

Cours du module : Algorithmique avancée et Python

Plan du cours

□ Structure de données en langage C

- Introduction
- Rappel sur le langage C
- Structures linéaires : Piles, Files, Listes...
- Structures non linéaires : arbre binaire

□ Langage Python:

- Bases du langage
- Structures de contrôle
- Structures du données
- Fonctions
- Calcul scientifique



Cours du module : Algorithmique avancée et Python

Introduction

Introduction

Qu'est-ce qu'un algorithme ?

- Enchaînement d'opérations destiné à résoudre un problème
- Spécification d'un schéma de calcul sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé
-

L'élaboration d'un algorithme exige :

- Description des données
- Description des méthodes
- Preuve de bon fonctionnement

La complexité d'un algorithme :

- Temps de calcul
- Espace nécessaire

Introduction

Type

Un **type de données**, ou simplement **type**, définit le genre de contenu d'une donnée et les **opérations** pouvant être effectuées sur la variable correspondante.

⇒ Le typage est l'association à un objet :

- un ensemble de valeurs possibles,
- et un ensemble d'opérations admissibles sur ces valeurs

Structure de données

Une **structure de données** est une manière particulière de stocker et d'organiser des données dans un ordinateur de façon à pouvoir être utilisées efficacement. On distingue :

- Structures de données du langage (types primitifs : Tableaux,...)
- Structures de données abstraites (piles, listes...)

Introduction

Structure de données abstraite

Un **type abstrait** ou une **structure de données abstraite** est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'elles peuvent effectuer.

On qualifie d'abstrait ce type de données car il correspond à un cahier des charges qu'une structure de données doit ensuite implémenter.

Structure dynamique

Une **structure dynamique** est une structure dont la taille peut varier en fonction des besoins.

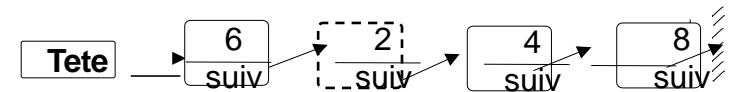
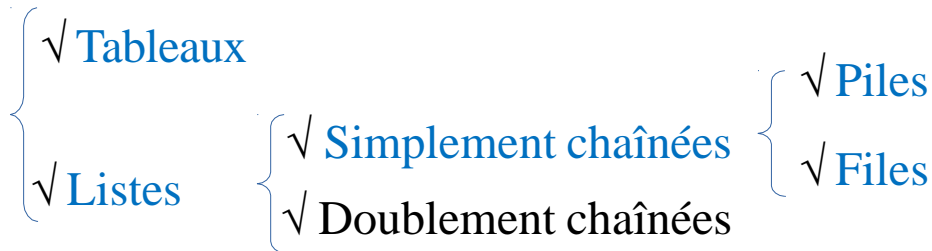


Ce cours exige la maîtrise des notions de **pointeurs** et de **gestion dynamique de la mémoire**.

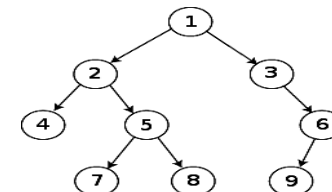
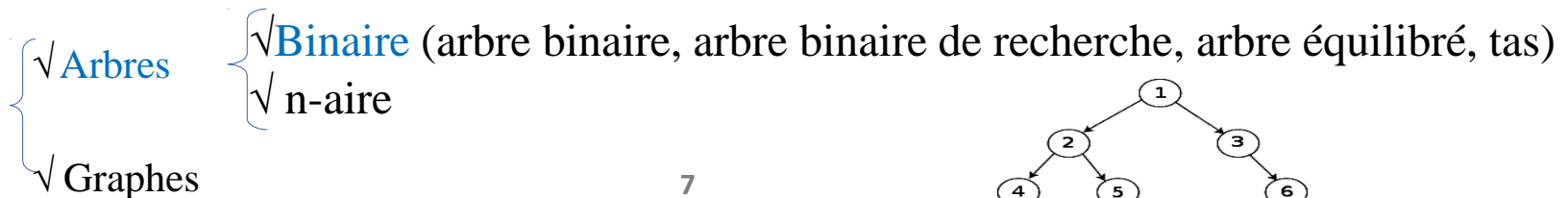
Introduction

On peut classer les structures de données en deux grandes catégories :

- **Les structures de données linéaires**, qui permettent de relier des données en séquence (on peut numéroté les éléments)



- **Les structures de données non linéaires**, qui permettent de relier un élément à plusieurs autres éléments





Chapitre 1 : Rappels sur le langage C

Chapitre 1 : Rappels sur le langage C

- ❑ Variables et constantes
- ❑ Types, Expressions et opérateurs
- ❑ Structures de contrôle
- ❑ Tableaux
- ❑ Pointeurs
- ❑ Fonctions
- ❑ Chaines de caractères
- ❑ Allocation dynamique de la mémoire
- ❑ Structures



Les variables & les constantes

Les variables

Définition

- Elles servent à **stocker les valeurs** des données utilisées pendant l'exécution d'un programme et elles doivent être **déclarées avant d'être utilisées**.
- Une variable est **caractérisée** par un :
 - **nom** (Identificateur) : constitué de lettres, de chiffres et du caractère
 - **type** (entier, réel, ...) : l'ensemble des valeurs qu'elle peut prendre.
- En langage C, il n'y a que 2 types de base : les **entiers** et les **réels** avec différentes variantes pour chaque type.

