguments.

```
#define symbole(a1,a2,....) chaine
```

⇨ Il n'y a pas d'espace entre le symbole et (

⇨ Lors de l'appel, symbole(x,y,...) est remplacé par la chaîne dans laquelle toutes les occurences des arguments formels a1, a2, sont remplacées par les arguments effectifs x, y, ...

⇨ *Exemple :*

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
avec cette définition la ligne :
x=MAX(58,x*y) sera remplacée par
x= ( (58) > (x*y) ? (58) : (x*y) )
```

⇨ Si on ne met pas de parenthèses sur les paramètres formels il peut y avoir des erreurs
⇨ Par exemple :

*#define carre(a) (a*a)*
....
x=carre(y+1) sera transformé en *x=(y+1*y+1)* soit
*x=2*y+1 !!!*

9.1.3. Troisième utilisation : Inclusion de fichier

⇨ Généralement les fichiers à inclure ont un nom qui se termine par .h (header) mais ce n'est pas obligatoire
⇨ ***#include <stdio.h>*** va chercher le fichier stdio.h dans le répertoire spécial contenant les fichiers header et l'incorpore au source.
⇨ ***#include "/user/dudule/monHeader.h"*** va chercher le fichier monHeader.h dans le répertoire /user/dudule et l'incorpore au source.

9.1.4. Quatrième utilisation : Compilation conditionnelle

⇨ Il existe 3 formes de tests possibles :
⇨ **Forme 1** : Tester si une expression est nulle

#if expression
lignes diverses
#endif

⇨ Si l'expression numérique constante est vraie (Non nulle) les lignes suivantes sont traitées par le préprocesseur sinon elles sont ignorées jusqu'au *#endif*

#define MAC 1
...
#if MAC
lignes diverses
#endif
...
#if SUN
lignes diverses
#endif
...
#if TEKTRO || HP
lignes diverses
#endif
...

⇨ **Formes 2 et 3** : Tester si un symbole a déjà été défini.

- Forme 2 : tester si un symbole est défini

  #ifdef bidule
  lignes diverses
  #endif

- Forme 3 : Tester si un symbole n'est pas défini

  #ifndef bidule
  lignes diverses
  #endif

⇨ C'est souvent utilisé pour éviter qu'un fichier header soit #inclus plusieurs fois.

Exemple : le fichier toto.h
*#ifndef _toto_*
*#define _toto_*
lignes diverses
#endif

9.1.5. Le préprocesseur : Opérateur

⇨ L'opérateur # ne peut s'employer que dans la définition d'une macro. Il permet de remplacer un argument formel par un argument effectif transformé automatiquement en

⇨ Exemples :

#define str(s) #s le préprocesseur remplacera alors
printf(str(bonjour)) par *printf("bonjour")*

Si on avait fait *#define str(s) "s",*
on aurait obtenu *printf("s") !!*

#define ipr(x) printf(#x "=%d", x)
Le préprocesseur remplace alors *ipr(i);* par
printf("i" "=%d",i); ce qui donne après concaténation
printf("i=%d", i);

9.1.6. Le préprocesseur : Opérateur

⇨ Cet opérateur ne peut s'employer que dans un #define (macro ou substitution de symbole).

⇨ Lorsque deux éléments sont séparés par ## ils sont concaténés pour former un nouveau symbole

⇨ Exemple :

#define JACOBONI tresfort
#define colle(a,b) a##b
colle(JACO,BONI) donnera *JACOBONI* puis *tresfort !*

⇨ Exemple plus intéressant :

#define trace(s,t) printf("x"#s"=%d, x"#t"=%s",x##s,x##t)
trace(1,2); sera transformé en
printf("x1=%d, x2=%s", x1,x2);

9.1.7. Exemple de fichier Header : def.h

```
#ifndef _def_
#define _def_
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
/*********************************************
    Amélioration malloc et calloc :
        Permet d'écrire
            int *p;
            p=Malloc(int) au lieu de
            p=(int *) malloc (sizeof(int))
        idem pour calloc
*********************************************/
#define Malloc(type)     (type *)malloc(sizeof(type))
#define Calloc(n,type)   (type *)calloc(n,sizeof(type))

/*********************************************
    Amélioration de la déclaration de structure

        structDEF (arbrebinint) {
            arbrebinint *fg;
            int valeur ;
            arbrebinint *fd;
```

```
        };
    au lieu de
        struct ARBREBININT;
        typedef struct ARBREBININT arbrebinint;
        struct ARBREBININT {
            arbrebinint *fg;
            int valeur ;
            arbrebinint *fd;
        };
**********************************************/
#define structdecl(type) struct _##type;\
    typedef struct _##type type
#define structdef(type)   struct _##type
#define structDEF(type)structdecl(type);structdef(type)
/**********************************************/
typedef enum { false, true} bool;
#endif
```

9.2. Compléments 2 : Arguments de la ligne de commande

⇨ Il est possible en C de récupérer les arguments passés sur la ligne de commande. Quand on appelle la fonction **main**, celle-ci reçoit trois paramètres :

⇨ Le premier est un entier qui représente le nombre d'arguments présents sur la ligne de commande. On l'appelle généralement **argc**.

⇨ Le second est un pointeur vers un tableau de chaînes de caractères qui contiennent les arguments proprement dits. On l'appelle généralement **argv**.

⇨ Exemple : soit le programme echo qui ne fait qu'afficher ses arguments.

