

Faculté Poly disciplinaire de Taza

Département MPI

Le langage C

Filière: SMI/SMA/S3

Pr: Anass El Affar

Références

- Livre
 - Méthodologie de la programmation en C : Bibliothèque standard – API Posix. Jean-Pierre Braquelaire. 3ème édition Dunod
 - Le langage C. B.W. Kernighan et D.M. Ritchie. 2ème édition, MASSON.
- Cours en ligne sur internet
 - Le langage C. Francois Pellegrini, ENSEIRB
 - www.developpez.com

Partie 1 : de l'algorithmique au C

Plan

- Rappel
- Introduction au langage C
- Structure d'un programme C
- De l'algorithmique au C

Rappel

- **Programmation?**
- L'ordinateur ne comprend que le binaire (langage machine)
 - Trop élémentaire, trop long à programmer et à débugger
- Il existe des langages de programmation dits « évolués » (proches du langage courant)
 - Meilleure expressivité et généricité
 - Moins de risques d'erreurs
- Pour chaque langage, il existe un programme « qui le traduit » en langage machine

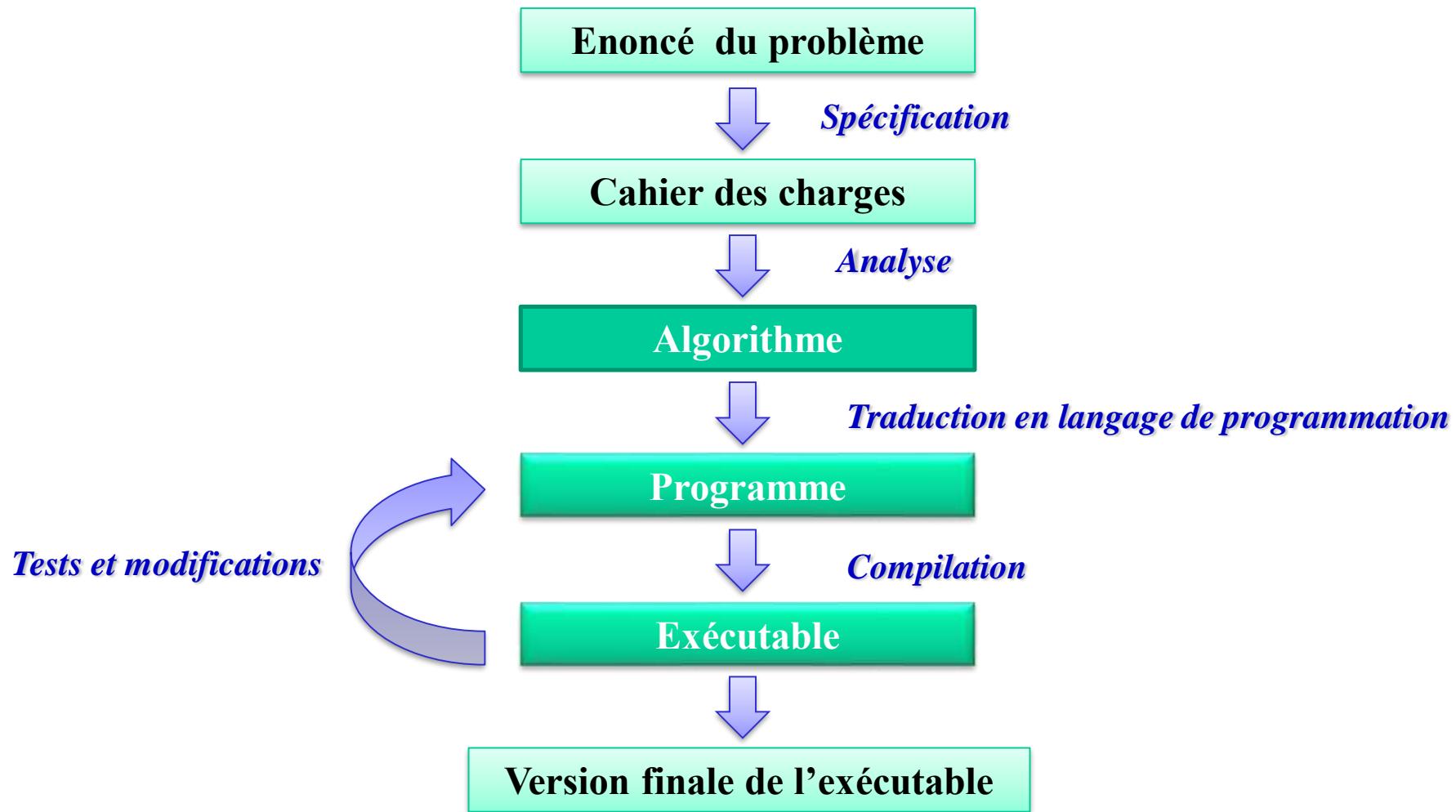


Modes de traduction : Compilation + Interprétation

Rappel

- Mais avant de programmer !!!!! Les bonnes pratiques

Les étapes de réalisation d'un programme



Introduction au C

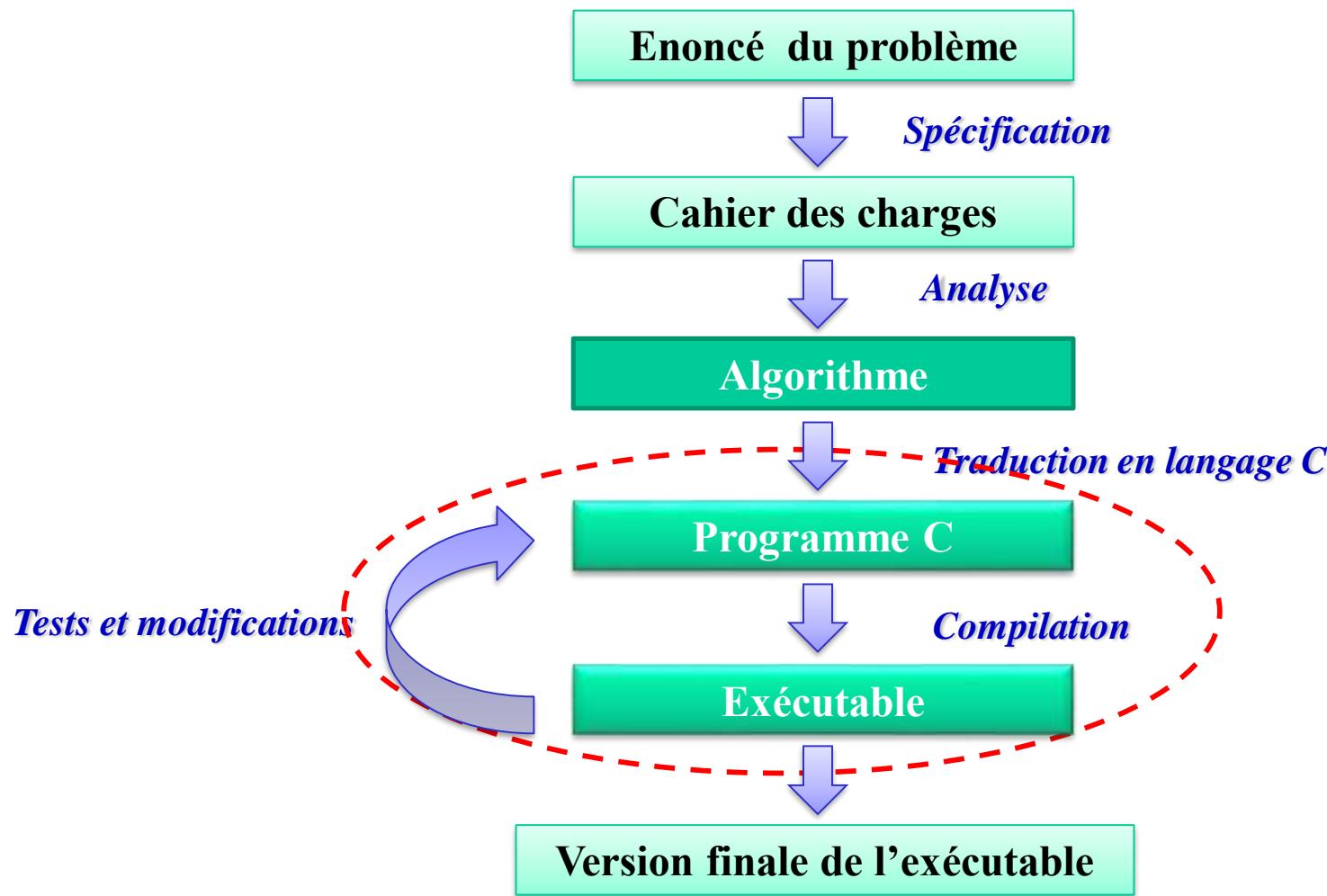
- Langage C
- Inventé aux Bell Labs / ATT en 1970
- Conçu pour être le langage de programmation d'Unix, premier système d'exploitation écrit dans langage autre qu'un langage machine
- Diffusé grâce à Unix
- Popularisé par sa concision, son expressivité et son efficacité
- Disponible actuellement sur quasiment toutes les plateformes

Introduction au C

- Langage C
- Un langage impératif : le programmeur spécifie explicitement l'enchaînement des instructions devant être exécutés :
 - Fais ceci, puis cela
 - Fais ceci, si cela est vrai
 - Fais ceci, tant de fois ou tant que cela est vrai
- Un langage de haut niveau
 - Programmation structurée
 - Organisation des données (regroupement structurel)
 - Organisation des traitements (fonctions)
 - Possibilité de programmer « façon objet »
- Un langage compilé : compilateur comme cc ou gcc

Introduction au C

- Ce que nous allons voir maintenant



Structure d'un programme C

- **Programme C**
- Un programme est la spécification d'un processus de traitement d'informations
- Un programme impératif spécifie précisément les traitements devant être réalisés sous la forme de suites d'instructions élémentaires
- Ces instructions opèrent sur les valeurs numériques contenues dans des variables nommées

Structure d'un programme C

- **Structure d'un programme C**

Par la suite, quelques mensonges bienveillants (omissions) vont glisser entre les lignes

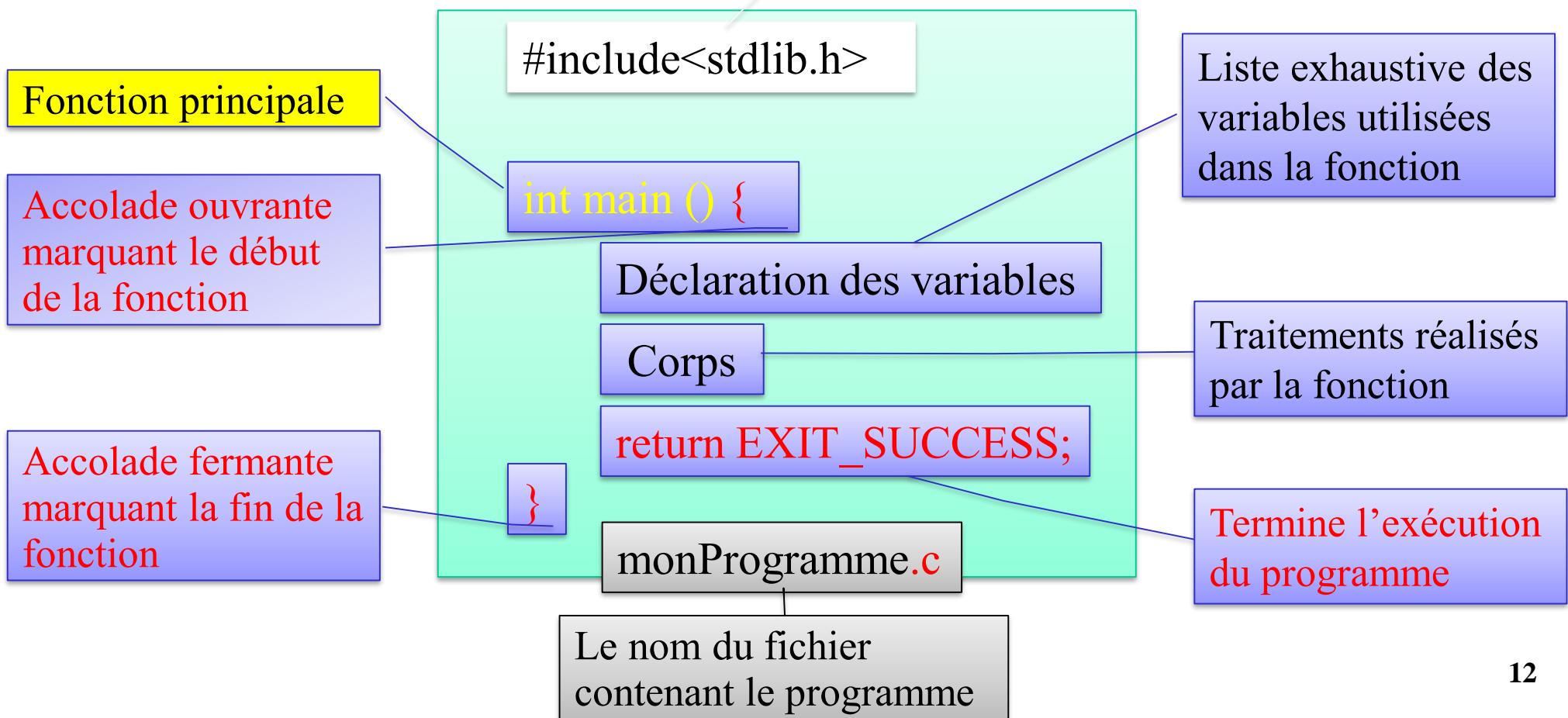
Votre programme doit obligatoirement contenir une fonction principale « main () », qui est exécutée lorsque le programme est lancé. La structure d'un programme C est la suivante :



Structure d'un programme C

- Structure d'un programme C : fonction principale

Fichier entête contenant des fonctionnalités nécessaires pour votre programme



Structure d'un programme C

- Mon premier programme : Hello World

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

int main()
{
    printf("Hello World");
    return EXIT_SUCCESS;
}
```

On sauvegarde ce programme dans un fichier qui se nomme **hello.c**

Une question : ça vous fait penser à quoi la fonction printf() ?

Une autre question : la machine comprend-elle mon programme C ?

La dernière (promis) : cela suffit-il pour pouvoir visionner le message₁₃ Hello World ? Alors que faut-il faire ?

Structure d'un programme C

- Mon premier programme : Hello World

La machine ne comprend que le langage machine, donc mon programme hello.c

Donc il faut traduire mon programme hello.c en langage machine à l'aide d'un traducteur du langage C vers le langage machine.

Un programme appelé compilateur (habituellement nommé cc ou gcc) vérifie la syntaxe de mon programme (on dit d'une façon générale, code source) et le traduit en code objet, compris par le processeur.

Le programme en code objet ainsi obtenu peut être exécuté sur la machine (après édition de liens)

Structure d'un programme C

- Mon premier programme : Hello World

Schéma simplifié de la compilation

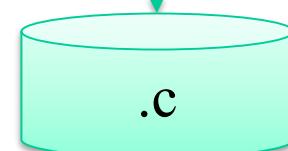
Editeur de texte ou environnement de développement

Fichier code source en langage C

Compilateur

Fichier exécutable

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hello World");
    return EXIT_SUCCESS;
}
```



Structure d'un programme C

- **Mon premier programme : Hello World**

En pratique deux façons sont possibles pour compiler hello.c :

- En ligne de commande (cmd sous windows et un terminal sous linux):
 - Vous utilisez un éditeur de texte comme Emacs (Linux) ou Bloc-notes (Windows), et un compilateur comme gcc (à télécharger gratuitement sur le net)
 - Dans le répertoire où se trouve le fichier à compiler, vous tapez : gcc hello.c
 - Dans ce cas un fichier exécutable nommé « a.out » est généré dans le même répertoire, pour l'exécuter, taper « ./a.out » sous linux et « a.out » sous windows
- En utilisant un environnement de développement
 - La plupart des environnements incluent un compilateur en plus de l'éditeur de texte
 - Pour compiler, il y a un bouton « compile », généralement dans le menu « build »
 - Pour exécuter votre programme, sélectionner toujours dans le même menu, l'icône « run »

Structure d'un programme C

- Mon premier programme : Hello World

Quelques environnements de développement

- Sous Windows

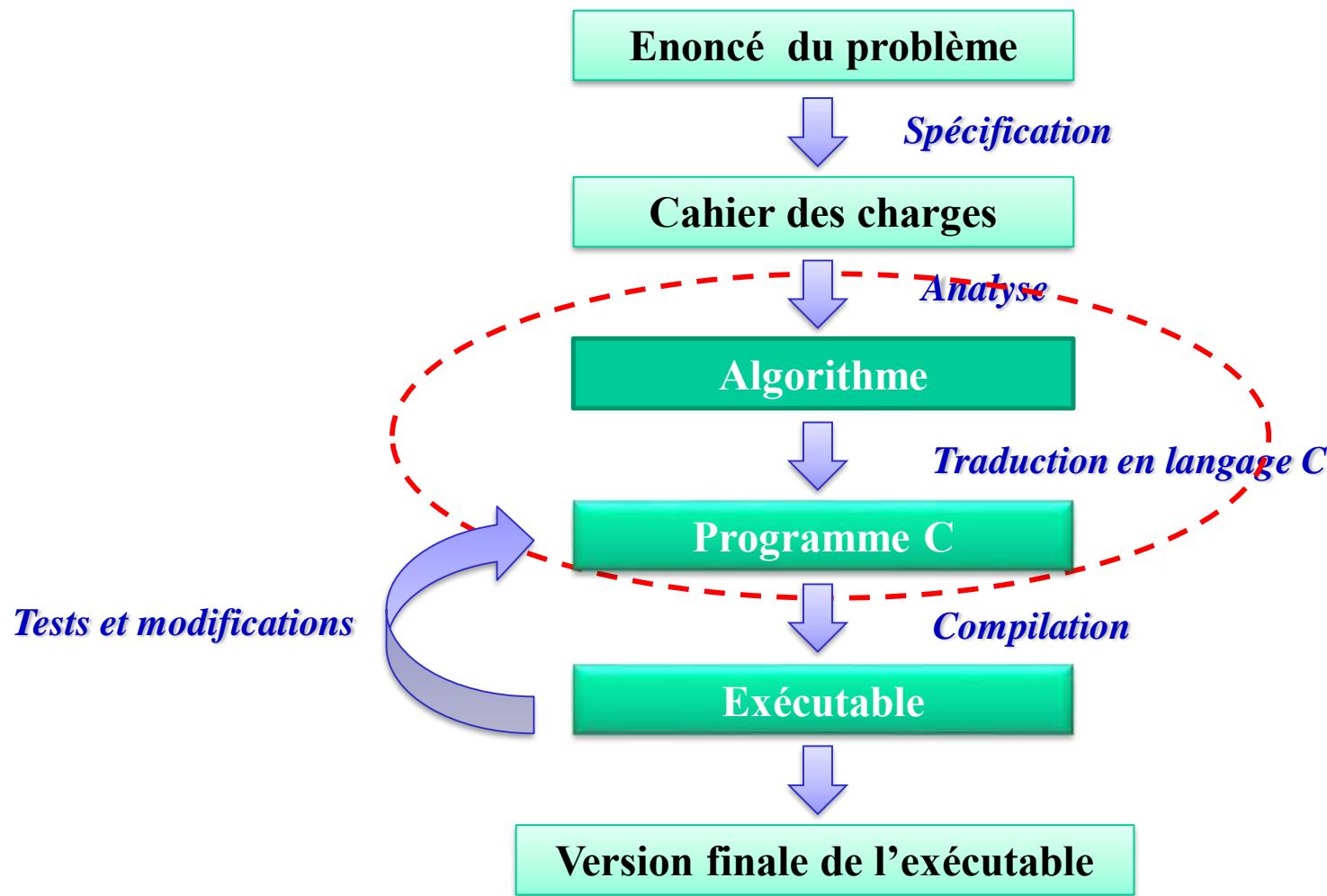
- Eclipse
- Netbeans
- Visual C++
- Turbo c++
- Dev-C++

- Sous Linux

- Eclipse
- Netbeans
- KDevelop
- ...

