Ministère de l'enseignement supérieur et de la recherche scientifique
* * * * * * * * * * * * * * * * * * *
Université de Carthage
* * * * * * * * * * * * * * * * * * *
Institut Supérieur des Sciences Appliquées et de Technologie
-- Mateur --
* * * * * * * * * * * * * * * * * * *
Département Informatique & Télécommunications

Support de cours
Programmation en langage C

Destiné aux étudiants des classes 1ères année
Licence Appliquée en Informatique Industrielle (L.A.I.I)

Mr Radhi Yazidi (P.T.C)

radhi.yazidi@gmail.com
https://sites.google.com/site/radhiyazidi

Année Universitaire 2012/2013

Fiche Matière

- Domaine : Sciences et Technologies
- Mention : Informatique Industrielle (I.I)
- Parcours : Maintenance des systèmes informatique
- Semestre : Semestre 1
- Unité d'enseignement : Algorithmique et programmation
- Pré-requis : Algorithmique (niveau baccalauréat)
- Volume horaire : 21h TD, 21h TP
- Programme officiel : Structure générale d'un programme en langage C ; Survol du langage C ; Présentation d'un environnement de développement ; Etude approfondie du langage C et des bibliothèques classiques au travers d'une série de TP ciblés (fonctions et modes de passage de paramètres, diverses structures de données).

Table des matières

Chapitre 1 : Les règles d'écriture

1. Introduction

```
#include <stdio.h>
#include <conio.h>
void main()
      {clrscr ( ) ;
       printf ("Bonjour");
      }
```

Ce programme permet d'afficher le mot « Bonjour », cette opération est réalisée par la fonction « **printf** » qui est une fonction standard du langage C, pour pouvoir l'utiliser, il faut inclure le fichier contenant sa déclaration, ce fichier est nommé « **stdio.h** ».
printf : permet d'afficher du texte.
clrscr : permet d'effacer l'écran.

Exemple de bibliothèques
math.h, conio.h, ctype.h, stdio.h...
Lorsque le compilateur C rencontre le mot clrscr, il regarde dans chacun des fichiers d'entête ".h" déclaré par l'instruction #include, si ce mot y est défini. Il trouve celui-ci dans la bibliothèque conio.h et remplace donc ce mot par le code qui lui est associé au moment de la compilation. A l'inverse, s'il ne le trouve pas, celui-ci émettra une erreur de syntaxe.

La fonction '**main**' est la fonction principale du programme, elle doit être présente dans tout programme C. Les accolades jouent le rôle des instructions début et fin en algorithmique.

2. Structure générale simplifiée d'un programme C

```
#include <bib1.h>
#include <bib2.h>
#include <….h>
void main ( )
      { Déclaration des variables ;
        Instr 1 ;
        Instr 2 ;
         … … ;
         … … ;
      }
```

3. Les mots clés

Ce sont des mots réservés par le langage à l'usage bien défini. Ils ne peuvent ni changer de signification ni être utilisés comme des identificateurs.

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

4. Les identificateurs

Ils servent à désigner les objets (fonctions, variables, constantes...) manipulés par le programme. Ils se présentent sous forme de chaînes alphanumériques de taille quelconque dans les limites acceptables par le compilateur.
En général, la taille des identificateurs ne dépasse pas 32 caractères, mais certains nouveaux compilateurs permettent d'aller jusqu'à 256 caractères. Au-delà de cette taille les caractères ne sont pas pris en considération.

1- Un identificateur doit commencer par une lettre.
2- Les caractères accentués ne sont pas acceptés.
3- Le caractère souligné (underscore '_') est considéré comme une lettre.
4- Un identificateur doit être différent des mots clés du langage.
5- Les majuscules et les minuscules sont considérées comme des caractères différents.

5. Les commentaires

Ils sont fondamentaux pour la compréhension des programmes. C'est pourquoi, il est conseillé de les utiliser autant que possible. Ils se présentent comme des portions de texte et ne sont pas pris en considération lors de la compilation. Ils sont insérés par les délimitations '/*' pour le début et '*/' pour la fin.

6. Le format libre

Le langage C autorise une mise en page libre. Ainsi une instruction peut s'étendre sur plusieurs lignes pourvue qu'elle se termine par ' ; '. De même une ligne peut comporter plusieurs instructions.

Chapitre 2 : Les notions de base

1. Les types de base

Les types de base du langage C sont :
int : pour les entiers standards.
float : pour définir les nombres flottants standards (réel).
double : permet de définir des nombres flottants en double précision.
char : pour les caractères.

Définition	Description	Nombre d'octets
int	entier standard	2
float	réel	4
double	Double précision	8
char	caractère	1

En plus de ces types, on peut définir des types dérivés qui sont obtenues en plaçant des spécifications supplémentaires :
long : il est utilisé pour définir des entiers ou des réelles de grande taille. 'long' peut s'appliquer à 'int' et à 'double'. Lorsqu'il est utilisé tout seul alors il désigne par défaut un grand entier.
short : il permet de manipuler les entiers les plus petits. Il peut être utilisé avec 'int' ou tout seul.
unsigned : (non signé) il se place devant les entiers ou les caractères qui doivent être considérés non signés. Il peut être utilisé seul, il désigne un entier non signé.

Remarques

Le type caractère, (char), peut servir pour le stockage des entiers compris entre (–128...127).
Les caractères en langage C sont stockés en tant qu'entier à travers leurs codes ASCII.

2. La déclaration des variables

Toute variable dans un programme C doit être déclarée. Elle possède un nom (identificateur définit suivant les règles mentionnées dans le chapitre I) et un type. Elle contient une information bien précise. Lors de l'exécution du programme, une zone mémoire dont la taille dépend du type lui sera réservée et contiendra cette information.