```
echo        Bonjour     Maître      Jacoboni
argv[0]     argv[1]     argv[2]     argv[3]  et argc vaut 4.
```

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    while (--argc>0)
        printf("%s%s", *++argv, (argc>1) ? " " : "");
/* ou printf( (argc>1) ? "%s " : "%s", *++argv);  */
    printf("\n");
    return 0;
}
```

⇨ Par convention, argv[0] est le nom du programme exécutable et donc argc vaut au moins 1.

10. Dixième partie : Types de Données abstraits

10.1. Les types de données abstraits

10.1.1. définition 1 :

- un ensemble de données et d'opérations sur ces données

10.1.2. définition 2 :

- un ensemble de données organisé de sorte que les spécifications des objets et des opérations sur ces objets soient séparées de la représentation interne des objets de de la mise en oeuvre des opérations
- nés de préoccupation de génie logiciel
  - ♦ abstraction
  - ♦ encapsulation
  - ♦ vérification de types
- exemples :
  - ♦ le type entier muni des opérations + - % / < > = >= =< est un TDA
  - ♦ une pile muni des opérations initialiser, empiler, dépiler, consulter le sommet de pile est un TDA
- les TDA généralisent les types prédéfinis.

10.1.3. Définition 3 : Incarnation d'un TDA

- une incarnation (une réalisation, une mise en œuvre) d'un TDA est la structure de données particulière et la définition des opérations primitives dans un langage particulier
- on parle aussi de type de données concret
- remarque : définir le jeu de primitives peut s'avérer un problème délicat délicat
- exemple : on peut mettre en œuvre le TDA pile en utilisant en mémoire des cellules contiguës (tableau) ou des cellules chaînées
- programmation à l'aide des TDA:
  - ♦ les programmeurs ont deux casquettes
    - ◊ le **concepteur** du TDA qui met en œuvre les primitives et doit connaître la représentation interne adoptée
    - ◊ **l'utilisateur** du TDA qui ne connaît que les services (les opérations) et n'accèdent jamais à la représentation interne

10.1.4. Avantages des TDA

- écriture de programmes en couches :
  - ♦ la couche supérieure dans les termes du domaine de problèmes
  - ♦ non avec les détails du langage de programmation
- séparation claire des offres de service et du codage
- facilité de compréhension et d'utilisation
- prise en compte de types complexes
- briques d'une structuration modulaire rigoureuse

10.1.5. Inconvénients des TDA

- l'utilisateur d'un TDA connaît les services mais ne connaît pas leur coût
- le concepteur du TDA connaît le coût des services mais ne connaît pas leurs conditions d'utilisation
- le choix des primitives est quelque fois difficile à faire

10.2. Le TDA Liste

10.2.1. définitions :

- une **liste** est une suite finie (éventuellement vide) d'éléments
- liste **homogène** : tous les éléments sont du même type
- liste **hétérogène** : les éléments sont de type différents

10.2.2. Hypothèses

- dans la suite nous interessons aux listes homogènes
- tous les éléments appartiennent à un TDA ELEMENT qui possède un élément vide et sur lequel on peut :
  - ♦ tester si deux éléments sont identiques (prédicat ElementIdentique)
  - ♦ affecter dans un élément un autre élément (ElementAffecter)
  - ♦ saisir un élément (ElementSaisir)
  - ♦ afficher un élément (ElementAfficher)
- on dispose d'un TDA POSITION permettant de repérer un élément dans la liste sur lequel on peut :
  - ♦ tester si deux positions sont égales (PositionIdentique)
  - ♦ affecter dans une position une autre position (PositionAffecter)
- Conventions de nommage:
  - ♦ nous préfixons l'opération par le nom du TDA pour distinguer par exemple ListeSaisir et ElementSaisir
  - ♦ pas d'utilisation du souligné mais mots séparés par des majuscules
- **ELEMENT** est d'un type quelconque (entier, caractère, chaîne ou tout type utilisateur muni des 4 opérations (affecter, tester l'égalité, saisir et afficher)
- **POSITION** peut être
  - ♦ le rang de l'élément : si l = (a b c d) le rang de a est 1 celui de d est 4
  - ♦ ou un pointeur sur la cellule contenant un élément (ou sur celle d'avant)
- Lors de l'incarnation du TDA, une POSITION sera un entier ou un pointeur (mais ici ce n'est pas le problème)