De l'algorithmique au C

- Algorithmique et le langage C



De l'algorithmique au C

- Rappel sur la structure d'un algorithme



- Il permet l'identification de l'algorithme
- Mot-clé : Algorithme



- Liste exhaustive des variables, constantes, des structures, des fonctions et des procédures utilisées dans le corps de l'algorithme
- Mots-clés : variable, constante, structure , fonction, ...



- C'est dans cette partie que les tâches (instructions, opérations,...) de l'algorithme sont placées
- Mots-clés : Début, Fin

De l'algorithmique au C

- Algorithme et programmation C

Algorithme somme

variable X, Y: Entier

Début

X←4

Ecrire("Donner Y ")

Lire(Y)

Ecrire(X+Y)

Fin

Entête

Déclarations

Corps

int main ()

{

 int X, Y ;

 X=4 ;

 printf("Donner Y");

 scanf("%d",&Y);

 printf("%d",X+Y);

 return EXIT_SUCCESS;

}

De l'algorithmique au C

- Traduction de l'entête d'un algorithme

Syntaxe en pseudo-code:

Algorithme <nom_algorithme>

Syntaxe en langage C :

int main ()

(éventuellement ajouter au début du fichier #include <stdio.h>)

De l'algorithmique au C

- Traduction des déclarations d'un algorithme : variables

Syntaxe en pseudo-code:

variable <nom_variable> : <type_variable>

Syntaxe en langage C :

<type_variable> <nom_variable> ;

Ne pas oublier le
point virgule ;

Exemple :

Syntaxe en algo	Syntaxe en C
variable X : Entier	int X;
variable Y, Z : Réel	float Y, Z;

De l'algorithmique au C

- Traduction des déclarations d'un algorithme : variables

Types en algorithmique	Types en langage C
Booléen	Type non défini en C mais on peut avoir des expressions booléennes (0 ou différent de zéro)
Entier	short = 16 bits int = taille du mot machine, long = 32 bits
Réel	float = virgule flottante à simple précision double = virgule flottante à double précision
Caractère	char = 8 bits
Chaîne de caractères	Type non défini, mais on utilise les pointeurs char * ou les tableaux char []

La langage C défini d'autres types (voir plus loin)

De l'algorithmique au C

- Traduction des déclarations d'un algorithme : Tableaux

Syntaxe en pseudo-code:

variable <nom_var> : Tableau[Taille] de <Type>

Syntaxe en langage C :

<Type> <nom_var> [Taille];

Ne pas oublier le
point virgule ;

Exemple :

Syntaxe en algo	Syntaxe en C
variable tab : Tableau[2] d'Entiers	int tab[2];
variable string : Tableau[3] de caractères	char string[3];

De l'algorithmique au C

- Traduction des déclarations d'un algorithme : Tableaux à deux dimension

Syntaxe en pseudo-code:

variable <nom_var> : Tableau[Taille1][Taille2] de <Type>

Syntaxe en langage C :

<Type> <nom_variable> [Taille1] [Taille2];

Exemple

Syntaxe en algo	Syntaxe en C
variable tab : Tableau[2][3] de Réels	float tab[2][3];

De l'algorithmique au C

- **Exercice**

Traduire en C

Algorithme saisieMatrice

 Variable matrice : Tableau[10][20] de Réels

 Variable i, j : Entier

 Variable string : Tableau[10] de Caractères

Début

Fin

De l'algorithmique au C

- Corrigé

Algorithme saisieMatrice

 Variable matrice : Tableau[10][20] de Réels

 Variable i, j : Entier

 Variable string : Tableau[10] de Caractères

Début

Fin

```
#include <stdlib.h>
int main ( )
{
    float matrice[10][20];
    int i, j ;
    char string[10] ;

    return EXIT_SUCCESS;
}
```

De l'algorithmique au C

- Traduction des instructions : affectation

Syntaxe de l'affectation en algo	Syntaxe de l'affectation en C
\leftarrow	=

Exemple :

Syntaxe en algo	Syntaxe de l'affectation en C
X \leftarrow 4	X = 4 ;

Remarque : chaque instruction en C se termine par un ;

De l'algorithmique au C

- Traduction des instructions : écriture

Syntaxe en algo	Syntaxe en C
Ecrire()	printf()

Exemple :

Type de la valeur	Syntaxe en algo	Syntaxe en C
Une chaîne	Ecrire("Bonjour")	printf("Bonjour") ;
X est entier	Ecrire(X)	printf("%d", X) ;
Y est un réel	Ecrire(Y)	printf("%f", Y) ;
Z est un caractère	Ecrire(Z)	printf("%c", Z) ;
Une expression	Ecrire(" La valeur de X =", X , " et de Y =", Y)	printf(" La valeur de X = %d et de Y =%f ", X, Y);

Pour pouvoir utiliser la fonction printf(), il faut ajouter au début de votre fichier :
`#include<stdio.h>`

De l'algorithmique au C

- **Exercice**

Traduire en C

Algorithme echange

Variables A, B, C : Entier

Début

 A \leftarrow 3

 B \leftarrow 2

 Ecrire(" Avant échange")

 Ecrire(" La valeur de A =", A , " et de B =", B)

 C \leftarrow B

 B \leftarrow A

 A \leftarrow C

 Ecrire(" Après échange")

 Ecrire(" La valeur de A =", A , " et de B =", B)

Fin

De l'algorithmique au C

- Corrigé

Traduire en C

Algorithme echange

Variables A, B, C : Entier

Début

A ← 3

B ← 2

Ecrire(" Avant échange")

Ecrire(" La valeur de A =", A , " et de B =", B)

C ← B

B ← A

A← C

Ecrire(" Après échange")

Ecrire(" La valeur de A =", A , " et de B =", B)

Fin

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

int main ()
{
    int A, B, C;
    A = 3;
    B = 2 ;
    printf(" Avant échange");
    printf("La valeur de A=%d et de B=%d",A, B) ;
    C = B ;
    B = A ;
    A = C ;
    printf(" Après échange") ;
    printf("La valeur de A=%d et de B=%d",A, B);

    return EXIT_SUCCESS;
}
```

De l'algorithmique au C

- Traduction des instructions : lecture

Syntaxe en algo	Syntaxe en C
Lire()	scanf()

Exemple :

Type de la valeur	Syntaxe en algo	Syntaxe en C
X est entier	Lire(X)	scanf("%d", &X) ;
Y est un réel	Lire(Y)	scanf("%f", &Y) ;
Z est un caractère	Lire(Z)	scanf("%c", &Z) ;

Pour pouvoir utiliser la fonction scanf(), il faut ajouter au début de votre fichier :

#include<stdio.h>

De l'algorithmique au C

- **Exercice**

Traduire en C

```
Algorithme echange
Variables A, B, C : Entier
Début
    Ecrire("Donner A")
    Lire(A)
    Ecrire("Donner B")
    Lire(B)
    C ← B
    B ← A
    A← C
    Ecrire("Après échange")
    Ecrire("La valeur de A =", A , " et de B =", B)
Fin
```

De l'algorithmique au C

- Corrigé

Traduire en C

Algorithme echange

Variables A, B, C : Entier

Début

Ecrire("Donner A")

Lire(A)

Ecrire("Donner B")

Lire(B)

C ← B

B ← A

A← C

Ecrire("Après échange")

Ecrire("La valeur de A =", A, " et de B =", B)

Fin

```
#include<stdio.h>
#include<stdlib.h>
int main ()
{
    int A, B, C;
    printf("Donner A");
    scanf(" %d",&A);
    printf("Donner B");
    scanf("%d ",&B) ;
    C = B ;
    B = A ;
    A = C ;
    printf("Après échange") ;
    printf("La valeur de A=%d et de B=%d",A, B);
    return EXIT_SUCCESS;
}
```

De l'algorithmique au C

- Traduction des instructions : schéma conditionnel

Syntaxe en algo	Syntaxe en C
Si condition Alors Début Si instructions Fin Si	if (condition) { instructions }
Si condition Alors Début Si instructions Fin Si Sinon Début Sinon instructions Fin Sinon	if (condition) { instructions } else { instructions }

De l'algorithmique au C

- Exercice

Traduire en C

```
{ Si X > 0 alors
    Début Si
        Ecrire("X supérieur à 0")
    Fin si
    Sinon
        Début Sinon
            Si X=0 alors
                Début Si
                    Ecrire("X égal à 0")
                Fin Si
            Sinon
                Début Sinon
                    Ecrire("X inférieur à 0")
                Fin Sinon
            FinSinon
        Fin Sinon
    Fin Sinon}
```

De l'algorithmique au C

- Corrigé

Traduire en C

```
{ if (X > 0)
  {
    printf("X supérieur à 0") ;
  }
else
{
  if (X==0)
  {
    printf("X égal à 0") ;
  }
  else
  {
    printf("X inférieur à 0") ;
  }
}
```

De l'algorithmique au C

- Traduction des instructions : boucle Pour

Syntaxe en algo	Syntaxe en C
Pour $i \leftarrow$ «valeur initiale» à « valeur finale» [de pas p] Faire instructions Fin Pour	<code>for (i = valeur initiale ; i <= « valeur finale» ; i = i + p)</code> { instructions }

Syntaxe en algo	Syntaxe en C
Pour $i \leftarrow 1$ à 100 de pas 1 Faire ... Fin Pour	<code>for (i = 1 ; i <= 100 ; i = i + 1)</code> { ... }
Pour $i \leftarrow 1$ à 100 Faire Fin Pour	<code>for (i = 1 ; i <= 100 ; i++)</code> { }

De l'algorithmique au C

- Exercice

Traduire en C

```
Algorithme Factorielle
    Variables n, fact : Entier
    Début
        fact ← 1
        Pour n ← 1 à 100 Faire
            fact ← (n * fact)
        Fin pour
        Ecrire ("La factorielle de 100 est ", fact)
    Fin
```

De l'algorithmique au C

- Corrigé

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

int main ( )
{
    int n, fact;
    fact = 1 ;
    for (n=1; n<=100; n=n+1)
    {
        fact = (n * fact) ;
    }
    printf ("La factorielle de 100 est %d", fact) ;

    return EXIT_SUCCESS;
}
```

De l'algorithmique au C

- Traduction des instructions : boucle Tant que et répéter tant que

Syntaxe en algo	Syntaxe en C
Tant que condition Faire instructions Fin Tant que	while (condition) { instructions }

Syntaxe en algo	Syntaxe en C
Répéter instructions Tant que condition	do { instructions }while (condition);

De l'algorithmique au C

- Exercice

Traduire en C

Algorithme Saisie

Variable n : Entier

Début

Lire (n)

Tant que $n > 0$ Faire

 Écrire ("Saisissez un nombre")

 Lire(n)

Fin Tant que

Fin

De l'algorithmique au C

- Corrigé

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

int main ( )
{
    int n ;
    scanf("%d",&n);
    while (n > 0)
    {
        printf("Saisissez un nombre");
        scanf("%d",&n);
    }
    return EXIT_SUCCESS;
}
```

Travaux pratiques

- **Exercice**

Ecrire un programme qui permet de calculer et d'afficher le produit deux entiers saisis au clavier

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int x, y, prod;
    printf("Entrer deux entiers :");
    scanf("%d ",&x);
    scanf("%d", &y);
    prod = x*y;
    printf("Le produit de %d et de %d est : %d", x, y, prod);
    return EXIT_SUCCESS;
}
```

Travaux pratiques

- **Exercice**

Ecrire un programme qui permet de calculer et d'afficher la somme de deux flottants saisis au clavier

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    float x, y, som;
    printf(" Entrer deux réels :");
    scanf("%f %f",&x, &y);
    som = x+y;
    printf("la somme de %f et de %f est : %f", x, y, som);
    return EXIT_SUCCESS;
}
```

Travaux pratiques

- **Exercice**

Ecrire un programme qui permet de calculer et d'afficher la somme de deux flottants saisis au clavier

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    float x, y, som;
    printf(" Entrer deux valeurs entiers :");
    scanf("%f %f",&x, &y);
    som = x+y;
    printf("la somme de %f et de %f est : %f", x, y, som);
    return EXIT_SUCCESS;
}
```

Travaux pratiques

- **Exercice**

Ecrire un programme qui permet de comparer deux entiers :

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int x, y ;
    printf("Entrer deux entiers ");
    scanf("%d %d", &x, &y);
    if (x > y){
        printf("%d est supérieur à %d", x, y);
    }
    else {
        printf("%d est supérieur à %d", y, x);
    }
    return EXIT_SUCCESS;
}
```

Partie 2 : variables et expressions

Plan

- Introduction
- Structure d'un programme C
- Commentaires
- Variables et Types
- Opérateurs et expressions
- Instructions

Introduction au C

- Langage C
- Un langage impératif : le programmeur spécifie explicitement l'enchaînement des instructions devant être exécutés :
 - Fais ceci, puis cela
 - Fais ceci, si cela est vrai
 - Fais ceci, tant de fois ou tant que cela est vrai
- Un langage de haut niveau
 - Programmation structurée
 - Organisation des données (regroupement structurel)
 - Organisation des traitements (fonctions)
 - Possibilité de programmer « façon objet »
- Un langage compilé : compilateur comme cc ou gcc

Structure d'un programme C

- **Structure d'un programme C**

Votre programme doit obligatoirement contenir une fonction principale « main () », qui est exécutée lorsque le programme est lancé. La structure d'un programme C est la suivante :

```
#include <stdlib.h>
```

```
int main () {
```

Déclaration des variables

Corps

```
return EXIT_SUCCESS;
```

```
}
```

```
monProgramme.c
```



Eviter

```
main () {
```

Déclaration des variables

Corps

```
return;
```

```
}
```

```
monProgramme.c
```

Structure d'un programme C

- Mon premier programme : Hello World

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

int main()
{
    printf("Hello World");
    return EXIT_SUCCESS;
}
```

```
#include <stdio.h>

main()
{
    printf("Hello World");
    return;
}
```

Les deux programmes sont (presque) identiques, ils affichent la même chose, le message Hello World, mais préférer la première solution (plus propre)

Commentaires

- Rôle des commentaires
- Commentaires = explications du code = documentation du code
 - Ceci est ..., il sert à
 - Ceci s'est utile pour cela et fait appel à cela
 - Ici, on fait ceci, puis cela
 - Ceci prend cela et retourne ...
- Les commentaires sont non seulement utiles, mais nécessaires à la compréhension d'un programme
 - Pensez à ceux qui vous suivront !
- Ils doivent donner une information de niveau d'abstraction plus élevé que le code qu'ils commentent
 - Sinon c'est de la paraphrase inutile

Commentaires

- Deux syntaxes pour les commentaires :
- Syntaxe standard : /* */

```
a = a +1;          /* Ceci est un commentaire de ligne */  
b = b +1          /* Et ceci en est un autre */
```

Le commentaire finit au premier « */ » rencontré

Un commentaire peut être sur une ligne ou plusieurs lignes

```
/* Ce commentaire, qui a un style de commentaire de bloc, s'étend  
** sur plusieurs lignes.  
*/
```

- Syntaxe à la « C++ » : //

```
a = a +1;          // Ceci est un commentaire de ligne  
b = b +1          // Et ceci en est un autre
```

Le commentaire finit à la fin de la ligne

Commentaires

- **Types de commentaire**

En général, trois types de commentaires sont possibles

- **Commentaires d'entête de fichier**

- Explique ce que contient ce fichier

- **Commentaires de fonction de bloc**

- Explique la fonction et l'usage du fragment du code qui suit

- **Commentaires de ligne**

- Déclaration de variable
- Début de boucle
- Astuces, etc.

```
/* Ce fichier contient le code d'un programme
** permettant le calcul de puissance de 2
** Date de création : 17/03/2016
** Auteur : I.B
** Version : v1
*/
```

