Université Mohamed V - Rabat
Ecole Nationale Supérieure d'Informatique
et d'Analyse des Systèmes

Programmation Procédurale

Prof. GUERMAH Hatim

Email: guermah.ensias@gmail.com

Année Universitaire: 2022-2023

Contenu du cours

- Les langages de Programmation
- Découvrir le langage C
- Eléments de programmation
- Structures de contrôle
- Les tableaux
- Les structures
- Gestion dynamique de la mémoire
- Programmation modulaire : les fonctions
- Le préprocesseur et la compilation conditionnelle
- La compilation séparée

Section 1

Les langages de Programmation

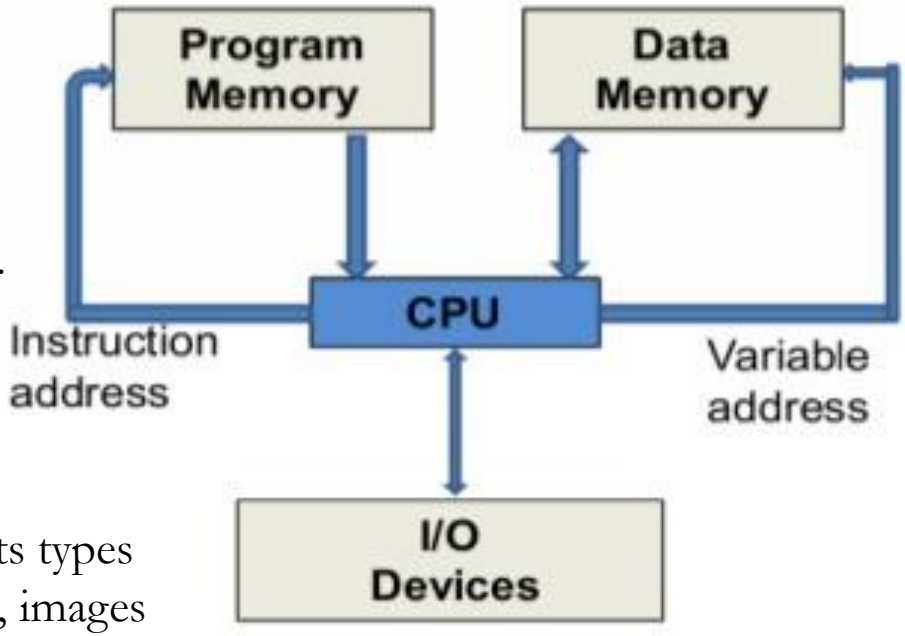
Rôle d'un langage de programmation

Notion de compilateur

Généralités

- Ordinateur : Machine électronique programmable capable de mémoriser une quantité d'information et de réaliser des calculs logiques sur des nombres binaires.

Ensemble des actions consécutives qu'un ordinateur doit exécuter.



Adresses et Valeurs

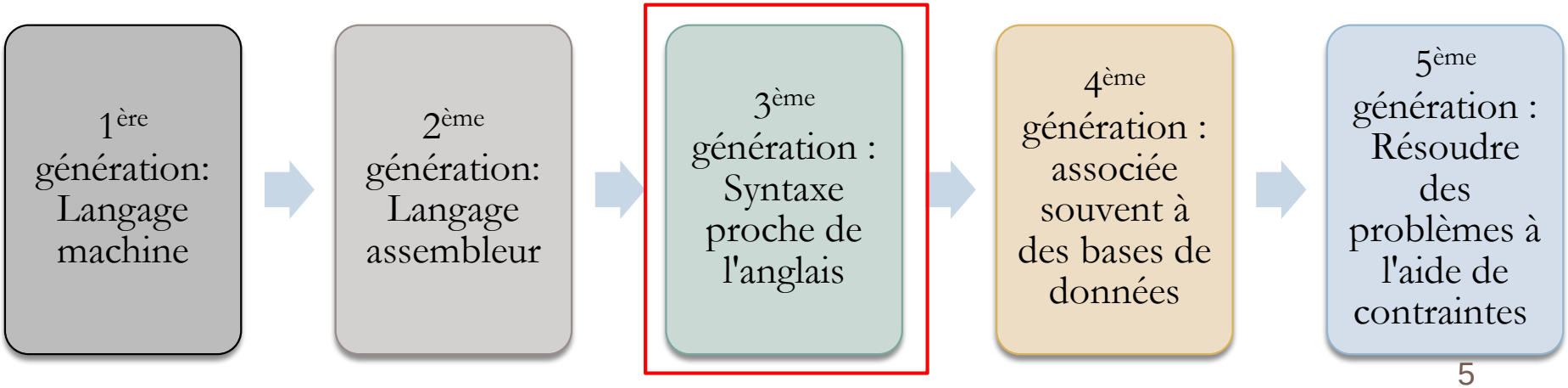
Echange de différents types d'informations (texte, images etc.)

Architecture Harvard

Langage de Programmation

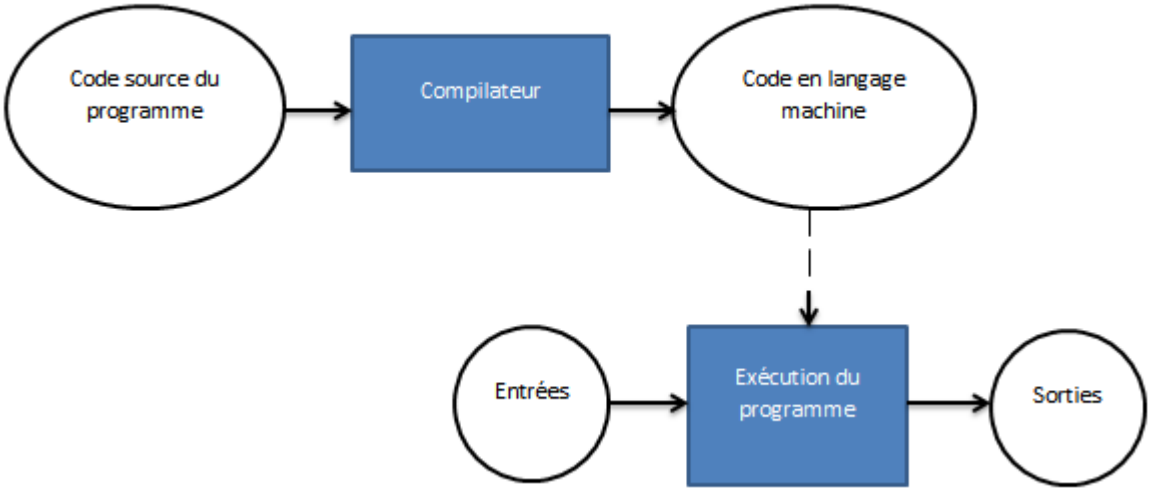
- Un langage de programmation est un code de communication, permettant à un être humain de dialoguer avec une machine en lui soumettant des instructions et en analysant les données matérielles fournies par l'ordinateur.

- On distingue aujourd'hui cinq générations de langages:



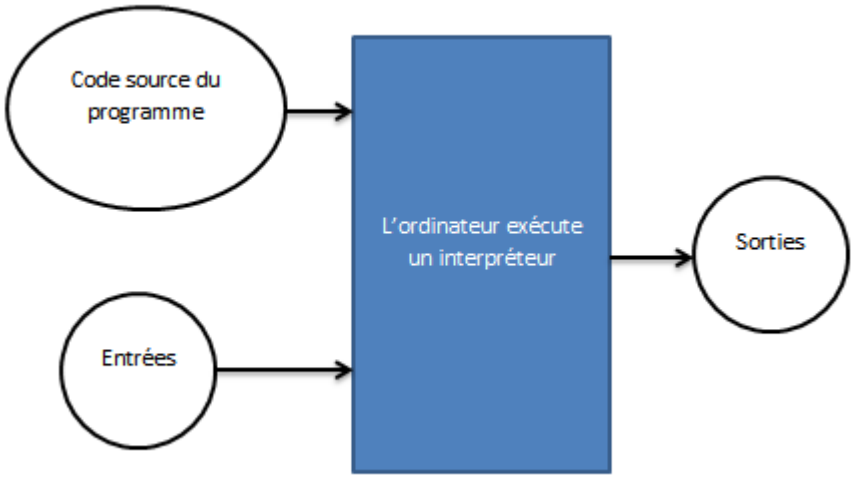
Langage compilé vs interprété

- Langage compilé : le code source du programme est transféré directement en langage compréhensible par la machine à l'aide d'un compilateur.

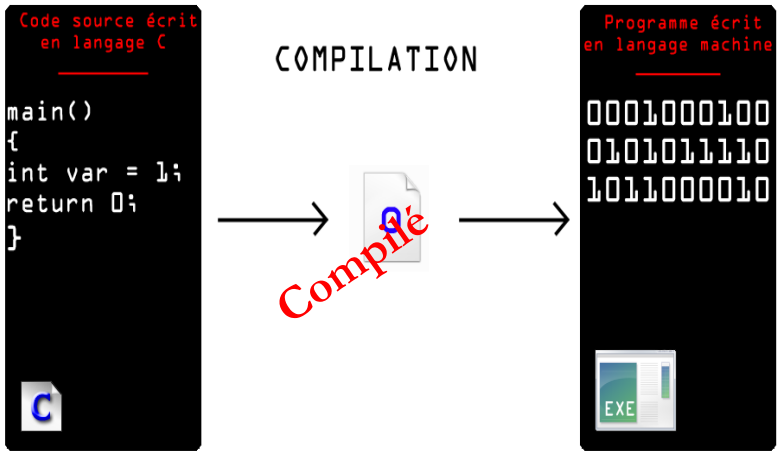


Langage compilé vs interprété

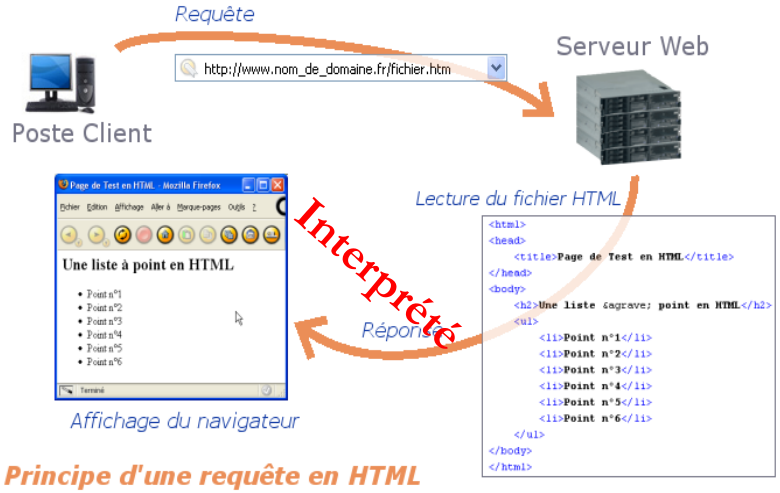
- Langage Interprété : Le code source écrit n'est pas exécuté directement par la machine, et a besoin d'un autre programme auxiliaire « interprétateur » pour traduire au fur et à mesure les instructions.



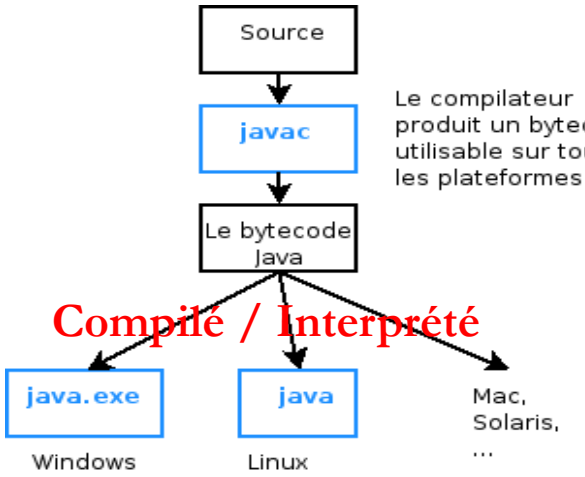
Exemple de langages



Langage C



Langage HTML

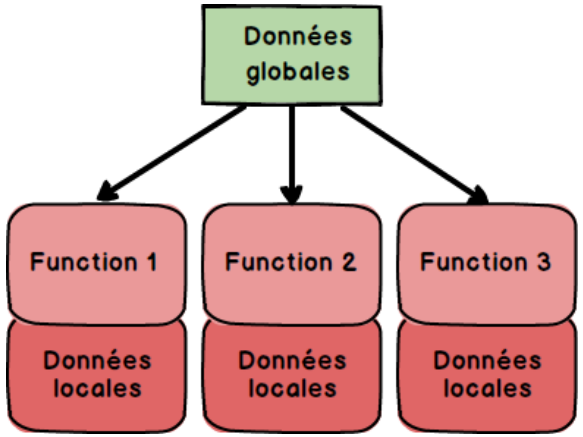


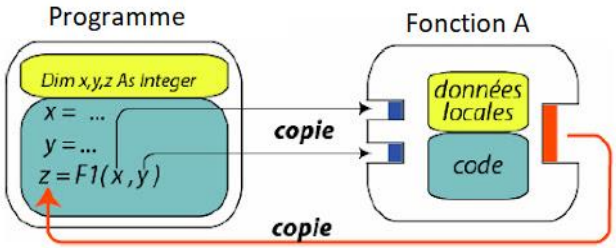
Langage Java

La programmation procédurale

- Permet d'écrire les étapes séquentielles nécessaires pour résoudre un problème
- Met l'accent sur les étapes pour réaliser une tâche
- Le programme : la liste des tâches et des opérations à exécuter