10.2.3. Signature du TDA LISTE

- **Élément particulier** : liste vide
- **Utilise** : ELEMENT, POSITION
- **Fonctions primitives :**
  - ♦ *opérations d'allocation, libération de mémoire*
    ListeCreer, ListeDetruire
  - ♦ *opérations de gestion*
    ListeAccèder, ListeInsérer, ListeSupprimer
  - ♦ *opérations de parcours*
    ListePremier, ListeSuivant, ListeSentinelle
  - ♦ *test*
    ListeVide
- **ListeCréer :** pas d'arguments ; alloue dynamiquement de la mémoire pour une liste vide et retourne cette liste
- **ListeDetruire :** un argument (la liste à détruire) ; libère la mémoire allouée dynamiquement pour la liste
- **ListeVide :** un argument (la liste à tester) ; retourne vrai si la liste est vide et faux sinon
- **ListeAccèder :** deux arguments (une position p et une liste l) ; retourne l'élément à la position p dans l (ou élément vide si l est vide ou p est erronée)

- **ListeInsérer :** trois arguments (un élément x, une position p et une liste l); modifie la liste l en insérant x à la position p dans l. retourne vrai si l'insertion

- **ListeSupprimer :** deux arguments (une position p et une liste l) ; supprime l'élément à la position p dans la liste l et retourne vrai si la suppression s''est

- **ListeSuivant :** deux arguments (une position p et une liste l) ; retourne la position qui suit p dans la liste (ou la sentinelle si la liste est vide ou si p n'est pas une position valide)
- **ListePremier :** un argument (la liste l) et retourne la première position dans la liste
- **ListeSentinelle :** un argument (la liste l) et retourne la position qui suit celle du dernier élément de la liste
- **Remarque**: ****une sentinelle est une astuce de programmation qui consiste à introduire un élément factice pour faciliter les recherches. La sentinelle est une position valide mais n'est la position d'aucun élément de la liste. Lorsqu'une liste est vide Premier(l) = Sentinelle(l).
- **Remarque sur la remarque** : Vous voyez que déjà on triche un petit peu sur les principes ; au niveau des primitives on ne devrait pas vous parler déjà de réalisation. Mais c'est une petite entorse, les sentinelles c'est le b-a-ba !

10.2.4. Opérations classiques sur les listes

- à l'aide de ces quelques primitives, on peut exprimer les opérations classiques sur les listes, et ce,
  - ♦ sans se soucier de la réalisation concrète des listes
  - ♦ sans manipuler ni pointeurs, ni tableaux, ni faux curseur
  - ♦ sans mettre les mains dans le cambouis
- deux exemples : ListePrecedent et ListeLocaliser

10.2.5.ListePrecedent

- signature:
  - ♦ deux arguments : une position p et une liste l
  - ♦ retourne la position précédent p dans l si elle existe, sinon retourne la sentinelle
- algorithme informel
  - ♦ parcourir l en faisant évoluer 2 positions (la position courante et celle d'avant)
  - ♦ arrêt du parcours quand la position courante est égale à p ou à la sentinelle
  - ♦ retourne la position avant ou la sentinelle
- Codage en C

```
***
POSITION ListePrecedent(POSITION p, LISTE l) {
    POSITION avant, courant, fin ;
    PositionAffecter(&avant, ListePremier(l)) ;
    PositionAffecter(&courant, ListeSuivant (avant, l)) ;
    PositionAffecter(&fin, ListeSentinelle(l)) ;
    while ( ! PositionIdentique(p, courant) && ! PositionIdentique(courant, fin)){
        PositionAffecter(&avant, courant) ;
        PositionAffecter(&courant, ListeSuivant(p, l)) ;
    }
    return (PositionIdentique(courant, p) ? avant : fin) ;
}
```

Remarque dans la pratique *: POSITION est un entier ou un pointeur, pour alléger les notations, nous noterons*

PositionIdentique ==

PositionAffecter =

```
***
POSITION ListePrecedent (POSITION p, LISTE L) {
    POSITION avant, courant, fin ;
    avant = ListePremier (L) ;
    courant = ListeSuivant (avant, L) ;
    fin = ListeSentinelle(L) ;
    for(;p !=courant && courant !=fin; avant=courant,
            courant=ListeSuivant(courant,L));
    return (courant == p ? avant : fin);
}
```