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

```
/* Fonction principale qui lance l'exécution */
int main() {
```

```
    int n ; // la puissance à calculer
```

```
    ...
```

```
    scanf("%d",&n);
```

```
/* On ne considère que la puissance dans N */
if (n>0){
```

```
    ...
}
```

```
return EXIT_SUCCESS; // pour éviter le
// Warning du compilateur
```

```
}
```

Variables et Types

- **Variable**

Les variables servent à stocker les données manipulées par le programme

- Les variables sont nommées par des identificateurs alphanumériques

- Le première lettre est comprise dans l'ensemble { a, .., z, A, ..., Z, _ } (mais éviter le « _ » au début de l'identificateur, réservé aux variables systèmes)
- Les autres lettres sont comprises dans l'ensemble {a, .., z, A, ..., Z, 0, ..., 9, -,_}
- Il y a une différence entre majuscules et minuscules

Alors que pensez vous des identificateurs suivants :

Brol
bRol
brol123

Brol_2_brol_1
Brol_
b1rol

1brol
123
ma variable

123_23
échange

_brol



Ok



Ok



Non



Non



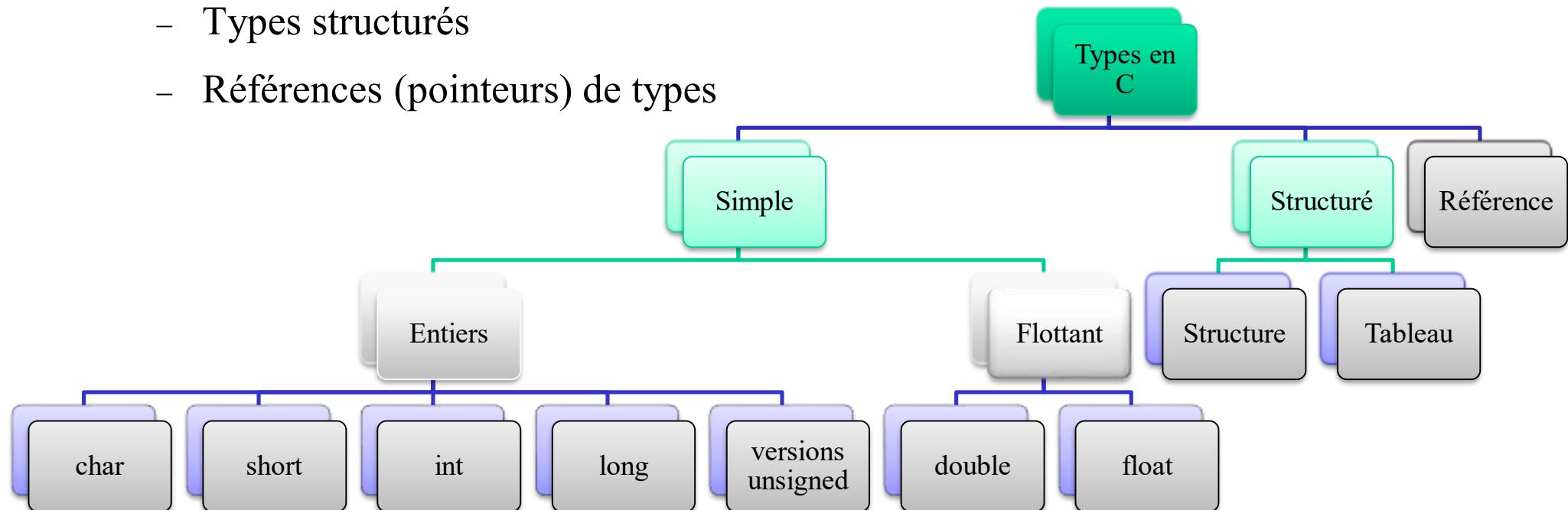
56
Eviter

Variables et Types

- **Types**

Toutes les variables sont typées :

- Types simples
- Types structurés
- Références (pointeurs) de types



C est un langage (assez) fortement typé : on ne pourra mettre dans une variable d'un type donné une valeur d'un type incompatible

Variables et Types

- **Types simples**

Deux familles principales :

- Types entiers :

- `char` : type caractère (et un seul caractère)
- `short`, `int`, `long` : types numériques servant au stockage des nombres entiers signés (positifs, négatifs ou nuls)
- Versions « `unsigned` » pour les entiers non signés (positifs ou nuls)

- Types flottants

- `float` : nombre à virgule flottante en simple précision
- `double` : nombres à virgule flottante en double précision

Le langage C ne définit pas de type booléen. Cependant toute expression nulle est considérée fausse et toute expression non nulle est considérée comme vraie

Variables et Types

- **Types simples : domaine des valeurs**

Le domaine de types simples dépend de l'architecture

Fichiers
float.h et
limits.h dans
/usr/include
sous Linux

Type	Bits	Valeur (architecture 32)
char	8	-128 à 127
unsigned char	8	0 à 255
short	16	-32768 à 32767
unsigned short	16	0 à 65535
int	32	-2147483648 à 2147483647
unsigned int	32	0 à 4294967295
long	32	-2147483648 à 2147483647
Unsigned long	32	0 à 4294967295
long long	64	-2^{64} à $2^{64}-1$
float	32	$-1,7 \times 10^{-38}$ à - 0.29 e ⁻³⁸ et 0.29 e ⁻³⁸ à $1,7 \times 10^{-38}$
double	64	-0.9×10^{-308} à - 0.5 e ⁻³⁰⁸ et 0.5e ⁻³⁰⁸ à 0.9×10^{-308}
long double		

Variables et Types

- **Types simples : domaine des valeurs**

Comme choisir le type de vos variables : la réponse est simple, selon les données manipulées et la précision souhaitée

- Dans mon programme, j'ai des données qui correspondent à des caractères

Type = char

- Dans mon programme, je manipule des valeurs entières

Type = int

- Dans mon programme, je manipule des valeurs entières positives

Type = unsigned int

- Dans mon programme, j'ai des réels avec peu de chiffres après la virgule

Type = float

- Dans mon programme, je fais des calculs sur des réels correspondants à des données scientifiques sensibles

Type = double

Variables et Types

- **Déclaration des variables**
- On spécifie le type, puis le nom de la variable
 - Autant de fois que nécessaire
`<type_de_la_variable> <nom_de_la_variable> ;`
- **Les déclarations se font au début d'un bloc d'instruction**
 - Après une accolade ouvrante : « { »
- On peut mettre plusieurs noms, séparés par une virgule : « , »

```
{  
    int          i;                      /* i : variable entière signée */  
    int          j, k;                  /* j et k : idem */  
    unsigned long l;                  /* l : entier long non signé */  
    double       d, e;                  /* d, e : flottants double précision */  
    float        f;                      /* f : flottant simple précision */  
    char         c;                      /* c : caractère */  
}
```

Variables et Types

- **Affectation de variables**
- On donne une valeur à une variable au moyen de l'instruction « `=` » :

```
i = 1;          /* i : variable entière signée */  
j = 3455;       /* j : idem */  
d = 7.45;        /* d : flottants double précision */  
k = 1.34e3;      /* k : flottants double précision */  
f=1.44589;      /* f : flottant simple précision */  
c = 'A';        /* c : caractère */
```

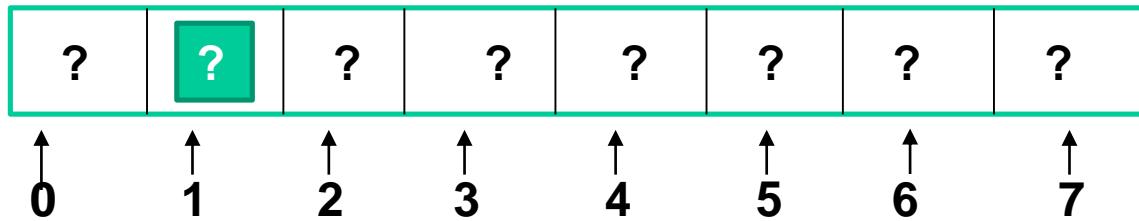
- Ce n'est pas une égalité au sens mathématique
 - Le sens exact est : prends la valeur de ce qui est à droite du « `=` » et mets-la dans la variable qui est à sa gauche.
- Il est possible de déclarer et d'initialiser une variable en même temps

```
int j=3, k =8 ;  
double d=7.45 ;  
char c='A';
```

Variables et Types

- **Types structurés : Tableau**
- Un tableau est une collection de variables de même type que l'on peut accéder individuellement par leur indice dans le tableau
- On déclare la taille du tableau entre « [] »

```
int t[8];          /* t est un tableau de 8 entier */
```



- On accède à un élément d'un tableau en donnant l'indice de l'élément entre « [] ». En C, les indices commencent à 0 et pas à 1

Variables et Types

- **Types structurés : Tableau**
- Attention, en C, aucun contrôle de débordement sur les indices !!!!!

```
int t[8];                                /* Tableau de 8 éléments entiers */  
int i ;                                    /* indice utilisé pour parcourir le tableau */  
  
for (i =0 ; i < 8 ; i= i+1)              /* On initialise tout le tableau */  
{  
    t[i] = i;  
}  
for (i =0 ; i < 8 ; i= i+2)              /* Pour tout les multiples de 2, on écrase avec */  
{  
    scanf("%d", &t[i]);  
}  
t[8] = 10;                                 /* On écrit en dehors du tableau, */  
                                            /* 64 */
```

Variables et Types

- **Types structurés : Tableau**
- On peut initialiser un tableau lors de sa déclaration
 - Liste des constantes du type adéquat, entre accolades

```
int t[8] = {1, 2, 3, 4, 5, 6, 7, 8 };
```

- Nombre d'éléments comptés par le compilateur

```
int t[] = {1, 2, 3, 4, 5, 6, 7, 8 };      /* Le compilateur comptera à votre place */
```

- La liste peut être partielle

```
int t[8] = {1, 2, 3, 4};                  /* 8 places prises, 4 initialisées */
```

Variables et Types

- **Types structurés : Tableau**

Supposer que vous avez un tableau tab de taille 9 qui contient des réels à simple précision (déjà rempli). Ecrire les instructions qui permettent d'afficher le contenu de ce tableau

```
int i;
for (i =0 ; i < 9 ; i= i+1)
{
    if(i==0){
        printf("La valeur du %d er élément du tableau est %f", i, t[i]);
    }
    else{
        printf("La valeur du %d ème élément du tableau est %f", i, t[i]);
    }
}
```

Variables et Types

- **Constantes**
- On peut avoir des :
 - Constantes entiers
 - Constantes à virgule flottante
 - Constantes caractères
- Les signes « + » et « - » sont optionnels
- Le marquer de type est aussi optionnel :
 - « U » ou « u » : pour les constantes non signées
 - « L » ou « l » : pour les constantes de type long
 - « F » ou « f » : pour les constantes à virgule flottante à simple précision

Variables et Types

- Constantes entières

Constantes décimales

- Base 10
- Suite de chiffres décimaux : [0-9]
- Pas de « 0 » au début du nombre

0
+0
43
-43
126367
-183899
4883084949U
9799092739389487UL

Constantes octales

- Base 8 (3bits)
- Suite de chiffres octaux : [0-7]
- Commencent par un « 0 »

0
05 $= 5 * 8^0 = 5$
-017 $= 1 * 8^1 + 7 * 8^0 = -15$

Ne mettez pas des « 0 » devant les constantes entières pour faire joli.....

Constantes hexadécimales

- Base 16 (4bits)
- Suite de chiffres en hexa : [0-9A-F]
- Commencent par « 0x » ou « 0X »

+0x0	= 0
0x0	= 0
0x1	= 1
0xA	= 10
0xB	= 11
0xF	= 15
-0x17	$= 1 * 16^1 + 7 * 16^0 = 23$

Variables et Types

- Constantes à virgule flottante

- Signe optionnel : « + », « - »

- Suite de chiffres et point décimal « . »

-12.48

+12.45

12.45

- Suite de chiffre après le « . » est optionnelle

-12.

+12.

12.

- Exposant « e » ou « E » est l'exposant entier décimal, optionnels

-12.32e3 = -12320

12.2E-2 = 0.122

- Marqueur de type, optionnel

- « F » : constante en simple précision

12.F

Variables et Types

- **Constantes caractères**
- Unique caractère encadré par des apostrophes « ' »
- Les caractères en C sont représentés par des entiers
 - La valeur d'une constante de type caractère est la valeur de son code dans le jeu de caractères de la machine
 - Exemple : 'O' correspond à 48 dans le codage ASCII
- Caractère d'échappement : la barre inversée « \ » (appelée « antislash » en anglais)

\0	Caractère nul	\ "	Guillemet	\t	Tabulation
\\"	Antislash	\r	Retour chariot	\b	Retour arrière
\'	Apostrophe	\n	Nouvelle ligne	\a	Bip Terminal

'A'	Lettre A	'\''	Apostrophe
''	Espace	'\n'	Nouvelle ligne
'\0'	Caractère nul	'\"'	Guillemet

Variables et Types

- **Constantes caractères**

Donner les instructions qui permettent d'afficher :

- « Bonjour, je suis une apostrophe : ' »

```
printf("Bonjour, je suis une apostrophe : \'");
```

- « Bonjour, je suis un guillemet : " »

```
printf("Bonjour, je suis un guillemet : \"");
```

- « Bonjour, je suis un antislash : \ »

```
printf("Bonjour, je suis un antislash: \\");
```

- Le message « Une nouvelle ligne va me suivre » et revient à la ligne

```
printf("Une nouvelle ligne va me suivre \n");
```

- Le message « Entre moi » suivie d'une tabulation, suivie du message « et lui, il y a une tabulation », puis suivie d'un bip du terminal et une nouvelle ligne

```
printf("Entre moi \t et lui, il y a une tabulation \a \n ");
```

Opérateurs et expressions

- **Vocabulaire**
- Un **opérateur** est un symbole d'opération qui permet d'agir sur des variables ou de faire des “calculs”
- Une **opérande** est une entité (variable, constante ou expression) utilisée par un opérateur
- Une **expression** est une combinaison d'opérateur(s) et d'opérande(s),
• **Une expression possède une valeur et un type.**
- Dans $a+b$:
 - a est l'opérande gauche
 - $+$ est l'opérateur
 - b est l'opérande droite
 - $a+b$ est appelé une expression
 - Si a et b sont des entiers, l'expression $a+b$ est un entier

Opérateurs et expressions

- Opérateurs arithmétiques

Opérateur	Syntaxe
Addition	<code>expr1 + expr2</code>
Soustraction	<code>expr1 - expr2</code>
Multiplication	<code>expr1 * expr2</code>
Division	<code>expr1 / expr2</code>
Modulo	<code>expr1 % expr2</code>

- Si les deux opérandes sont entiers alors / est une division entière (5/2 vaut 2 et non 2.5)
- % calcule le reste de la division entière de deux entiers (5%2 vaut 1)
- Il existe des opérateurs unitaires – et + qui changent le signe de leur opérande

Opérateurs et expressions

- Expression booléenne
- Une expression booléenne est une fonction logique qui ne retourne que les valeurs VRAI ou FAUX
- En C, il n'existe pas de type booléen propre
 - Type `bool` introduit dans la norme C99 seulement
- Le type booléen est émulé par le type `int`, avec les conventions suivantes :
 - FAUX : valeur 0
 - VRAI : toutes les autres valeurs non nulles

En C, toute expression non nulle est considérée vraie et toute expression nulle est considérée fausse

Opérateurs et expressions

- Opérateurs relationnels, d'égalité et logiques

Opérateurs relationnels	
Inférieur	$\text{expr1} < \text{expr2}$
Inférieur ou égale	$\text{expr1} \leq \text{expr2}$
Supérieur	$\text{expr1} > \text{expr2}$
Supérieur ou égale	$\text{expr1} \geq \text{expr2}$

- Toutes ces opérations donnent 0 si la comparaison indiquée est fausse et 1 si elle est vraie

Opérateurs et expressions

- Opérateurs relationnels, d'égalité et logiques

Opérateurs d'égalité	
Égal à	<code>expr1 == expr2</code>
Différent de	<code>expr1 != expr2</code>

- Toutes ces opérations donnent **0** si la comparaison indiquée est fausse et **1** si elle est vraie

Opérateurs et expressions

- Opérateurs relationnels, d'égalité et logiques

Opérations logiques	
Négation logique	<code>! expr</code>
Conjonction (ET logique)	<code>expr1 && expr2</code>
Disjonction (OU logique)	<code>expr1 expr2</code>

- Toutes ces opérations **donnent 0** si la comparaison indiquée est fausse et **1 si elle est vraie**
- Pour **&&**, expr1 est d'abord évaluée. Si elle est fausse, **&& retourne 0** sans évaluer expr2. Sinon expr2 est évaluée
- Pour **||**, expr1 est d'abord évaluée. Si elle est vraie, **|| retourne 1** sans évaluer expr2. Sinon expr2 est évaluée

Opérateurs et expressions

- Opérateurs relationnels, d'égalité et logiques

Avec une expression booléenne, on peut

- Stocker le résultat d'un expression booléenne dans une variable entière

```
int a = ( (b>4) && ((c== 'A') || (c == 'a'))) ;
```

```
int m = (c!= 'A') || ! b ;
```

Opérateurs et expressions

- Opérateurs relationnels, d'égalité et logiques

Avec une expression booléenne, on peut

- Utiliser le résultat d'une expression entière comme valeur de condition

```
if(4){  
    printf("Raté \n ");  
}  
else {  
    printf("Ben Oui \n ");  
}
```



```
int a = 4;  
if(a){  
    printf("Raté \n ");  
}  
else {  
    printf("Ben Oui \n ");  
}
```



```
int i = 4;  
int a =3, b=4, c=5;  
i= ((a>b) && (a>c));  
printf("la valeur de i est %d \n", i );
```

0

```
if(0){  
    printf("Raté \n ");  
}  
else {  
    printf("Ben Oui \n ");  
}
```



```
int a = 0;  
if(a){  
    printf("Raté \n ");  
}  
else {  
    printf("Ben Oui \n ");  
}
```



Opérateurs et expressions

- **Opérateurs d'affectation**

L'opération d'affectation en C se note « `=` ». Contrairement à beaucoup de langages de programmation, l'affectation en C n'est pas une instruction, mais une expression, car elle a