Syntaxe

Type identificateur ;

Exemples

float moyenne ;
int a, b, c ;

Initialisation des variables

Une valeur initiale peut être affectée à une variable au moment de sa déclaration.
int i = 5 ;
La valeur d'initialisation d'une variable peut être le résultat d'une expression.
int i = 3*6 ;

3. Les constantes

Une constante est une donnée dont la valeur reste fixe tout au long du programme. Elle peut être un nombre, un caractère ou une chaîne de caractère.

a) Les constantes entières

Elles peuvent utiliser plusieurs notations suivant la base de codage utilisée.
- base décimale : (0, 1, 2..., 9) : int i ; i = 15 ;
- base octale : (0, 1...7) la définition d'une constante dans cette base se fait par l'ajout du préfixe '0' ou '\'. (int i = 017 ; int j = \17 ;)
- base héxa : (0, 1...9, A,..., F) la définition d'une constante dans cette base se fait par l'ajout du préfixe : 0x, 0X, \x, \X.

b) Les constantes flottantes

Deux notations sont possibles :
- La notation littérale avec virgule flottante (10.5, 0.4 ...).
- La notation scientifique : utilisant la virgule flottante et un exposant noté 'e' ou 'E'.

Exemple
5.69 E4 = 5.69 E+4 = 56.9 e3

c) Les constantes caractères

Deux notations sont possibles :
- La notation utilisant la valeur du caractère placé entre deux côtes.

Exemple
char car = 'a' ;
- La notation utilisant le code du caractère dans la base, le tout placé entre deux côtes.

Exemple
char car = '0x41' ; $((0x41)_{Héxa}$ = a)

d) Les constantes chaînes de caractères

Une chaîne est une séquence de caractères placés les un à côté des autres. Une constante de ce type est définit en délimitant cette séquence par deux guillemets.
Exemple : "Ceci est une chaîne";

e) Les constants symboliques

Il est possible d'associer à une valeur constante un nom d'identificateur qui lui fait référence, cette association se fait à l'aide du mot clé **define** de la manière suivante :

```
#define  identificateur  valeur
```

Exemple
#define N 50
#define max 100

4. Les opérateurs

a) Les opérateurs standards

Avant de nous lancer dans les spécialités du langage C, retrouvons d'abord les opérateurs correspondant à ceux que nous connaissons déjà en langage descriptif.

- Les opérateurs arithmétiques

+	addition
-	soustraction
*	multiplication
/	division (entière)
%	modulo (reste d'une div. entière)

- Les opérateurs logiques

&&	et logique (**and**)
||	ou logique (**or**)
!	négation logique (**not**)

- Les opérateurs de comparaison

= =	égal à
! =	différent de
<, <=, >, >=	plus petit que,...

- Résultats des opérations

Les résultats des opérations de comparaison et des opérateurs logiques sont du type **int** :
- la valeur 1 correspond à la valeur booléenne **« vrai »**.
- la valeur 0 correspond à la valeur booléenne **« faux »**.

Les opérateurs logiques considèrent toute valeur différente de zéro comme **vrai** et zéro comme **faux**.

b) Les opérateurs du langage C

- Les opérateurs d'affectation

En pratique, nous retrouvons souvent des affectations comme :

```
i = i + 2 ;
```

En C, nous utiliserons plutôt la formulation plus compacte :

```
i += 2 ;
```

L'opérateur **+=** est un opérateur d'affectation.

+=	ajouter à
- =	diminuer de
***=**	multiplier par
/=	diviser par
%=	modulo

- Les opérateurs d'incrémentation et de décrémentation

Les affectations les plus fréquentes sont du type :

i = i + 1 **et** **i = i - 1**

En C, nous disposons de deux opérateurs inhabituels pour ces affectations :

I++ ou **++I**	pour l'incrémentation	(augmentation d'une unité)
I-- ou **--I**	pour la décrémentation	(diminution d'une unité)

Les opérateurs ++ et -- sont employés dans les cas suivants :
Incrémenter/décrémenter une variable. Dans ce cas il n'y a pas de différence entre la notation **préfixe** (**++I --I**) et la notation **postfixé** (**I++ I--**).
Incrémenter/décrémenter une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, nous devons choisir entre la notation préfixé et postfixé.

X = I++	passe d'abord la valeur de I à X et **incrémente après**
X = I- -	passe d'abord la valeur de I à X et **décrémente après**
X = ++I	**incrémente d'abord** et passe la valeur incrémentée à X
X = - -I	**décrémente d'abord** et passe la valeur décrémentée à X

Exemple
Supposons que la valeur de N est égale à *5*

Incrém. postfixe:	**X = N++;**	Résultat: N=6 et X=*5*
Incrém. préfixe:	**X = ++N;**	Résultat: N=6 et X=6

c) Quelques opérateurs particuliers du langage C

- L'opérateur Ternaire (... ? ... : ...)

Test d'une expression **?** valeur renvoyé si vrai **:** valeur renvoyé si faux

Exemple

```
int i = 5, j = 2 ;
i < j ? i + j : i * j ;       //le résultat est 10
```

- L'opérateur sizeof (...)

Cet opérateur prend comme paramètre un type ou une variable et retourne la taille en octet qu'occupe cette variable ou ce type.

Exemple
```
int T ;
T = sizeof ( float ) ;
```

d) Forçage du type (casting)

On peut imposer une conversion de type en préfixant l'élément à convertir du type en question placé entre deux parenthèses.

Exemple
```
int i = 5,  j = 2 ;
float r = i / j ;              //r = 2.0
float v = (float) i / j ;     //v = 2.5
```

5. Les expressions et les instructions

a) Les expressions

La formation des expressions est définie par récurrence :

- Les constantes et les variables sont des expressions.
- Les expressions peuvent être combinées entre elles par des opérateurs et former ainsi des expressions plus complexes.
- Les expressions peuvent contenir des appels de fonctions et elles peuvent apparaître comme paramètres dans des appels de fonctions.

Exemples
```
a = (5*x + 10*y)*2
(a + b) >= 100
```

b) Les instructions

Une expression comme **I = 0** ou **I++** ou **printf (...)** devient une instruction, si elle est suivie d'un point-virgule.

Exemples
```
i = 0 ;
i ++ ;
printf (" Bonjour ") ;
a = (5*x+10*y)*2 ;
```

Chapitre 3 : Les entrées/sorties

Pour transmettre les données saisies au clavier à un programme ou pour faire l'affichage de ces données, des commandes adéquates sont nécessaires.
Le langage C n'offre pas dans son vocabulaire de base telle commandes. Ces opérations sont plutôt réalisées à l'aide des fonctions faisant partie de la bibliothèque '**stdio.h**' fournie avec le compilateur.
La bibliothèque standard '**stdio.h**' contient un ensemble de fonctions qui assurent la communication de la machine avec le monde extérieur.