10.2.6. ListeLocaliser

- signature
  - ♦ deux arguments : un élément x et une liste l
  - ♦ retourne la première position où on rencontre x dans l et sinon retourne la sentinelle
- algorithme informel de recherche linéaire
  - ♦ parcourir l avec une position courante
  - ♦ arrêt quand l'élément courant est égal à x ou quand on est arrivé à la fin de la liste
  - ♦ retourner la position courante ou la sentinnelle
- Codage en C

```
****
POSITION ListeLocaliser (ELEMENT x, LISTE L) {
    POSITION p, fin ;
    p = ListePremier (L) ;
    fin = ListeSentinelle(L) ;

    for (; p != fin && !ElementIdentique(x, ListeAcceder(p, L)) ;)
        p = ListeSuivant (p,L) ;
    return p ;
}
```

10.2.7.Algorithmes abstraits sur le TDA LISTE

- algorithmes utilisant les opérations sur les listes
- exemple : ListePurger qui élimine les répétitions dans une liste
- signature : trois signatures sont possibles
  - ♦ ListePurger modifie physiquement la liste initiale (qui est donc perdue)
  - ♦ ListePurger retourne le résultat de la purge dans un argument
  - ♦ ListePurger retourne une copie de la liste initiale copie où les répétitions

- Remarques :
  - ♦ dans les deux premières versions, pour que le paramètre soit modifié physiquement par la fonction ListePurger il faut qu'une liste soit une adresse, un pointeur sur quelque chose,

    void ListePurger (LISTE l) modification physique de l ssi le type liste est un pointeur sur une structure

    void ListePurger (LISTE l, LISTE ll) modification physique de ll ssi idem
  - ♦ pour la solution 3 : la liste résultat est créée localement à la fonction listePurger ; pour qu'elle puisse être retournée il faut que la liste soit allouée dynamiquement dans la fonction ; qui la libérera ? Problèmes de

```
LISTE ListePurger (LISTE L) {
    LISTE LL ;
    ELEMENT x ;
    POSITION p, q, fin ;
    LL = ListeCreer() ;

    fin = ListeSentinelle (L) ;
    p = ListePremier (L); q =  ListePremier (L);
    for ( ; p != fin ; p = ListeSuivant(p, L) ){
            x = ListeAcceder(p, L) ;
       if (ListeLocaliser(x, LL) == ListeSentinelle(LL)) {
            ListeInserer (x, q, LL) ;
            q = ListeSuivant(q, LL);
            }
     }
    return LL ;
    /* attention retour d'un pointeur déclaré dans la fonction
     La zone pointée a-t-elle été allouée dynamiquement (sinon pointeur fou) ? Si
    oui, Qui la libérera? (fuites de mémoires*/
}
```

- algorithme informel pour la solution 2

parcourir ll du début à la fin
si l'element courant x n'appartient pas à l l'inserer
(au début, à la fin ?) dans l (l est supposée vide au départ)

10.3. Réalisations du TDA LISTE

réaliser le TDA consiste à

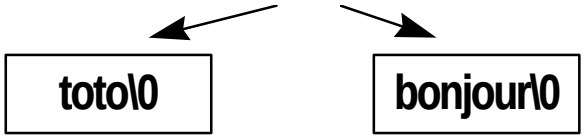
- définir le TDA ELEMENT
- choisir une structure de donnée pour représenter une liste (cellules contiguës ou cellules chaînées) pour la représentation interne de la liste
- définir les primitives dans un langage de programmation

le TDA ELEMENT peut être

- un objet (stockage direct)
- une adresse (stockage indirect)

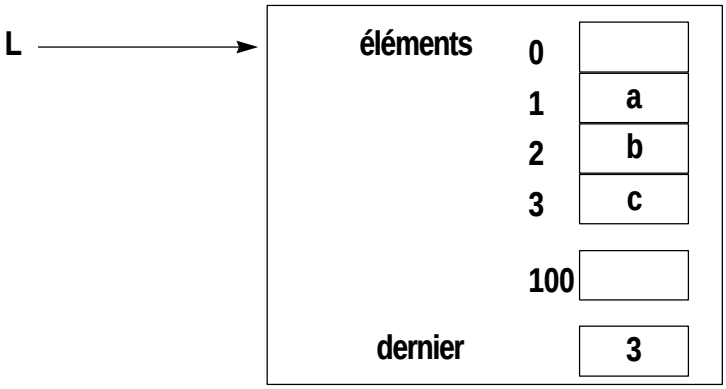
• exemples:

- stockage direct : l = (1 2 3 4)
- stockage indirect : l = (ad1 ad2)



10.3.1.Mise en œuvre des listes par cellules contiguës

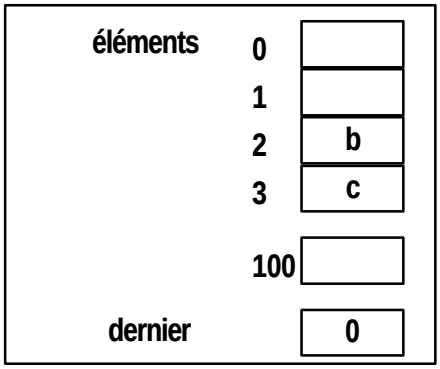
- plusieurs représentations sont possibles nous choisissons : une liste est un pointeur sur une structure à deux champs
  - un tableau qui contient les éléments de la liste
  - un entier indiquant l'indice du dernier élément de la liste