Types Entier : 4 variantes d'entiers :

- **char** : caractères (entier sur 1 octet)
- **short ou short int** : entier court (entier sur 2 octets)
- **int** : entier standard (entier sur 2 ou 4 octets)
- **long ou long int** : entier long (4 octets)

Types Réel : 3 variantes de réels :

- **float** : réel simple précision codé sur 4 octets
- **double** : réel double précision codé sur 8 octets
- **long double** : réel très grande précision codé sur 10 octets

Les variables

Déclaration

- La déclaration introduit les **variables** qui seront utilisées, leur **type** et parfois aussi leur **valeur** de départ (initialisation).

- **Syntaxe** :

`Type NomVar1, NomVar2, ..., NomVarN;`

- **Exemples** :

```
int i, j, k;  
float x, y ;  
double z=1.5;  
short compteur;  
char c=`A` ;
```

Les constantes

Définition et déclaration

□ Définition :

- Une **constante** est une **zone mémoire** qui conserve sa valeur pendant toute l'exécution d'un programme.

□ Déclaration :

- On associe une valeur à une constante en utilisant :
 - la directive **#define** : **#define nom_constante valeur;**
 - le mot clé **const** : **const type nom_constante = valeur;**

□ Exemples :

```
#define Pi 3.14;    (la constante ne possède pas de type)
#define MIP "Math-Informatique-Physique";
const float Pi =3.14; (la constante est typée)
```



Expressions et opérateurs

Expressions et opérateurs

Définition

- **Expression** :
 - ▣ peut être une **valeur**, une **variable** ou une **opération**.
 - ▣ **Exemples** : 1, b, $a*2$, $a+3*b-c$, ...
- **Opérateur** :
 - ▣ **symbole** qui permet de manipuler des **variables** (opérandes) pour produire un **résultat**.
 - ▣ On distingue :
 - les **opérateurs binaires** : nécessitent **deux opérandes** (ex : $a + b$)
 - les **opérateurs unaires** : nécessitent **un seul opérande** (ex: $a++$)
 - l'**opérateur conditionnel ?** : nécessite **trois opérandes**.
- Une **expression** fournit une seule **valeur**, elle est évaluée en respectant des règles de **priorité** et d'**associativité**.

Expressions et opérateurs

Opérateurs du langage C

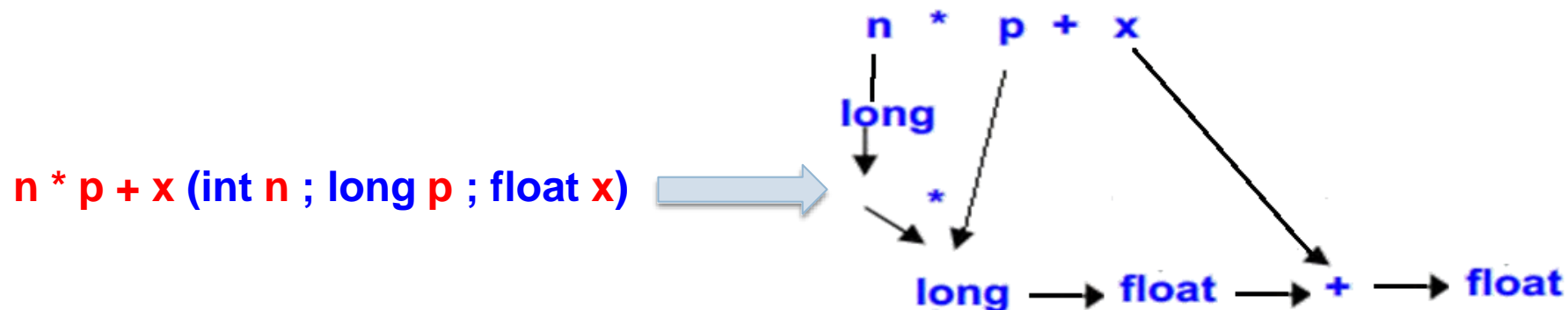
□ Exemples :

- opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%` (modulo)
- opérateurs d'affectation : `=`, `+=`, `-=`, `*=`, `/=`, ...
- opérateurs logiques : `&&`, `||`, `!`
- opérateurs de comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`
- opérateurs d'incrément et de décrémentation : `++`, `--`
- opérateurs sur les bits : `&`, `|`, `~`, `^`, `<<`, `>>`
- autres opérateurs particuliers : `?:`, `sizeof`, `cast`, ...

Expressions et opérateurs

Opérateurs arithmétiques

- Les **opérandes** peuvent être des **entiers** ou des **réels** sauf pour **%** qui agit uniquement sur des entiers.
- Lorsque les types des deux opérandes sont différents il y'a **conversion implicite** dans le **type le plus fort**.
- La **conversion implicite** se fait comme suit :
`int → long → float → double → long double.`
- **NB:** (**short** et **char** sont convertis en **int**).
- Exemple de conversion implicite :



Expressions et opérateurs

Opérateur de forçage de type (cast)

- Il est possible d'effectuer des **conversions explicites (casting)** ou de **forcer le type** d'une expression.