→ Convient mieux aux applications ne nécessitant pas ou peu d'interaction avec les usagers

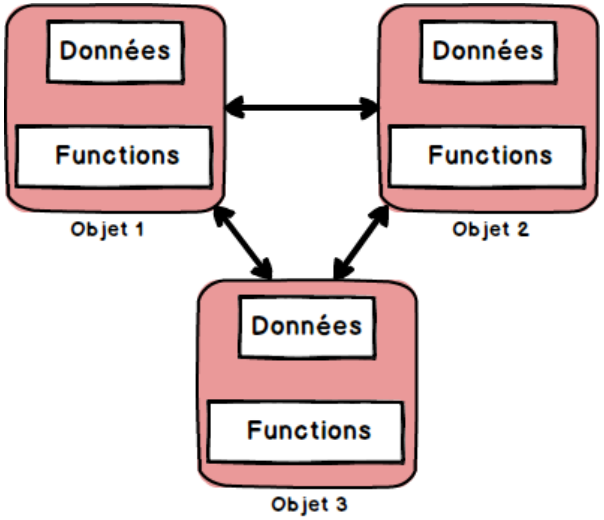




La programmation Orientée Objet

- Permet de modéliser le problème par un ensemble d'objets
- Met l'accent sur les objets requis pour résoudre un problème
- Le programme est l'ensemble des objets et des interactions entre ces objets

→ En programmation orientée objet, on cherche à identifier les objets impliqués et leurs responsabilités respectives



Section 2

Découvrir le langage C

Présentation du langage C
Forme Générale d'un programme
Compilation d'un programme
Types de données de base

Un peu d'historique

- Le langage C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs dans le but d'écrire le système d'exploitation Unix.
- En 1974, Bell Labs ont accordé des licences UNIX aux universités :
  - → 1978: Le standard K&R-C publié dans le livre « The C Programming language »
  - → 1980: Bjarne Stroustrup a étendu le langage C avec le concept de classe.
  - → 1983: Le langage Objective-C a été créé par Brad Cox afin d'apporter un support de la programmation orientée objet inspiré de Smalltalk.
  - → 1989-1990: la normalisation du langage par l'ANSI (American National Standards Institute) en définissant ANSI C.
  - → 1998: La standardisation du langage C++ par ANSI et ISO.

Succès du Langage C : Pourquoi

C est un langage:

- **Universel:** Il n'est pas orienté vers un domaine d'applications.
- **Compact:** la formulation d'expressions simples, limitées mais efficaces.
- **Moderne:** C est un langage structuré, déclaratif et récursif.
- **Près de la machine et rapide :** C offre des opérateurs et des fonctions très proches de ceux du langage machine.
- **Indépendant de la machine :** C peut être utilisé sur n'importe quel système en possession d'un compilateur.
- **Portable :** en respectant le standard ANSI-C, le même programme est utilisable sur tout autre système.
- **Extensible:** le langage est animé par des bibliothèques de fonctions privées ou livrées par de nombreuses maisons de développement.

Exemple

Exercice : Ecrire un programme qui permet d'afficher :

LANGAGE C

Bonjour… !

Algorithmique

```
Debut
  ecrire(' LANGAGE C')
  retourLigne()
  ecrire(' Bonjour… !')
fin
```

Bonjour.c

```
#include <stdio.h>
void main()
{
  printf ("LANGAGE C \nBonjour… ! " );
}
```

Structure d ’un programme

Struct.c

[directives au préprocesseur]
[déclarations de variables externes, des macros]
[fonctions secondaires]
main()
{
déclarations de variables internes
Instructions : Bloc d ’instructions ou Macro-instruction

}

Directives au préprocesseur: #include

- Généralement, les programmes C font appel à des fonctions externes pour réaliser des opérations d'entrée-sortie ou autres.

- Le programmeur doit informer le compilateur sur l'existence de ces fonctions en incluant le fichier en-tête approprié :

#include <fichier.h>

- La bibliothèque la plus utilisée est **stdio.h** qui contient l'ensemble des fonctions standard d'entrée-sortie parmi lesquelles on trouve printf.

Les Fonctions

- Un programme source C se présente sous la forme d'une ou de plusieurs fonctions.
- Chaque fonction comporte un ou plusieurs blocs d'instructions pour réaliser une tâche précise.
- Chaque fonction peut faire appel à une autre fonction du programme.
- La fonction principale de tout programme C s'appelle main().

Les Bloc d ’instruction

- Le corps d’un bloc débute par le signe { et se termine par }
- Le concept de « Bloc » structure les programmes complexes.
- Les blocs peuvent être emboîtables, conditionnels ou inconditionnels.
- Les blocs contiennent soit des instructions simples, soit des appels à d’autres fonctions ou à des sous-programmes externes.

Instruction / Macro-instruction

- Une instruction est un mot clé permettant d'exécuter une tâche bien donnée.

- C permet aussi d'utiliser des macro-instructions traitées par un préprocesseur au moment même de la compilation du programme.

- Une macro-instruction peut par exemple remplacer une chaîne de caractères, inclure un fichier de code, réaliser une compilation conditionnelle.

Commentaires

- Les commentaires incorporés dans le programme sont placés entre les délimiteurs /* et */ , ou bien après //

- Toute instruction C se termine par un point-virgule excepté les déclarations de fonctions, les accolades et les macro-instructions du préprocesseur.

Section 3

Eléments de programmation

Constantes et Variables
Expressions, Opérateurs
Conversion de types
Entrées-sorties printf, scanf

Mots réservés

- Les Mots clés suivants sont réservés au langage C

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

- Il ne peuvent pas être utilisés comme identificateurs

Les Constantes

- Une constante donnée inchangée qui conserve sa valeur pendant toute l'exécution d'un programme
- En C, on associe une valeur à une constante en utilisant :

  la directive #define :
  #define nom_constante valeur
  Ici la constante ne possède pas de type.
  exemple: #define Pi 3.141592

  le mot clé const :
  const type nom = expression ;
  Dans cette instruction la constante est typée
  exemple : const float Pi =3.141592

- L'intérêt des constantes est de donner un nom parlant à une valeur, par exemple NB_LIGNES, aussi ça facilite la modification du code.

Types de Constantes : Entières

- Constantes entières: On distingue 3 formes de constantes entières :
  - Forme Décimale : c'est l'écriture usuelle. Ex : 372, 200
  - Forme Octale (base 8) : on commence par un 0 suivi de chiffres octaux. Ex : 0477
  - Forme Hexadécimale (base 16) : on commence par 0x (ou 0X) suivis de chiffres hexadécimaux (0-9 a-f). Ex : 0x5a2b, 0Xa9f.
- Le compilateur attribue automatiquement un type aux constantes entières. Il attribue en général le type le plus économique parmi (int, unsigned int, long int, unsigned long int);
- On peut forcer la machine à utiliser un type de notre choix:
  - u ou U pour unsigned int, Ex : 100U, 0xAu
  - l ou L pour long, Ex : 15l, 0127L
  - ul ou UL pour unsigned long, Ex : 1236UL, 035ul

Types de Constantes : Réelles

- Constantes entières: Une constante réelle représente un nombre à virgule flottante (de type float ou double) sous forme décimale ou exponentielle.

- On distingue 2 notations :
  - Notation Décimale Ex : 123.4, 5.27.
  - Notation Exponentielle Ex : 1234e-1 ou 1234E-1

- Les constantes réelles sont par défaut de type double.
- On peut forcer la machine à utiliser un type de notre choix:
  - f ou F pour le type float. Ex: 1.25f
  - l ou L pour le type long double. EX: 1.0L

Types de Constantes : Caractères

- Constante Caractère: représente un seul caractère indiqué entre des apostrophes, Ex : ‘b’, ‘A’, ‘?’
- La valeur d’une constante caractère est le code ASCII du caractère
- Les caractères constants peuvent apparaître dans des opérations arithmétiques ou logiques
- Les constantes caractères sont de type int

code	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	LF	VT	NP	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	”	#	$	%	&	’
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	‘	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	-	DEL		

Séquences d ’échappement

- L’affichage du texte peut être contrôlé à l'aide des séquences d'échappement : composées d’un antislash suivi d’une suite de caractères.

\a	alerte (avertisseur sonore)
\b	retour arrière (backspace)
\c	suppression du retour-chariot final
\f	saut de page
\n	nouvelle ligne
\r	retour-chariot
\t	tabulation horizontale
\v	tabulation verticale
\\	backslash
\nnn	le caractère dont le code ASCII octal vaut nnn (un à trois chiffres)
\xnnn	le caractère dont le code ASCII hexadécimal vaut nnn (un à trois chiffres)

Variables

- Variable : un emplacement mémoire contenant une donnée modifiable, caractérisée par un nom identificateur et un type.
- Les variables doivent être déclarées avant d'être utilisées

Syntaxe :

type ***nomvar1, nomvar2, nomvar3,… ;***

- **type** représente un type de donnée
- le nom d'une variable est composé de lettres, de chiffres et du caractère souligné, et ne doit pas commencer par un chiffre.
- Le compilateur différencie les majuscules et les minuscules dans les noms des variables.

Variables : Types de base

- Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre et le nombre d'octets à lui réserver en mémoire.
- En langage C, il n'y a que deux types de base les entiers et les réels avec différentes variantes pour chaque type.
- 4 variantes d'entiers :
  - char : caractères (entier sur 1 octet : - 128 à 127)
  - short ou short int : entier court (entier sur 2 octets : - 32768 à 32767)
  - int : entier standard (entier sur 2 ou 4 octets)
  - long ou long int : entier long (4 octets : - 2147483648 à 2147483648)
- 3 variantes de réels :
  - float : réel simple précision codé sur 4 octets de -3.4*Và 3.4*10^{38}
  - double : réel double précision codé sur 8 octets de -1.7*10^{308} à 1.7*10^{308}
  - long double : réel très grande précision codé sur 10 octets de -3.4*10^{4932} à 3.4*10^{4932}

Variables : Exemple

Variables.c

```
#include <stdio.h>
main()
{
char  _L, 2A ;

int  A, B, prix  HT, prix.TTC, iNbre_1 ;

float  X, Y, fNbre1, PRIX- -HT  ;

}
```

incorrect

Opérateurs en C

- Le langage C est riche en opérateurs. Outre les opérateurs standards, il comporte des opérateurs originaux d'affectation, d'incrémentation et de manipulation de bits.

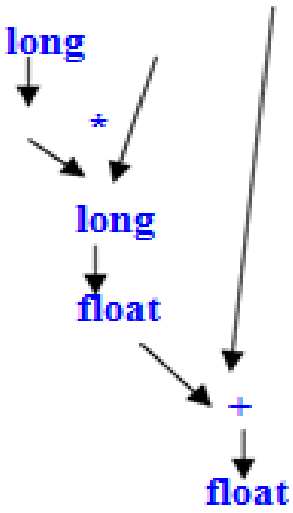
- On distingue les opérateurs suivants en C :
  - les opérateurs arithmétiques : +, -, *, /, %
  - les opérateurs d'affectation : =, +=, -=,*=,/=,…
  - les opérateurs logiques : &&, ||, !
  - les opérateurs de comparaison : ==, !=, <, >, <=, >=
  - les opérateurs d'incrémentation et de décrémentation : ++, --
  - les opérateurs sur les bits : <<, >>, &, |, ~, ^
  - d'autres opérateurs particuliers : ?:, sizeof, cast

Opérateurs arithmétiques

- binaires : + - * / et % (modulo) et unaire : -
- Les opérandes peuvent être des entiers ou des réels sauf pour % qui agit uniquement sur des entiers.
- Lorsque les types des deux opérandes sont différents il y'a conversion implicite dans le type le plus fort
- Les types short et char sont systématiquement convertis en int indépendamment des autres opérandes
- La conversion se fait en général selon une hiérarchie qui n'altère pas les valeurs int → long →float → double → long double

Exemple de conversion

Exemple : n * p + x (int n ; long p ; float x)



conversion de n en long

multiplication par p

n * p de type long

conversion de n * p en float

addition

résultat de type float

Opérateur d'affectation =

- L'opérateur = affecte une valeur ou une expression à une variable

  Exemple: double x,y,z; x=2.5; y=0.7; z=x*y-3;

- Le terme à gauche de l'affectation est appelé lvalue (left value)
- L'affectation est interprétée comme une expression. La valeur de l'expression est la valeur affectée
- On peut enchainer des affectations, l'évaluation se fait de droite à gauche

  Exemple : i = j = k= 5 (est équivalente à k = 5, j=k et ensuite i=j)

- La valeur affectée est toujours convertie dans le type de la lvalue, même si ce type est plus faible (ex : conversion de float en int, avec perte d'information)

Opérateurs relationnels

- Opérateurs:
  - < : inférieur à
  - > : supérieur à
  - == : égal à
  - <= : inférieur ou égal à
  - >= : supérieur ou égal à
  - != : différent de

- Le résultat de la comparaison n'est pas une valeur booléenne, mais 0 si le résultat est faux et 1 si le résultat est vrai
- Les expressions relationnelles peuvent donc intervenir dans des expressions arithmétiques
- Exemple: a=2, b=7, c=4
  - b==3 → 0 (faux)
  - a!=b → 1(vrai)
  - 4*(a<b) + 2*(c>=b)→ 4

Opérateurs logiques

- Opérateurs:
  - && : ET logique
  - || : OU logique
  - ! : négation logique

- && retourne vrai si les deux opérandes sont vrais (valent 1) et 0 sinon
- || retourne vrai si l'une des opérandes est vrai (vaut 1) et 0 sinon
- Les valeurs numériques sont acceptées : toute valeur non nulle correspond à vraie et 0 correspond à faux
- Le 2ème opérande est évalué uniquement en cas de nécessité
  - a && b : b évalué uniquement si a vaut vrai
  - a || b : b évalué uniquement si a vaut faux.