ANALYSE

• Une position est un entier : l'indice de l'élément dans le tableau
• Le premier élément de la liste est dans la case n°1 du tableau (pas dans la case 0)
• La sentinelle est la position qui suit celle du dernier (ici 4)
• ****

liste est vide si le dernier élément du tableau est 0
quand la liste est vide :
la position du premier élément de la liste est 1
la sentinelle est aussi 1



PRIMITIVES

• **ListeCréer** : allouer de la mémoire pour une telle structure et si l'allocation réussit mettre à 0 le champ dernier avant de retourner l'adresse de la structure
• **ListeDetruire** : il suffit d'appeler free sur la liste L
• Les **fonctions de parcours et d'accès** sont immédiates

• Les **fonctions de gestion** :
ListeInserer : pour insérer à la position p il faut commencer par décaler à partir du dernier élément jusqu'au pième ; puis on insère dans la pième case du tableau et on incrémente dernier
ListeSupprimer : consiste à tasser à partir de la position p jusqu'au dernier et à décrementer dernier

COMPLEXITÉ:
****

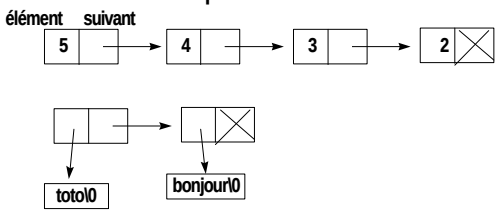
- ListeInserer et ListeSupprimer sont au pire des cas en o(n) (de l'ordre de n si n est le nombre d'éléments dans la liste) à cause de décalages
- toutes les autres opérations sont en o(1) (en temps constant)
- insérer et supprimer des éléments en fin de liste est en o(1) puisqu'alors on n'a pas besoin de décaler

10.3.2.Mise en œuvre des listes par cellules chaînées

- Cellules chaînées

une cellule est composée de deux champs :

- ♦ élément, contenant un élément de la liste
- ♦ suivant, contenant un pointeur sur une autre cellule



♦ Les cellules ne sont pas rangés séquentiellement en mémoire ; d'une exécution à l'autre leur localisation peut changer

♦ Une position est un pointeur sur une cellule

♦ On a donc les déclarations suivantes :

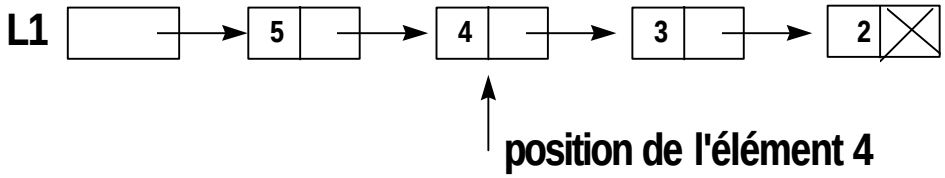
```
typedef struct cell{
        ELEMENT element;
        struct cell * suivant;
        } cellule, *LISTE;
typedef LISTE POSITION;
```

10.3.3. Listes simplement chaînées sans en-tête

- une liste est un pointeur sur la cellule qui contient le premier élément de la liste
- la position d'un élément est le pointeur sur la cellule qui contient cet élément
- la liste vide est le pointeur NULL
- inconvénients en C :

♦ problèmes lors de l'insertion ou de la suppression d'un élément au début de la liste

♦ suppression d'un élément en o(n)



- Problème 1 : quand on veut insérer ou supprimer au début de la liste

♦ ce n'est pas pareil qu'au milieu : donc à chaque fois il faut tester (si p = Premier(L)...)

♦ de plus

♦ on modifie physiquement la liste abstraite (OK)

♦ on modifie le pointeur sur la première cellule ; donc il faut passer un pointeur sur la liste en paramètre à ces procédures