- **Syntaxe** :

(type) expression;

- **Exemple** :

int n, p;

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

Expressions et opérateurs

Opérateur conditionnel ?

- Syntaxe :

exp1 ? exp2 : exp3

exp1 est évaluée, si sa valeur est non nulle c'est **exp2** qui est exécutée, sinon **exp3** qui est exécutée.

- Exemple 1 : **max = (a > b) ? a : b;**

Si $a > b$ alors on affecte à max le contenu de a, sinon on lui affecte b.

- Exemple 2 : **int i=10;**

(a > b) ? i++ : i--;

Si $(a > b)$ on incrémente i sinon on décrémente i.

Expressions et opérateurs

Opérateur SIZEOF

- Fournit la **taille** en octets d'un **type** ou d'une **variable**.
- **Syntaxe** :
`sizeof (type) ;` ou `sizeof (variable) ;`
- **Exemples** :
`double n;`
`printf ("%d \n", sizeof(int)) ;` affiche 4
`printf ("%d \n", sizeof(n)) ;` affiche 8

Expressions et opérateurs

Priorités des opérateurs en C

Priorité 1 (la plus forte):	()
Priorité 2:	! ++ --
Priorité 3:	* / %
Priorité 4:	+ -
Priorité 5:	< <= > >=
Priorité 6:	== !=
Priorité 7:	&&
Priorité 8:	
Priorité 9 (la plus faible):	= += -= *= /= %=



Structures de contrôle

Structures de contrôle

Introduction

- Les structures de contrôle **conditionnent** l'exécution des instructions en fonction de la **valeur d'une expression**.
- On distingue **deux types** de **structures de contrôle** :
 - **structures alternatives (tests)** : permettent d'effectuer des choix : **if ... else** et **switch**.
 - **structures répétitives (boucles)** : permettent de répéter plusieurs fois l'exécution d'un ensemble d'instructions : **while, do...while** et **for**.

Structures de contrôle

Structure `if...else`

□ Structure `if...else` :

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

□ Imbrication de `if...else` :

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

□ Remarques :

- ***else*** est toujours associé au dernier ***if*** qui ne possède pas une partie ***else***.
- pour éviter toute ambiguïté ou pour forcer une certaine interprétation, il vaut mieux utiliser les **accolades** : {...}.

Structures de contrôle

Structures : switch, while, et do ...while

□ Structure switch :

```
main( )
{ char c;
  switch (c) {
    case 'a':printf("voyelle\n"); break ;
    case 'b':printf("consonne\n");
              }
}
```

□ Structure while :

```
main( )
{ int i, som;
  i =0; som= 0;
  while (som <=10)
    { i++;
      som+=i;}
  printf (" La valeur cherchée est N= %d\n ", i);}
```

□ Structure do .. while:

```
main()
{ int N;
  do {
    printf (" Entrez une note entre 0 et 20 \n");
    scanf("%d",&N);
  } while (N < 0 || N > 20);
}
```

Structures de contrôle

Structure for

□ Structure for :

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

- **expr1** : effectue l'initialisation (évaluée 1 fois au début de la boucle).
- **expr2** : constitue le test de continuation .
- **expr3** : pour réinitialiser les données de la boucle.

□ Exemple : un programme qui calcule **x** à la puissance **n** :

```
main ( ) {
    float x, puiss;
    int n, i;
    { printf (" Entrez les valeurs de x et n \n");
      scanf ("%f %d" , &x, &n);
      for (puiss =1, i=1; i<=n; i++)
          puiss*=x;
      printf (" %f à la puissance %d est égal à : %f", x,n,puiss);
    }
}
```

Structures de contrôle

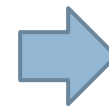
Les instructions **break** et **continue**

□ **break** :

- peut être utilisée dans une boucle (**for**, **while**, ou **do .. while**) pour **arrêter le déroulement de la boucle** et **passer** à la première instruction qui la suit .
- En cas d'imbrication de boucles, **break** ne quitte que la boucle (ou le switch) le plus interne..

□ Exemple :

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



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

Structures de contrôle

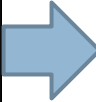
Les instructions `break` et `continue`

□ `continue` :

- ▣ peut être utilisée dans une boucle (`for`, `while`, ou `do .. while`) pour **abandonner l'itération courante** et passer à l'itération suivante.

▣ Exemple :

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



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

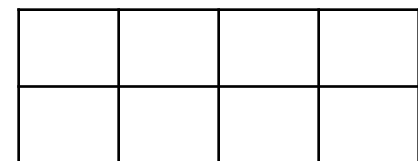
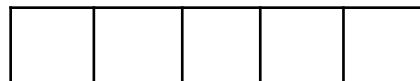


Les tableaux

Les tableaux

Caractéristiques

- **Un tableau** est une **variable structurée** composée d'un nombre de **variables** simples de **même type** appelées **éléments**.
- Les **éléments** du tableau sont stockées en mémoire à des emplacements **contigus** (l'un après l'autre).
- Le type des éléments du tableau peut être :
 - **un type simple** : char, int, float, double, ...
 - **un pointeur**.
 - **une structure**.
- Un tableau peut être **unidimensionnels** ou **multidimensionnel**.