- Un type : le type du membre droit
- Une valeur : la valeur du membre droit

Ainsi, « `var = expr` » range à l'adresse désigné par « `var` » la valeur « `expr` » convertie dans le type de « `var` » et de plus vaut cette valeur

On peut donc les utiliser comme termes d'expressions englobant.

```
i = j = 3;          /* j=3 est une expression dont la valeur est 3, cette valeur est stockée dans i */
```

Cependant c'est une source potentielle de problèmes

```
if(a=3)  
    printf("Attention Danger");
```

```
if(a=0)  
    printf("Attention Danger");
```

Opérateurs et expressions

- **Opérateurs d'affectation combinée**

Comme de nombreuses instructions arithmétiques sont des modification du contenu de variables, il existe des raccourcis spécifiques pour l'affectation

$$\text{var} = \text{var op expr;} \quad \Leftrightarrow \quad \text{var} \quad \text{op} = \text{expr ;}$$

On trouve des raccourcis pour tous les opérateurs arithmétiques binaires : « $+=$ », « $-=$ », « $*=$ », « $/=$ », « $\%=$ »

i = 3; i += 12	/* i =15	*/
i = 3; i -= 7	/* i =-4 (i de type int)	*/
i = 3; i *= 4	/* i =12	*/
i = 3; i /= 2	/* i = 1 (division entière)	*/
i = 3; i %= 2	/* i =1	*/

Opérateurs et expressions

- **Opérateurs d'incrémentation et de décrémentation**

Les opérateurs mythiques pour tout programmeur C sont « ++ » qui ajoute 1 à son opérande et « -- » qui retranche 1 de son opérande.

```
var = var +1; ⇔ var ++;  
var = var - 1; ⇔ var -- ;
```

Ces instructions sont également des expressions qui ont une valeur et un type et peuvent être utilisées dans une expression englobant

```
i = 3;  
a = i ++; /* L'expression « i ++ » a un type et une valeur */
```

Opérateurs et expressions

- **Opérateurs d'incrémentation et de décrémentation**

Selon que les opérations sont positionnés avant ou après la variable sur laquelle ils opèrent, on lit la valeur de la variable avant ou après l'opération : on a donc deux modes de fonctionnement

- Le suffixe :

Post-incrémantation	var ++;
Post-décrémantation	var -- ;

- Le préfixe

Pré-incrémantation	++ var;
Pré-décrémantation	-- var ;

Ce qu'il faut retenir :

$$a = i++; \Leftrightarrow \begin{array}{l} a = i; \\ i = i + 1; \end{array}$$

$$a = i--; \Leftrightarrow \begin{array}{l} a = i; \\ i = i - 1; \end{array}$$

$$a = -- i; \Leftrightarrow \begin{array}{l} i = i - 1; \\ a = i; \end{array}$$

$$a = ++ i; \Leftrightarrow \begin{array}{l} i = i + 1; \\ a = i; \end{array}$$

Opérateurs et expressions

- Opérateurs d'incrémentation et de décrémentation

Exemple :

```
i =3;  
a = i ++;  
printf("Valeur de i=%d et de a=%d"); /* i=4 et a=3 */
```

```
i =3;  
a = i --;  
printf("Valeur de i=%d et de a=%d"); /* i=2 et a=3 */
```

```
i =3;  
a = ++i;  
printf("Valeur de i=%d et de a=%d"); /* i=4 et a=4 */
```

```
i =3;  
a = --i;  
printf("Valeur de i=%d et de a=%d"); /* i=2 et a=2 */
```

Opérateurs et expressions

- Opérateurs d'incrémentation et de décrémentation

Exemple :

```
i =3;
```

```
i++;
```

/* i = i+1 */

```
a = i ;
```

```
printf("Valeur de i=%d et de a=%d"); /* i=4 et a=4 */
```

```
i =3;
```

```
i--;
```

/* i = i-1 */

```
a = i ;
```

```
printf("Valeur de i=%d et de a=%d"); /* i=2 et a=2 */
```

```
i =3;
```

```
++i;
```

/* i = i+1 */

```
a = i;
```

```
printf("Valeur de i=%d et de a=%d"); /* i=4 et a=4 */
```

```
i =3;
```

```
--i;
```

/* i = i-1 */

```
a = i;
```

```
printf("Valeur de i=%d et de a=%d"); /* i=2 et a=2 */
```

Opérateurs et expressions

- Opérateurs d'incrémentation et de décrémentation

Exercice : Donner, pour chaque cas, le message affiché, ainsi que les valeurs de a et c à la fin des instructions suivantes :

```
a = 3;  
b = 3;  
  
if (a++ > b)  
{  
    printf("Raté");  
}  
else {  
    printf("Ok");  
}
```

Ok
a vaut 4

```
a = 3;  
b = 3;  
  
if (++a > b)  
{  
    printf("Raté");  
}  
else {  
    printf("Ok");  
}
```

Raté
a vaut 4

```
a = 3;  
b = 3;  
c = 2;  
if ((++a > b) || (a > c --))  
{  
    printf("Raté");  
}  
else {  
    printf("Ok");  
}
```

Effet de bord

Raté
a vaut 4 et c = 2

Opérateurs et expressions

- Conversion de type
- Lorsque deux variables de types différents mais comparables, et que l'on désire par exemple affecter la valeur de l'une à l'autre, il se pose un problème des différences de structure internes entre ces variables (par exemple le nombre de bits affecté à la variable). En C, il y a, avant l'évaluation d'une expression, une conversion implicite du type le plus faible vers le type le plus fort :

char < short ≤ int ≤ long < float < double

- Les constantes entières sont de type int
- Les constantes flottantes sont de type double (si on précise pas autre chose)

'A' + 1	: une expression de type int car 1 est un entier qui est plus fort qu'un char
var + 12.45	: une expression de type double car 12.45 est un double (pas un float)
12 / 5.F	: une expression de type float car 23.F est float (le résultat est 2.5 et non 2)

Opérateurs et expressions

- **Conversion de type et affectation**

Lors d'une affectation, le type du membre droit est converti dans le type du membre gauche

Si le type de destination (gauche) est plus faible que le type du membre droit :

- Le résultat peut être indéfini pour les types flottants si le nombre ne peut être représenté
- Il y a troncature pour les types entiers

```
char c; int i=2 ;  
c = 'A' +1;          /* Aucun problème  
i = i * 12.34;      /* D'abord on calcul 2 * 12.34 ce qui donne 24.68 puis on tronque 34,  
                     ce qui donne 12. Attention toutefois avec si on dépasse la capacité  
                     d'un entier, on peut se retrouver avec des chiffre négatives */
```

C'est vraiment délicat, c'est le compilateur qui fait de lui-même, il faut faire attention, voir l'exemple du TP

Opérateurs et expressions

- Conversion de type : casting
- L'opérateur parenthésé de conversion de type (aussi appelé « type cast ») permet les conversions explicites de types.
 - Permet de spécifier explicitement des conversions au compilateur
 - Supprime les avertissements du compilateur lors de la conversions vers des types plus faibles

```
int diviseur;  
int reste;  
double nombre;  
reste = (int ) nombre % diviseur;
```

```
int a = 2 ;  
int b = 3;  
double resultat1, résultat2,  
resultat3, resultat4;  
resultat1 = b / a;  
resultat2 = (double) b / a;  
resultat3 = b / (double) a;  
resultat4 = (double) b / (double) a;  
resultat5 = (double) ( b / a);
```

resultat1 =1.
resultat2 =1.5
resultat3 =1.5
resultat4 =1.5
resultat5 =1.

Opérateurs et expressions

- Tableau partiel des priorités des opérateurs déjà abordés

Opérateur	Signification	Associativité
()	Parenthésage	Gauche Droite
-	Négation unitaire	Droite Gauche
! ++ --	Non/Inc/décrémentation	Droite Gauche
* / %	Mult/Div/modulo	Gauche Droite
+ -	Addition/Soustraction	Gauche Droite
< <= > >=	Comparaison	Gauche Droite
== !=	Egalité /Inégalité	Gauche Droite
&&	Et logique	Gauche Droite
	Ou logique	Gauche Droite
= += -= *= /= %=	Affectation	Droite Gauche

L'ordre d'évaluation d'une expression dépend des priorités des opérateurs

Partie 3 : structures de contrôle

Plan

- Structures de contrôle

- Lecture : printf
- Ecriture : scanf
- if ... else
- Opérateur ternaire ?... : ...
- switch ... case
- for
- while
- do ... while
- Instruction break
- Instruction continue

Instructions

- Fonctions d'écriture printf()
- Sert à afficher du texte
- Affiche la chaîne de caractère passée en paramètre, entre guillemets "
- Les séquences d'échappement valables pour les caractères s'appliquent également

```
printf("Je dis \"Bonjour!\"\n");
printf("Avec l'\apostrrophe \n");
printf("Avec l'apostrrophe \n"); /* le \" n'est pas nécessaire */
```

Pour pouvoir utiliser la fonction printf(), il faut ajouter au début de votre fichier :

#include<stdio.h>

Instructions

- Fonctions d'écriture printf()
- On peut indiquer comment afficher les autres paramètres au moyen des séquences d'échappement "%"

Type de la valeur	Syntaxe en C
Une chaîne	printf("Bonjour") ;
%	printf("%%%") ;
C est char	printf("%c", C) ;
X est int	printf("%d", X) ;
Y est long	printf("%ld", Y) ;
Z est float	printf("%f", Z) ;
U est double	printf("%lf", U) ;
Une expression	printf(" La valeur de X = %d et de Y =%f ", X, Y);

```
printf("%lf est plus petite que %lf \n",d1,d2);
```

Instructions

- Fonctions de lecture scanf()
- Fonction inverse de printf
- Sert à analyser le texte tapé et le convertir en valeurs placées dans les variables passées en paramètres, selon la chaîne de format passée en premier paramètre
 - Pas d'espace dans la chaîne de format
 - On doit mettre un « & » avant le nom de la variable de types simples (référence)

Type de la valeur	Syntaxe en C
C est char	scanf("%c", &C) ;
X est int	scanf("%d", &X) ;
Y est long	scanf("%ld", &Y) ;
Z est float	scanf("%f", &Z) ;
U est double	scanf("%lf", &Z) ;

Instructions

- **Structures de contrôle**

Les structures de contrôle servent à orienter l'exécution du programme en fonction de la valeur courante d'une expression

- Exécution conditionnelle d'un fragment de code si une certaine condition est vérifiée
- Exécution répétitive d'un fragment du code si une certaine condition est vérifiée

Les conditions de contrôle sont des expressions logiques booléennes

Instructions

- **Instruction if ... else**

Syntaxe :

- **if(expression) instruction1**
- **if(expression) instruction1 else instruction2**

Si l'expression est vrai, l'instruction1 est exécutée, sinon l'instruction 2 est exécutée si elle existe

```
if((an%4) !=0) {          /* Si l'année n'est pas bissextile */  
    nbjours =365;  
}  
else {  
    nbjours = 366;  
}
```

Les accolades déterminent le bloc d'instructions qui dépend du if ou du else : si la condition est vraie toutes les instructions du bloc sont exécutées

Instructions

- **Instruction if ... else**

Lorsqu'on a plusieurs cas à tester, il est possible d'enchaîner les if ... else

```
if((an%4) !=0) {          /* Si l'année n'est pas bissextile */  
    nbjours =365;  
}  
else {  
    if ((an%4) ==0) {  
        nbjours = 366;  
    }  
    else {  
        nbjours = 365;  
    }  
}
```

Le else se rapporte toujours au premier if le précédent

Instructions

- **Instruction if ... else**

Lorsque le bloc d'instructions qui dépend d'un if ou d'un else se compose d'une seule instruction, il est possible d'omettre les accolades : **MAIS ATTENTION**

```
if((an%4) !=0) {  
    nbjours =365;  
}  
else {  
    if ((an/400) ==0) {  
        nbjours = 366;  
    }  
    else {  
        nbjours = 366;  
    }  
}
```

```
if((an%4) !=0)  
    nbjours =365;  
else {  
    if ((an/400) ==0)  
        nbjours = 366;  
    else  
        nbjours = 366;  
}
```

```
if((an%4) !=0)  
    nbjours =365;  
else if ((an/400) ==0)  
    nbjours = 366;  
else  
    nbjours = 366;
```

Le else se rapporte toujours au premier if le précédent

Instructions

- Instruction if ... else

Exemples : que affiche se code suivant

```
int a =3;  
if(a==3)  
printf("a vaut 3 \n");  
printf("Merci \n");
```

a vaut 3
Merci

```
int a =3;  
if(a==3)  
    printf("a vaut 3 \n");  
    printf("Merci \n");
```

Conclusion : Si vous n'êtes pas sûr de ce que vous faites, mettez les accolades, c'est plus simple

```
int a =3;  
if(a==3){  
    printf("a vaut 3 \n ");  
}  
printf("Merci \n");
```

Mettez des accolades

```
int a =3, b= 0;  
if(a==3)  
    if(b==0) printf("Bonjour ");  
else printf("Merci \n");
```

Bonjour

```
int a =3, b= 1;  
if(a==3)  
    if(b==0) printf("Bonjour ");  
else printf("Merci \n");
```

Merci

```
int a =0, b= 0;  
if(a==3)  
    if(b==0) printf("Bonjour ");  
else printf("Merci \n");
```

(n'affiche rien)

100

Instructions

- **Opérateur ternaire ? : ...**

Dans de nombreux cas, une instruction conditionnelle sert juste à positionner une variable

- Redondance du code

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

```
max = b;
if(a > b)
    max = a;
```

L'opérateur ternaire « ? : » renvoie une valeur différente selon la validité de l'expression qui le précède

```
max = (a >b) ? a : b ;
```

Instructions

- **Switch ... case**

L'instruction switch ... case sert à traiter des choix multiples en fonction d'une expression entière (des caractères ou des entiers)

```
printf("Entrez votre choix \n");
scanf("%c",&c);
switch(c) { /* évaluation d'une expression entière */
    case 'A':
        printf("Vous avez saisi A \n");           // Instructions exécutées jusqu'au break
        printf("Enter un entier");
        scanf("%d",&i);
        i++;
        break;                                // break permet de sortir de la sélection
    case 'B':
        printf("Vous avez saisi B \n");
        break;
    default :
        printf("Choix \"%c\" invalide\n",c); // comportement par défaut (optionnel)
        break;
}
```

Instructions

- **Switch ... case**

Supposer qu'on a oublié un break dans le bloc de case « A » et que l'utilisateur a saisi A, quelle sont les instructions qui seront exécutées?

```
printf("Entrez votre choix \n");
scanf("%c",&c);
switch(c) { /* évaluation d'une expression entière */
    case 'A':
        printf("Vous avez saisi A \n");
        printf("Enter un entier");
        scanf("%d",&i);
        i++;
        /* break; */ // Instructions exécutées jusqu'au break
    case 'B':
        printf("Vous avez saisi B \n");
        break;
    default :
        printf("Choix '\"%c'\" invalide\n");
        break; // comportement par défaut
}
```

The diagram shows a red box highlighting the code block for the 'A' case. A red arrow originates from the opening brace of the 'default:' block and points to the bottom edge of the red box, indicating that the execution flow continues from the end of the 'A' case block.

Instructions

- **Switch ... case**

Bien sur, au lieu de saisir une caractère, on peut saisir un entier

```
printf("Entrez votre choix \n");
scanf("%d",&k);
switch(k) { /* évaluation d'une expression entière */
    case 1:
        printf("Vous avez saisi A \n");           // Instructions exécutées jusqu'au break
        printf("Enter un entier");
        scanf("%d",&i);
        i++;
        /* break; */                         // on a oublié de faire un break
    case 2 :
        printf("Vous avez saisi B \n");
        break;
    default :
        printf("Choix \'%c\' invalide\n");      // comportement par défaut
        break;
}
```

Instructions

- **Instruction while**

Syntaxe :

while (expression) instruction

Tant que l'expression est vraie, on exécute l'instruction

Remarque : une instruction doit modifier l'expression, sinon on a une boucle infinie

```
printf("Entrez le nombre de départ : ");
scanf("%d",&n);

while (n >1) {
    n = ((n%2)==0) ? (n / 2 ) : ( (n-1) / 2);
    printf("%d \n", n);
}
```

Instructions

- **Instruction for**

Syntaxe :

```
for (instruction1 ; expression ; instruction2)  
    instruction3
```

L'instruction1 est d'abord exécutée (initialisation). Puis, tant que l'expression est vraie, on exécute l'instruction3 (corps de la boucle), puis l'instruction 2 (itérateur)

```
printf("Je compte jusqu'à 10 : \n ");
```

```
for ( i= 1 ; i <= 10 ; i ++)  
    printf("%d", i );
```

```
printf("%d", i );
```

```
for ( ; ; )  
    printf("Je suis une boucle infinie \n ");
```

Instructions

- **Instruction for**

L'instruction for est sémantiquement équivalente à l'instruction while.

```
printf("Je compte jusqu'à 10 : \n ");
for ( i= 1 ; i <= 10 ; i++)
    printf("%d", i );
```

```
printf("Je compte jusqu'à 10 : \n ");
i = 1;
while (i <= 10){
    printf("%d", i );
    i ++ ;
}
```

Instructions

- **Instruction do ... while**

Syntaxe :

do instruction while (expression) ;

L'instruction est d'abord exécutée, puis l'expression est évaluée. Si elle est vraie, on reboucle sur l'exécution de l'instruction.

A la différence de la boucle while, l'instruction est toujours exécutée au moins une fois.

```
do {  
    printf("Entrez un nombre entre 1 et 10 : ");  
    scanf("%d",&n);  
}while ( (n <1) || (n>10) ) ;
```

Instructions

- **Instruction break**

L'instruction break permet de quitter la boucle la plus interne, ou bien la case, dans lesquels elle se trouve.