♦ ListeInserer(x, p, &L) aïe aïe_ les bugs prévisibles

♦

♦ Exemple : insérer 42 au début de la liste L1

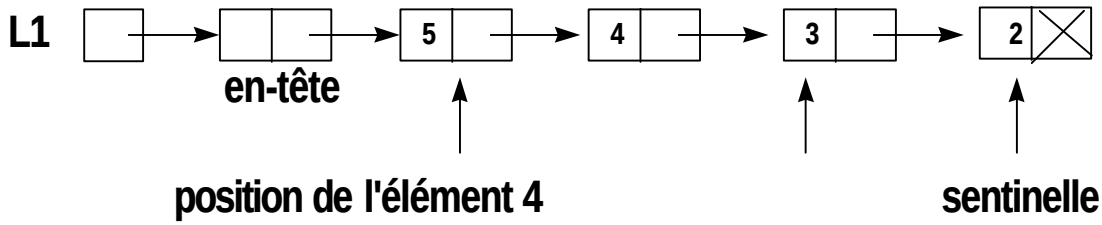
```
***
1/ créer une cellule
cell = (cellule *) malloc (sizeof(cellule))
      (et tester si tout va bien)
2/ initialiser la cellule avec 42 et un pointeur sur la cellule qui contient 5
ElementAffecter(&cell->element, x) ;
cell->suivant = p ->suivant ;
3/ accrocher la cellule créer à la liste : modifier le pointeur L1 qui doit donc être

* ptrl = cell ;
bool ListeInserer(ELEMENT x, POSITION p, LISTE * ptrl)
```

- Problème 2 : pour supprimer l'élément à la position p il faut d'abord trouver le précédent (recherche en o(n) dans le pire des cas) avant de pouvoir raccrocher la cellule suivante

10.3.4. Listes simplement chaînées avec en-tête

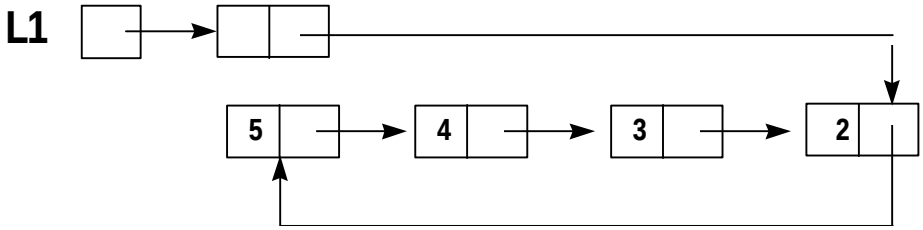
- une cellule d'en-tête ne contient pas d'élément et pointe sur la cellule qui contient le premier élément de la liste
- une liste est un pointeur sur la cellule d'en-tête

- une liste vide ne contient que sa cellule d'en-tête
- avantages : les insertions et suppressions sont en o(1)



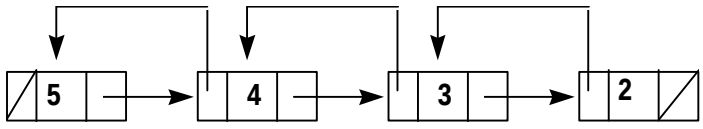
10.3.5. Listes simplement chaînées circulaires

- une liste est un pointeur sur une cellule d'en-tête
- la cellule d'en-tête pointe sur la cellule qui contient le dernier élément et celle-ci pointe sur le premier élément de la liste
- avantages
  - ♦ toutes les opérations sont en o(1) : insertion au début et à la fin, suppression au début. représentation utilisée pour représenter des files on entre à la fin (en o(1)) et on sort au début (en o(1) aussi)
  - ♦ sauf suppression en fin de liste (et évidemment la recherche linéaire...) en o(n)



10.3.6.Listes doublement chaînées

- pour parcourir facilement la liste dans les deux sens, on utilise des cellules qui gèrent deux pointeurs :
  - ♦ un pointeur avant : champ précédent de la cellule
  - ♦ un pointeur arrière : champ suivant de la cellule
- Les listes doublement chaînées peuvent être
  - ♦ avec ou sans en-tête
  - ♦ circulaires ou non



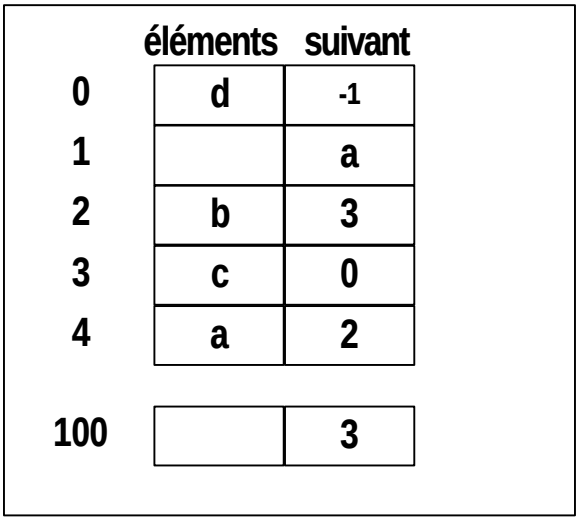
10.3.7.Mise en œuvre des listes par curseurs (faux-pointeurs)

- idée : simuler les pointeurs avec un tableau qui peut contenir plusieurs listes
- on dispose d'un grand tableau (simulant la mémoire, le tas) de cellules dont le premier champ contient l'élément et le deuxième champ l'indice dans le tableau du suivant
- une liste est alors l'indice du premier élément de la liste
- Nécessite de gérer les cellules disponibles