Opérateurs d'affectation Et d'incrémentation

- Les opérateurs ++ et -- sont des opérateurs unaires permettant respectivement d'ajouter et de retrancher 1 au contenu de leur opérande: effectuée après ou avant l'évaluation de l'expression :
  - k = i++ (post -incrémentation) : affecte d'abord la valeur de i à k et incrémente après (k = i++ ; → k = i ; i = i+1 ;)
  - k = ++i (pré-incrémentation) incrémente d'abord et après affecte la valeur incrémentée à k (k = ++i ; → i = i+1 ; k = i ;)
- L'expression *exp1= exp1 op exp2* peut s'écrire en général de façon équivalente sous la forme *exp1 op= exp2.*
- Exemple :
  - i = 5 ; n = ++i - 5 ; — i vaut 6 et n vaut 1
  - i = 5 ; n = i++ - 5; — i vaut 6 et n vaut 0
  - a=a+b s'écrit : a+=b et n=n%2 s'écrit : n%=2

Opérateurs de manipulations de bits

- Opérateurs arithmétiques bit à bit :
  - & : ET logique
  - | : OU logique
  - ^ : OU exclusif
  - ~ : négation

- Les opérandes sont de type entier. Les opérations s'effectuent bit à bit suivant la logique binaire

b1	b2	~b1	b1&b2	b1|b2	b1^b2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Exemple : 14= 1110 , 9=1001→ 14 & 9= 1000=8, 14 | 9 =1111=15

Opérateurs de décalage de bits

- Opérateurs arithmétiques bit à bit :
  - >> : décalage à droite
  - << : décalage à gauche

- Dans le cas d'un décalage à gauche les bits les plus à gauche sont perdus. Les positions binaires rendues vacantes sont remplies par des 0

  Exemple : char x=14; (14=00001110) → 14<<2 = 00111000 = 56

- Lors d'un décalage à droite les bits les plus à droite sont perdus:
  - si l'entier à décaler est non signé, les positions binaires rendues vacantes sont remplies par des 0.
  - s'il est signé le remplissage dépend de l'implémentation (en général le remplissage se fait par le bit du signe).

  Exemple : Ex : char x=14; (14=00001110) → 14>>2 = 00000011 = 3

Opérateur forçage de type (cast) et SIZEOF

- Il est possible d'effectuer des conversions explicites ou de forcer le type d'une expression.
  - Syntaxe : (<type>) <expression>
  - Exemple : int n, p ;

    (double) (n / p); // convertit l'entier n / p en double

- Il est possible de manipuler la taille en octets d'un type ou d'une variable
  - Syntaxe : sizeof (<type>) ou sizeof (<variable>)

- Exemple

```
float n = 4.6, p = 1.5 ;
(int) n / (int) p = 4 / 1 = 4
(int) n / p = 4 / 1.5 = 2.66
n / (int) p = 4.6 / 1 = 4.6
n / p = 4.6 / 1.5 = 3.06
```

```
int n;
printf ("%d \n",sizeof(int));    // affiche 4
printf ("%d \n",sizeof(n));  // affiche 4
```

Bibliothèque de fonctions E/S

- Il s'agit des instructions permettant à la machine de dialoguer avec l'utilisateur
  - Dans un sens la **lecture** permet à l'utilisateur d'entrer des valeurs au clavier pour qu'elles soient utilisées par le programme
  - Dans l'autre sens, l'**écriture** permet au programme de communiquer des valeurs à l'utilisateur en les affichant à l'écran (ou en les écrivant dans un fichier)

- Le langage C dispose d'une bibliothèque de fonctions (stdio.h) pour gérer les entrées-sorties.
- Il suffit d'inclure, au début du programme, l'instruction préprocesseur :

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

Ecriture formatée de données: printf ()

- la fonction printf est utilisée pour afficher à l'écran du texte, des valeurs de variables ou des résultats d'expressions

- Syntaxe : printf("format", expr1, expr2, …);
  - Expr : sont les variables et les expressions dont les valeurs sont à représenter
  - Format : est une chaîne de caractères qui peut contenir
    - du texte
    - des séquences d'échappement ('\n', '\t', …)
    - des spécificateurs de format : un ou deux caractères précédés du symbole %, indiquant le format d'affichage

Fonction printf : codes de formats

- Les codes de formats permettent d'interpréter ou convertir les données des arguments à afficher.
- Ces codes doivent correspondre en nombre et en type aux arguments dans la fonction printf.

SYMBOLE	TYPE	AFFICHAGECOMME
%dou%i	*int*	Entier relatif
%u	*Unsinged int*	Entier naturel non signé
%c	*char*	caractère
%o	*int*	Entier sous forme octale
%xou%X	*int*	Entier sous forme hexadécimale
%f	*float, double*	Réel en notation décimale
%eou%E	*float, double*	Réel en notation exponentielle
%s	*char**	Chaîne de caractères

Fonction printf : Exemple

- #include<stdio.h>

```
void main()    {
        printf(" Génie ") ;
        printf("  Logiciel\n " ) ;
        printf(" Formation Langage C\n\n " ) ;
        printf("   Les caractères %c et %c ont les codes ASCII :%d et %d \n\n ”, 'A', 'a', 'A', 'a') ;
        printf("  %f est un nombre réel \n\ n" , 1.234) ;
        printf ("  Réels en notation exponentielle :%e  %e  %e %e \n " , 123.4, 0.00123, 0.01234e5, 123.40e-6) ;
        printf ("  %c" , " Fin des exemples " );
        printf ("  %s " , " Fin des exemples " );
        }
```

Lecture formatée de données: scanf ()

- La foction scanf permet de lire les données saisies à partir du clavier.

- Syntaxe : scanf("format", AdrVar1, AdrVar2, …);
  - AdrVar: adresses des variables auxquelles les données seront attribuées. L'adresse d'une variable est indiquée par le **nom** de la variable **précédé** du signe **&**
  - Format : le format de lecture de données, est le même que pour printf

Fonction scanf : Exemple

```
#include<stdio.h>
void main() {
        char  lettre ;
        int Nbre1, Nbre2 ;
        float  Nbre3 ;
        printf(" Taper une lettre : ") ;
        scanf(" %c", &lettre) ;
        printf(" Vous avez taper la lettre %c\n ", lettre) ;
        printf(" Taper trois nombres : ") ;
        scanf(" %d %d %f ", & Nbre1, &Nbre2, &Nbre3) ;
        printf  ("Les nombres entrés sont : \n \t %d \t %d  \t %f ", Nbre1,
        Nbre2, Nbre3) ;
        }
```

Fonctions getchar/putchar

- getchar () et putchar (): permet la saisie ou l'affichage non formaté d'un seul caractère :

Exemple:

```
#include <stdio.h>
void main(){
        char x ;
        printf(" Taper une lettre et valider par entrée : ") ;
        x=getchar() ;
        printf("Vous avez  tapé la lettre ") ;
        putchar(x) ;
        }
```

Section 4

Structures de contrôle

Instruction if
Instruction switch
La boucle while
La boucle do … while
La boucle for
Instructions break et continue

Structures de contrôle

- Les structures de contrôle définissent la façon avec laquelle les instructions sont effectuées. Elles conditionnent l'exécution d'instructions à la valeur d'une expression.

- On distingue :
  - Les structures alternatives (tests) : permettent d'effectuer des choix càd de se comporter différemment suivant les circonstances (valeur d'une expression). En C, on dispose des instructions : if…else et switch.
  - Les structures répétitives (boucles) : permettent de répéter plusieurs fois un ensemble donné d'instructions. Cette famille dispose des instructions : while, do…while et for.

L'instruction if…else

- Syntaxe : If (expression)
  bloc-instruction1
  else
  bloc-instruction2

  - bloc-instruction peut être une seule instruction terminée par un point-virgule ou une suite d'instructions délimitées par des accolades { }
  - expression est évaluée, si elle est vraie (valeur différente de 0), alors bloc-instruction1 est exécuté. Si elle est fausse (valeur 0) alors bloc-instruction2 est exécuté.
- La partie else est facultative. S'il n'y a pas de traitement à réaliser quand la condition est fausse, on utilisera simplement la forme :

  If (expression) bloc-instruction1

if…else : exemples

- Exemple 1:

```
float a, b, max;
if (a > b)
        max = a;
else
        max = b;
```

- Exemple 2:

```
int a;
if ((a%2)==0)
        printf(" %d est paire" ,a);
else
        printf(" %d est impaire ",a);
```

if…else : exemples

```
void main() {
        int i, j;
        printf("i et j ?\n");
        scanf("%d%d ",&i, &j);
        if (i>j)      printf( "i est plus grand que j \n");
          else        printf("i est plus petit ou égal à j\n");
        if (i<j) {
                     printf( "i est plus petit ");
                     printf(" que j \n  ");
                }
          else       printf("i est plus grand ou égal à j\n");

        }
```

if…else : Exercices

Exercice 1 :

Écrire un programme C demandant à l 'utilisateur de deviner une lettre. Ce programme devra indiquer si la réponse donnée est correcte ou non. La bonne réponse est ' a ' ou ' A '.

Exercice 2 :

Écrire un programme C où l 'utilisateur saisit un caractère, le programme teste s 'il s 'agit d 'une lettre majuscule, si oui il renvoie cette lettre en minuscule, sinon il renvoie un message d 'erreur.

Imbrication des instructions if

- Imbrication de plusieurs instructions if…else: on peut conduire à des confusions, par exemple :

```
if (N>0)
        if (A>B)        MAX=A;
        else            MAX=B;
```

→ interprétation 1 : si N=0 alors MAX prend la valeur de A ou B

→ interprétation 2 : si N=0 alors MAX ne change pas

- En C un else est toujours associé au dernier if qui ne possède pas une partie else.
- Conseil : pour éviter toute ambiguïté ou pour forcer une certaine interprétation dans l'imbrication des **if**, il vaut mieux utiliser les accolades.

L'instruction d'aiguillage switch

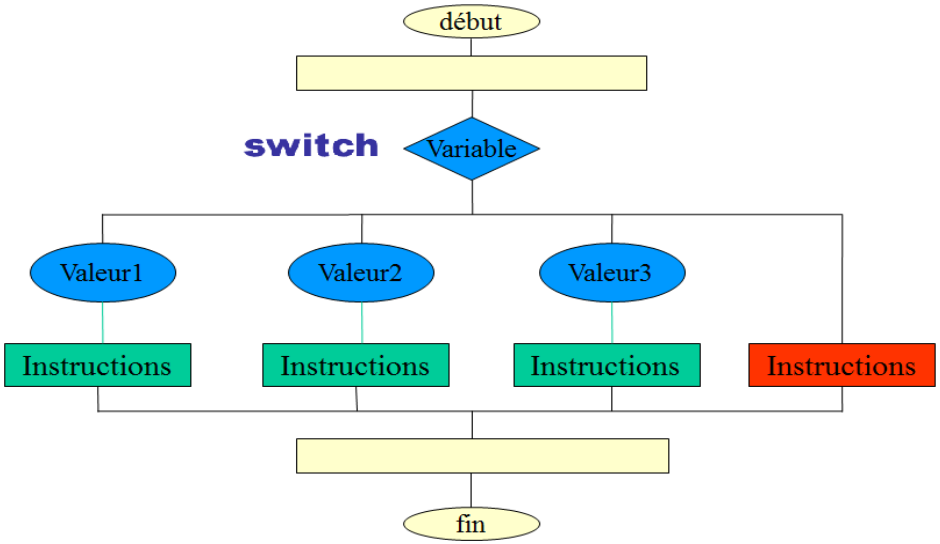
- Permet de choisir des instructions à exécuter selon la valeur d'une expression qui doit être de type entier.
- la syntaxe est :

```
switch (expression) {
        case expression_constante1 : instructions_1;    break;
        case expression_constante2 : instructions_2;  break;
                …
        case expression_constante n : instructions_n;  break;
        default : instructions;
}
```

  - expression_constantei doit être une expression constante entière.
  - Instructions_i peut être une instruction simple ou composée.
  - break et default sont optionnels et peuvent ne pas figurer.

L'instruction d'aiguillage switch

- L'expression est évaluée: si sa valeur est égale à une expression_constante i, on se branche à ce cas et on exécute les instructions_i qui lui correspondent
  - → On exécute aussi les instructions des cas suivants jusqu'à la fin du bloc ou jusqu'à une instruction break.
- si la valeur de l'expression n'est égale à aucune des expressions constantes:
  - → Si default existe, alors on exécute les instructions qui le suivent
  - → Sinon aucune instruction n'est exécutée



L'instruction switch : exemples

```
void main()          {
        int i;
        printf("i ?\n");
        scanf(%d,&i);
        switch(i)     {
                case 0:        printf( " Bonjour \n");
                               printf("la valeur est 0\n");
                case 1:        printf("la valeur est 1\n");
                case 2:        printf ("la valeur est 1 ou 2\n");
                               break;
                case 3:        break;
                default:       printf ("autre valeur\n");
                        }
        }
```

L’instruction switch : Exercices

Exercice 1 :

Écrire un programme C demandant à l ’utilisateur de deviner une lettre. Ce programme devra indiquer si la réponse donnée est correcte ou non. La bonne réponse est ‘ a ’ ou ‘ A ’.