- On ne sort que d'un seul niveau
- Ne concerne pas les if ... else

```
printf("Je calcule la somme de 10 entiers positifs \n ");

somme =0 ;

for ( i= 1 ; i <= 10 ; i ++ ) {           /* Lecture d'au plus 10 entiers */
    scanf("%d", &n );
    if ( n <= 0)                         // si un entier n'est pas strictement positif
        break;                            // On quitte la boucle for

    somme += n;
}
```

Instructions

- **Instruction break**

L'instruction break peut être utilisée pour sortir d'une boucle déclarée comme boucle infinie

```
while (1) {                      /* Boucle infinie */
    scanf("%c", &c );
    if ( (c == 'Q') || (c == 'q') )      // si c vaut la valeur de sortie : q ou Q
        break;                          // On quitte la boucle while
    switch (c) {
        case 'A' :
        case 'a' :
            ....
            break;                      // se break là, ne fait sortir que du switch
            ....
    }
}
```

Instructions

- **Instruction continue**

L'instruction continue sert à sauter l'itération courante de la boucle la plus interne, dans laquelle elle se trouve, et à passer à l'itération suivante (évite les goto)

```
printf("Je calcule la somme de 10 entiers positifs \n ");
somme =0 ;
for ( i= 1 ; i <= 10 ; i ++ ) { /* Lecture d'au plus 10 entiers */
    scanf("%d", &n );
    if ( n <= 0 )
        continue;
    somme += n;
}
```

/* Lecture d'au plus 10 entiers */
// si un entier n'est pas strictement positif
// On ne quitte pas la boucle for, mais on
//passe à l'itération suivante (au nombre suivant)

Partie 4 : sous-programmes et fonctions

Plan

- Analyse descendante
- Sous-programme et fonctions
- Fonctions en C
- Paramètres des fonctions
- Visibilité des variables
- Récursivité

Analyse descendante

- **Problématique**

Problème : Comment traiter les problèmes trop complexes pour être appréhendés en un seul bloc (dans la fonction main par exemple) ?

Solution : On peut appliquer la méthode "Diviser pour régner", et décomposer le problème en sous-problèmes plus simples : c'est l'analyse descendante

Analyse descendante

- **Principe d'analyse descendante**

Rôle : Méthode pour écrire un programme de qualité : lecture plus facile de programme, modularité du code

Principe :

- Abstraire

Repousser la plus loin possible l'écriture de l'algorithme (codage)

- Décomposer

Décomposer la résolution du problème initial en une suite de “sous problèmes” que l'on considère comme résolus

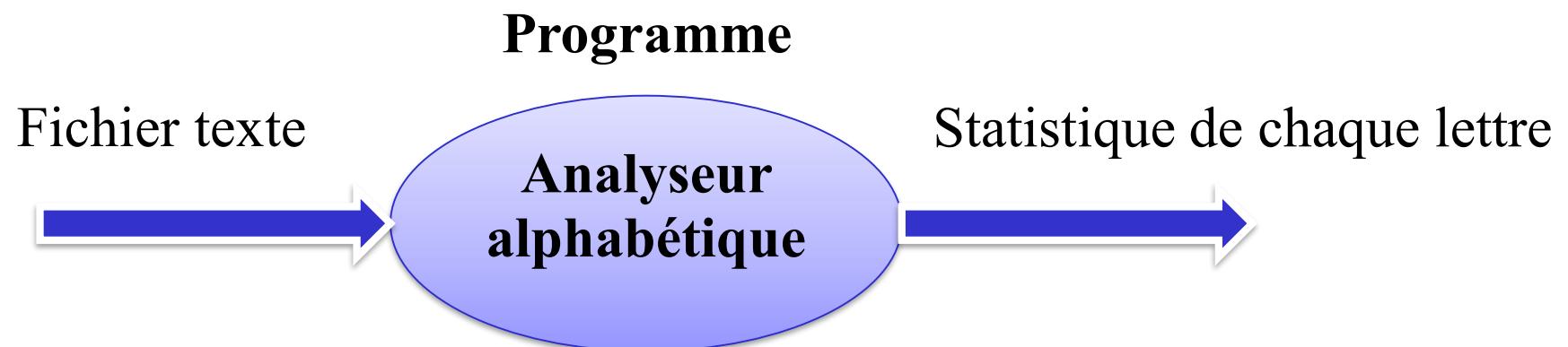
- Combiner

Résoudre le problème initial par combinaison des abstractions des “sous-problèmes”

Analyse descendante

- Exemple

Analyseur alphabétique : Écrire un programme analysant un fichier texte et indiquant le nombre d'occurrence de chaque lettre de l'alphabet dans le fichier

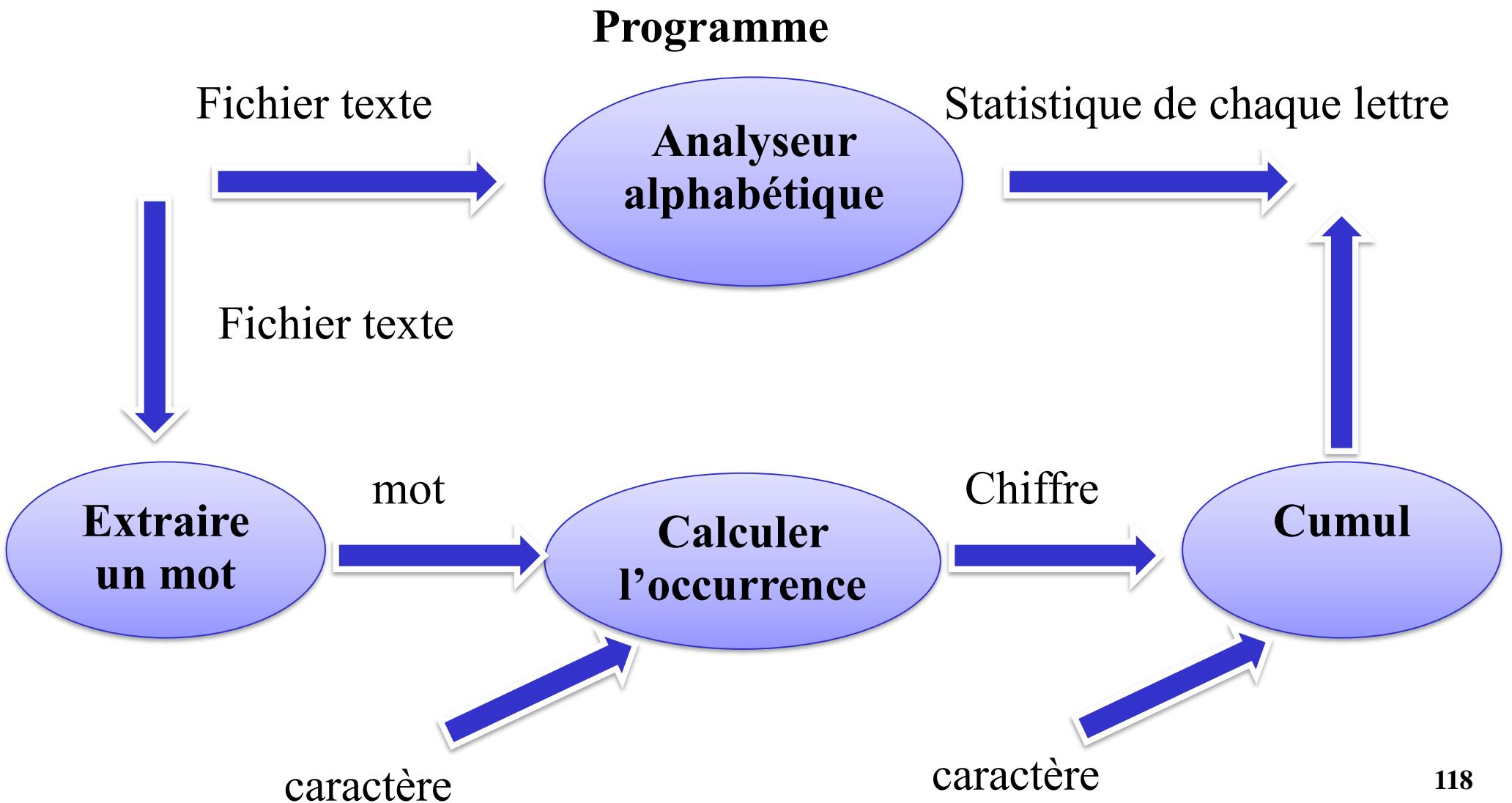


Analyse descendante

- Exemple
- Résoudre le problème revient à :
 - Extraire un mot du fichier
 - Pour chaque, lettre de l'alphabet, compter le nombre d'occurrence
 - Calculer le cumul d'occurrence de chaque lettre
 - Répéter la traitement jusqu'au dernier mots du fichier.
- Chacun de ces sous-problèmes devient un nouveau problème à résoudre
- Si on considère que l'on sait résoudre ces sous-problèmes, alors on sait “quasiment” résoudre le problème initial

Analyse descendante

- Exemple



Fonctions et sous-programmes

- **Sous-programmes et fonctions**

Donc écrire un programme qui résout un problème revient toujours à écrire des sous-programmes qui résolvent des sous parties du problème initial. Il faut comprendre les mots “programme” et de “sous-programme” comme “programme algorithmique” indépendant de toute implantation

- En algorithmique il existe deux types de sous-programmes :
 - Les fonctions : réalisent des traitements en se servant des valeurs de certaines variables et renvoient un résultat. Elles se comportent comme des fonctions mathématiques : $y=f(x, y, \dots)$
 - Les procédures : réalisent seulement des traitements mais ne renvoient aucun résultat

Fonctions et sous-programmes

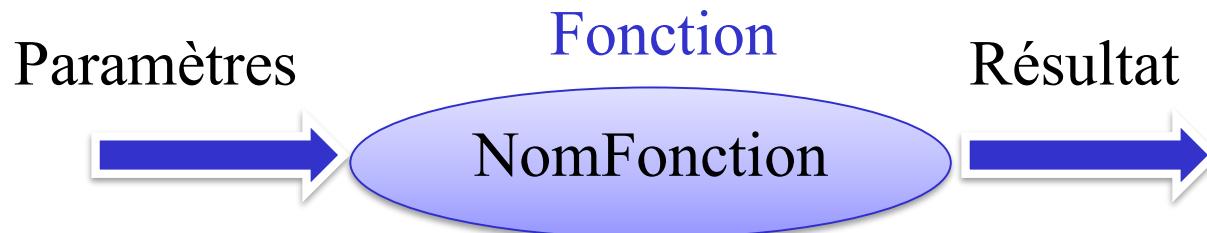
- **Fonctions et procédures**

Les **fonctions** et les **procédures** sont des groupes d'instructions indépendants désignés par un nom. Elles ont plusieurs **intérêts** :

- permettent de "**factoriser**" **les programmes**, c-à-d de mettre en commun les parties qui se répètent
- permettent une **structuration** et une **meilleure lisibilité** des programmes
- **facilitent la maintenance** du code (il suffit de modifier une seule fois)
- ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes

Fonctions en C

- Fonctions en C
- En C, il n'y a pas de différence entre fonctions et procédures :
 - il n'y a que des fonctions censées renvoyer une valeur d'un type spécifié
 - Un sous-programme (procédure) est une fonction qui ne renvoie rien (type void)
- Une fonction est un bloc d'instructions ayant :
 - Un type pour les valeurs qu'elle retourne (type du résultat)
 - Un nom (nom de la fonction)
 - Une liste de paramètres typés, entre parenthèses (paramètres)



Fonctions en C

- **Exemple**

La fonction SommeCarre suivante calcule la somme des carrées de deux réels x et y :

```
float SommeCarre (float x, float y )
{
    float z;
    z = x*x+y*y;
    return z;          // on retourne le résultat à l'aide de l'instruction return
}
```

La fonction Pair suivante détermine si un nombre est pair :

```
int Pair(int x)
{
    if (x%2 == 0)
        return 1;      // vue qu'il n'y a pas de type booléen en C, on retourne 1
    else                  //pour dire que c'est vrai et 0 pour faux
        return 0;
}
```

Fonctions en C

- Exemple

La fonction Affiche permet d'afficher le message Bonjour à l'écran:

```
void Affiche ()  
{  
    printf("Bonjour \n");  
    /* L'affichage à l'écran n'est pas considéré comme un résultat en C, donc pas  
    besoin de l'instruction return */  
}
```

La fonction afficheInt permet d'afficher le contenu d'une variable entière à l'écran:

```
void AfficheInt ( int i )  
{  
    printf("La valeur de i est %d \n", i );  
    /* L'affichage à l'écran n'est pas considéré comme un résultat en C, donc pas  
    besoin de l'instruction return */  
}
```

Fonctions en C

- **Remarque**

Les paramètres des fonctions sont vus comme des variables locales au bloc de code de la fonction. On peut définir d'autres variables dans le bloc.

Si la fonction ne renvoie aucun résultat, on utilise le mot réservé **void**. L'affichage à l'écran n'est pas considéré comme un résultat (un retour)

Si la fonction n'a besoin d'aucun paramètre on écrit simplement () ou (void)

```
int ValeurAbsolue ( int X )
{
    int valeur;
    if (X >= 0)
        valeur = X;
    else
        valeur = - X;

    return valeur;
}
```

Fonctions en C

- **Instruction return**

On spécifie la valeur que renvoie une fonction au moyen de l'instruction **return**

- **Valeur de même type que le type de retour déclaré de la fonction**

```
int ValeurAbsolue ( int X ) // retourne la valeur absolue d'un entier passé en paramètre
{
    int valeur;
    if (X >= 0 )
        valeur = X;
    else
        valeur = - X;

    return valeur;
/* remarquer que le type de la variable valeur est le même que celui du retour de la fonction
ValeurAbsolue
*/
}
```

Fonctions en C

- **Instruction return**

L'instruction return permet la terminaison anticipée de la fonction

- **Peut exister en plusieurs exemplaires**

```
int Pair(int x)
{
    if (x%2 == 0)
        return 1;      // vue qu'il n'y a pas de type booléen en C, on retourne 1
    else                  //pour dire que c'est vrai et 0 pour faux
        return 0;

    printf ("Ce message ne sera jamais affiché \n");
}
```

Fonctions en C

- **Instruction return**

Pour les fonctions ne retournant rien, soit on utilise l'instruction return sans argument, soit on met rien. Dans ce cas, le type de retour de la fonction est void

```
void Affiche ()  
{  
    printf("Bonjour \n");  
}
```

```
void Affiche ()  
{  
    printf("Bonjour \n");  
  
    return;          // optionnel  
}
```

Fonctions en C

- **Exercice**

Ecrire une fonction permettant de retourner le max de deux réels.

```
float max2 ( float a, float b )
{
    float max;          // le résultat final sera stocké dans cette variable

    if (a >= b)
        max = a;
    else
        max = b;

    return max;
}.
```

Fonctions en C

- **Exercice**

Ecrire une fonction permettant de retourner le factoriel d'un entier donné

```
int factoriel ( int n )
{
    int fac;          // le résultat final sera stocké dans cette variable
    int i;            // utilisée dans la boucle

    fac = 1;          // on initialise la variable à 1 pour pouvoir calculer les produits

    for (i = 1; i <= n; i++)
    {
        fac = fac * i;           // ou encore fac *= i;
    }

    return fac;
}
```

Fonctions en C

- Fonction main : fonction principale

int main () est une fonction particulier qui retourne un entier et dont la liste des paramètres est vide. Elle est appelée la fonction principale.

Tout programme doit contenir obligatoirement la fonction principale, qui est exécutée lorsque le programme est lancé.

```
int main ()
{
    ...
    return EXIT_SUCCESS;
}
```

Fonctions en C

- **Exercice**

Ecrire une fonction permettant d'afficher le contenu d'un tableau d'entiers passé en paramètre. La taille du tableau est aussi passé en paramètre de la fonction

```
void afficheTableau ( int tab[], int n )
{
    int i;          // utilisée dans la boucle
    for (i =0; i < n; i++)
        printf("la valeur de la case d'indice %d est : %d", i, tab[i]);
}
```

Fonctions en C

- **Appel de fonction**

On appelle une fonction en donnant son nom, suivi de la valeur des paramètres de l'appel, dans l'ordre de définition

Le nom de fonction avec ses arguments est une expression typée utilisable normalement

```
int ValeurAbsolue ( int X )  
{  
    int valeur;  
    if (X >= 0 )  
        valeur = X;  
    else  
        valeur = - X;  
  
    return valeur;  
}
```

```
int main ()  
{  
    int val1;  
    val1 = ValeurAbsolue (3);      ← Appel de Fonction  
  
    printf("la valeur absolue est : %d ", val1);  
  
    return EXIT_SUCCESS;  
}
```

Fonctions en C

- Appel de fonction

```
int ValeurAbsolue ( int X )  
{  
    int valeur;  
    if (X >=0 )  
        valeur = X;  
    else  
        valeur = - X;  
  
    return valeur;  
}
```

```
int main ()  
{  
    int val1, val2;  
  
    printf("Saisir un entier");  
    scanf ("%d", &val1);  
  
    val2 = ValeurAbsolue(val1);  
    printf("la valeur absolue de %d est : %d ", val1, val2);  
  
    return EXIT_SUCCESS;  
}
```

Fonctions en C

- Appel de fonction

```
int ValeurAbsolue ( int X )  
{  
    int valeur;  
    if (X >=0 )  
        valeur = X;  
    else  
        valeur = - X;  
  
    return valeur;  
}
```

```
void AfficheInt ( int i )  
{  
    printf("La valeur de i est %d \n", i );  
}
```

```
int main ()  
{  
    int val1, val2;  
  
    printf("Saisir un entier");  
    scanf ("%d", &val1);  
  
    val2 = ValeurAbsolue(val1);  
    AfficheInt(val2);  
  
    return EXIT_SUCCESS;  
}
```

Fonctions en C

- **Exercice**

Dans un exercice, nous avons défini une fonction qui permet de calculer le max de deux réel. Maintenant, on veut écrire une fonction max3 qui calcule de max de trois réels. Donner le code de max3 , en faisant un appel à la fonction max2

```
float max2 ( float a, float b )  
{  
    int max;  
  
    if (a >= b)  
        max = a;  
    else  
        max = b;  
  
    return max;  
}  
. . .
```

```
float max3 ( float a, float b, float c )  
{  
    int m, max;  
  
    m = max2 (a,b);  
  
    max = max2(m,c)  
  
    return max;  
}  
. . .
```