(a b c d) est représentée par

l = 4

l = 52



10.4. Comparaison des réalisations

Rappel sur la complexité

- o(1) : temps d'exécution constant (indépendant du nombre d'éléments de la liste)
- o(log n) : complexité logarithmique (temps d'exécution proportionnel au log du
- o(n) : complexité linéaire (temps d'exécution proportionnel au nombre
- o(n log n) : complexité de l'ordre de n log n
- o(n^2): complexité quadratique
- o(e^n) : complexité exponentielle

réalisations par cellules contiguës

- avantages
  - ♦ simples à programmer
  - ♦ facilite les opérations insérer en fin, longueur, précédent, sentinelle en o(1)
  - ♦ intéressant si les listes ont une taille qui varie peu
- inconvénients
  - ♦ nécessite de connaître à l'avance la taille maximum de la liste
  - ♦ coûteux en mémoire si on a des listes de taille très variable
  - ♦ rend coûteuses les opérations d'insertion et de suppression (en o(n))

réalisations par cellules chaînées

- avantages
  - ♦ économise la mémoire pour des listes de taille très variables
  - ♦ facilite les opérations insérer et supprimer en o(1)
  - ♦ intéressant pour les listes où on fait beaucoup d'insertion et de suppression
- inconvénients
  - ♦ risque de mauvaise manipulation sur les pointeurs (contrôle de la validité des positions qui augmente la complexité ou fragilise le TDA)
  - ♦ coûteux en mémoire si beaucoup de grosses listes (un pointeur par cellules)
  - ♦ rend coûteuses les opérations longueur, précédent, sentinelle, insérer en fin (en o(n))

10.5. Conclusion

- Avantages du TDA :
  - ♦ algorithme abstrait indépendant des choix de réalisation
- Inconvénients du TDA :
  - ♦ pour certaines opérations on peut gagner du temps en mettant les mains dans le cambouis (i.e. en utilisant une représentation adéquate).
- Règle empirique :
  - ♦ Utilisez les algorithmes abstraits puis pour les fonctions importantes optimisez les si besoin en en faisant des primitives.

10.6. Listing de la réalisation du TDA Liste par cellules contigües

- **Le TDA ELEMENT**

```
/*****************************************************************
*       Fichier :       ELTPRIM.H
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :Delozanne, Futtersack
*       Contenu :       Déclaration des primitives du TDA ELEMENT.
*
*****************************************************************/

#ifndef _ELTPRIM_H              /* pour l'inclusion conditionnelle */
#define  _ELTPRIM_H

/* inclusion de la déclaration du type concret ELEMENT */
#include "eltsdd.h"

/*Déclaration des types auxiliaires : bool */
typedef enum {FAUX, VRAI} bool ;

/* Déclaration des primitives du TDA ELEMENT */

void ElementLire(ELEMENT *);
void ElementAfficher(ELEMENT);
ELEMENT ElementAffecter(ELEMENT*, ELEMENT);
      /* affecte le deuxieme argument dans le premier
      qui est donc modifié et passé par adresse */
bool ElementIdentique(ELEMENT, ELEMENT);
      /* retourne vrai si les deux arguments sont identiques */

#endif
```

- **LE TDA LISTE**

```
/*****************************************************************
*       Fichier :       LSTPRIM.H
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :          Delozanne, Futtersack
*       Contenu :       Déclaration des primitives du TDA LISTE
*                       (correspondant au cours 1).
*****************************************************************/
#ifndef _LSTPRIM_H              /* pour l'inclusion conditionnelle */
#define  _LSTPRIM_H

#include "LSTSDd.H"      /* inclusion du fichier où est indiquée la structure de
      donnée retenue pour réaliser le TDA LISTE

/***********************************/
/* Déclaration des primitives du TDA LISTE*/
```

```
LISTE ListeCreer(void);
      /* crée et retourne une liste vide en lui allouant de la mémoire dynamique*/
bool ListeVide (LISTE);
      /* teste si la liste est vide */
POSITION ListeSentinelle(LISTE);
      /* retourne la position qui suit la position du dernier élément de la liste*/
POSITION ListePremier(LISTE);
      /* retourne la première position de la liste si la liste est non vide
      ou ListeSentinelle */
POSITION ListeSuivant(POSITION,LISTE);
      /* retourne la position qui suit la position paramètre dans la liste si la liste est
      non vide
      ou ListeSentinelle */
bool ListeInserer (ELEMENT,POSITION,LISTE);
      /* modifie la liste en insérant l'élément à la position ;
      retourne faux si la liste est pleine ou si la position est mauvaise */
bool ListeSupprimer (POSITION, LISTE);
      /*  supprime de la liste l'élément passé en paramètre ;
      retourne faux si la liste est vide ou si la position est mauvaise */
ELEMENT ListeAcceder(POSITION,LISTE);
      /*retourne l'élément à la position p dans la liste, sans modifier la liste
      retourne l'élément vide si la liste est vide ou si la position est mauvaise */
void ListeDetruire(LISTE);
      /* libére la mémoire dynamique allouée pour la liste */
#endif
```