Les tableaux

Tableaux unidimensionnels

- **Syntaxe de déclaration :** `Type nom[dimension];`

- Exemple : `float notes[30];`

- **Initialisation à la déclaration :**

- `Type nom[dimension] = {val1, val2, val3, ...};`

- Exemple : `int A[5] = {1, 2, 3, 4, 5};`

- **Accès aux éléments d'un tableau :**

- ▣ L'accès à un élément du tableau se fait au moyen de l'indice **T[i]**.

```
int T[5] = {9, 8, 7, 6, 5};
```



`T[0]=9, T[1]=8, T[2]=7, T[3]=6, T[4]=5`

Les tableaux

Tableaux multidimensionnels

- Un tableau **multidimensionnel** est un **tableau** qui **contient des tableaux**.

- **Syntaxe** : **Type Nom[d1] [d2]...[dn] ;**

- **Exemple** : Tableaux à deux dimensions :

short A[2] [3] ;

- **A** est un tableau comportant **2 éléments**, chacun d'entre eux étant un **tableau de 3 éléments**.
- On peut représenter le tableau A de la manière suivante :

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

- **Initialisation à la déclaration :**

float A[3][4] = { {-1.5, 2.1, 3.4, 0}, {8, 7.5, 1, 2.7 }, {3.1, 0, 2, -1} } ;

Les tableaux

Remarque

- En C, le **nom du tableau (T)** représente l'**adresse du premier élément** du tableau : **$T = \&T[0]$**

- Exemple :

```
short T[5] = {100, 200, 300, 400, 500};
```

⇒ **$T = \&T[0] = 1E06$**

- On peut écrire également :

⇒ **$\text{short } *P;$**

⇒ **$P = T;$**

1E05	
T → 1E06	100
1E08	200
1E0A	300
1E0C	400
1E0E	500
1E0F	

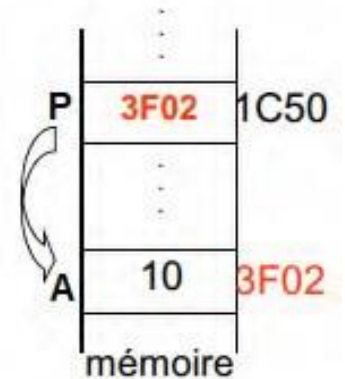


Les pointeurs

Les pointeurs

Définition

- Un **pointeur** est une **variable spéciale** qui peut contenir l'**adresse** d'une autre variable.
- **Exemple** :
 - ▣ A est une **variable** qui contient la valeur 10.
 - ▣ P est un **pointeur** qui contient l'**adresse de A** (*P pointe sur A*).
- **Remarques** :
 - ▣ Le **nom** d'une **variable** permet d'accéder **directement** à sa valeur → **adressage direct**.
 - ▣ Un **pointeur** qui contient l'adresse de la variable, permet d'accéder **indirectement** à la valeur de cette variable → **adressage indirect**.
 - ▣ Le **nom** d'une **variable** est lié à la **même adresse**, alors qu'un **pointeur** peut pointer sur **différentes adresses**.



Les pointeurs

Intérêts

- Manipuler de façon simple des données pouvant être **importantes**.
 - **Exemple** : **passage par référence** pour les paramètres des fonctions (au lieu de passer à une fonction un élément **très grand en taille**, on pourra par exemple lui fournir un pointeur vers cet élément...).
- Permettent d'**allouer dynamiquement la mémoire**.
- Permettent de créer des **structures de données** (**listes, piles, files, arbres**) dont le nombre d'éléments peut évoluer de façon dynamique.
- ...

Les pointeurs

Déclaration

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

- **Exemples** :
 - `int *pi;` // **pi** est un pointeur vers une variable de type **int**.
 - `float *pf;` // **pf** est un pointeur vers une variable de type **float**.

- **Remarque** :

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

Les pointeurs

Opérateurs de manipulation

- Lors du travail avec des pointeurs, nous utilisons :
 - L'opérateur **&** : '**adresse de**' : pour obtenir l'**adresse** d'une **variable**.
 - L'opérateur ***** : '**contenu de**' : pour accéder au **contenu** d'une **adresse** (un pointeur).