1. L'affichage formaté

La fonction **printf** est utilisée pour transférer du texte, des valeurs de variables ou des résultats d'expressions vers le fichier de sortie standard **stdout** (par défaut l'écran).

a) Écriture formatée en C

printf("<format>",<Expr1>,<Expr2>, ...) ;

"<format>" : Format de représentation.
<Expr1>,... : Variables et expressions dont les valeurs sont à représenter.

La partie "**<format>**" est en fait une chaîne de caractères qui peut contenir:

* du texte.
* des séquences d'échappement (\n, \t, \a...).
* des caractères de formatage (commencent toujours par le symbole '**%**').

b) Les caractères de formatage pour ' printf '

SYMBOLE	TYPE	IMPRESSION COMME
%d ou %i	int	entier relatif
%u	int	entier naturel (unsigned)
%o	int	entier exprimé en octal
%x	int	entier exprimé en hexadécimal
%c	char	caractère
%f	double	rationnel en notation décimale
%e	double	rationnel en notation scientifique
%s	char*	chaîne de caractères

Exemples

```
#include <stdio.h>
void main ( )
   {int i = 90 ;
    printf (" %d décimal donne l'octal %o et l'hexadécimal %x ", i, i, i) ;
   }
```

```
#include <stdio.h>
void main ( )
      { int i = 5 ; int j = 8 ;
        printf ("le contenu de i est  %d \n", i) ;
        printf ("le contenu de i est %d et celui de j est %d \n", i, j) ;
        printf ("la somme de i et j est : %d \n", i+j) ;
      }
```

```
#include <stdio.h>
void main ( )
{printf (" %c et %c ont les codes ASCII %d et %d \n", 'a', 'A', 'a', 'A') ;
}
```

c) Quelques séquences d'échappement du langage

Notation :	Signification :
\n	Génère une nouvelle ligne.
\t	Tabulation horizontale.
\v	Tabulation verticale.
\b	Retour d'un caractère en arrière.
\r	Retour chariot.

d) Contrôle du gabarit et de la précision

Entre le % et le caractère de formatage, plusieurs autres symboles et caractères peuvent être placé permettant alors de contrôler le gabarit d'affichage et la précision. Ainsi, on peut placer :

- Un chiffre qui indique le nombre minimum de caractère à afficher avec un remplissage par des zéros si ce chiffre est lui-même précédé d'un zéro.
- Un signe (-) pour réclamer un cadrage à gauche.
- Un chiffre précédé d'un point pour la précision.

e) La macro 'putchar'

Cette macro joue le même rôle que 'printf ' pour les caractères.

Exemple

```
#include <stdio.h>
void main ( )
      { char k = 'a' ;
         putchar (k) ;        //ou  printf ("%c", k) ;
      }
```

2. La saisie formatée

La fonction **scanf** est la fonction symétrique à **printf**, elle nous offre pratiquement les mêmes conversions que **printf**, mais en sens inverse.

a) Lecture formatée en C

scanf("<format>",<AdrVar1>,<AdrVar2>, ...)

"<format>" : Format de lecture des données.

<AdrVar1>,... : Adresses des variables auxquelles les données seront attribuées.

* La fonction **'scanf'** reçoit ses données à partir du fichier d'entrée standard **'stdin'** (par défaut le clavier).
* La chaîne de format détermine comment les données reçues doivent être interprétées.
* Les données reçues correctement sont mémorisées successivement aux adresses indiquées par <AdrVar1>,...
* L'adresse d'une variable est indiquée par le nom de la variable précédé du signe '&'.

Exemple

```
int i ; float j ; char k ;
printf ("Donnez les valeurs de i, j et k") ;
scanf ("%d", &i) ;
scanf ("%f", &j) ;      //ou scanf ("%d%f%c", &i, &j, &k) ;
scanf ("%c", &k) ;
```

b) Remarques

1- La spécification du format, les valeurs saisies et les arguments doivent correspondre en types et en nombres.
2- S'il n'y a plus de valeurs saisies que de format et argument, les valeurs en trop seront ignorées.
3- Si le type de la valeur saisie ne correspond pas à l'indication de formatage dans 'scanf', on obtiendra une erreur.
4- Lors de la saisie de plusieurs valeurs, ces dernières doivent être séparées par des blancs (espace, tabulation, saut de lignes...) et la saisie doit se terminer par un retour chariot (entrée).

c) Spécification du nombre maximum des caractères à considérer

Le nombre maximum de caractère à considérer lors de la saisie peut être spécifié avec un chiffre placé entre le % et le caractère de formatage.

d) La macro 'getchar( )'

Cette macro joue le même rôle que 'scanf ' pour les caractères.

Exemple

```
char k ;
printf (" Donner un caractère : ") ;
k = getchar( ) ;        //ou  scanf ("%c", &k) ;
```

Chapitre 4 : Les instructions de contrôle

Le langage C dispose de 3 catégories de structure de contrôle : les structures de branchement conditionnel, inconditionnel et les structures répétitives.

1. Branchement conditionnel

a) Branchement conditionnel simple

if (<expression>)
    <Bloc d'instructions 1> ;
else
    <Bloc d'instructions 2> ;

* Si l'<expression> fournit une valeur différente de zéro (vrai), alors le <bloc d'instructions 1> est exécuté.
* Si l'<expression> fournit la valeur zéro (faux), alors le <bloc d'instructions 2> est exécuté.

Exemples

1) Donner un programme C qui permet d'indiquer si un nombre saisi par l'utilisateur peut être utilisé comme diviseur.

```
#include <stdio.h>
void main()
      { int i ;
         printf ("Donner un entier : ") ;
         scanf ("%d", &i) ;
         if ( i == 0)
            printf ("Impossible de l'utiliser comme diviseur") ;
      }
```

2) Donner un programme C qui permet de vérifier si un nombre saisi par l'utilisateur est pair ou impair.

```
#include <stdio.h>
void main()
      { int i ;
         printf ("Donner un entier : ") ;
         scanf ("%d", &i) ;
         if ((i % 2) == 0)
            printf ("Le nombre est pair. ") ;
         else
            printf ("Le nombre est impair. ") ;
      }
```

b) Branchement conditionnel imbriqué

En combinant plusieurs structures ' **if – else** ' en une expression nous obtenons une structure qui est très courante pour prendre des décisions entre plusieurs alternatives.

if (<expr1>)
<bloc 1>
else if (<expr2>)
<bloc 2>
else if (<expr3>)
<bloc 3>
else if (<exprN>)
<bloc N>
else <bloc N+1>

Les expressions <expr1>... <exprN> sont évaluées du haut vers le bas jusqu'à ce que l'une d'elles soit différente de zéro. Le bloc d'instructions y lié est alors exécuté et le traitement de la commande est terminé.