Exercice 2:

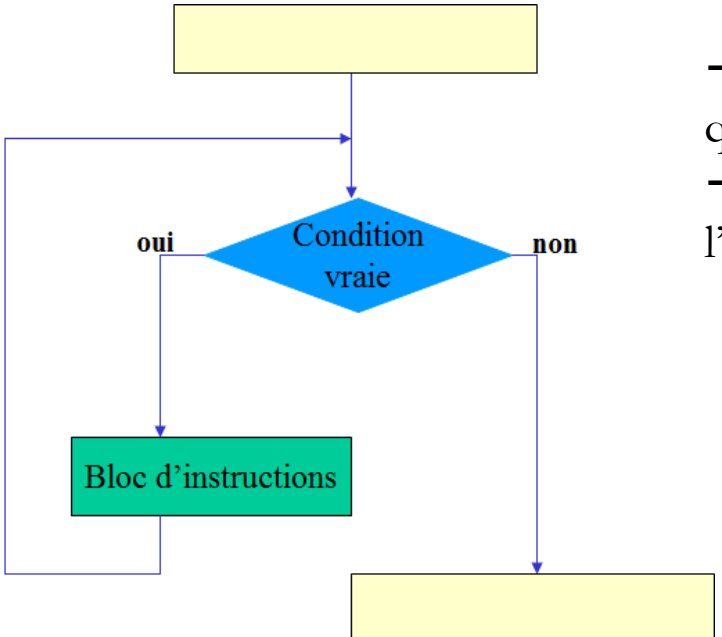
Un professeur décide d'inscrire les appréciations suivantes en fonction des résultats obtenus

- entre 9 et 10 BRAVO
- entre 8 et 9 non inclus Très Bien
- entre 6 et 8 non inclus : Bien
- entre 5 et 6 non inclus : A AMELIORER
- entre 0 et 5 non inclus: INSUFFISANT

Ecrire un programme qui demande la note (un nombre entre 0 et 10) et qui affiche l'appréciation.

L'instruction while

- la syntaxe : while (expression)
  instruction
  - instruction peut être une seule instruction terminée par un point- virgule ou une suite d'instructions délimitées par des accolades { }
  - expression est évaluée, si elle est vraie (valeur différente de 0), alors bloc-instruction est exécuté:



→ L'instruction simple ou composée est exécutée tant que la condition a pour valeur VRAI (différente de 0).
→ Si l'expression de la condition vaut dès l'entrée, l'instruction n'est jamais effectuée.

L'instruction while : Exemples

```
void main()
{
        int i = 2, j = 7;

while (i<j)
{
   printf("i: %d et j: %d\n", i,j);
          i++;
          j--;
}
}
```

Boucle limitée

```
void main()
{
int i = 0, j = 3;

while (!i)
        printf("i: %d\n", i);
}
```

Boucle infinie

L’instruction while : Exercices

Exercice 1 :

Reprendre l ’exercice des devinettes (if-else, switch), sauf que cette fois-ci, l ’utilisateur a le droit d ’essayer jusqu'à ce que sa réponse soit correcte.

Exercice 2 :

Saisir une suite de caractères, compter et afficher le nombre de lettres e et d’espaces.

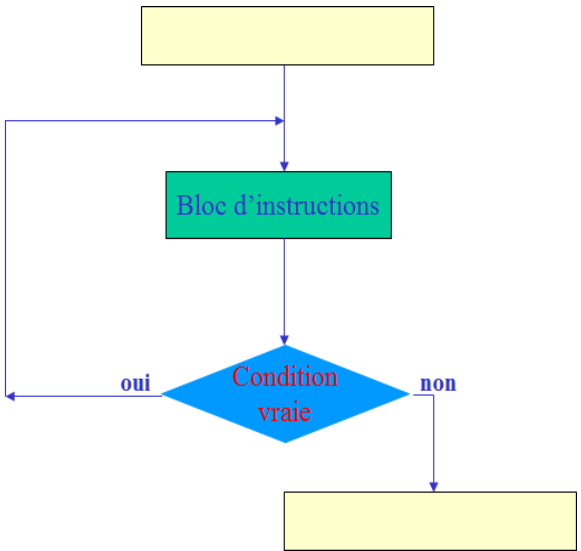
L'instruction do ... while

- la syntaxe : do

  instructions

  while (condition);

  - instruction peut être une seule instruction terminée par un point- virgule ou une suite d'instructions délimitées par des accolades { }
  - expression est évaluée, si elle est vraie (valeur différente de 0), alors bloc-instruction est exécuté:



→ L'instruction simple ou composée est exécutée jusqu'à ce que la condition prenne la valeur (FAUX)

→ Noter le point virgule à la fin de chaque instruction : obligatoire

L’instruction do while: exemples

```
void main()
{
        int i = 5, j = 0;
        do
        {
                printf("i :%d et j: %d\n", i,j);
                j++;
                i--;
        }
        while (i>j);
}
```

L’instruction do while : Exercices

Exercice 1 :

Même exercice1 que while, en utilisant cette fois la boucle do-while.

Exercice 2 :

Écrire un programme affichant les multiples de 2. La sortie se fait soit par la fonction kbhit, ou par une temporisation (de valeur égale à 5000).

La fonction kbhit appartient à la bibliothèque conio.h. Cette fonction teste si un caractère a été frappé au clavier. Tant que ce n ’est pas vrai kbhit renvoie 0 (ceci signifie que la valeur de la fonction kbhit est 0)

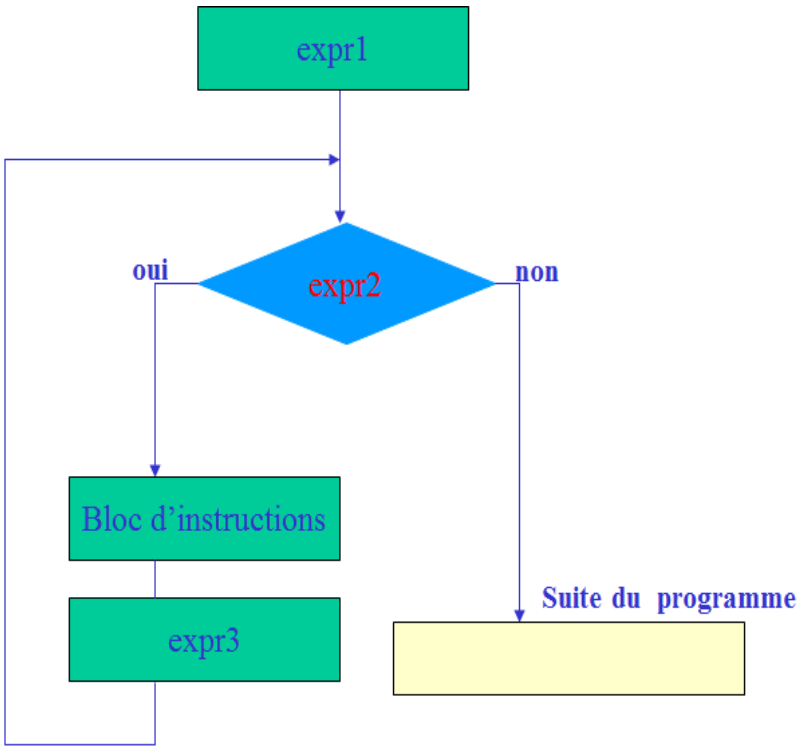
L'instruction for

- la syntaxe :

```
for (expr1 ; expr2 ; expr3)
        {
                instructions
        }
```

- L'expression expr1 est évaluée une seule fois au début de l'exécution de la boucle. Elle effectue l'initialisation des données de la boucle.
- L'expression expr2 est évaluée et testée avant chaque passage dans la boucle. Elle constitue le test de continuation de la boucle.
- L'expression expr3 est évaluée après chaque passage. Elle est utilisée pour réinitialiser les données de la boucle.

L'instruction for



For (expr1; expr2; expr3)
instructions

équivalent

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

En pratique, expr1 et expr3 contiennent souvent plusieurs initialisations ou réinitialisations, séparées par des virgules

L'instruction for: exemples

- Exemple 1:

```
void main()  {
    int i;
    for (i=2; i<5; i++)
        printf("i:%d \n",i);
    }
```

- Exemple 2:

```
void main() {
    int i,j;
    for (i=2, j=4; i<5 && j>2; i++, j--)
        printf("i:%d et j:%d\n",i,j);
    }
```

L’instruction for : Exercices

Exercice 1 :

Même exercice que pour while et do-while, cette fois-ci l ’utilisateur n ’a droit qu ’à 3 essais.

Exercice 2 :

Écrire un programme C où l ’utilisateur saisit un entier n. Le programme devra calculer et afficher n!.

L'instruction break

- L'instruction **break** peut être utilisée dans une boucle (for, while, ou do .. while). Elle permet d'arrêter le déroulement de la boucle et le passage à la première instruction qui la suit.
  - → En cas de boucles imbriquées, break ne met fin qu' à la boucle la plus interne

- L'instruction **continue** peut être utilisée dans une boucle (for, while, ou do .. while). Elle permet l'abandon de l'itération courante et le passage à l'itération suivante

L'instruction break

- L'instruction **break** peut être utilisée dans une boucle (for, while, ou do .. while). Elle permet d'arrêter le déroulement de la boucle et le passage à la première instruction qui la suit.

  →En cas de boucles imbriquées, break ne met fin qu' à la boucle la plus interne

- Exemple:

```
void main () {  int i,j;
                for(i=0;i<4;i++)
                        for (j=0;j<4;j++)
                         { if(j==1) break;
                                 printf("i=%d,j=%d\n ",i,j);
                        }}
```

Résultat:

```
i=0,j=0
i=1,j=0
i=2,j=0
i=3,j=0
```

L'instruction continue

- L'instruction **continue** peut être utilisée dans une boucle (for, while, ou do .. while). Elle permet l'abandon de l'itération courante et le passage à l'itération suivante.
- Exemple:

```
void main () {
        int i;
        for(i=1;i<5;i++)   {
        printf("début itération %d\n " ,i);
                if(i<3) continue;
                printf("  fin itération %d\n " ,i);
    } }
```

Résultat:

```
début itération 1
début itération 2
début itération 3
fin itération 3
début itération 4
fin itération 4
```

L'instruction continue et break: exemples

```
void main()
{
  int i, j;
  … /* instruction affectant i et j */
      for (;i>0 && j>0;i--,j--)
      {
            if (i==5) continue;
        printf("i :% d et j : %d\n", i,j);
            if (j==5) break;
      }
}
```

Conditions initiales	affichage
i = 2 et j = 3	i : 2 et j : 3 i: 1 et j : 2
i = 5 et j = 3	i : 4 et j : 2 i : 3 et j : 1
i = 3 et j = 5	i : 3 et j : 5

Section 5

Les Tableaux et Structures

Les Tableaux

- Un tableau est une suite séquentielle de cellules, chacune d'elles pouvant contenir une donnée de même type désignées par un seul identificateur.

- Les éléments ou composantes du tableau sont stockées en mémoire à des emplacements contigus .

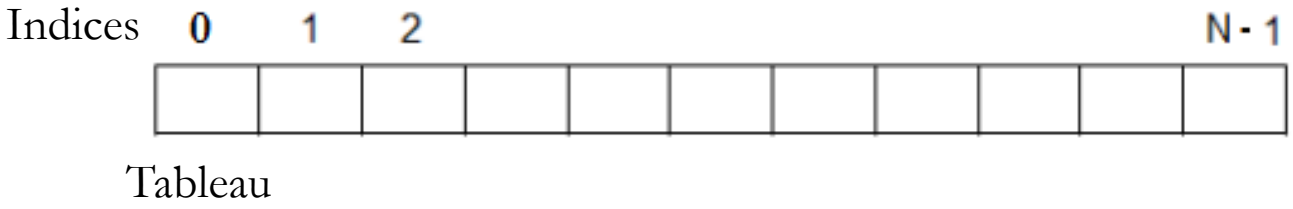
- Le type des éléments du tableau peut être :
  - Simple : char, int, float, double, …
  - Pointeur ou structure (chapitres suivants).

- On peut définir des tableaux :
  - à une dimension (tableau unidimensionnel ou vecteur)
  - à plusieurs dimensions (tableau multidimensionnel)

Tableaux à une dimension : Déclaration

- La déclaration d'un tableau à une dimension s'effectue en précisant le type de ses éléments et sa dimension :
- la syntaxe : Type identificateur [dimension];
  - dimension : nombre d 'éléments du tableau
  - identificateur: nom du tableau
  - Type : type des éléments du tableau

- La déclaration d'un tableau permet de lui réserver un espace mémoire dont la taille (en octets) est égal à : dimension * taille du type
- Exemple:
  - int N; float Tableau [N]; // on réserve 4 * N octets



Initialisation et Accès au tableau

- L'initialisation des éléments d'un tableau lors de la déclaration: indiquer la liste des valeurs respectives entre accolades.
  - Si la liste ne contient pas assez de valeurs, les composantes restantes sont initialisées par zéro.
  - la liste ne doit pas contenir plus de valeurs que la dimension du tableau.
  - Il est possible de ne pas indiquer la dimension lors de l'initialisation.

- L'accès à un élément du tableau se fait au moyen de l'indice.
  - T[i] donne la valeur de l'élément i du tableau T.
  - L'indice du premier élément du tableau est 0 et l'indice du dernier élément est égal à la dimension-1.
  - les tableaux sont traités élément par élément de façon répétitive en utilisant des boucles: on ne peut pas écrire printf(" %d",T) ou scanf(" %d",&T).