- **Exemples :**

```
int * p;    //on déclare un pointeur vers une variable de type int

int i=10, j=30; // deux variables de type int

p=&i;       // on met dans p, l'adresse de i (p pointe sur i)

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

*p=20;      // met 20 dans la case pointée par p (i vaut 20 après l'instruction)

p=&j;       // p pointe sur J

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

Les pointeurs

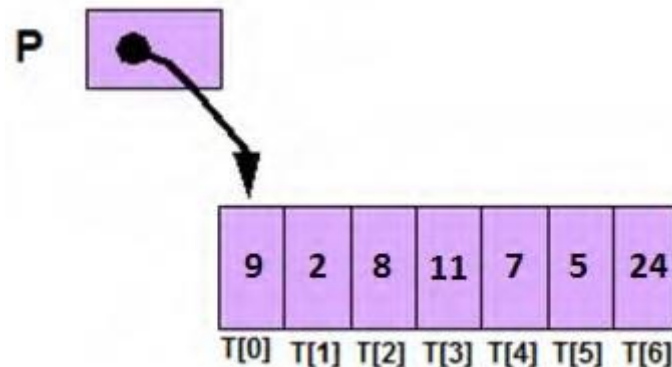
Initialisation

- A la déclaration d'un pointeur p (`int *p;`), on ne sait pas sur quel **zone mémoire il pointe**. Ceci peut générer des problèmes.
- Conseil :
 - Toute **utilisation d'un pointeur** doit être précédée par une **initialisation**.
 - On peut initialiser un pointeur en lui affectant :
 - l'**adresse d'une variable** : Exemple : `int a, *p1; p1=&a;`
 - un **autre pointeur déjà initialisé** : Exemple : `int *p2; p2=p1;`
 - la **valeur 0** désignée par le symbole **NULL** :
 - Exemple : `int *p; p=0;`
ou `p=NULL;` (on dit que p pointe 'nulle part': aucune adresse).

Les pointeurs

Pointeurs et tableaux

- Le **nom d'un tableau** représente l'**adresse** de son premier élément.
- Si on déclare un tableau **T**, alors : **T=&T[0]**.
- On peut dire que **T** est un **pointeur constant** sur le 1^{er} élément du tableau.
- En déclarant un tableau **T** et un pointeur **P** du même type, l'instruction **P=T** fait pointer **P** sur le premier élément de **T** : **P=&T[0]**.

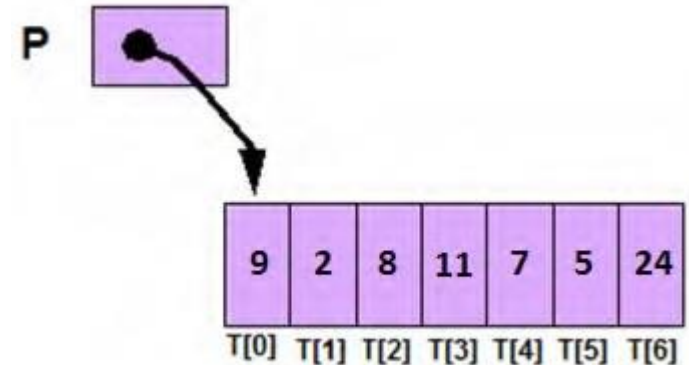


Les pointeurs

Pointeurs et tableaux

□ A partir de là, on peut manipuler le tableau **T** en utilisant **P**, en effet :

- **P** pointe sur **T[0]** et ***P** désigne **T[0]**
- **P+1** pointe sur **T[1]** et ***(P+1)** désigne **T[1]**
-
- **P+i** pointe sur **T[i]** et ***(P+i)** désigne **T[i]**



□ **Exemple :**

```
short x, A[7]={5,0,9,2,1,3,8};  
short *P;  
P=A;  
x=*(P+5);
```



x = A[5] = 3



Les fonctions

Les fonctions

Syntaxe de définition

- Une **fonction** est définie en dehors de la fonction principale **main ()** par la syntaxe suivante :

```
type nom_fonction (type1 arg1,..., typeN argN){ // en-tête
    instructions ; //corps de la fonction
    return (expression);
}
```

- L'**en-tête (prototype)** de la fonction contient :
 - **type** : c'est le type du résultat retourné. Si la fonction n'a pas de résultat à retourner, elle est de type **void**.
 - **nom de la fonction** : Identificateur de la fonction
 - **parenthèses** : on spécifie les **arguments** de la fonction et leurs types. S'il n'y a pas de paramètres, on peut déclarer les paramètres comme **()**.
- Pour fournir un **résultat** en quittant une fonction, on utilise la commande **return**.

Les fonctions

Appel d'une fonction

- L'appel d'une fonction se fait par simple écriture de son nom avec la liste des paramètres : **nom_fonction (para1, para2, ..., paraN) ;**
- Lors de la définition d'une fonction, les paramètres sont appelés **paramètres formels**.
- Lors de l'appel, les paramètres sont appelés **paramètres effectifs**.
- L'**ordre** et les **types** des **paramètres effectifs** doivent correspondre à ceux des **paramètres formels**.
- **Exemple** :

```
main( )
{
    double z;
    int A[5] = {1, 2, 3, 4, 5};
    z=Som(2.5, 7.3);
    AfficheTab(A,5);
}