Exemple

```
#include <stdio.h>
void main( )
      { int A, B;
        printf("Entrez deux nombres entiers : ") ;
        scanf("%i %i", &A, &B) ;
        if (A > B)
            printf("%i est plus grand que %i \n", A, B) ;
        else if (A < B)
                  printf("%i est plus petit que %i \n", A, B) ;
             else
                  printf("%i est égal à %i \n", A, B) ;
      }
```

La dernière partie '**else**' traite le cas où aucune des conditions n'a été remplie. Elle est optionnelle, mais elle peut être utilisée très confortablement pour détecter des erreurs.

c) Branchement conditionnel multiple :

switch (valeur à tester)
{ case constante 1 **:**
Bloc 1 ;
break ;
case constante 2 **:**
Bloc 2 ;
break ;
...
...
case constante n **:**
Bloc n ;
break ;
default :
Bloc d'instructions par défaut ;
}

Remarques

- Le mot clé '**break**' engendre une sortie du bloc associé au '**switch**'.
- Le mot clé '**default**' désigne un bloc d'instructions qui sera exécuté par défaut, si '**switch**' n'est pas interrompu par un '**break**'.

2. Les Structures de contrôle répétitives

En C, nous disposons de trois structures qui nous permettent la définition de boucles conditionnelles. Théoriquement, ces structures sont interchangeables, c à d il serait possible de programmer toutes sortes de boucles conditionnelles en n'utilisant qu'une seule des trois structures.

a) La boucle 'while' :

while (expression)
{Bloc d'instructions ;**}**

* Tant que l'expression fournit une valeur différente de zéro, le bloc d'instructions est exécuté.
* Si l'expression fournit la valeur zéro, l'exécution continue avec l'instruction qui suit le bloc d'instructions.
* Le bloc d'instructions est exécuté zéro ou plusieurs fois.

La partie expression peut désigner :

Une variable d'un type numérique.
Une expression fournissant un résultat numérique.

La partie bloc d'instructions peut désigner :

Un bloc d'instructions compris entre accolades.
Une seule instruction terminée par un point-virgule.

Exemple

```
//Afficher les nombres de 0 à 9
int i = 0;
while (i<10)
      {
       printf("%d \n", i);
       i++;
      }
```

b) la boucle 'do...while' :

do
{Bloc d'instructions ;**}**
while (expression)**;**

La structure '**do ... while**'est semblable à la structure '**while**', avec la différence suivante :

* '**while**' évalue la condition avant d'exécuter le bloc d'instructions,
* '**do ... while**' évalue la condition après avoir exécuté le bloc d'instructions. Ainsi le bloc d'instructions est exécuté au moins une fois.
* Le bloc d'instructions est exécuté au moins une fois et aussi longtemps que l'expression fournit une valeur différente de zéro.

Exemple

```
float N;
do
      { printf("Introduisez un nombre entre 1 et 10 : ") ;
         scanf(" %f ", &N);
      }
while (N<1 || N>10);
```

c) La boucle 'for' :

for (expr1 ; expr2 ; expr3)
{Bloc d'instructions ;}

expr1 est évaluée une fois avant la première itération de la boucle. Elle est utilisée pour initialiser les données de la boucle.
expr2 est évaluée avant chaque itération de la boucle. Elle est utilisée pour décider si la boucle est répétée ou non.
expr3 est évaluée à la fin de chaque itération de la boucle. Elle est utilisée pour réinitialiser les données de la boucle.

Exemples

```
int I;
for (I=0 ; I<=20 ; I++)
     printf("Le carré de %d est %d \n", I, I*I);
```

```
int n, total;
for (tot=0, n=1 ; n<101 ; n++)
      total+=n;
printf("La somme des nombres de 1 à 100 est %d", total);
```

Conseils généraux

* Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez **while** ou **for**.
* Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez **do - while**.
* Si le nombre d'exécutions du bloc d'instructions dépend d'une ou de plusieurs variables qui sont modifiées à la fin de chaque répétition, alors utilisez **for**.
* Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez **while**.

Le choix entre **for** et **while** n'est souvent qu'une question de préférence ou d'habitudes :
* **for** nous permet de réunir avantageusement les instructions qui influencent le nombre de répétitions au début de la structure.
* **while** a l'avantage de correspondre plus exactement aux structures d'autres langages de programmation.
* **for** a le désavantage de favoriser la programmation de structures surchargées et par la suite illisibles.
* **while** a le désavantage de mener parfois à de longues structures, dans lesquelles il faut chercher pour trouver les instructions qui influencent la condition de répétition.

3. Branchement inconditionnel

Le langage C dispose de 3 instructions pour le branchement inconditionnel : break, continue, goto.

a) break

En plus de son utilisation avec '**switch**', le mot clé '**break**' peut être utilisé dans n'importe quelle boucle. Il engendre une interruption et une sortie immédiate de la boucle. L'exécution du programme continue au niveau de l'instruction juste après la boucle. Si on a plusieurs boucles imbriquées alors le '**break**' fait sortir de la boucle la plus interne.

Syntaxe break ;

b) continue

Cette instruction intervient également pour interrompre l'exécution des boucles, mais contrairement à '**break**', elle ne provoque pas la sortie complète de la boucle mais plutôt l'interruption de l'itération courante et le passage à la suivante.

Syntaxe continue ;

c) goto

Elle provoque un branchement immédiat du programme à un endroit prédéfini. Les boucles, les tests et les autres instructions sont interrompus. L'endroit où reprend le programme est défini par une étiquette suivi par le symbole ' **:**'.

Syntaxe

```
goto Nom_Etiquette ;
… ;
… ;
Nom_Etiquette :  … ;
```

Chapitre 5 : Les tableaux

Un tableau est un type particulier de variables qui permet d'associer sous le même nom un ensemble fini de valeurs de même nature. Tous les types de base et les types personnalisés peuvent servir à la définition d'un tableau.