Fonctions en C

- **Exercice**

Ecrire un programme qui demande à l'utilisateur de saisir trois réels et qui affiche leur max

Fonctions en C

- Corrigé

```
/* Fichier maximum.c
 * Ce fichier contient le programme qui
 * calcule le max de trois entiers
 * Auteur : SMI2008
 * Version : v1
 * Date création : aujourd'hui
 */
#include < stdlib.h>
#include < stdio.h>

/* Fonction qui calcule le max de 2 réels
float max2 ( float a, float b )
{
    int max;
    if (a >= b)
        max = a;
    else
        max = b;
    return max;
}
```

```
/* Fonction qui calcule le max de 3 réels */
float max3 ( float a, float b, float c )
{
    int m, max;
    m = max2 (a,b);
    max = max2(m,c)
    return max;
}

/* Fonction principale qui lance l'exécution */
int main ()
{
    float x, y, z, max ;
    printf("Saisir trois réel : ");
    scanf ("%f%f%f",&x,&y,&z);

    max = max3(x,y,z);
    printf("La max est %f : ", max);
    return EXIT_SUCCESS;
}
```

Paramètres des fonctions

- **Paramètres formels et effectifs**

Les paramètres servent à échanger des données entre la fonction appelante et la fonction appelée

- Vocabulaire

- Un paramètre formel est aussi appel aussi paramètre
 - Un paramètre effectif est aussi appel argument

- Signification

- Les paramètres placés dans la déclaration d'une fonctions sont des paramètres formels. Ils peuvent prendre toutes les valeurs possibles dans le type déclaré mais ils sont abstraits (n'existent pas réellement)
 - Les paramètres placés dans l'appel d'une fonction sont des paramètres effectifs. ils contiennent les valeurs pour effectuer le 139 traitement

Paramètres des fonctions

- **Paramètres formels et effectifs**

Un paramètre effectif est une variable ou constante (numérique ou définie par le programmeur)

Le paramètre formel et le paramètre effectif sont associés lors de l'appel de la fonction. Donc,

- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels.
- L'ordre et le type des paramètres doivent correspondre

Paramètres des fonctions

- Exemple

```
int ValeurAbsolue ( int X )  
{  
    int valeur;  
    if (X >= 0 )  
        valeur = X;  
    else  
        valeur = - X;  
  
    return valeur;  
}
```

Paramètres formels

```
void AfficheInt ( int i )  
{  
    printf("La valeur de i est %d \n", i );  
}
```

```
int main ()  
{  
    int val1, val2;  
  
    printf("Saisir un entier");  
    scanf ("%d", &val1);  
  
    val2 = ValeurAbsolue(val1);  
    AfficheInt(val2); // ok  
    AfficheInt(3); // Ok  
    AfficheInt(Tab[ ]); //erreur  
  
    return EXIT_SUCCESS;  
}
```

Paramètres effectifs

Paramètres des fonctions

- **Transmission des paramètres**

Il existe deux modes de transmission de paramètres en C :

- La transmission par valeur : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la fonction. Dans ce mode le paramètre effectif ne subit aucune modification.
 - Lorsque le type des paramètres est un type simple, la transmission est par valeur
- La transmission par adresse (ou par référence) : les adresses des paramètres effectifs sont transmises à la fonction appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel lors de l'exécution de la fonction
 - Remarque : le paramètre effectif doit être une variable (et non une valeur) lorsqu'il s'agit d'une transmission par adresse.
 - Lorsque le type des paramètres est un type tableau ou une référence (pointeur),¹⁴² la transmission est par référence

Paramètres des fonctions

- Transmission des paramètres : Exemple

```
/* Passage par valeur */
void incrementer (int X, int Y )
{
    X = X+1;
    Y = Y +1;
}
```

```
void affiche( int i, char c )
{
    printf("La valeur de %c est %d \n", c, i );
}
```

La valeur de x est 1
La valeur de y est 2

```
int main ( )
{
    int x, y;
    x = 1;
    y = 2;
    afficher(x, 'x');
    afficher(y, 'y');

    incrementer (x, y);
    afficher(x, 'x');
    afficher(y, 'y');

    return EXIT_SUCCESS;
}
```

Paramètres des fonctions

- Transmission des paramètres : Exemple

```
/* Passage par valeur */
void echange (float y, float x )
{
    float z;
    z =x;
    x = y;
    y = z
}
```

```
void affiche( float i, char c )
{
    printf("La valeur de %c est %f \n", c, i );
}
```

La valeur de x est 1
La valeur de y est 2

La valeur de x est 1
La valeur de y est 2

```
int main ()
{
    float x, y;

    x = 1;
    y = 2;

    afficher(x, 'x');
    afficher(y, 'y');

    echange (x, y);

    afficher(x, 'x');
    afficher(y, 'y');

    return EXIT_SUCCESS;
}
```

Paramètres des fonctions

- Transmission des paramètres : Exemple

```
/* Passage par référence */
void remplirTab1 ( int tab [ ],int taille)
{
    int i;
    for (i =0; i < taille; i++)
        tab[i] = i;
}
```

```
void affiche( int tab[], int taille )
{
    int i =0;
    for (i =0; i < taille; i++)
        printf("La valeur de la case %d est %d\n", i, tab[i]);
```

La valeur de la case 0 est 0
La valeur de la case 1 est 1
La valeur de la case 2 est 2

```
/* Passage par référence */
void remplirTab2 ( int tab [ ],int taille)
{
    int i;
    for (i =0; i < taille; i++)
        tab[i] = 0;
}
```

```
main()
{
    int tab [3];
    remplirTab1( tab, 3);
    afficher(tab, 3);
}
```

La valeur de la case 0 est 0
La valeur de la case 1 est 0
La valeur de la case 2 est 0

```
remplirTab2( tab, 3);
afficher(tab, 3);
return EXIT_SUCCESS;
}
```

Paramètres des fonctions

- Transmission des paramètres : conclusion

Types simples : int, float, char,

Passage par valeur

Types : tableaux, pointeurs

Passage par référence

Paramètres des fonctions

- **Exercice**

Ecrire une fonction une fonction remplirTab qui permet de remplir un tableau de réels. Les valeurs sont saisies par l'utilisateur

```
/* Passage par référence */
void remplirTab ( float tab [ ],int taille)
{
    int i;
    for (i =0; i < taille; i++)
    {
        printf("Saisir la valeur de la case %d ", i);
        scanf("%f", &tab[i]);
    }
}
```

Paramètres des fonctions

- **Exercice**

Ecrire une fonction afficheTab qui permet d'afficher un tableau de réels.

```
void afficheTab( float t[], int taille )
{
    int i =0;
    for (i =0; i < taille; i++)
        printf("La valeur de la case %d est %f \n", i, t[i] );
}
```

Paramètres des fonctions

- **Exercice**

Ecrire une fonction incTab qui permet d'incrémenter les valeurs des cases d'un tableau

```
void incrTab( float t[], int taille )
{
    int i =0;
    for (i =0; i < taille; i++)
        t[i] = t[i]+1;
}
```

Paramètres des fonctions

- **Exercice**

Ecrire un programme qui demande à l'utilisateur de

- le nombre de notes à saisir,
- saisi ces notes
- incrémente les valeurs saisies et
- les affiche

Paramètres des fonctions

- Corrigé

```
/* Fichier incremente.c
 * Ce fichier contient le programme qui permet
 * d'incrémenter les valeurs saisies par
 * l'utilisateur
 * Auteur : SMI2008
 * Version : v1
 * Date création : aujourd'hui
 */
#include < stdlib.h>
#include < stdio.h>

void remplirTab ( float tab [ ],int taille)
{
    int i;
    for (i =0; i < taille; i++)
    {
        printf("Saisir la valeur de la case %d ", i);
        scanf("%f", &tab[i]);
    }
}
```

```
void afficheTab( int t[], int taille )
{
    int i =0;
    for (i =0; i < taille; i++)
        printf("La valeur de la case %d est %d \n", i, t[i] );
}

void incrTab( float t[], int taille )
{
    int i =0;
    for (i =0; i < taille; i++)
        t[i] = t[i]+1;
}

int main ( )
{
    int n;
    float tab[10];
    printf("Saisir le nombre des notes ( max est 10 )");
    scanf ("%d",&n);
    remplirTab(tab, n);
    incrTab(tab,n);
    afficheTab(tab,n);
    return EXIT_SUCCESS;
}
```

Paramètres des fonctions

- **Exercice**

Ecrire une fonction chercherMotif qui permet de chercher un entier dans un tableau, elle retourne 1 si l'entier existe et 0 sinon

```
int chercherMotif (int T[], int taille, int motif)
{
    int i;
    for (i =0; i < taille; i++)
    {
        if( t[i] == motif)
            return 1; // on termine la fonction
    }

    return 0;
}
```

```
int chercherMotif (int T[], int taille, int motif)
{
    int i, isIn =0 ;
    for (i =0; i < taille; i++)
    {
        if( t[i] == motif)
        {
            isIn = 1; // on l'a trouvé
        }
        else
    }

    return isIn;
}

// C'est quoi l'inconvénient de cette solution
```

Paramètres des fonctions

- **Exercice**

Ecrire un programme qui demande à l'utilisateur le nombre d'éléments à saisir, saisi ces éléments et cherche un élément donné.

Paramètres des fonctions

- Corrigé

```
/* Fichier chercher.c
 * Ce fichier contient le programme qui permet
 * d'un élément parmi les valeurs saisies par
 * l'utilisateur
 * Auteur : SMI2008
 * Version : v1
 * Date création : aujourd'hui
 */
#include < stdlib.h>
#include < stdio.h>

int chercherMotif(int T[], int taille, int motif)
{
    int i;
    for (i =0; i < taille; i++)
    {
        if( t[i] == motif)
            return 1;
    }
}
```

```
    return 0;
}

void remplirTab ( float tab [ ],int taille)
{
    /*ajouter ici le code de l'exo précédent (manque de place) */
}

int main ( )
{
    float n, motif, tab[20] ;
    int result;
    printf("Saisir le nombre d'éléments( max est 20 )");
    scanf ("%d",&n);
    remplirTab(tab, n);
    printf("Elément recherché ? :");
    scanf ("%d",&motif);
    result = chercherMotif(tab,n, motif);
    if (result == 1)
        printf("L'élément n'existe pas ");
    else
        printf("L'élément existe ");

    return EXIT_SUCCESS;
}
```

Paramètres des fonctions

- **Exercice**

Ecrire une fonction minTab qui permet de chercher le plus petit élément dans un tableau d'entier à partir d'une case donnée index :

```
int minTab(int T[], int index, int taille )
{
    int i;
    int tmp;
    tmp = T[index];           // on suppose que le plus est T[ indice ] : initialisation
    for (i = index + 1; i < taille; i++) // la boucle commence à partir d' index + 1
    {
        if( T[i] < tmp )      // si je trouve un plus petit, je change le min
        {
            tmp = T[i];
        }
    }

    return tmp;
}
```

Paramètres des fonctions

- **Exercice**

Ecrire une fonction tri qui permet de trier un tableau d'entier par ordre croissant sans utiliser de tableau intermédiaire (faire deux boucles)

```
void tri (int T[], int taille )  
{  
    int i,j ;  
    int tmp;  
    for (i =0; i < taille; i++)  
    {  
        for (j = i+1; j < taille; j++)  
        {  
            if( T[j] < T[i] )  
            {  
                tmp = T[i];  
                T[i] = T[j];  
                T[j] = tmp;  
            }  
        }  
    }  
}
```

Visibilité des variables

- **Variables locales et globales**

- On peut déclarer une variable soit

- Dans un bloc d'instruction : variable locale
- En dehors d'un bloc d'instruction, dans un fichier : variable globale

```
/* Fichier test.c
*/
#include < stdlib.h>
#include < stdio.h>

int b =1;      // variable globale
void remplirTab ( float tab [ ],int taille)
{
    int i;      // variable locale
    for (i =0; i < taille; i++)
        scanf("%f", &tab[i]);
}
```

Visibilité des variables

- **Variables locales et globales**

On peut manipuler 2 types de variables dans un module (fichier, fonction) : des variables locales et des variables globales.

- Une variable locale n'est connue qu'à l'intérieur du module où elle a été définie. Elle est créée à l'appel du module et **détruite** à la fin de son exécution
- Une variable globale est connue par l'ensemble des modules et le programme principale. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme

Visibilité des variables

- **Variables globales**

Une variable déclarée en dehors du corps d'une fonction est visible de toutes les fonctions du fichier dans lequel elle est déclarée

```
int i;          // variable visible dans tout le fichier

void f( )
{
    i=4;        // On la voit et on l'utilise ici
}

int main( )
{
    i=3;        // on la voit et on l'utilise ici
    f();
    printf("%d",i); // 3 ou 4 ?
    return EXIT_SUCCESS;
}
```

Visibilité des variables

- **Variables locales**

Une variable déclarée à l'intérieur du corps d'une fonction ou d'un bloc d'instruction n'est visible qu'à l'intérieur du corps du bloc.

```
void f( )
{
    int i;      // une première variable locale i
    i=4;        // On la voit et on l'utilise ici
}
void g()
{
    int i;      // une autre, différente de la première
    i=3;        // On la voit et on l'utilise ici
}

int main( )
{
    i=5;        // ici, on ne voit ni l'une ni l'autre : erreur
    f();
    g();
    return EXIT_SUCCESS;
}
```

Visibilité des variables

- **Variables locales**

Lorsqu'une variable locale a le même nom qu'une variable globale, celle-ci est masquée à l'intérieur du bloc ou la variable locale est définie.

```
int i;                      // variable visible dans tout le fichier

void f( )
{
    int i;                  // la variable locale masque la variable globale
    i=4;                    // Ici, on utilise donc la variable locale
}

int main( )
{
    i=3;                   // on la voit et on l'utilise ici
    f( );
    printf("%d",i);        // 3 ou 4 ?
    return EXIT_SUCCESS;
}
```

Récursivité

- **Principe**

Une fonction peut s'appeler elle-même: on dit que c'est une fonction **récursive**

Toute fonction récursive doit posséder un cas limite (cas trivial) qui arrête la récursivité

```
int factoriel (int n )
{
    if ( n==1 )
        return 1;
    else
        return ( n * factoriel (n-1) );
}
```

Récursivité

- **Exercice**

Ecrivez une fonction récursive qui calcule le terme n de la suite de Fibonacci définie par :

$$U(0)=U(1)=1$$

$$U(n)=U(n-1)+U(n-2)$$

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

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

Partie 5 : pointeurs et tableaux

Plan

- Pointeurs
- Occupation mémoire
- Tableaux
- Tableaux et pointeurs
- Arithmétique des pointeurs
- Pointeurs et fonctions
- Chaînes de caractère

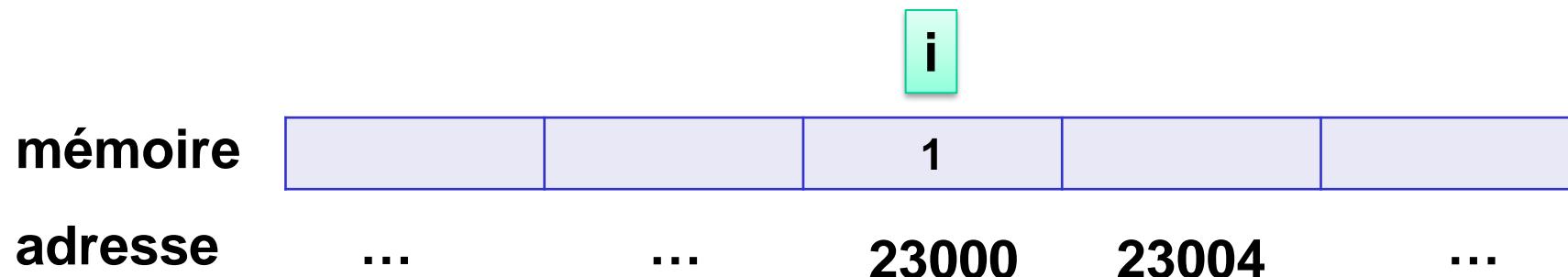
Pointeurs

- **Variables**

Les variables servent à stocker les données manipulées par le programme.

Lorsque l'on déclare une variable, par exemple un entier i, l'ordinateur réserve un espace mémoire pour y stocker les valeurs de i.

L'emplacement de cet espace dans la mémoire est nommé adresse



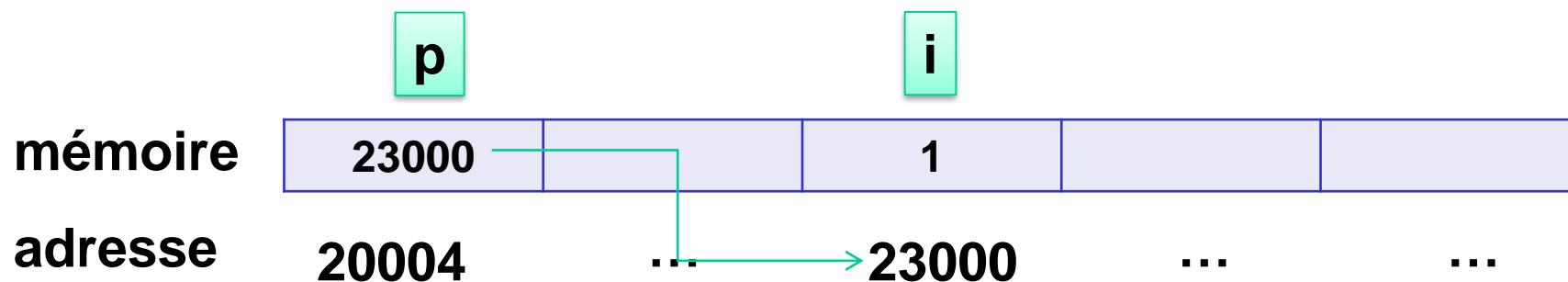
Pour connaître l'adresse d'une variable donnée, **on utilise l'opérateur &**. Ainsi, &i vaut 23000

Pointeurs

- **Variable pointeur**

Un pointeur est une variable dont le contenu est une adresse.