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

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

Les fonctions

Syntaxe de déclaration

- La déclaration se fait par le **prototype** qui indique le **type de la valeur de retour** et les **types des paramètres** et de la **fonction**.
- **Syntaxe** : `type nom_fonction (type1, ..., typeN) ;`
- Il est **interdit** en C de définir des fonctions à l'intérieur d'autres fonctions.
- La **déclaration des fonctions** est obligatoire si ces **fonctions sont définies après** la fonction principale **main()**.
- **Exemple** : Déclaration et appel d'une fonction :

```
#include<stdio.h>
float ValeurAbsolue(float); //Déclaration de la fonction ValeurAbsolue
main( ) { float x=-5.7,y;
        y= ValeurAbsolue(x); //Appel de la fonction ValeurAbsolue
        printf("La valeur absolue de %f est : %f \n " , x,y); }

float ValeurAbsolue(float a) //Définition de la fonction ValeurAbsolue
{ if (a<0) a=-a;
  return a; }
```

Les fonctions

Variables locales et globales

- On peut manipuler 2 types de variables qui se distinguent par leur **portée**:

- **variable locale** :

- définie à l'intérieur d'une fonction
- n'est connue qu'à l'intérieur de cette fonction.
- Elle est créée à l'appel de la fonction et détruite à la fin de son exécution.

- **variable globale** :

- définie à l'extérieur des fonctions et est définie durant toute l'application;
- Elle peut être utilisée et modifiée n'importe où dans le programme.

```
#include <stdio.h>
int i = 0;    GLOBALE
int main() {
    float i = 0.3; LOCALE
    printf("Bonjour ");
    return 0;
}
int autre() {
    float f = 3.0; LOCALE
}
```

- **Remarques** :

- Les **variables** déclarées au **début** de la fonction principale **main()** ne sont pas des **variables globales**, mais elles sont **locales** à **main()**.
- En général, les **variables globales** sont déclarées immédiatement derrière les instructions **#include** au début du programme.
- Une variable déclarée dans un **bloc d'instructions {...}** est **locale** à ce **bloc**.

Les fonctions

Transmission des paramètres

- Il existe deux modes de transmission de paramètres :
 - Transmission par valeur.
 - Transmission par adresse (par référence).
- ❶ **Transmission par valeur :**
 - La **variable extérieure** (portant le **même nom** que le **paramètre de la fonction**) ne subit **aucune modification**.
 - **Exemple :**

```
#include<stdio.h>
float ValeurAbsolue(float);

main( ) {
    float x=-5.7,y;
    y= ValeurAbsolue(x);
}

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

Les fonctions

Transmission des paramètres

❑ ② Transmission par adresse (par référence) :

- ❑ La fonction modifie une **variable extérieure** à cette fonction (portant le **même nom** que le **paramètre de la fonction**).

❑ Exemple :

```
#include<stdio.h>
float ValeurAbsolue(float*);

main( ) {
    float x=-5.7,y;
    y= ValeurAbsolue( &x);
}

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

L'appel de la fonction ValeurAbsolue provoque :

- le calcul de la valeur absolue de -5.7 et son stockage dans y → **y=5.7**
- la modification de la variable x (*paramètre effectif*) qui va changer de valeur → **x=5.7**

❑ Remarque: Pour effectuer une transmission **par adresse**, on doit :

- ❑ déclarer le **paramètre formel** de type **pointeur**.
- ❑ envoyer l'**adresse (&)** et non la valeur du **paramètre effectif** lors de l'appel.

Les fonctions

Récurtivité

- Une fonction **récursive** est une fonction qui fait **appel à elle-même**.
- Toute fonction **récursive** doit posséder un **cas limite** (cas trivial) qui arrête la récursivité.
- **Exemple** : Une fonction qui calcule la factorielle d'un entier :

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



Les chaînes de caractères

Les chaînes de caractères

Déclaration

- En langage C, il n'existe pas de **type** spécial pour les **chaînes de caractères**. Une **chaîne de caractères** est traitée en C comme un **tableau de caractères**.

F	S	T	H	\0
---	---	---	---	----

- Le dernier élément d'une **chaîne de caractères** vaut le caractère **'\0'**.
- **Syntaxe** : `char NomVariable [Longueur];`
- **Exemple** : `char Adresse[15];`
- **Remarques** :
 - Pour une chaîne de **N caractères**, on a besoin de **N+1** octets en mémoire (le dernier octet est réservé pour le caractère **'\0'**).
 - Le nom d'une chaîne de caractères est le représentant de l'**adresse du 1^{er} caractère** de la chaîne (on peut manipuler les chaînes de caractères en utilisant des **pointeurs**).
 - Il existe plusieurs **fonctions** prédéfinies pour le traitement des chaînes de caractères.

Les chaines de caractères

Initialisation

- On peut **initialiser** une chaîne de caractères à la définition par l'une des trois méthodes suivantes :

1) comme un tableau.

Exemple : `char ch[] = {'e', 'c', 'o', 'l', 'e', '\0'};`

2) par une chaîne constante.

Exemple : `char ch[] = "ecole";`

- ### 3) par un pointeur
- : en attribuant **l'adresse d'une chaîne de caractères constante** à un **pointeur** de type `char`.