1. Déclaration d'un tableau

a) Syntaxe

Type nom_tableau [taille] ;

Exemples

int T[5] ; ⟶ T

0	1	2	3	4
int	int	int	int	int

double F[10] ;

b) Remarques

Lors de la déclaration, la dimension d'un tableau ne peut être qu'une constante ou une expression constante. Elle ne peut pas être une constante symbolique.
En C, le nom d'un tableau est le représentant de l'adresse du premier élément du tableau. Les adresses des autres composantes sont calculées (automatiquement) relativement à cette adresse.

2. Accès à un élément du tableau

En déclarant un tableau par :

int A[5];

Nous avons défini un tableau A avec cinq composantes, auxquelles nous pouvant accéder par:

A[0], A[1]... A[4]

Remarques

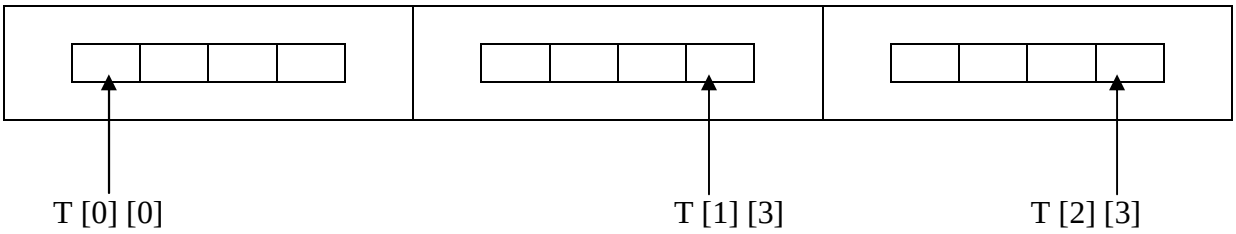
- L'accès au premier élément du tableau se fait par A [0].
- L'accès au dernier élément du tableau se fait par A [N-1].
- La référence d'un élément qui n'existe pas (exemple : A [6]) n'est pas signalée comme une erreur lors de la compilation. En effet, le programme fait comme si la case existait. La seule manifestation de cette erreur se fait à travers l'obtention de valeurs erronées. Elle peut même planter l'ordinateur.

3. Tableaux à plusieurs dimensions

En C, en plus des tableaux à une dimension, il est possible de déclarer des tableaux à plusieurs dimensions. Ainsi et à titre d'exemple, la déclaration d'un tableau à deux dimensions :

Type Tab [L][C] ;

Type Tab [Dim 1][Dim 2]…[Dim n] ;
Exemple
int T[3][4] ;



4. Initialisation des tableaux

a) Tableau à une dimension

```
int tab[3]={1,7,4} ;
int tab[ ]={1,6,8,10,15,45} ;
int tab[4]={    , 2,    , 7} ;
```

Les valeurs non déclarées seront initialisées à 0.

b) Tableau à deux dimensions

```
int tab [3][4]={1,2,3,4,5,6,7,8,9,10,11,12} ;
int tab [3][4]={ {1,2,3,4} , {5,6,7,8} , {9,10,11,12} } ;
int tab [3][4]={ {1,  ,3,4},        ,{9,10,   ,12} } ;
```

Initialisées à 0.

5. Quelques règles d'écriture

* Les éléments d'un tableau peuvent être incrémentés ou décrémentés.

```
T [ 4 ] = 10 ;
T [ 4 ] ++ ;      //T[4] devient 11
```

* Les indices peuvent être des expressions arithmétiques :

```
int i,  j,  k ;
T[2 * i - 1] ... ;
Tab [ i - 3 ] [ j + k ]... ;
```

* En C, il n'est pas possible de faire des affectations globales entre tableaux.

```
int T1[4] = { 20, 5, 30, 8} ;
int T2[4] ;
T2 = T1 ;       //ERREUR
```

Application
Donner un programme en C qui effectue la copie d'un tableau d'entiers initialisé dans un autre tableau de même dimension.

Solution

```
void main( )
      { int i ;
      int T1[4] = {20, 5, 30, 8} ;
      int T2[4] ;
      for ( i = 0 ; i < 4 ; i++)
              T2[i] = T1[i] ;
}
```

6. Saisie et affichage des éléments d'un tableau :

La saisie et l'affichage d'un élément d'un tableau se font de la même manière que n'importe quelle variable possédant le même type que celui des éléments du tableau.

Application

Donner un programme en C qui fait la saisie d'un tableau de 5 caractères et l'affiche.

Solution

```
#include <stdio.h>
void main()
      { int i ;
      char T[5] ;
      /* Saisie */
      for ( i = 0 ; i < 5 ; i++)
            { printf ("Donner le caractère numéro :%d", i+1);
              scanf ("%c", &T[i]) ;
            }
      /* Affichage */
      for ( i = 0 ; i < 5 ; i++)
            printf ("%c \t", T[i]) ;
      }
```

Chapitre 6 : Les chaînes de caractères

1. Définition

Les chaînes de caractères sont implémentées en C comme des tableaux de caractères dont le dernier élément est le caractère nul '**\0**' (code ASCII 0) indiquant la fin de la chaîne.

2. Déclaration et initialisation des chaînes de caractères

Il existe plusieurs façons pour déclarer et initialiser une chaîne de caractères.

```
char ch  [ ] = "ABCDE" ;      //tableau de 5 caractères plus '\0'
char ch1 [6] = "ABCDE" ;
char ch2 [ ] = { 'A', 'B', 'C', 'D', 'E' , '\0' } ;
char ch3 [ ] = { 65, 66, 67, 68, 69 , 0 } ;
```

3. Saisie et Affichage des chaînes de caractères

Pour afficher une chaîne, on utilise la fonction printf avec le caractère de formatage %s et on met en argument le nom du tableau qui contient la chaîne :

```
char ch [ ] = "Informatique" ;
printf ("%s Industrielle \n", ch) ;
```

Pour saisir une chaîne, On utilise la fonction scanf avec le caractère de formatage %s et on met en argument le nom du tableau qui contient la chaîne. Attention, les tableaux sont des cas particuliers et on n'a pas besoin de mettre le caractère & devant le nom :

```
char ch [512] ;
printf ("Donner une chaîne : ") ;
scanf ("%s", ch) ;
```

4. Copie et comparaison des chaînes de caractères

Une chaîne est un tableau, on ne peut pas donc réaliser directement une affectation ou une copie. Pour copier une chaîne dans un tableau, on utilise la fonction **strcpy (string copy)** :

```
char ch1 [ ] = "Info" ;
char ch2 [512] ;
strcpy (ch2, ch1) ;
printf ("%s \n", ch2) ;
```

Pour comparer deux chaînes, on utilise la fonction **strcmp (string compare)**. Une autre fonction utile est la fonction **strlen (string length)** qui sert à mesurer la longueur d'une chaîne (caractère nul non compris).