Les Tableaux : Exemples

- float B[4] = {-1.5, 3.3, 7e-2, -2.5E3};
- short Tab[10] = {1, 2, 3, 4, 5}; //les éléments restantes sont initialisées par zéro
- short Ta[3] = {1, 2, 3, 4, 5}; → Erreur
- short Tb[ ] = {1, 2, 3, 4, 5}; // Taille déclarée explicitement tableau de 5 éléments
- int tab [7]; tab[0]=2; tab[5]=2; ou bien tab[0]=tab[5]=2;
- tab[2*i-2] = 6; // L'indice peut être une variable ou une expression arithmétique
- for (i=0; i<5; i++) V[i] = 0; ou int V[5] = {0, 0, 0, 0, 0};
  ou bien int V[5] = {0}; //Initialisation au moment même de la déclaration du tableau

- Saisie / Affichage des éléments d'un tableau T d'entiers de taille n :

```
for(i=0;i< tailleTableau;i++){
    printf ("Entrez l'élément %d \n ",i + 1);
    scanf(" %d" , &T[i]);
    }
```

```
for(i=0;i< tailleTableau;i++){
    printf (" %d \t",T[i]);
    }
```

Les Tableaux : Exercices

Exercice 1 :

Réaliser un programme qui cherche le nombre maximal du tableau et qui affiche, par la suite, le nombre d'occurrence.

Exercice 2 :

Écrire un programme qui calcul le nombre d'étudiants qui ont eu une note supérieure à 10.

Exercice 3 :

Écrire un programme qui compte les fréquences des voyelles dans une saisie.

Le caractère de fin de saisie EOF est produit par une combinaison de touches : CTRL+Z

Tableaux à plusieurs dimensions

- La déclaration d'un tableau à une dimension s'effectue de la façon suivante:

  Type Nom_du_Tableau[D1][D2]…[Dn];

  - où Di est le nombre d'éléments dans la dimension i.

- Exemple : pour stocker les notes de 20 étudiants en 5 modules dans deux examens, on peut déclarer un tableau :

  float notes[20][5][2];

  → notes[i][j][k] est la note de l'examen k dans le module j pour l'étudiant i

- Cas particulier : un tableau de deux dimension (un tableau des tableaux.)

  Type nom_du_Tableau[nombre_ligne][nombre_colonne];

- Exemple : short A[2][3]; le tableau A est représenté de la manière suivante :

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]

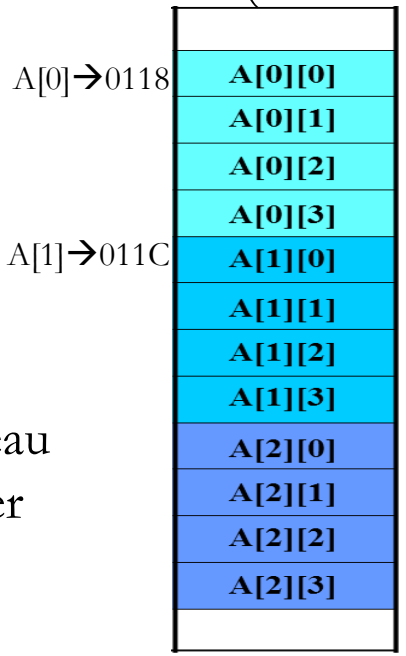
Tableaux à deux dimensions

- Un tableau à deux dimensions A[n][m] est à interpréter comme un tableau unidimensionnel de dimension n dont chaque composante A[i] est un tableau unidimensionnel de dimension m.

- Un tableau à deux dimensions A[n][m] contient n* m composantes. Ainsi lors de la déclaration, on lui réserve un espace mémoire dont la taille (en octets) est égal à : n*m* taille du type.

- Les éléments d'un tableau sont stockées en mémoire à des emplacements contigus ligne après ligne

- Comme pour les tableaux unidimensionnels, le nom d'un tableau A à deux dimensions est le représentant de l'adresse du premier élément A=&A[0][0]



Tableaux à deux dimension : Exercices

Exercice 1 :

Réaliser un programme qui permet de calculer la trace d'une matrice.

Exercice 2 :

Ecrire un programme qui permet de vérifier si une matrice est symétrique.

Le problème du tri

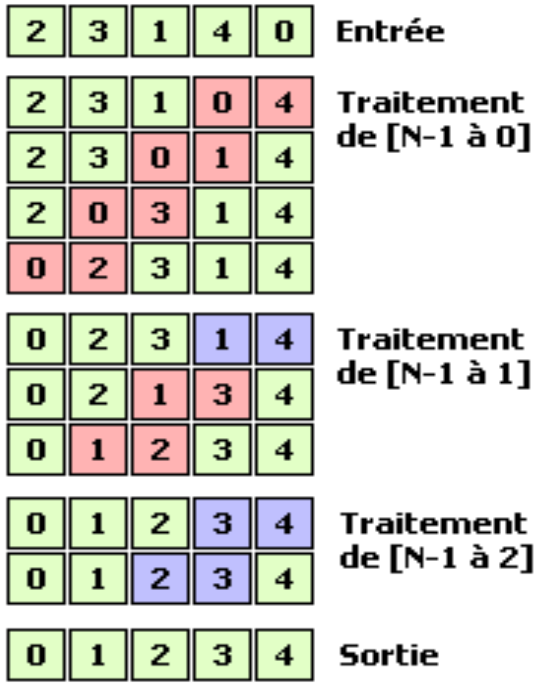
- Opération de tri: ordonner un ensemble d'éléments dans un tableau en fonction de clés sur lesquelles est définie une relation d'ordre.

- Il faut connaitre qu'il existe plusieurs types de tris (parmi les quels) :
  - Tri a bulle
  - Tri par sélection
  - Tri rapide
  - Tri par insertion
  - Tri par permutation
  - Tri par fusion

Tri par bulle

- Tri par bulle : cet algorithme parcourt le tableau en comparant 2 cases successives , lorsqu'il trouve qu'elles ne sont pas dans l'ordre souhaité (croissant dans ce cas) , il permute ces 2 cases. a la fin d'un parcours complet on aura le déplacement du minimum a la fin du tableau . en faisant cet opération N fois , le tableau serait donc trié.

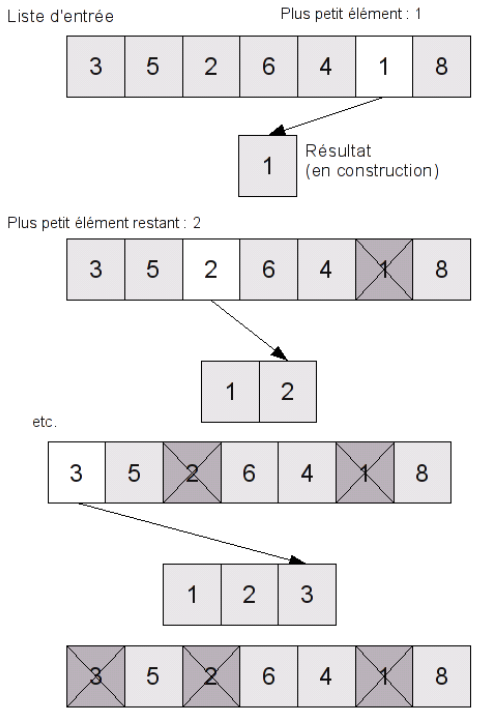
```
for(j=1;j<=N;j++) // pour faire l'opération N fois
   for(i=0;i<N-1;i++)
      if ( T[i] > T[i+1] ) {
            c = T[i];
            T[i] = T[i+1];
            T[i+1] = c;
      }
```



Tri par sélection

- Cette méthode consiste rechercher le plus grand élément (ou le plus petit), le placer en fin de tableau (ou en début), recommencer avec le second plus grand (ou le second plus petit), le placer en avant-dernière position (ou en seconde position) et ainsi de suite jusqu'à avoir parcouru la totalité du tableau.

```
int i,j,c;
for(i=0;i<N-1;i++)
   for(j=i+1;j<N;j++)
      if ( T[i] > T[j] ) {
         c = T[i];
         T[i] = T[j];
         T[j] = c;
      }
```



Les Structures

- Un tableau permet de regrouper des éléments de même type (codés sur le même nombre de bits et de la même façon).
  - Objectif : rassembler des éléments de type différent tels que des entiers et des chaînes de caractères.

- Une structure permet de rassembler sous un même nom des données de types différents

- Une structure peut contenir des donnés entières, flottantes, tableaux , caractères, pointeurs, etc...

- Les objets contenus dans la structure sont appelés champs de la structure

Les Structures : Déclaration

- La déclaration d'une structure s'effectue en précisant les variables qu'elles la composent, appelées champs, et en spécifiant leurs types (peut être n'importe quel autre type, même une structure)

- la syntaxe :

```
struct nom_struct {
//déclarations des champs;
}nom_var;
```

Ou

```
typedef nom_struct {
//déclarations des champs;
} nom_struct;
```

- Les champs d'une structure peuvent être :
  - De n'importe quel type de base
  - Des tableaux d'éléments de type quelconque.
  - Des pointeurs.
  - Des structures (déclarés au préalable).

Les Structures : Exemples de déclaration

```
struct compte {
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
        };
struct compte a,b,c; /*déclaration de 3 variables de ce type*/
```

```
struct compte {
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
        } a, b;   /*déclaration de 2 variables de ce type*/
struct compte c; /*déclaration de 1 variable de ce type*/
```

```
struct  { /* le nom de la structure est facultatif */
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
      } a,b,c;   /*déclaration de variables de ce type ici */
```

Déconseillé : plus de possibilité de déclarer d'autres variables de ce type

```
typedef  struct compte {
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
        } cpt ;   /* cpt est alors un type équivalent à struct  compte*/
cpt a,b,c; /*déclaration de variables de ce type*/
```

Recommandé : on ne se sert plus de "struct nom_struct " par la suite

Initialisation et Accès au Structure

- La portée du nom d'un membre est limité à la structure dans laquelle il est défini: On peut avoir des membres homonymes dans des structures distinctes.

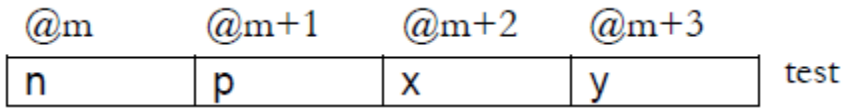
```
struct st1 {                    struct st2 {
        int x;                          char x;
        char y;  };                     int y;   };
```

- Une structure peut comporter des champs de type struct mais il faut les déclarer au préalable.
- Pour accéder à un membre d'une structure, utiliser la syntaxe :

nom_variable.nom_membre ou nom_pointeur->nom_membre

- Le langage c impose aux objets de type struct les deux contraintes suivantes :
  - les champs alloués selon leur ordre d'apparition dans la structure.
  - l'adresse d'une structure correspond à l'adresse de son premier champ.

```
struct st1 {
        int n,p;
        char x,y;  };
struct st1 test ;
```



Les Structures : Exemples

```
main()
{
  struct signe{ int jour, mois,
                        char zodiac[10] ;
                } ;
   struct signe d1, d2 ;
      // 2 variables de type  signe

   d1.jour = 14 ;
   strcpy(d1.zodiac, " bélier ") ;
   d2.mois = 11 ;
   d2.jour = d1. jour +1 ;
}
```

Accès direct

```
main()
{
   struct signe { int jour, mois ;
                        char zodiac[10] ;
                } ;
  struct signe d3,      // variable de struct signe
  *p1 ;            /*pointeur vers var de struct signe */
   p1=&d3 ; // p1 contient l'adresse du 1er octet de d3
   p1->mois = 12 ;     /* d3. Mois vaut 12 */
  strcpy(p1->zodiac, " bélier ") ;
   printf(" mois:\nsigne :%s \n ",p1->mois,p1->
zodiac) ;
}
```

Accès via un pointeur

Les Structures : Exercices

Exercice :

Reprendre l'exercice des devinettes, avec 3 essais maximum.

Utilisez une structure stock, où on peut trouver la lettre à deviner en minuscules, en majuscules, ainsi que le nombre d'essais maximum.

Section 6

Les fonctions

Notions de fonctions

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre.
  - les découper en des parties : les fonctions.

- Les fonctions sont des groupes d'instructions permettent de décomposer un programme à des parties bien définies et de simplifier à la fois la réalisation et la mise au point du problème.

- Les fonctions permettent de :
  - factoriser les programmes: mettre en commun les parties qui se répètent
  - structurer et d'offrir une meilleure lisibilité des programmes
  - faciliter la maintenance du code (une seule modification)
  - être réutilisées dans d'autres programmes

Les fonctions: Déclaration

- Syntaxe : Type nom_fonction (type1 p1, type2 p2…, typen pn)

```
{       [Définition de variables locales]
        [Définitions de variables externes]
        [Instructions]
        [Résultats retournés]
        }
```

  - Type : le type du résultat retourné. Si la fonction n'a pas de résultat à retourner, elle est de type **void**.
  - **Typei pi** (entre parenthèses) : les arguments de la fonction et leurs types.
  - Pour fournir un résultat en quittant une fonction, on dispose de la commande **return**.

Appel d'une fonction

- L'appel d'une fonction se fait par écriture de son nom avec la liste des paramètres :

  nom_fonction (para1,…, paraN)

- L'ordre et les types des **paramètres effectifs** (spécifiés lors de l'appel) doivent correspondre à ceux des **paramètres formels** (spécifiés lors de la définition).
- Il est interdit en C de définir des fonctions à l'intérieur d'autres fonctions: soit avant, soit après la fonction principale main.
- Si une fonction est définie après son premier appel (après main), elle doit être déclarée auparavant à l'aide de son prototype :

  type nom_fonction (type1,…, typeN)

- Une fonction peut se trouver :
  - dans le même module de texte (toutes les déclarations et instructions qui la composent s'y trouvent).
  - être incluse automatiquement dans le programme (par une directive #include)
  - ou compilée séparément (réunie au programme lors de la phase d'édition de liens).