Exemple : `char *ch = "ecole";`

Les chaînes de caractères

Manipulation

- Le langage C dispose de plusieurs **bibliothèques** contenant des **fonctions** pour le traitement de chaînes de caractères.
- Les principales bibliothèques et fonctions sont :

- **Bibliothèque `<stdio.h>` :**

Fonction	Rôle	Exemple
<code>printf()</code>	permet d'afficher une chaîne en utilisant le spécificateur de format %s.	<code>char ch[]= " Bonsoir " ; printf(" %s ", ch);</code>
<code>puts(ch)</code>	affiche la chaîne désignée par ch et provoque un retour à la ligne.	<code>char *ch= " Bonsoir " ; puts(ch);</code>
<code>scanf()</code>	permet de saisir une chaîne de caractères en utilisant le spécificateur de format %s	<code>char Nom[15]; printf("entrez votre nom"); scanf(" %s ", &Nom);</code>
<code>gets(ch)</code>	lit la chaîne de caractères désignée par ch .	<code>char phrase[100]; printf("entrez une phrase"); gets(phrase);</code>

Les chaines de caractères

Manipulation

■ Bibliothèque `<string.h>` :

Fonction	Rôle	Exemple
<code>strlen(ch)</code>	fournit la longueur de la chaîne ch sans compter le <code>'\0'</code>	<code>char ch[]= " Test";</code> <code>printf("%d",strlen(ch)); //affiche 4</code>
<code>strcat(ch1, ch2)</code>	ajoute ch2 à la fin de ch1 . Le <code>'\0'</code> de ch1 est écrasé par le 1 ^{er} caractère de ch2	<code>char ch1[20]=" Bonne", ch2=" chance ";</code> <code>strcat(ch1, ch2) ;</code> <code>printf(" %s", ch1); //Bonne chance</code>
<code>strcmp(ch1, ch2)</code>	compare ch1 et ch2 du point de vue lexicographique.	retourne une valeur : nulle si ch1 et ch2 sont identiques, négative si ch1 précède ch2, positive si ch1 suit ch2.
<code>strcpy(ch1, ch2)</code>	copie ch2 dans ch1 y compris le caractère <code>'\0'</code>	<code>char ch[10];</code> <code>strcpy(ch, " Bonjour ");</code> <code>puts(ch); // Bonjour</code>



Allocation dynamique de la mémoire

Allocation dynamique de la mémoire

Introduction

- Dans un programme, si **on ne connaît pas** la **taille des données** au moment de la programmation, on **réserve l'espace maximal prévisible**.
- Exemple :
 - On considère un tableau de `int` : `int tableau[100];`
 - Le tableau occupera alors en réalité $4 * 100 = 400$ octets en mémoire.
 - Même si le tableau est **vide**, il prend **400 octets** !! La place en mémoire est réservée, aucun autre programme n'a le droit d'y toucher.



Gaspillage de la mémoire

Allocation dynamique de la mémoire

Introduction

- Pour éviter le **gaspillage de la mémoire**, il faut **allouer la mémoire** en fonction des données à saisir (par exemple la dimension d'un tableau).



c'est l'**allocation dynamique de la mémoire**.

- En langage C, la bibliothèque **<stdlib.h>** contient deux fonctions pour **allouer** ou **libérer** de la mémoire :
 - ▣ **malloc()** : (**m**emory **a**llocation) : demande au système d'exploitation la permission d'utiliser de la mémoire.
 - ▣ **free()** : (**L**ibérer) : permet d'indiquer à l'OS que l'on n'a plus besoin de la mémoire qu'on avait demandée (un autre programme peut s'en servir).

Allocation dynamique de la mémoire

Fonction malloc()

- **Syntaxe** : `malloc(N)` ; //N:nombre d'octets
- La fonction `malloc` retourne :
 - un **pointeur** de type **char *** pointant vers le **premier octet**.
 - le **pointeur NULL** s'il n'y a pas assez de mémoire libre à allouer.

□ **Exemple** : réserver la mémoire pour un texte de **100 caractères** :

```
char *T;
```

```
T = malloc(100) ;
```

- Cette instruction fournit l'**adresse** d'un bloc de 100 octets et l'affecte à **T**.
- S'il n'y a pas assez de mémoire, **T** obtient la valeur **NULL**.

Allocation dynamique de la mémoire

Fonction free()

- Une fois qu'on a fini d'utiliser la mémoire, on doit la libérer à l'aide de la fonction **free**.
- **Syntaxe** : **free (pointeur) ;**
- **Exemple** : Libérer la mémoire réservé pour un texte de 100 caractères :

```
char *T;  
T = malloc(100) ;  
... //utilisation de la mémoire  
free(T) ;
```
- **Remarques** :
 - Si on appelle pas la fonction **free**, on s'expose à des **fuites de mémoire** (le programme risque au final de prendre beaucoup de mémoire alors qu'il n'a en réalité plus besoin de tout cet espace).
 - Si on ne libère pas explicitement la mémoire à l'aide de **free**, alors elle est libérée **automatiquement** à la fin du programme.

Allocation dynamique de la mémoire

Exemple

- Un programme qui stocke l'âge de tous les amis de l'utilisateur dans un tableau, puis affiche ces âges.

Le problème est que : on connaît pas le nombre d'amis de l'utilisateur ==>
Donc on connaît pas la **taille à donner au tableau**.