5. Quelques fonctions sur les chaînes de caractères (string.h)

size_t strlen (const char *s) : Retourner la longueur de la chaîne s.

char *stpcpy (char *s1, const char *s2) : Copier s2 dans s1.

char *strcat (char *s1, const char *s2) : Concaténer s2 à la fin de s1.

int strcmp (const char *s1, const char*s2) : Comparer s1 et s2 suivant l'ordre lexicographique et retourner un résultat :

< 0	Si	s1 < s2
= = 0	Si	s1 = = s2
> 0	Si	s1 > s2

char *strncat (char *s1, const char *s2, size_t n) : Concatèner les n premiers caractères de s2 après s1.

int strncmp (const char *s1, const char *s2, size_t n) : Comparer les n premiers caractères de s1 et s2.

6. Quelques fonctions sur les caractères (ctype.h)

int isalpha (int c) : vérifie si c appartient à 'a...z' ou 'A...Z'.

int islower (int c) : vérifie si c appartient à 'a...z'.

int isdigit (int c) : vérifie si c est un chiffre (0...9).

int ispunct (int c) : vérifie si c est un caractère de ponctuation (: ; ! ? , .).

int isspace (int c) : vérifie si c est un caractère d'espacement (' ' '\n' '\t' '\v' '\r' ...).

Chapitre 7 : Les pointeurs

Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable. En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.
Si un pointeur P contient l'adresse d'une variable A, on dit que 'P pointe sur A'.

1. La déclaration des pointeurs

a) Syntaxe

Type * Nom_pointeur ;

Exemple

```
int * a ;      //Réserve un emplacement pour stocker une adresse mémoire
float * b ;
```

b) Déclaration multiple

```
int * p, * q ;
int * pt, i ;     //i est un entier
```

c) Remarque

Les pointeurs et les noms de variables ont le même rôle, ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :
* Un pointeur est une variable qui peut 'pointer' sur différentes adresses.
* Le nom d'une variable reste toujours lié à la même adresse.

2. L'initialisation des pointeurs

Comme pour les variables, il est possible d'initialiser une variable de type pointeur. La valeur initiale est dans ce cas l'adresse d'une donnée, possédant comme type, le type vers lequel pointe le pointeur en question.
Exemple

```
int i ;
int * pi ;  //Réserve un emplacement pour stocker une adresse mémoire
pi = &i;    //Ecrit l'adresse de 'i' dans cet emplacement
```

Si on indique à un pointeur une valeur d'initialisation l'adresse d'une variable ayant un autre type que celui vers lequel pointe, le compilateur affiche un message d'erreur.
Exemple

```
float f ;
char * c = &f ;   //ERREUR
```

3. L'accès indirect aux variables

Pour pouvoir accéder à travers un pointeur à la donnée contenue dans la variable sur laquelle il pointe, on utilise l'opérateur d'indirection noté ' * '.

Exemple

```
int a, b ;
int * pt ;
a = 123 ;
pt = &a ;
b = *pt ;
printf ("%d",b) ;       //Affiche 123
```

4. Saisie et affichage d'une variable à travers un pointeur

Exemple 1

```
int i, *pt = &i ;
scanf("%d", &i) ;       //On saisit la valeur 5
printf("%d", *pt) ;     //On obtient 5
```

Exemple 2

```
int * pt, i, j ;
i = 1 ; pt = &i ;
j = *pt + 4 ;           //j = 5
```

Exemple 3

```
#include <stdio.h>
void main( )
      { int i = 125 ; int * pt = &i ;
         printf("La valeur de i est : %d", i) ;      //125
         printf("La valeur de *pt est : %d", *pt) ;  //125
         i++ ;
         printf("La valeur de i est : %d", i) ;      //126
         printf("La valeur de *pt est : %d", *pt) ;  //126
         (*pt)++ ;
         printf("La valeur de i est : %d", i) ;      //127
         printf("La valeur de *pt est : %d", *pt) ;  //127
      }
```

5. Incrémentation et Addition de pointeurs

a) L'incrémentation

L'incrémentation d'un pointeur donne l'adresse situé à {sizeof(type) octets} à partir de la valeur courante du pointeur.

Exemple

```
int i ;
int *pt = &i ;          // si &i = 1600 (octet)
pt++ ;                  // 1604  (int sur 4 octets)
```

b) L'addition

L'addition d'un pointeur et un entier A donne l'adresse situé à {A * sizeof(type)} à partir de la valeur courante.

Exemple

```
int *i, *j ;
int k ;           // &k = 1500
i = &k ;          // i = 1500
j = i+10 ;        // j = 1540
j++ ;             // j = 1544
```

Pour les pointeurs, l'addition n'est permise qu'entre un pointeur et un entier. Il est également interdit de faire une addition entre deux pointeurs.

6. Soustraction et décrémentation de pointeurs

On peut faire une soustraction entre deux pointeurs et entre un pointeur et un entier. La soustraction d'un entier à partir d'un pointeur et la décrémentation d'un pointeur fonctionnent de la même façon que pour l'addition et l'incrémentation.

Exemple

```
int *a, *b ;
int tab[6] ;
a = &tab[5] ;
b = &tab[1] ;
printf("%d", a – b) ;   //4
```

7. Le pointeur 'NULL'

Il est possible de définir un pointeur qui ne pointe vers aucune donnée, ce pointeur (appelé pointeur nul) contient l'adresse nulle.
Le pointeur nul (NULL) peut être comparé à n'importe quel pointeur. Il peut être affecté à n'importe quel pointeur.