Par exemple, si on déclare une variable entière initialisée à 1 et que l'on déclare un pointeur p dans lequel on range l'adresse de i (on dit que p pointe sur i), on a un schéma qui ressemble à :



Pour connaître l'adresse d'une variable donnée, on utilise l'opérateur &. Ainsi, &i vaut 23000

Pointeurs

- **Déclaration d'un pointeur**

On déclare un pointeur au moyen de l'opération d'indirection « * »

Syntaxe :

type * var ;

(type : le type du contenu pointé par var)

Attention :

Le contenu de var est une adresse. Le type des données pointé par var est « type ».

Exemple :

```
int i, j;           // déclaration de deux variables de type entier
int * p;           // déclaration d'un pointeur sur une variable entière
char * s, * t;     // déclaration de deux pointeurs
```

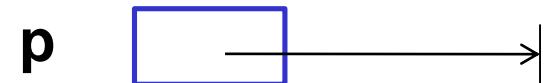
Pointeurs

- Initialisation d'un pointeur

Par défaut, lorsqu'on déclare un pointeur, on ne sait pas sur quoi il pointe. Come toute variable, il faut l'initialiser

- On peut dire que le pointeur ne pointe sur rien en lui affectant la valeur NULL (le masse)

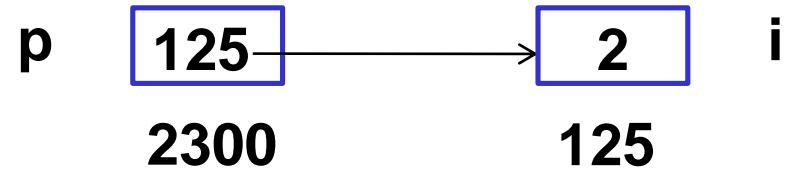
```
int * p = NULL;
```



- On peut dire qu'il pointe sur une variable à l'aide de l'opérateur de &

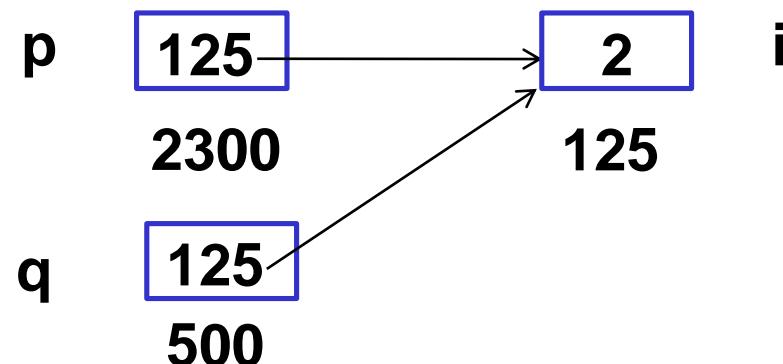
```
int i = 2;
```

```
int * p = &i;
```



- On peut dire qu'il point sur la même adresse qu'un autre pointeur

```
int *q = p;
```

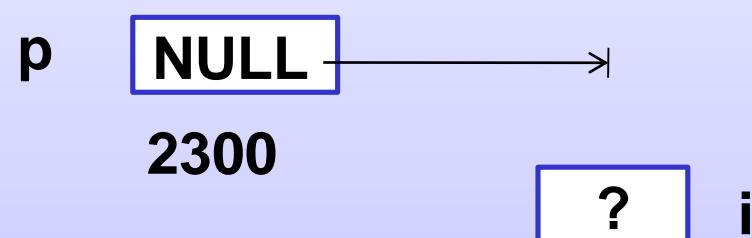


Pointeurs

- Accès au contenu d'un pointeur

Pour accéder au contenu pointé par un pointeur, on utilise « * »

```
int * p = NULL;
```



```
int i;
```



```
p = &i;
```



```
*p = 3;
```



```
(*p) ++;
```



170

printf(" le contenu de la valeur pointé par p est %d et la valeur de i est :", *p, i);

Pointeurs

- Accès au contenu d'un pointeur

Pour accéder au contenu pointé par un pointeur, on utilise « * »

```
int a;  
int b;  
int * maxptr;  
  
a=5;  
b=17;  
maxptr = (a>b) ? &a : &b;    // on obtient la référence du max  
(*maxptr)++;                  // incrémente le max de 1  
  
printf("Le maximum incrémenté est %d \n", *maxptr);
```

Occupation mémoire

- **Taille d'une variable**

A toute variable créée est associée une zone de la mémoire, servant à stocker le contenu de cette variable

La taille de cette zone mémoire dépend du type de la variable considérée :

- char : 1 octet
- int : 2, 4 ou 8 octets (selon l'architecture du système)
- float : 4 octets
- double : 8 octets
- etc

Occupation mémoire

- **Taille d'une variable**

L'opérateur « `sizeof()` » donne la taille en octets du type ou de la variable passée en paramètre

- Ce n'est pas une fonction normale
- Evalué et remplacé par la constante entière correspondante lors de la compilation

```
printf("Sur mon système un \" int \" fait %d octets \n", sizeof(int)) ; // taille d'un type ici, int  
  
double d;  
printf("Sur mon système un \" double \" fait %d octets \n", sizeof(d)) ; // taille d'une variable
```

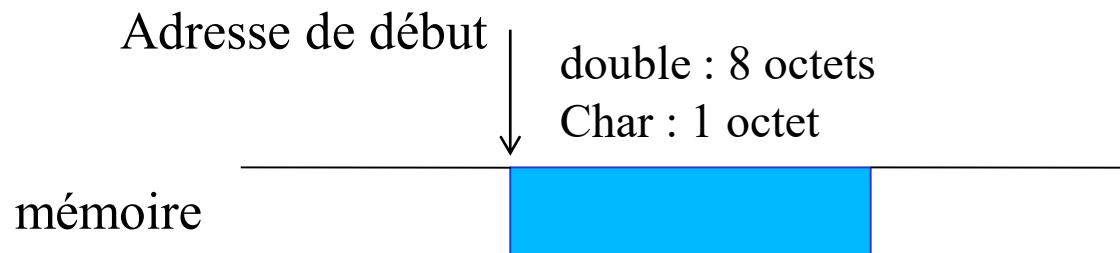
Occupation mémoire

- **Taille d'une variable**

Tout mot (octet) de la mémoire est identifié par un numéro unique : son adresse mémoire

On peut donc identifier toute zone mémoire servant au stockage d'une variable par :

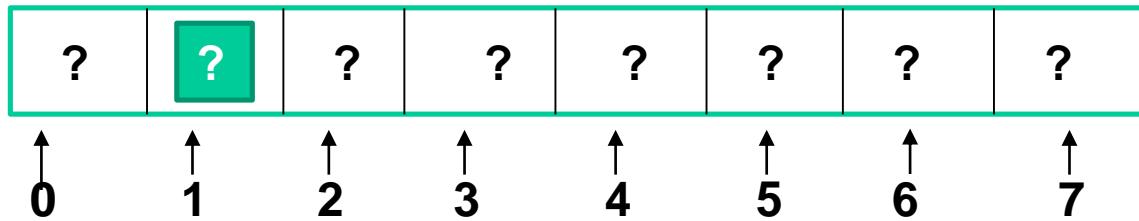
- Son adresse de début
- Sa taille (dépend du type de la variable)



Tableaux

- **Tableau**
- Un tableau est une collection de variables de même type que l'on peut accéder individuellement par leur indice dans le tableau
- On déclare la taille du tableau entre « [] »

```
int t[8]; /* t est un tableau de 8 entier */
```



- On accède à un élément d'un tableau en donnant l'indice de l'élément entre « [] ». En C, les indices commencent à 0 et pas à 1

Tableaux

- **Tableau**
- On peut initialiser un tableau lors de sa déclaration
 - Liste des constantes du type adéquat, entre accolades

```
int t[8] = {1, 2, 3, 4, 5, 6, 7, 8 };
```

- Nombre d'éléments comptés par le compilateur

```
int t[] = {1, 2, 3, 4, 5, 6, 7, 8 };      /* Le compilateur comptera à votre place */
```

- La liste peut être partielle

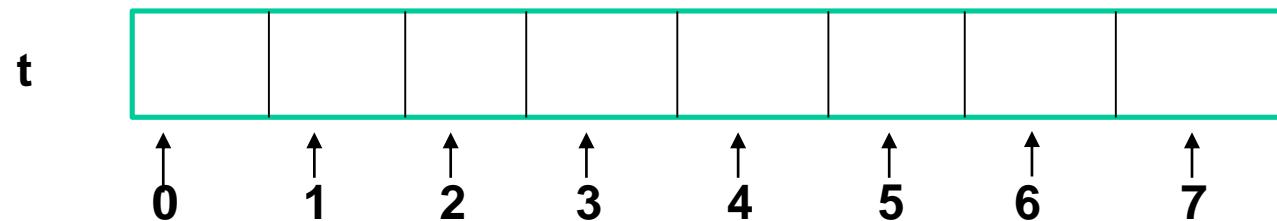
```
int t[8] = {1, 2, 3, 4};                  /* 8 places prises, 4 initialisées */
```

Tableaux

- **Tableau**

On réalité, un tableau est une référence sur une zone mémoire de variables contiguës de même type

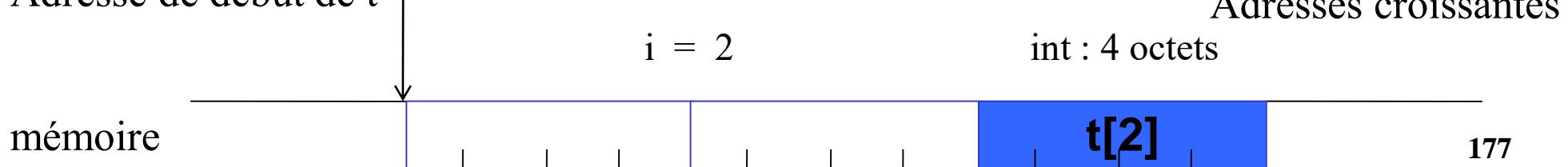
int t[8];



Si `t` est de type « type », `t[i]` est donc le contenu de la zone mémoire :

- Commençant à l'octet d'adresse : $(t + i * \text{sizeof(type)})$
- De taille, `sizeof(type)`

Adresse de début de `t`



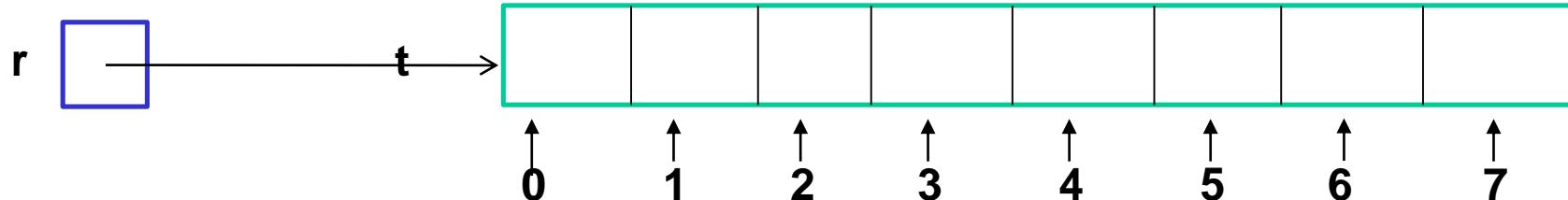
Tableaux et pointeurs

- **Tableaux et pointeurs**

Tableaux et pointeurs étant du même type, on peut donc :

- Utiliser le nom d'un tableau comme une adresse

```
int t[8];           /* t est un tableau de 8 entier */  
int *r;             /* pointeur sur un entier */  
int a, b, c ;  
  
r = t ;            /* t est en de type ( int *) */  
a = t[0];           /* Lecture du premier élément du tableau */  
b = *r ;            /* Comme r pointe sur t, on lit aussi t[0] */  
c = * t;             /* Comme t est un (int *), ceci marche aussi */  
t = r;              /* ne marche pas, car t est une constante */
```



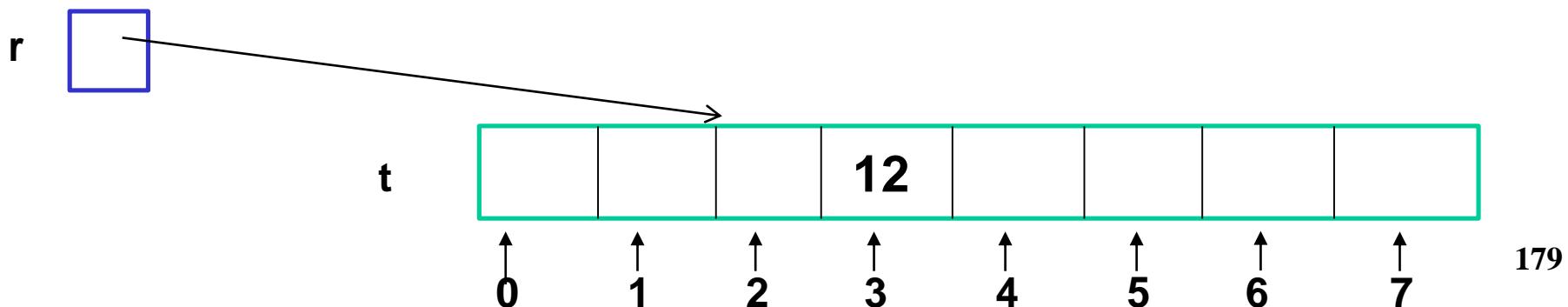
Tableaux et pointeurs

- **Tableaux et pointeurs**

Tableaux et pointeurs étant du même type, on peut donc :

- Utiliser les notations « [] » à partir des variables de types pointeurs :

```
int t[8];          /* t est un tableau de 8 entier */  
int *r;           /* pointeur sur un entier */  
  
r = &t[2];         /* on pointe sur la 3ème case du tableau */  
r[1] = 12;         /* on met 12 dans r[1], c'est-à-dire dans t[3] */
```



Arithmétique des pointeurs

- **Incrémantion et décrémentation**

A partir d'un pointeur p , on peut définir un pointeur q du même type mais pointant à une case en amont ou en aval

$$q = \&p[i]$$

On peut simplifier cette écriture en définissant une arithmétique des pointeurs

- $p+i \Leftrightarrow \&p[i]$
- $p-i \Leftrightarrow \&p[-i]$

```
int t[8];          /* t est un tableau de 8 entier */  
int *r;           /* pointeur sur un entier */  
  
r = t +2         /* même chose que r = &t[2] */
```

Arithmétique des pointeurs

- **Opération traditionnelle**

Tous les opérateurs arithmétiques d'addition et de soustraction d'entiers existent également pour les pointeurs :

- « += », « -= », « ++ », « -- »,

```
int t[8];          /* t est un tableau de 8 entier      */
int *r;           /* pointeur sur un entier          */
int i

for ( i=0; i < 8; i++)        /* boucle sur les indices          */
    scanf("%d", &t[i]);       /* ou bien scanf("%d", t+i);      */

for ( r = t; r < t +8; r++)   /* boucle sur les références      */
    scanf("%d", r);
```

Pointeurs et fonctions

- **Transmission des paramètres**
 - On a dit qu'il existe deux modes de transmission de paramètres en C :
 - La transmission par valeur : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la fonction.
 - La transmission par adresse (ou par référence) : les adresses des paramètres effectifs sont transmises à la fonction appelante.