- **Réalisation du TDA ELEMENT PAR DES ENTIERS**

```
/*****************************************************************
*       Fichier :       ELTSDD.H
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :Delozanne, Futtersack
*       Contenu :       Inclusion du fichier où est déclarée la structure de données
*                       adoptée pour réaliser le TDA ELEMENT.
*
*****************************************************************/

/* ce fichier sera modifié quand on voudra changer la représentation adoptée */

#ifndef _ELTSDD_H               /* pour l'inclusion conditionnelle */
#define _ELTSDD_H

#include "ELTINT.H"     /* inclusion du fichier où est déclarée la structure de
      donnée retenue pour réaliser le TDA ELEMENT */

#endif

/*****************************************************************
*       Fichier :       ELTINT.H
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :          Delozanne, Futtersack
*       Contenu :       Déclaration de la structure de données adoptée
*                       pour une réalisation du TDA ELEMENT.
*
*****************************************************************/

#ifndef _ELTINT_H               /* pour l'inclusion conditionnelle */
#define  _ELTINT_H

/* Déclaration d'un type concret ELEMENT
```

```
- un élément est dans ce fichier de type int ; par convention l'élément vide est
      32767;
 le stockage est direct*/

/* plus généralement pour un stockage direct un élément de type simple ou "pas
      trop gros"
par exemple : un nombre complexe, un point */

typedef int ELEMENT;

#define ELEMENT_VIDE 32767

#endif

/*****************************************************************
*       Fichier :       ELTINT.C
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :          Delozanne, Futtersack
*       Contenu :       Définition des primitives pour une réalisatioon
*                       par des entiers du TDA ELEMENT.
*
*****************************************************************/

/* définition des primitives du type concret ELEMENT entier */

#include "ELTPRIM.H"
#include <stdio.h>

void ElementLire(int * i) {
      scanf("%d",i);
}

void ElementAfficher(ELEMENT elt)     {
      printf(" %d  ",elt);
}

ELEMENT ElementAffecter(ELEMENT * e1, ELEMENT e2)   {
      return  *e1 = e2  ;
}

bool ElementIdentique(ELEMENT e1, ELEMENT e2)  {
      /* retourne vrai si les deux arguments sont identiques */
      return       e1 == e2 ;
}
```

- **Réalisation du TDA LISTE par tableaux**

```
/*****************************************************************
*       Fichier :       LSTSDD.H
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :          Delozanne, Futtersack
*       Contenu :       Inclusion du fichier où est déclarée la structure de données
*                       adoptée pour réaliser le TDA LISTE.
*
*****************************************************************/

/* ce fichier sera modifié quand on voudra changer la représentation adoptée*/

#ifndef _LSTSDD_H               /* pour l'inclusion conditionnelle */
#define _LSTSDD_H
```

```
#include "LSTTAB.H"       /* inclusion du fichier où est déclarée la structure de
      donnée retenue pour réaliser le TDA LISTE */

#endif

/*****************************************************************
*       Fichier :       LSTTAB.H
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :          Delozanne , Futtersack
*       Contenu :       Déclaration de la strucure de données
*                       pour réaliser le TDA LISTE PAR TABLEAU
*                       (correspondant au cours 1).
*
*****************************************************************/

#ifndef _LSTTAB_H               /* pour l'inclusion conditionnelle */
#define  _LSTTAB_H


#include "ELTPRIM.H"    /* inclusion du fichier des primitives du TDA ELEMENT*/


/***********************************/
/* déclaration de la strucure de données adoptée pour
une réalisation par tableau du TDA LISTE
Rappel :
- une liste est un pointeur sur une structure de données fondamentale (SDF)
- si la liste L est vide, le premier élement de la liste est la sentinelle*/

/* une liste est un pointeur sur une structure composée d'un tableau
et du nombre d'élements significatifs du tableau*/

#define LongMax 100   /* longueur maximale d'une liste */

typedef int POSITION ;

typedef struct {
      ELEMENT elements[LongMax];
      int dernier;
} laStruct,*LISTE;

#endif

/*****************************************************************
*       Fichier :       LISTTAB.C
*       Format :        Source C
*       Version  :      19/9/96
*       Programmeurs :          Delozanne, Futtersack
*       Contenu :       Définition des primitives pour une réalisatioon
*                       par tableau du TDA LISTE.
*
*****************************************************************/

/* définition des primitives du type concret LISTE par tableau*/

#include "LSTPRIM.H"

#include <stdlib.h>
#include <stdio.h>
/* Implantation des primitives du TAD LISTE dans une réalisation par tableau*/
```