Les Fonctions: Exemples

Une fonction qui calcule la somme de deux réels et une autre qui affiche le résultat :

```
double Som (double x, double y )
{ return (x+y);  }

void AfficheSom (double x, double y)
{ printf (" %lf", x+y ); }
```

Une fonction qui calcule le triple d'un entier nb :

```
int triple(int nb)
 {    return(nb*3) ;      }

main()  {
    int i, j ;
    i = 2 ;
     j = triple(i) ;
     j = triple(4) ;   }
```

Une fonction qui affiche les éléments d'un tableau d'entiers:

```
void AfficheTab (int T[ ], int n)
{
int i; for(i=0;i<n;i++)
printf (" %d \t", T[i]);
}
```

Une fonction qui calcule la valeur absolue d'un reel :

```
float ValeurAbsolue(float);
main( )
{ float x=-5.7,y;
y= ValeurAbsolue(x);
printf("La valeur absolue de %f est : %f  \n " , x,y); }

float ValeurAbsolue(float a)
{ if (a<0) a=-a; return a;  }
```

Variables locales et globales

- Le langage C permet de manipuler deux types de variables distinguées par leur portée (espace de visibilité et durée de vie):
  - Variable locale: définie à l'intérieur d'une fonction et n'est connue qu'à l'intérieur de cette fonction. Elle est créée à l'appel de la fonction et détruite à la fin de son exécution.
  - Variable globale: définie à l'extérieur des fonctions. Elle est définie durant toute l'application et peut être utilisée par les différentes fonctions du programme.

- Une variable locale cache la variable globale qui a le même nom.
- Il est recommandé d'utiliser autant que possible des variables locales: économiser la mémoire et assurer l'indépendance de la fonction.
- Les variables déclarées au début de la fonction principale main ne sont pas des variables globales, mais elles sont locales à main.

Variables locales et globales : Exemple

```
#include<stdio.h>
int x = 7;
int f(int);
int g(int);
main( ){
        printf("x = %d\t", x);
        {  int x = 6;    printf("x = %d\t", x); }
        printf("f(%d) = %d\t", x, f(x));
        printf("g(%d) = %d\t", x, g(x));
                }
        int f(int a)  { int x = 9;    return (a + x);   }
        int g(int a) { return (a * x); }
```

Résultat : x=7 x=6 f(7)=16 g(7) = 49

```
#include<stdio.h>
void f(void);
int i;
main( ){
        int k = 5;
        i=3;
        f();
        f();
        printf("i = %d et k=%d \n", i,k);     }
        void f(void)  { int k = 1;
        printf("i = %d et k=%d \n", i,k);
        i++;k++;     }
```

Résultat : i=3 et k=1
i=4 et k=1
i=5 et k=5

Transmission des paramètres

- Il existe deux modes de transmission de paramètres :
  - La transmission **par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la fonction ou procédure: le paramètre effectif ne subit aucune modification
  - La transmission **par adresse** : les adresses des paramètres effectifs sont transmises à la fonction appelante.: le paramètre effectif subit les mêmes modifications que le paramètre formel.
- La transmission des paramètres en C se fait toujours par valeur: Pour effectuer une transmission par adresse en C, il faut déclarer le paramètre formel de type pointeur et lors d'un appel de la fonction, envoyer l'adresse et non la valeur du paramètre effectif.

```
void Increment (int x,  int *y)
{ x=x+1;     *y =*y+1; }
   main( ){
                int n = 3, m=3;
                Increment (n, &m);
                printf("n = %d et m=%d \n", n,m);
                }
```

Résultat: n=3 et m=4

Transmission des paramètres : Exercices

Exercice :

Réaliser une fonction qui permet d'échanger le contenu de deux variables.

Solution:

```
void Echange (float *x,  float *y){
        float z;
         z = *x;
        *x = *y;
        *y = z;
 }
```

La notion de Récursivité

- la programmation récursive est une technique de programmation qui remplace les instructions de boucle (while, for, etc.) par des appels de fonction.
- Une fonction qui fait appel à elle-même est une fonction récursive: Il possède un cas limite (cas trivial) qui arrête la récursivité.
- l'ordre de calcul d'une fonction récursive est l'ordre inverse de l'appel de la fonction

Exemple : Calcul du factorielle

```
int  fact (int n)  {
        if (n==0)     /*cas trivial*/
        return (1);
        else          return (n* fact(n-1) );
         }
```

Fonction Itérative vs Récursive

Exercice :

Ecrivez une fonction itérative puis récursive qui calcule le terme n de la suite de Fibonacci définie par : U(0)=U(1)=1 et U(n)=U(n-1)+U(n-2)

Solution:

Fonction itérative :

```
int  Fib (int n)  {
        int i, AvantDernier, Dernier, Nouveau;
       if (n==0 || n==1)
          return (1);
        AvantDernier=1; Dernier =1;
        for (i=2; i<=n; i++)
        { Nouveau= Dernier+ AvantDernier;
           AvantDernier = Dernier;
          Dernier = Nouveau;
          }
          return (Nouveau);
                    }
```

Fonction Récursive :

```
int  Fib (int n)  {
       if (n==0 || n==1)
       return (1);
       else
       return ( Fib(n-1)+Fib(n-2));
                    }
```

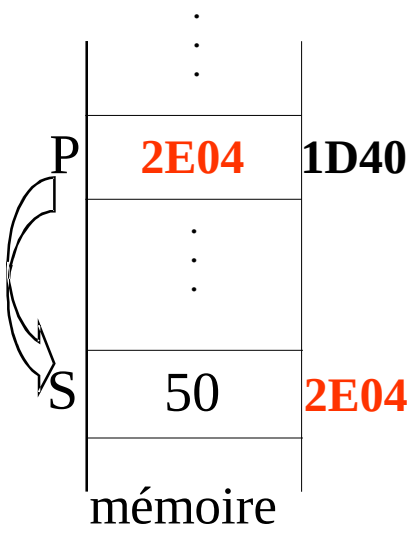
Section 7

Les Pointeurs

Notion de Pointeur

- Un pointeur est une variable spéciale qui peut contenir des adresses mémoire, d'autres variables.

Soit S une variable contenant la valeur 50 et P un pointeur qui contient l'adresse de S (on dit que P pointe sur A) .



- Le nom d'une variable permet d'accéder directement à sa valeur (adressage direct).
- Un pointeur qui contient l'adresse de la variable, permet d'accéder indirectement à sa valeur (adressage indirect).
- Le nom d'une variable est lié à la même adresse, alors qu'un pointeur peut pointer sur différentes adresses.

Les pointeurs: Déclaration

- Un pointeur est limité à un type de donnée et dépend du type de la variable pointée (même si la valeur d'un pointeur, une adresse, est un entier).

  → Connaître la taille de la valeur pointée.

- Syntaxe : Type *nom-du-pointeur ;
  - Type : le type de la variable pointée
  - * : l'opérateur qui indiquera au compilateur que c'est un pointeur.

- la valeur d'un pointeur donne l'adresse du premier octet parmi les n octets où la variable est stockée

- Pour manipuler les pointeurs, nous utilisons :
  - un opérateur 'adresse de': **&** pour obtenir l'adresse d'une variable
  - un opérateur 'contenu de': ***** pour accéder au contenu d'une adresse

Les pointeurs: Exemple de manipulation

Exemple 1:

```
int * p;   //on déclare un pointeur vers une variable de type int
int i=10, j=30; // deux variables de type int

p=&i; // p pointe sur i

printf("*p = %d \n",*p); //affiche : *p = 10

*p=20; // met la valeur 20 dans la case mémoire pointée par p (i vaut 20 après cette instruction)

p=&j; // p pointe sur j

i=*p; // on affecte le contenu de p à i (i vaut 30 après cette instruction)
```

Exemple 2:

```
float a, *p;
p=&a;

printf("Entrez une valeur : \n");
scanf("%f ",p); //supposons qu'on saisit la valeur 1.5

printf("Adresse de a= %x, contenu de a= %f\n" , p,*p);
*p+=0.5;
 printf ("a= %f\n" , a); //affiche a=2.0
```

si un pointeur P pointe sur une variable A, alors *P peut être utilisé partout où on peut écrire A

Les pointeurs: Opérations arithmétiques

- A la déclaration d'un pointeur p, on ne sait pas sur quel zone mémoire il pointe: Toute utilisation d'un pointeur doit être précédée par une initialisation
- On peut initialiser un pointeur en lui affectant :
  - l'adresse d'une variable : `int a, *p1; p1=&a;`
  - un autre pointeur déjà initialisé `int *p2; p2=p1;`
  - la valeur 0 désignée par le symbole **NULL**, défini dans <stddef.h>:

    `int *p; p=0;ou p=NULL; // on dit que p pointe 'nulle part'`
- La valeur d'un pointeur étant un entier, certaines opérations arithmétiques sont possibles (i un entier et p, p1 et p2 des pointeurs sur une variable de type T):
  - p+i (resp p-i) : désigne un pointeur sur une variable de type T. Sa valeur est égale à celle de p incrémentée (resp décrémentée) de i*sizeof(T).
  - p1-p2 : Le résultat est un entier dont la valeur est égale à (différence des adresses)/sizeof(T).
  - on peut également utiliser les opérateurs ++ et -- avec les pointeurs
  - la somme de deux pointeurs n'est pas autorisée.

Les pointeurs: Opérations arithmétiques

Exemple :

```
int *p1, *p2;
int  z =1;
p1=&z;   //si l'adresse de z correspond à 22ff44
printf("Adresse p1 = %x \n",p1);
p1++;
p2=p1+1;
printf("Adresse p1 = %x \t Adresse p2 = %x\n",p1,p2);
printf("p2-p1 = %d \n",p2-p1);
```

Affichage :

Adresse p1 = 22ff44
Adresse p1 = 22ff48 Adresse p2 = 22ff4c

p2-p1=1

Les pointeurs et les tableaux

- Le nom d'un tableau T représente l'adresse de son premier élément (T=&T[0]). Avec le formalisme pointeur, T est un pointeur constant sur le premier élément du tableau.
- En déclarant un tableau T et un pointeur P du même type, l'instruction P=T , T un tableau et P un pointeur du même type, fait pointer P sur le premier élément de T (P=&T[0]) : on peut manipuler le tableau T en utilisant P :
  - P pointe sur T[0] et *P désigne T[0]
  - P+1 pointe sur T[1] et *(P+1) désigne T[1]
  - ….
  - P+i pointe sur T[i] et *(P+i) désigne T[i]

- Exemple: short x, A[7]={5,0,9,2,1,3,8};

```
short *P; // On obtient l'adresse P+5 en ajoutant 5*sizeof(short) = 10 octets à l'adresse dans P
P=A;      // les composantes du tableau sont stockées à des emplacements contigus et &A[5]=
             &A[0]+sizeof(short)*5=A+10
x=*(P+5);  // x= A[5]
```

Pointeurs : Saisie et Affichage d'un tableau à une dimension

```
main()
  { float T[100] , *pt;
    int i,n;
    do {printf("Entrez n \n " );
        scanf(" %d" ,&n);
        }while(n<0 ||n>100);

    pt=T;
    for(i=0;i<n;i++)
        { printf ("Entrez T[%d] \n ",i );
          scanf(" %f" , pt+i);
         }

     for(i=0;i<n;i++)
          printf (" %f \t",*(pt+i));
  }
```

En utilisant i un entier pour parcourir

```
main()
  { float T[100] , *pt;
    int n;
    do {printf("Entrez n \n " );
        scanf(" %d" ,&n);
        }while(n<0 ||n>100);

     for(pt=T;pt<T+n;pt++)
        { printf ("Entrez T[%d] \n ",pt-T );
          scanf(" %f" , pt);
         }

        for(pt=T;pt<T+n;pt++)
          printf (" %f \t",*pt);
  }
```

sans utiliser un entier i

Les pointeurs et les tableaux

- Le langage C permet de définir :
  - Un tableau de pointeurs : Ex : int *T[10]; //un tableau de 10 pointeurs d'entiers
  - Un pointeur de tableaux : Ex : int (*pt)[20]; // pointeur sur tableaux de 20 éléments
  - Un pointeur de pointeurs : Ex : int **pt; //pointeur qui pointe sur des pointeurs d'entiers
- Le nom d'un tableau A à deux dimensions est un pointeur constant sur le premier élément du tableau càd A[0][0].
- En déclarant un tableau A[n][m] et un pointeur P du même type, on peut manipuler le tableau A en utilisant le pointeur P en faisant pointer P sur le premier élément de A (P=&A[0][0]), Ainsi :
  - P pointe sur A[0][0] et *P désigne A[0][0]
  - P+1 pointe sur A[0][1] et *(P+1) désigne A[0][1]
  - P+M pointe sur A[1][0] et *(P+M) désigne A[1][0]
  - ….
  - P+i*M pointe sur A[i][0] et *(P+i*M) désigne A[i][0]
  - P+i*M+j pointe sur A[i][j] et *(P+i*M+j) désigne A[i][j]

Pointeurs et tableaux à deux dimension

Exercice :

Ecrivez une programme qui permet la saisie et l'affichage d'un tableau à deux dimension en utilisant les pointeurs.