Dans tous les cas, ce sont des copies des variables qui sont transmises

Pointeurs et fonctions

- Passage par valeur : type simple

```
/* Passage par valeur */
void echange (float x, float y )
{
    float z;
    z =x;
    x = y;
    y = z
}
```

La valeur de x est 1
La valeur de y est 2

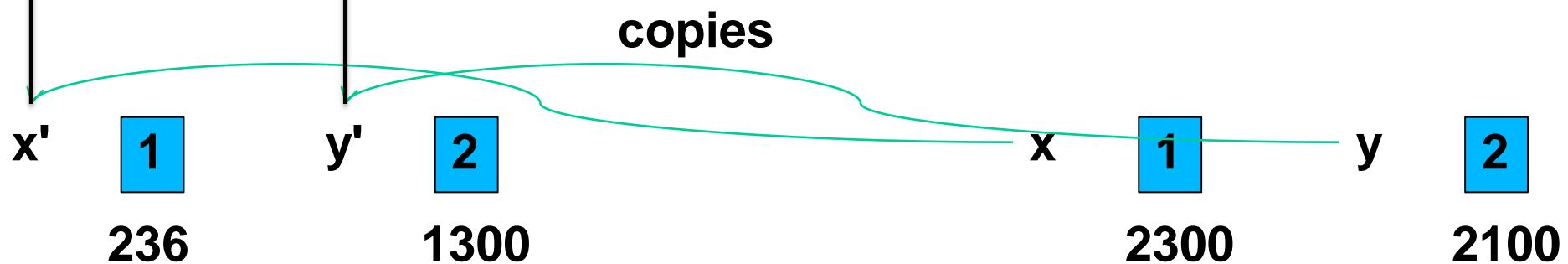
La valeur de x est 1
La valeur de y est 2

```
int main ()
{
    float x, y;
    x = 1;
    y = 2;
    echange (x, y);
    printf(" x = %d \n", x );
    printf(" y = %d \n", y );
    return EXIT_SUCCESS;
}
```

Pointeurs et fonctions

- Passage par valeur : type simple

```
/* Passage par valeur */
void echange (float x, float y )
{
    float z;
    z =x;
    x = y;
    y = z
}
```



```
int main ()
{
    float x, y;

    x = 1;
    y = 2;

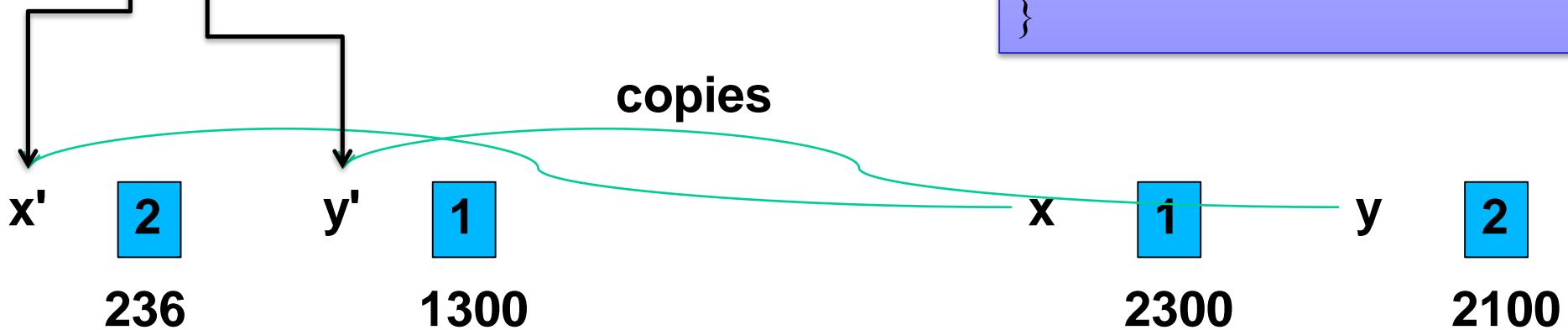
    echange (x, y);
    ...
}
```

Les arguments qui sont passés à echange() sont des copies de x et y

Pointeurs et fonctions

- Passage par valeur : type simple

```
/* Passage par valeur */
void echange (float x, float y )
{
    float z;
    z =x;
    x = y;
    y = z
}
```



```
int main ()
{
    float x, y;

    x = 1;
    y = 2;

    echange (x, y);
    ...
}
```

echange() modifie le contenu des copies qui sont détruites à la sortie de la fonction ¹⁸⁵

Pointeurs et fonctions

- Passage par adresse : pointeur

```
/* Passage par adresse */
void echange (float * x, float * y )
{
    float *z;
    z =x;
    x = y;
    y = z
}
```

```
int main ()
{
    float x, y;
    float *p, q;
    x = 1;
    y = 2;

    p = &x;
    q= &y ;
    echange (x, y); // erreur

    echange (p, q); // ou echange(&x,&y)

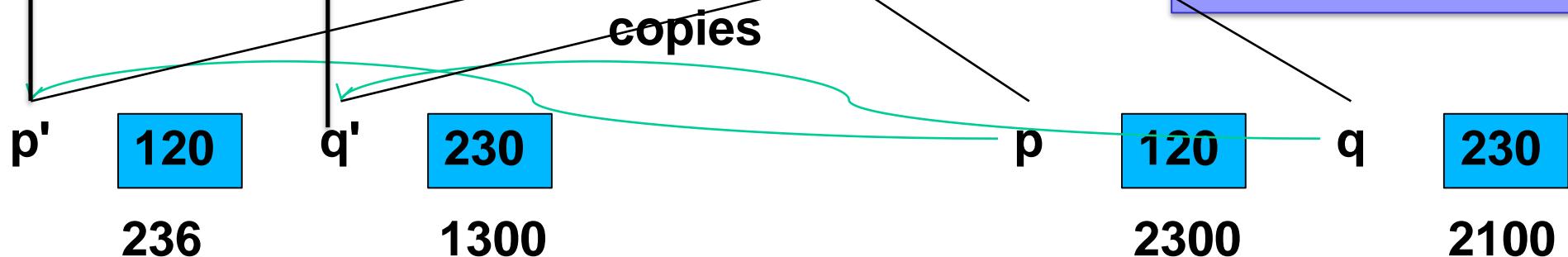
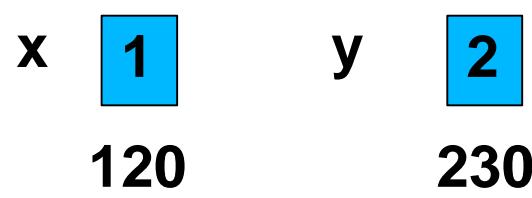
    printf(" x = %d \n", x );
    printf(" y = %d \n", y );

    return EXIT_SUCCESS;
}
```

Pointeurs et fonctions

- Passage par adresse : pointeur

```
/* Passage par adresse */
void echange (float * x, float * y )
{
    float *z;
    z =x;
    x = y;
    y = z
}
```



```
int main ()
{
    float x, y;
    float *p, q;
    x = 1;
    y = 2;

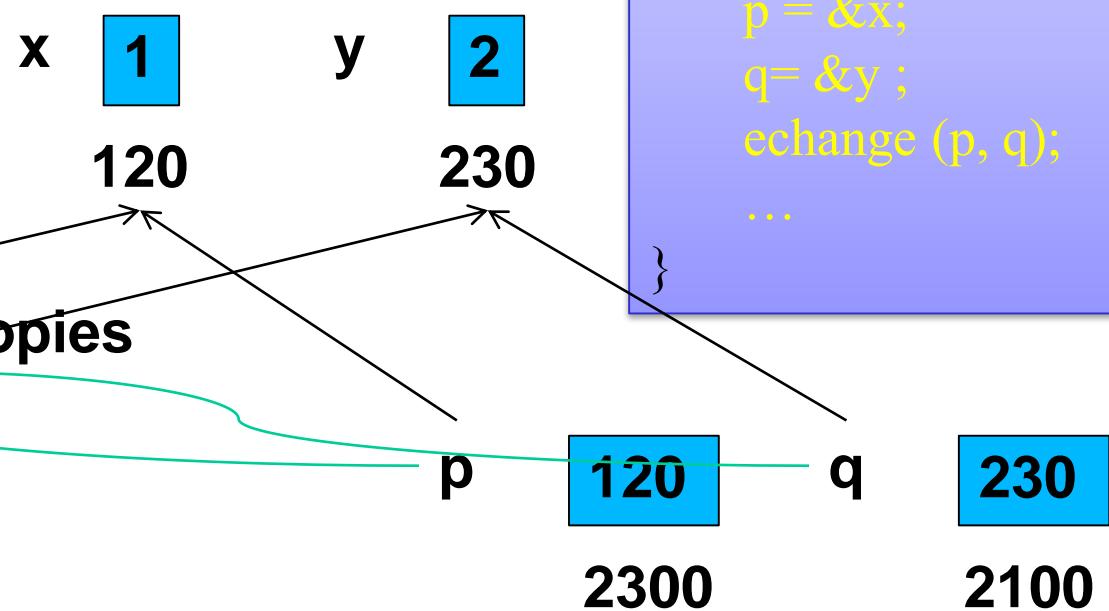
    p = &x;
    q = &y ;
    echange (p, q);
    ...
}
```

Les arguments qui sont passés à `echange()` sont des copies de `p` et `q` : `p'` et `q'`¹⁸⁷

Pointeurs et fonctions

- Passage par adresse : pointeur

```
/* Passage par adresse */
void echange (float * x, float * y )
{
    float *z;
    z =x;
    x = y;
    y = z
}
```



```
int main ()
```

```
{
```

```
    float x, y;  
    float *p, q;  
    x = 1;  
    y = 2;
```

```
    p = &x;  
    q= &y ;  
    echange (p, q);  
    ...
```

`echange()` modifie des copies qui sont détruites à la sortie de la fonction

Pointeurs et fonctions

- Passage par adresse : pointeur

```
/* Passage par adresse */
void echange (float * x, float * y )
{
    float z;
    z = *x;
    *x = *y;
    *y = z
}
```

```
int main ()
{
    float x, y;
    float *p, q;
    x = 1;
    y = 2;

    p = &x;
    q = &y ;
    echange (x, y); // erreur

    echange (p, q); // ou echange(&x,&y)

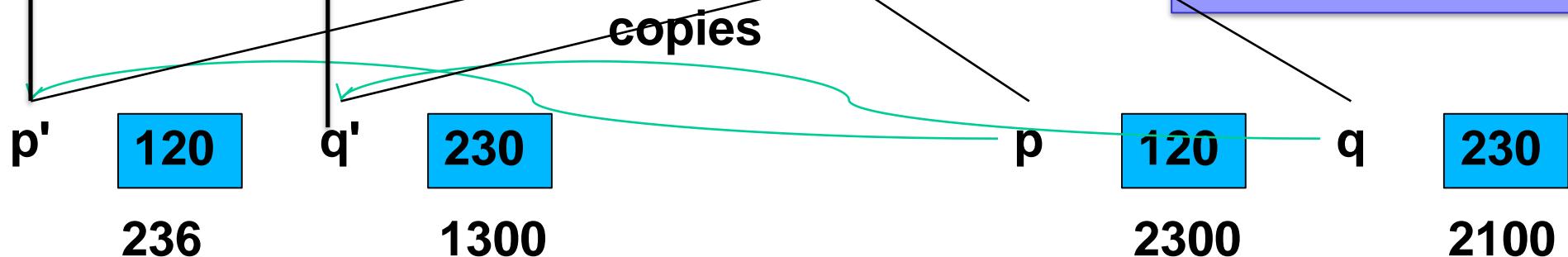
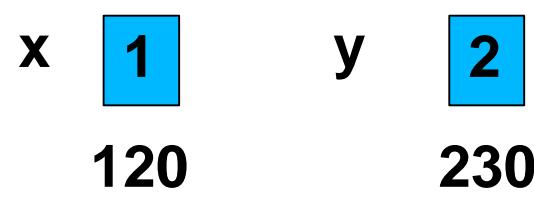
    printf(" x = %d \n", x );
    printf(" y = %d \n", y );

    return EXIT_SUCCESS;
}
```

Pointeurs et fonctions

- Passage par adresse : pointeur

```
/* Passage par adresse */
void echange (float * x, float * y )
{
    float z;
    z =*x;
    *x = *y;
    *y = z
}
```



```
int main ()
{
    float x, y;
    float *p, q;
    x = 1;
    y = 2;

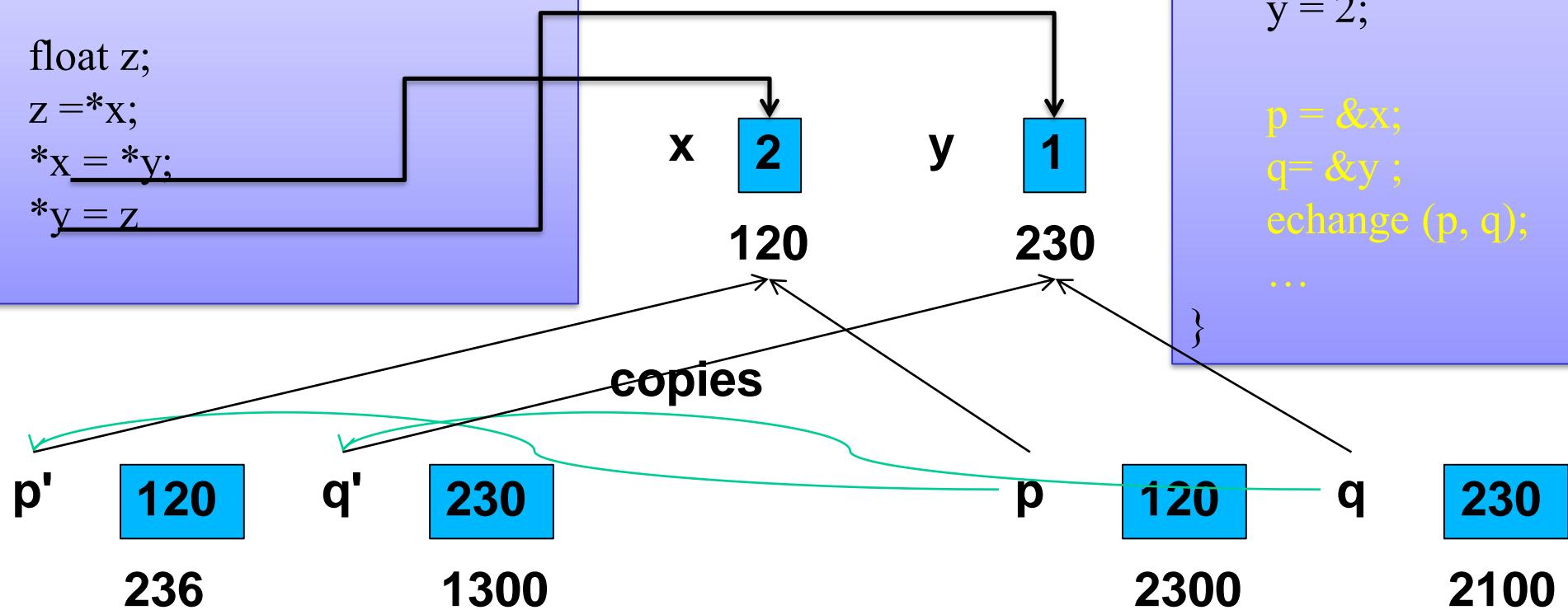
    p = &x;
    q = &y ;
    echange (p, q);
    ...
}
```

Les arguments qui sont passés à `echange()` sont des copies de **p** et **q** : **p'** et **q'**

Pointeurs et fonctions

- Passage par adresse : pointeur

```
/* Passage par adresse */
void echange (float * x, float * y )
{
    float z;
    z =*x;
    *x = *y;
    *y = z
}
```



```
int main ()
```

```
{
```

```
    float x, y;  
    float *p, q;  
    x = 1;  
    y = 2;
```

```
    p = &x;  
    q = &y ;  
    echange (p, q);  
    ...
```

Les arguments qui sont passés à `echange()` sont des copies de `p` et `q` : `p'` et `q'` ¹⁹¹

Pointeurs et fonctions

- Passage par adresse

```
/* Passage par référence */
void remplirTab1 ( int tab [ ],int taille)
{
    int i;
    for (i =0; i < taille; i++)
        tab[i] = i;
}
```

```
void affiche( int tab[], int taille )
{
    int i =0;
    for (i =0; i < taille; i++)
        printf("La valeur de la case %d est %d
```

La valeur de la case 0 est 0
La valeur de la case 1 est 1
La valeur de la case 2 est 2

```
/* Passage par référence */
void remplirTab2 ( int tab [ ],int taille)
{
    int i;
    for (i =0; i < taille; i++)
        tab[i] = 0;
}
```

La valeur de la case 0 est 0
La valeur de la case 1 est 0
La valeur de la case 2 est 0

```
main()
{
    int tab [3];
    remplirTab1( tab, 3);
    afficher(tab, 3);

    remplirTab2( tab, 3);
    afficher(tab, 3);
    return EXIT_SUCCESS;
}
```

Pointeurs et fonctions

- Transmission des paramètres : conclusion

Types simples : int, float, char,

Passage par valeur

Types : tableaux, pointeurs

Passage par référence

Chaînes de caractères

- Il n'existe pas de type chaîne spécifique

Une chaîne de caractères en C est un tableau unidimensionnel de caractères

- Le nom de la chaîne fait référence à l'adresse de début du premier caractère de la chaîne

Une chaîne de caractères bien formée est toujours terminée par un caractère nul '\0'

- Indique où le contenu utile de la chaîne finit
- Ne pas oublier de compter et de réservé sa place!!

Chaînes de caractères

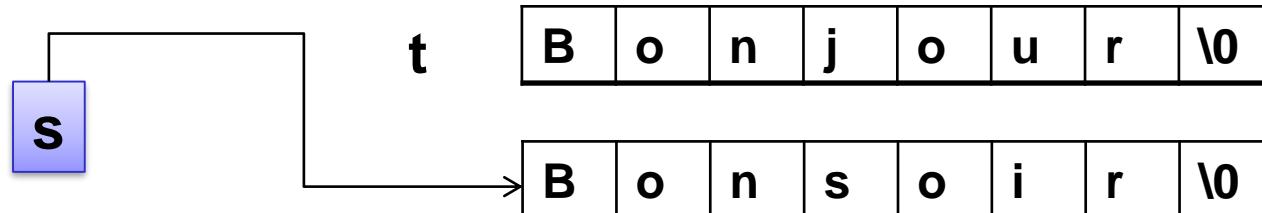
- **Déclaration d'une chaîne**

On peut déclarer et initialiser une chaîne de caractères comme on déclare un tableau de caractère.

```
char t[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

Il est préférable d'utiliser les constantes

```
char t[] = "Bonjour";           // tableau modifiable en mémoire
char * s ="Bonsoir";           // pointeur sur une constante
t[1] = 'Z';                    // légal car t est modifiable
s[1] = 'A';                    // illégal car s pointeur sur une zone constante
s = t;                         // légal car s est une variable
```



Chaînes de caractères

- **Affichage**
- Directement dans printf si chaîne simple : %s pour affiche les chaînes
- La chaîne est affichée jusqu'au premier « \0 » rencontré

```
char t[] = "Bonjour";      // tableau modifiable en mémoire
char * s ="Bonsoir";      // pointeur sur une constante
```

```
printf("%s", t);
printf("%s", s);
```

Chaînes de caractères

- **Lecture**
- On utilise scanf
- Code « %s » aussi
- Pas besoin de « & » car une chaîne est déjà une référence sur une zone mémoire
- Limiter la taille avec la syntaxe « %**nums** » pour éviter les débordements de tampons

```
char str[10];           // Une chaîne de 10 caractères
int a;

printf("Enterez votre nom : ");
scanf("%s", str);       // pas de & car str est une référence

scanf("%9s", str);      // on limite à 9 pour \0
```

Chaînes de caractères

- **Fonctions de manipulation de chaînes**

Dans la bibliothéque stdlib.h (#include <stdlib.h>)

- strlen() : renvoie la taille courante d'une chaîne (pas la taille maximale du tableau)
- strcpy () : copie d'une chaîne source vers un tableau de caractères destination
- strcat () : ajout d'une chaîne source à la fin d'une chaîne destination
- Strchr () : recherche de la première occurrence d'un caractère dans une chaîne
- Etc, ...