Exemple

```
int *p ;
if (p == NULL)
      printf("ERREUR") ;
```

8. Les pointeurs et les tableaux

En C, et de point de vue type, le nom d'un tableau est assimilé à une constante d'adresse définissant l'emplacement à partir duquel vont se succéder les éléments. Ainsi, l'identifiant d'un tableau est considéré comme un pointeur lorsqu'il est employé seul.

Exemples

int t[10] ;

t ⟺ &t[0] *t ⟺ t[0]

t + i ⟺ &t[i] *(t + i) ⟺ t[i]

```
int t[10], i ;
for(i = 0 ; i < 10 ; i++)
      t[i] = 1 ;
```

⟺

```
int t[10], i ;
for(i = 0 ; i < 10 ; i++)
      *(t + i) = 1 ;
```

Si P pointe sur un tableau, alors :

P	désigne l'adresse de la première composante.
P+i	désigne l'adresse de la $i^{ème}$ composante après P.
*(P+i)	désigne le contenu de la $i^{ème}$ composante après P.

9. L'allocation dynamique de la mémoire

Nous avons vu que l'utilisation de pointeurs nous permet de mémoriser économiquement des données de différentes grandeurs. Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de l'allocation dynamique de la mémoire.

a) La fonction 'malloc'

Syntaxe

void* malloc (n * sizeof(type))

Cette fonction alloue une zone mémoire de (n * la taille du type). Elle retourne un pointeur générique donnant l'adresse de début du bloc alloué ou un pointeur NULL si l'allocation a échoué pour insuffisance d'espace.
Cette fonction est définie dans les bibliothèques 'stdlib.h' et 'alloc.h'.

b) La fonction 'free'

Syntaxe

void free (Nom_Pointeur)

Cette fonction permet de libérer un espace mémoire déjà alloué.

Exemple

```
#include <stdio.h>
#include <alloc.h>
void main( )
      { int *adr, i;
         adr = malloc (10*sizeof(int));
         for (i = 0 ; i < 10 ; i++)
            *(adr+i) = 1;
         for (i = 0 ; i < 10 ; i++)
            printf("%d \t", *(adr+i))
         free (adr) ;
      }
```

Chapitre 8 : Les fonctions

1. Définition

En langage C, il n'existe qu'une seule sorte de module, nommée fonction. Mais elle peut prendre des aspects différents (fournir une valeur, un résultat non scalaire ou modifier les valeurs de certains paramètres ...).
- Une fonction est une portion de programme regroupant un ensemble d'instructions délimité par deux accolades et portant un nom qui est le nom de la fonction.
- Elle peut être déclaré dans le programme principal (main) ou dans une autre fonction.
- Elle est dite paramétrée si elle a besoin de prendre des données en entrée, ils sont appelées arguments ou paramètres.
- Généralement, une fonction retourne une valeur qui représente le résultat de son exécution. Toutefois, il est possible en C de définir une fonction qui ne retourne aucune valeur.

2. Déclaration des fonctions en langage C

En C, il faut déclarer chaque fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur du type des paramètres et du résultat de la fonction.

Déclaration : Prototype d'une fonction

```
TypeRés  NomFonct (TypePar1  NomPar1, TypePar2  NomPar2, ...);
```

3. Définition des fonctions en langage C

a) Syntaxe

```
TypeRés  NomFonct (TypePar1  NomPar1, TypePar2  NomPar2, ...)
        { //déclarations locales
          //instructions
          return ... ;
        }
```

b) Valeur de retour d'une fonction

- Si une fonction ne fournit pas de résultat, il faut indiquer **void** (vide) comme type du résultat.
- Le type par défaut est **int**, autrement dit, si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement de type **int**.
- L'instruction **return** provoque un arrêt immédiat de l'exécution de la fonction et renvoie une valeur qu'elle prend comme paramètre.
- Si le **return** n'est pas présent dans la fonction, alors l'exécution des instructions continue jusqu'à la fin du bloc de la fonction.

Exemple 1

```
int min(int a, int b)
      {if(a<=b)
            return a ;
      else  return b ;
      }
```

c) Remarques

- ✓ Si une fonction n'a pas de paramètres, on peut déclarer la liste des paramètres comme **(void)** ou simplement comme **( )**.
- ✓ L'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée ou définie avant d'être appelée.

Exemple 2

```
#include <stdio.h>
void signe(int n)
      { if(n<0)
            printf("n est strictement négatif") ;
        else if (n>0)
            printf("n est strictement positif") ;
                  else
            printf("n est nulle") ;
      }
void main( )
      {int nb ;
       printf("Donner le nombre à tester : ") ;
       scanf("%d",&nb) ;
       signe(nb) ;
      }
```

4. Les types des paramètres dans les fonctions

Suivant la nature de l'instruction dans laquelle intervient la fonction (définition, déclaration, appel), il existe deux types de paramètres.

a) Les paramètres formels :

Ce sont les paramètres qui figurent dans la définition de la fonction, ils sont utilisés par les instructions faisant partie du bloc de la fonction. Ils sont considérés comme des variables locales à la fonction.

b) Les paramètres effectifs :

Ce sont les paramètres qui figurent dans l'instruction d'appel de la fonction. Ils sont substitués aux paramètres formels.

Exemple

```
#include <stdio.h>
int montant(int a, int b) ;        //Paramètres formels : facultatifs

void main( )
      {int nb, res, val ;
       printf("Donner le nombre d'articles le prix : ") ;
       scanf("%d%d",&nb,&val) ;
       res=montant(nb,val) ;        //Paramètres effectifs
       printf("le montant est %d : ",res) ;
      }

int montant(int nbart, int p)      //Paramètres formels : obligatoires
      {int total ;
       total=nbart*p ;
       return total ;
      }
```

c) Les variables locales et les variables globales :

Une variable n'est visible qu'à l'intérieur du bloc où elle est déclarée, donc les variables locales ne sont vues qu'à l'intérieur de la fonction où elles sont déclarées, leur portée est limitée à cette fonction.