Solution:

```
#define N 10
#define M 20
main( )
   { int i, j, A[N][M], *pt;
    pt=&A[0][0];
    for(i=0;i<N;i++)
          for(j=0;j<M;j++)
               { printf ("Entrez A[%d][%d]\n ",i,j );
                  scanf(" %d" , pt+i*M+j);
                                                    }
    for(i=0;i<N;i++)
               { for(j=0;j<M;j++)
               printf (" %d \t",*(pt+i*M+j));
               printf ("\n");
               }   }
```

Allocation dynamique de mémoire

- Quand on déclare une variable dans un programme, on lui réserve implicitement un certain nombre d'octets en mémoire. Ce nombre est connu avant l'exécution du programme

- Or, il arrive souvent qu'on ne connaît pas la taille des données au moment de la programmation. On réserve alors l'espace maximal prévisible, ce qui conduit à un gaspillage de la mémoire

- Il serait souhaitable d'allouer la mémoire en fonction des données à saisir (par exemple la dimension d'un tableau).

- Il faut donc un moyen pour allouer la mémoire lors de l'exécution du programme : c'est l'allocation dynamique de mémoire

La fonction malloc et free

- La fonction malloc de la bibliothèque <stdlib> permet de localiser et de réserver de la mémoire, sa syntaxe est : malloc(N)
- Cette fonction retourne un pointeur de type char * pointant vers le premier octet d'une zone mémoire libre de N octets ou le pointeur NULL s'il n'y a pas assez de mémoire libre à allouer.

→ Exemple : Si on veut réserver la mémoire pour un texte de 1000 caractères, on peut déclarer un pointeur pt sur char (char *pt).

- Si on veut réserver de la mémoire pour des données qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un cast.

→ Exemple : on peut réserver la mémoire pour 2 variables contiguës de type int avec l'instruction : p = (int*)malloc(2 * sizeof(int)); où p est un pointeur sur int (int *p).

- Si on n'a plus besoin d'un bloc de mémoire réservé par malloc, alors on peut le libérer à l'aide de la fonction free , dont la synatxe est : free(pointeur);

Pointeur : malloc et free

Exercice :

Ecrivez une programme qui permet la saisie et l'affichage d'un tableau en utilisant l'allocation dynamique de la mémoire.

Solution:

```
#include<stdio.h>
#include<stdlib.h>
main()
  { float  *pt;
    int i,n;
    printf("Entrez la taille du tableau \n" );
    scanf(" %d" ,&n);

    pt=(float*) malloc(n*sizeof(float));
    if (pt==Null)
      {
       printf( " pas assez de mémoire \n" );
       system(" pause " );
      }
```

```
printf(" Saisie du tableau \n " );
for(i=0;i<n;i++)
  { printf ("Élément  %d ? \n ",i+1);
    scanf(" %f" , pt+i);
   }

printf(" Affichage du tableau \n " );
for(i=0;i<n;i++)
     printf (" %f \t",*(pt+i));
free(pt);
}
```

Section 8

Les chaines de caractères

Notion de chaine de caractère

- les chaînes de caractères servent à stocker les informations non numériques « du texte » (comme les noms des personnes).

- Le langage C n'offre pas la possibilité de définir un type spécial chaîne ou string : Une chaîne de caractères est traitée comme un tableau à une dimension de caractères (vecteur de caractères).

- La représentation interne d'une chaîne de caractères est terminée par le symbole '\0' (NULL) ce qui permet de détecter la fin de la chaîne .

- Il existe plusieurs fonctions prédéfinies pour le traitement des chaînes de caractères (ou tableaux de caractères)

Les Chaînes de caractères : Déclaration

- La déclaration d'une chaîne de caractères est identique à un tableau normal de type char

  char Nom_Variable [Longueur]; //tableau de caractères

  - Exemple : char Adresse [15];
- Pour une chaîne de n caractères, on a besoin de n+1 octets en mémoire (le dernier octet est réservé pour le caractère '\0')
  - le compilateur C ne contrôle pas si nous avons réservé un octet pour le symbole de fin de chaîne: l'erreur se fera seulement remarquer lors de l'exécution du programme

- Le nom d'une chaîne de caractères est le représentant de l'adresse du 1er caractère de la chaîne.
- On peut aussi manipuler les chaînes de caractères en utilisant des pointeurs : un pointeur sur char peut pointer sur les éléments d'un tableau de caractères

Initialisation des chaines de caractères

- On peut initialiser une chaîne de caractères à la définition :
  - comme un tableau : char ch[ ] = {‘e’,’c’,’o’,’l’,’e’,’\0’};
  - par une chaîne constante : char ch[ ] = "école " ;
  - en attribuant l'adresse d'une chaîne de caractères constante à un pointeur sur char : char *ch = "école" ;
- Il est possible de préciser le nombre d'octets à réserver à condition que celui- -ci soit supérieur ou égal à la longueur de la chaîne d'initialisation:
  - char ch[6] = "école"; //est valide
  - char ch[4] = "école" ; //provoque une erreur à la compilation
  - char ch[5] = "école"; //provoque une erreur à l’exécution
- Le langage C dispose d’un ensemble de bibliothèques qui contiennent des fonctions spéciales pour le traitement de chaînes de caractères: La bibliothèque <stdio.h>, la bibliothèque <string.h> et la bibliothèque <stdlib.h>.

Fonctions de la bibliothèque <stdio.h>

- printf( ) : affiche une chaîne de caractères en utilisant le spécificateur de format %s.
  - Exemple : char ch[ ]= " Bonsoir " ;
    printf(" %s ", ch);
- puts(<chaine>) : affiche la chaîne de caractères désignée et provoque un retour à la ligne.
  - Exemple : char *ch= " Bonsoir " ;
    puts(ch); /*équivalente à printf("%s\ \n ", ch);*/

- scanf( ) : saisie une chaîne de caractères en utilisant le spécificateur de format %s.
  - Exemple : char Nom[15];
  - printf("entrez votre nom");
  - scanf(" %s ", Nom);
- gets(<chaine>) : lit la chaîne de caractères désignée par <Chaîne>
  - Exemple : char phrase[100];
    printf(""entrez une phrase");
    gets(phrase);

Fonctions de la bibliothèque <string.h>

- strlen (<chaine>) : fournit la longueur de la chaîne sans compter le '\0' final
  - Exemple : char s[ ]= " Test";
    printf("%d",strlen(s)); //affiche 4

- strcat(ch1, ch2) : ajoute ch2 à la fin de ch1. Le caractère ' '\\0' de ch1 est écrasé par le 1er caractère de ch2
  - Exemple : char ch1[20]=" Bonne ", *ch2=" chance ";
    strcat(ch1, ch2) ;
    printf(" %s", ch1); // affiche Bonne chance

- strcmp(ch1, ch2): compare ch1 et ch2 lexico graphiquement et retourne une valeur :
  - nul si ch1 et ch2 sont identiques
  - négative si ch1 précède ch2
  - positive si ch1 suit ch2

Fonctions de la bibliothèque <string.h>

- strcpy(ch1, ch2) : copie ch2 dans ch1 y compris le caractère ‘ ‘\\0’
  - Exemple : char ch[10];
  - strcpy(ch, " Bonjour ");
  - puts(ch); // affiche Bonjour

- strchr(char *s, char c) : recherche la 1ère occurrence du caractère c dans la chaîne s et retourne un pointeur sur cette 1ère occurrence si c’est un caractère de s, sinon le pointeur NULL

  - Exemple : char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strchr(chaine, 'd');
    if (suiteChaine != NULL) // Si on a trouvé quelque chose
    printf("Voici la fin de la chaine a partir du premier d : %s", suiteChaine);
    // affiche Voici la fin de la chaine a partir du premier d : de test

Fonctions de la bibliothèque <stdlib.h>

- <stdlib.h> contient des fonctions pour la conversion de nombres en chaînes de caractères et vice- -versa.

- atoi(ch): retourne la valeur numérique représentée par ch comme int
- atof(ch): retourne la valeur numérique représentée par ch comme float (si aucun caractère n'est valide, ces fonctions retournent 0)
  - Exemple : int x, float y;
  - char *s= " 123 ", ch[]= " 4.56 ";
  - x=atoi(s); y=atof(ch); // x=123 et y=4.56

- itoa(int n, char * ch, int b) : convertit l'entier n en une chaîne de caractères qui sera attribué à ch. La conversion se fait en base b
  - Exemple : char ch[30]; int p=18;
  - itoa(p, ch, 2); // ch= " 10010 ";

Exercice

Exercice :

Essayer d'écrire une fonction similaire à Char* strchr(char *s, char c);

Section 9

Les fichiers

Notion de fichier

- Un programme a en général besoin :
  - De lire des données (texte, nombres, images, sons, mesures, …)
  - De sauvegarder des résultats (texte, nombres, images, sons, signaux générés, …)

    → Lecture et Ecriture dans des fichiers
- Le langage C dispose d'un ensemble de bibliothèques qui contiennent des fonctions spéciales pour le traitement des fichiers: stdio.h et cstdio.h (« standard input/output).

- Pour manipuler un fichier, on utilise un pointeur sur une donnée spécifique dont le type est FILE.

  Syntaxe: FILE *fichier
- La variable fichier contiendra l'adresse en mémoire du début du fichier.

Ouverture de fichiers

- Ouverture d'un fichier à l'aide de la fonction fopen (définie dans le fichier stdio.h) *:* Cette fonction renvoie un pointeur sur le fichier ouvert

  Syntaxe: FILE *fopen (char *Nom, char *Mode)

  - Nom :une chaîne de caractères (tableau de caractères) contenant le nom du fichier ou bien un flot de données standard.
  - Mode : désigne le type de traitement des données
    - "r" (read) : lecture (si le fichier existe)
    - "w" (write) : écriture (le fichier est écrasé s'il existe et s'il n'existe pas, il est crée)
    - "a" (append) : écriture à la fin d'un fichier existant

Exemple : fichier = fopen("C:/Data/fichier1.txt","r");

Localisation et fermeture de fichiers

- Localisation du fichier :
  - Dans le répertoire de travail (plus exactement là où est le fichier exécutable)
  - Le chemin absolu d'accès au fichier peut être donné (attention les \ de windows deviennent des / en C)
- L'extension du fichier : permet à l'OS de l'ordinateur de sélectionner le programme permettant d'ouvrir le fichier.

Exemple : fichier = fopen("C:/Data/fichier1.txt","r");

fichier = fopen("fichier1","r");

- Fermeture d'un fichier à l'aide de la fonction fclose (Il faut toujours fermer un fichier après l'avoir utilisé afin de libérer la mémoire) :

Syntaxe: int fclose(FILE *fichier) ;

Exemple : fclose(fichier) ;

Types de fichiers en C

- Il existe en C deux types de fichiers :
  - Fichier Texte : il s'agit d'une recopie directe de la mémoire:
    - organisé en lignes (séparées par '\n')
    - traduction 1 octet = 1 caractère
  - Le format binaire : recopier directement le contenu de la mémoire sous forme « binaire »:
    - séquence d'octets les uns derrière les autres sans traduction
    - adapté au données hétérogènes et structures complexes.

	Fichiers textes	**Fichiers binaires**
Taille	Peu compacte	compacte
Lisible avec un programme courant	oui	non
Lisible avec un programme spécifique	oui	oui
Écriture/Lecture par blocs	**non**	**oui**
Fonctions	fopen(mode r,w ou a) fclose() fprintf() fscanf()	fopen(mode rb, wb ou ab) fclose() fwrite() fread()

Écriture-lecture de fichiers textes

- Ecriture dans un fichier texte :

  Syntaxe : `fprintf(FILE *fichier, char *proto,…);`

  - Exemple :
    ```
    double a;
    fprintf(fichier,«%lf»,a);
    ```

  → Presque même syntaxe que printf !

- Lecture d'un fichier texte :

  Syntaxe : `fscanf(FILE *fichier,char *proto,…);`

  - Exemple :
    ```
    double a;
    fscanf(fichier,"%lf",&a);
    ```

  → Presque même syntaxe que Scanf!

Exemple : écriture dans un fichier texte

Exercice :

Ecrire un programme qui permet d'écrire deux nombres (double) dans un fichier et un autre programme permettant de les afficher.

```
include <stdio.h>
void main()
{
double a=1.5, b=2.5;
FILE *fichier;
// Ouverture du fichier en écriture grâce à "w"
fichier = fopen("essai.txt","w");       // Verifier que le fichier a bien été ouvert
if (fichier != NULL)
{     // Ecriture
        fprintf(fichier,"%lf\n",a);
        fprintf(fichier,"%lf\n",b);
fclose(fichier);  // Fermeture du fichier
}}
```

Exemple : lecture à partir d'un fichier texte

```
#include <stdio.h>
void main()
{
int i;
double tab[2];
FILE *fichier;        // Ouverture du fichier en lecture grâce à "r"
fichier = fopen("essai.txt","r");
if (fichier != NULL)
{
        for(i=0;i<2;i++)
        fscanf(fichier,"%lf\n",tab+i);
        fclose(fichier);   }
for(i=0;i<2;i++)       printf("%lf\n",tab[i]);
}
```

Écriture-lecture de fichiers binaires

- Ecriture / lecture d'un bloc de données en binaire:

  Syntaxe Ecriture : int fwrite(void * source, int taille_type, int nombre, FILE *flot)

  Syntaxe lecture : int fread(void *destination, int taille_type, int nombre, FILE *flot)

  - Source/ destination: Pointeur sur les données
  - taille_type: la taille du type écrit
  - Nombre : Nombre d'éléments qu'on veut écrire dans le fichier
  - Flot: Pointeur sur le fichier

→ Fwrite : Ecrit tout un bloc de données en un seul appel et retourne un entier : nombre d'éléments effectivement écrits.