Exemple

```
#include <stdio.h>
int n ;          //n est une variable globale
void main()
      {... ;
       ... ;
      }
int f1(...)
      {... ;
      }
float f2()
      {... ;
       ... ;
      }
```

5. Fonction locale et fonction globale

a) Déclaration locale :

Une fonction peut être déclarée localement dans la fonction qui l'appelle (avant la déclaration des variables). Elle est alors disponible à cette fonction seulement.

b) Déclaration globale :

Une fonction peut être déclarée globalement (en dehors de toute fonction). Elle est alors disponible à toutes les fonctions du programme.

Exemple 1 : Déclaration locale par rapport à main

```
#include <stdio.h>
void main ()
      {... ;
       int min (...) ;         //La fonction min est locale à main
       ... ;
      }
void f1 ()
      {... ;
      }
```

Exemple 2 : Déclaration globale par rapport à main

```
#include <stdio.h>
int min (...) ;               //La fonction min est globale
void main ()
      {... ;
       ... ;
       ... ;
      }
```

6. Les modes de passage des paramètres

En langage C, le passage des paramètres d'une fonction se fait par valeur. Ce mode de passage interdit la modification du contenu des arguments. L'exemple suivant illustre le problème de la permutation.

Exemple

```
#include <stdio.h>
void permut (int a, int b)
      { int x ;
        printf("Début échange :%d %d\n", a, b) ;
        x = a ;
        a = b ;
        b = x ;
        printf("Fin échange :%d %d\n", a, b) ;
      }
void main ()
      { int n = 2, p = 4 ;
        printf("Avant l'appel :%d %d\n", n, p) ;
        permut (n, p) ;
        printf("Après l'appel :%d %d\n", n, p) ;
      }
```

En effet, lors de l'appel de la fonction « permut », **les valeurs de n et p sont recopiées dans a et b**. C'est effectivement sur ces copies qu'a travaillé la fonction « permut ». Pour remédier à ce problème, on peut transmettre en argument **l'adresse de la variable** (en utilisant la notion des pointeurs), la fonction pourra éventuellement agir sur le contenu de cette adresse.

Proposition d'une solution

```
#include <stdio.h>
void permut (int *a, int *b)
      { int x ;
        printf("Début échange :%d %d\n", a, b) ;
        x = *a ;
        *a = *b ;
        *b = x ;
        printf("Fin échange :%d %d\n", a, b) ;
      }
void main ()
      { int n = 2, p = 4 ;
        printf("Avant l'appel :%d %d\n", n, p) ;
        permut (&n, &p) ;
        printf("Après l'appel :%d %d\n", n, p) ;
      }
```

Chapitre 9 : Les structures

1. Définition d'une structure

C'est une liste d'éléments de différents types à laquelle on associe un nom globale, chaque élément de cette structure, appelé membre ou champ, possède un nom et un type. Ce type peut être un type de base ou personnalisé. La structure elle même est considérée comme un type personnalisé.

a) Définition d'une structure

```
struct  Nom_de_la_ structure
        { Type_membre1      Nom_membre1 ;
          Type_membre2      Nom_membre2 ;
          ...
        } ;
```

b) Déclaration d'une variable

```
struct  Nom_de_la_structure      var ;
```

Nouveau type — Nom de la variable

Exemple

```
struct  Fiche
      { int age ;
        float poids, taille ;
        char nom[10] ;
      } ;
struct Fiche Personne ;   //Personne est une variable de type Fiche
```

2. Manipulation des membres d'une structure

Un membre d'une structure peut être manipulé en spécifiant le nom de la variable structure suivi de l'opérateur de champ '.' suivi du nom du membre considéré.

Exemple

```
scanf ("%d", &Personne.age) ;       //Saisie
printf ("%d", Personne.age) ;       //Affichage
```

Le membre d'une structure est manipulé de la même manière que toutes autres variables du même type.

3. Utilisation globale d'une structure

On ne peut pas faire d'une manière globale la saisie ou l'affichage du contenu d'une structure, ces opérations doivent s'effectuer élément par élément.

On ne peut pas comparer globalement deux structures.

Exemple

```
struct Fiche    P1, P2 ;
...
...
```

~~P1 < P2~~ ~~P1 == P2~~ ...

4. Initialisation des variables d'une structure

Elle se fait de la même manière que pour les tableaux.

Exemple

```
struct Etudiant
      { char Nom[10] ;
        float taille, poids ;
      } ;
struct Etudiant E = {"Aymen", 1.85, 78.5} ;
```

5. Imbrication de structures

Le type d'un membre d'une structure peut être lui-même une structure.
Exemple

```
struct Date
      { int jj, mm, aa ;
      } ;
struct Personne
      {char Nom[10], Prenom[10] ;
       struct Date  DN ;
      } ;
```

struct Personne P ;
Comment accéder à l'année de naissance de la personne P ?
P.DN.aa

Comment initialiser une structure comportant une autre structure ?
struct Personne P1 = {"Aymen", "Ali", {27, 1, 1988}} ;

6. Les tableaux de structures

Comme tout autre type, on peut déclarer des tableaux de structures.
Exemple

```
struct Point
      { int x, y ;
      } ;
struct Point  Courbe [100] ;       //Courbe est un tableau de points
```

Accès au 10ème point :
Courbe [9].x
Courbe [9].y

7. Les pointeurs et les structures

Il est possible en langage C de définir des pointeurs sur des variables de type structure.
Exemple

```
struct  Etudiant
      { float poids ;
          ... ;
      } ;
struct  Etudiant  E ;         //une variable E de type Etudiant
struct  Etudiant *pE ;        //un pointeur pE sur un Etudiant
pE = &E ;                     //pE pointe vers E
```

L'accès au membre de la structure se fait alors à l'aide de flèche '->'.
pE -> poids , E.poids et (*pE).poids sont équivalentes.

8. Utilisation de '*typedef*' avec les structures

Le mot clé typedef permet d'associer un nouveau nom à un type déjà existant. Il ne permet en aucun cas de définir un nouveau type.

Syntaxe

typedef Nom_type_existant Nouveau_Nom ;

Exemple

```
typedef    int    entier ;    //on a donné un autre nom au type int
entier   a, b ;
```

Application de typedef aux structures

```
typedef  struct
      { float x ;
        float y ;
      } Point ;
Point P1, P2 ;          //on utilise Point au lieu de struct Point
```

Bibliographie

Claude Delannoy, Programmer en langage C : cours et exercices corrigés, Edition Eyrolles, Paris, 1997