→Fread: Lit un bloc de données en un seul appel et retourne un entier : le nombre d'éléments effectivement lus

Exemple : fwrite(tableau,sizeof(float),DIM,fichier);

fread(tableau,sizeof(float),DIM,fichier);

Exemple : Écriture-lecture de fichiers binaires

Exercice :

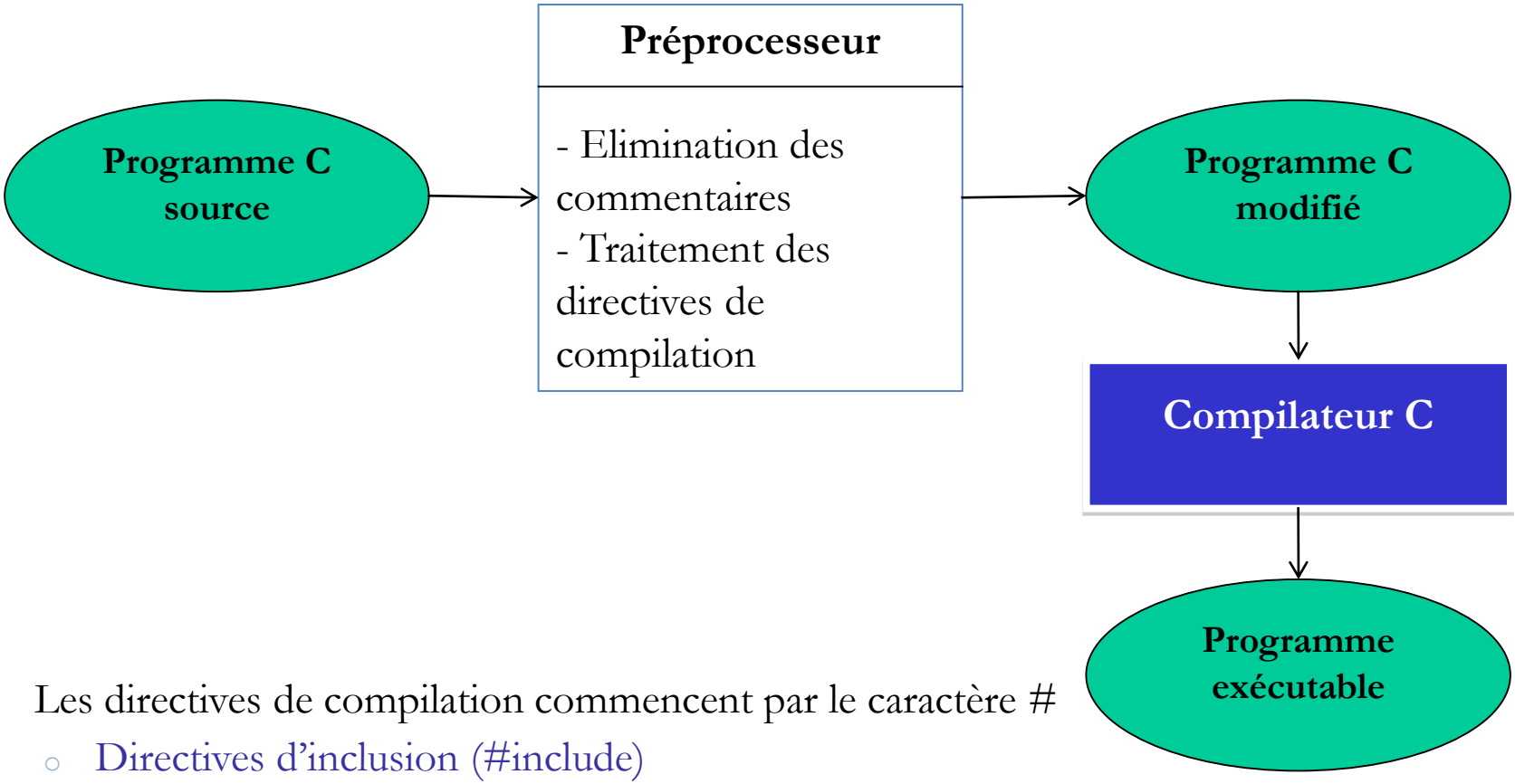
Ecrire un programme qui permet de remplir un tableau et d'insérer ses éléments dans un fichier binaire et de les récupérer

```
#define DIM 10000
void main(){
int i;
double sum,tab1[DIM],tab2[DIM];
FILE *fichier;       // Remplissage du tableau
for(i=0;i<DIM;i++)
tab1[i]=i*atan(1);    // Ecriture du fichier au format binaire
fichier = fopen("essai2.bin","wb");
if (fichier != NULL){     fwrite(tab1,sizeof(double),DIM,fichier);
                          fclose(fichier);    }
fichier = fopen("essai2.bin","rb"); // Lecture du fichier
if (fichier != NULL)  {    fread(tab2,sizeof(double),DIM,fichier);
                           fclose(fichier);}
}
```

Section 10

Le préprocesseur et la Compilation séparée

Le Préprocesseur



- Les directives de compilation commencent par le caractère #
  - Directives d'inclusion (#include)
  - Directives de substitution symbolique
  - Directives de compilation conditionnelle

Directives d'inclusion et des symboles

- Directives d'inclusion : Permettent d'incorporer, avant compilation, le texte figurant dans un fichier quelconque.

  → Exemple d'utilisation : Ecrire une seule fois les déclarations communes à plusieurs fichiers sources

Syntaxe : # include <nom_fichier> ou # include "nom_fichier"

- Directives de substitution symbolique: Le préprocesseur remplace chaque occurrence du Symbole par Equivalent excepté :
  - Dans les lignes commençant par le caractère #
  - Dans les constantes chaînes de caractères
  - Si le Symbole fait partie d'un identificateur
- Syntaxe : # define Symbole Equivalent

  → Symbole doit être un identificateuret la directive #undef Symbole annule la précédente directive #define Symbole Equivalent

Directives d'inclusion et des symboles: exemple

```
#define THEN            // Equivalent : texte vide
#defin BEGIN   {
#define  END  ;}
#include<stdio.h>

main()
BEGIN
int i;
scanf("%d",&i);
if  (i>3) THEN
   BEGIN
   printf("i est supérieur à 3\n")
   END
END
```

Macro-instructions

- Une macro est semblable à une fonction mais plus rapide (le texte est directement inséré à l'endroit voulu mais pas de gestion de pile)

  Syntaxe : define Symbole(param1, param2, …) Equivalent

- Les paramètres (param1, param2, …) sont substitués lors du pré-traitement par le préprocesseur

```
#include<stdio.h>
#define ABS(x)   ((x>0)?x:-x)
#define CUBE(x)  x*x*x
#define DIFF(x,y)  x-y
#define AFFICHE(message)    printf("%s", message)
main()
{ int a=5, x=4, y=5, d, v;
  char *s;
int v=ABS(a);
printf("Valeur absolue de %d=%d\n", a,v);
d=DIFF(x,y);
Printf("Différence %d-%d=%d\n", x,y,d);
s="Bonjour";
AFFICHE(s);      }
```

Directives de compilation conditionnelle

- Ces directives permettent d'exclure ou d'incorporer des portions du programme source dans le programme généré par le préprocesseur: les conditions utilisées par ces directives :
  - Soit existence ou inexistence d'un symbole
  - Soit valeur d'une expression

Syntaxe

```
# if comparaison1
…
#elif comparaison2
…
#else
…
#endif
```

```
#ifdef Symbole
…
#else
…
#endif
```

Directives de compilation conditionnelle : exemple

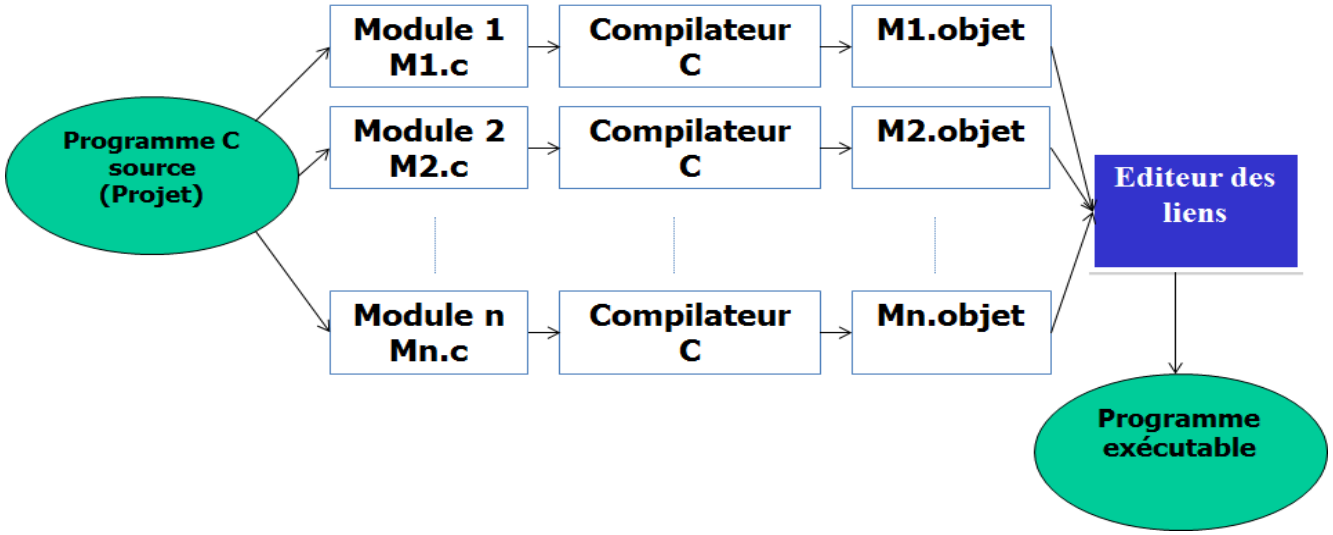
Exemple 1 :

```
# define x  3
…
#if x<10
printf("Bonjour");
#else
scanf("%d",&a);
printf("Bonsoir");
#endif
```

Exemple 2 :

```
# define Partie1
…
#ifdef Partie1
printf( "OK1");
#else
printf( "OK2");
#endif
```

Compilation séparée : Pourquoi



- Clarté et lisibilité
  - Projet modulaire, bien organisé et facile à maintenir
  - Réalisation distribuée au sein d'une équipe de programmeurs
- Réutilisabilité: Un module réalisé pour un projet pourra être réutilisé dans un autre projet
- Confidentialité: Possibilité de ne pas fournir le fichier source du module mais seulement le fichier objet

Compilation séparée : Pourquoi

- Un module C est constitué de :
  - un fichier (.c) contenant la définition d'un certain nombre de fonctions
  - un fichier (.h) (fichier en-tête : head file) contenant les éléments que devra connaître un utilisateur du module (définition de constantes, Macro-instructions, déclarations de variables globales, déclaration des fonctions (prototypes) définies dans le fichier (.c)).
  - → Le fichier (.h) constitue l'interface du module et doit être inclut dans le fichier (.c)

M1.h

```
#include<stdio.h>
#define N 50
#define CARRE(x) x*x
typedef struct type
{int x;
int y;
int z; } type1;
float f1(char *s);
float *f2(type1 R);
```

M1.c

```
#include"M1.h"

float f1(char *s)
{…};

float *f2(type1 R);
{…};

static float *f3(type1 R);
{y=CARRE(x);
…};
```

F3 est déclarée avec le mot clé **static** pour qu'elle ne sera pas visible

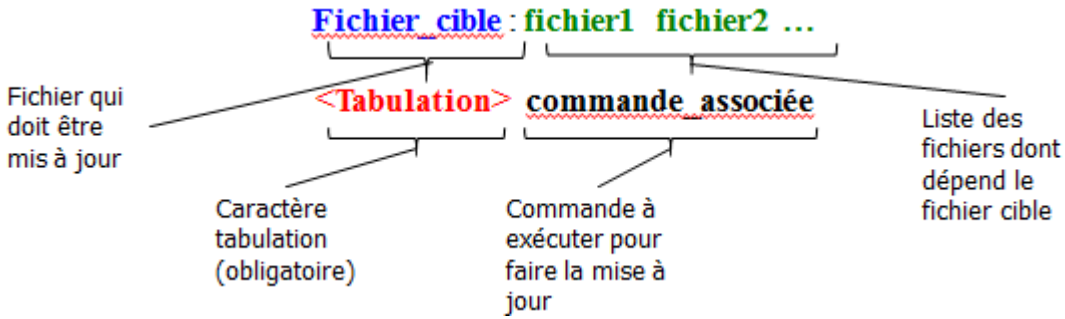
Compilation et édition des liens

- Compilation d'un module : gcc -c nom_module.c

  → nom_module.o

- Edition des liens: gcc -o nom_exécutable nom_module1.o nom_module2.o …

- La fonction make: permet de gérer la compilation séparée d'un projet de telle sorte que seules les commandes de compilation ou d'éditions de liens nécessaires soient exécutées, lors de la mise à jour d'un fichier exécutable ou objet: Nécessité de définir les dépendances entre les différents fichiers constituant un projet.
- Syntaxe du fichier de description des dépendances:



Fichier makefile sous linux

- Si le fichier de description des dépendances s'appelle makefile ou Makefile, alors pour lancer make, il suffit de taper : **>> make**
- Si le fichier porte un autre nom par exemple fd, il faut taper : **>> make –f fd**
- **Exemple d'un fichier makefile sous linux**

```
# Création de l'exécutable par édition de liens
        principal : M2.o M1.o
        gcc -o principal M2.o M1.o
# Création de M2.o par compilation sans édition de liens
        M2.o : M2.c M2.h
        gcc -c M2.c
# Création de M1.o par compilation sans édition de liens
        M1.o : M1.c M1.h
        gcc -c M1.c
```