L'intérêt de l'**allocation dynamique** est là : on va demander le nombre d'amis à l'utilisateur, puis on fera une **allocation dynamique** pour créer un tableau ayant exactement la taille nécessaire (ni trop petit, ni trop grand).

Allocation dynamique de la mémoire

Exemple

- Un programme qui stocke l'âge de tous les amis de l'utilisateur dans un **tableau**, puis **affiche** l'âge de tous ces amis.

On ne connaît pas le **nombre d'amis de l'utilisateur** ==>
Donc on connaît pas la **taille**
à donner au tableau.



Allocation dynamique

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

int main(){
    int nombreDAmis = 0, i = 0;    int* ageAmis = NULL;

    printf("Combien d'amis avez-vous ? ");
    scanf("%d", &nombreDAmis);

    ageAmis = malloc(nombreDAmis * sizeof(int));
    if (ageAmis == NULL) { exit(0); }

    // On demande l'âge des amis un à un
    for (i = 0 ; i < nombreDAmis ; i++)
    {    printf("Ouel age a l'ami numero %d ? ", i + 1);
        scanf("%d", &ageAmis[i]);    }

    // On affiche les âges stockés un à un
    printf("Vos amis ont les ages suivants :\n");
    for (i = 0 ; i < nombreDAmis ; i++)
    {    printf("%d ans\n", ageAmis[i]);    }

    free(ageAmis);

    return 0;
}
```



Structures

Structures

Définition

- A partir des **types prédéfinis** du langage C (entiers, flottants, caractères, ...), on peut créer de **nouveaux types**, appelés **types composés** : *tableaux, structures, listes, piles, files, arbres, ...*
- Une **structure** est une **entité** qui rassemble des **variables**, qui peuvent être de **types différents**, sous un **seul nom**.
- Les **structures** permettent de **simplifier** l'écriture d'un programme en **regroupant** des données liées entre elles.
-
- **Exemples :**

structure Personne
Nom
Prenom
Date_naissance

structure Voiture
Catégorie
Carburant
NombreChevaux

structure Animal
Nom
Type
Couleur

Structures

Déclaration et initialisation d'une structure

- En C, on définit une structure à l'aide du mot-clé **struct** suivi d'un **identificateur** (nom de la structure) et de la **liste des variables** qu'elle doit contenir.
- Chaque variable de la structure est appelée un **champ** ou un **membre**.

- **Syntaxe :**

```
struct NomStructure {  
    type1 membre-1;  
    type2 membre-2;  
    ...  
    typeN membre-n;  
};
```

- **Exemple :**

```
struct Etudiant {  
    char Nom[20];  
    char Prenom[50];  
    int CNE;  
    float Note;  
};
```

- Définir une structure consiste en fait à définir un **nouveau type**.
- Une structure peut être **initialisée** par une liste **d'expressions constantes** à la manière des initialisations de tableau.
- **Exemple :** `struct Etudiant e = {"Alami", "Kamal", 12345, 15.25};`

Structures

Déclaration d'une variable de type structure

- Il faut distinguer la *déclaration de la structure* de la *déclaration d'une variable de type structure* correspondant à une structure donnée.
- Pour déclarer une **variable** de type **structure**, on distingue 2 cas :
 - Cas 1 : la structure est **déjà définie** : => on déclare la **variable** comme suit :

```
struct NomStructure Var;
```

Exemple : struct Etudiant e1;
 struct Etudiant e2;

- Cas 2 : la structure **n'a pas été déclarée au préalable** : => on déclare la **variable** comme suit :

```
struct NomStructure  
{ type1 membre1;  
  ...  
  typeN membreN;  
} Var;
```

Structures

Accès aux membres d'une structure

- On accède aux champs d'une structure grâce à l'opérateur « . » (*membre de structure*).

□ **Exemple** : soit la structure de données « *Etudiant* » suivante :

```
struct Etudiant
{
    char Nom[20];
    char Prenom[20];
    int CNE;
    float Note;
} e1;
```

- Le CNE de l'étudiant e1 est désigné par l'expression : **e1.CNE;**
 - La Note de l'étudiant e1 est désignée par l'expression : **e1.Note;**
- **NB** : On peut effectuer sur les champs de la structure toutes les opérations valides.

Structures

Affectation et comparaison de structures

□ Affectation de structures

- ▣ On peut affecter une structure à une variable structure de même type, grâce à l'opérateur d'affectation :

```
struct personne
{
    char nom[20];
    char prenom[20];
    int no_employe;
};

struct personne p1 = {"Jean", "Dupond", 7845};
struct personne p2=p1;
```

□ Comparaison de structures

- ▣ Aucune comparaison n'est possible sur les structures (même par les opérateurs == et !=).

Structures

Tableaux de structures

- Une déclaration de **tableau de structures** se fait selon la même syntaxe que la déclaration d'un tableau dont les éléments sont de **type simple**.

- **Exemple :**

```
struct personne
{
    char nom[20];
    char prenom[20];
    int no_employe;
};
struct personne t[100];
```

tableau t :

nom	nom	nom	nom	nom
prenom	prenom	prenom	prenom	prenom	
no_employe	no_employe	no_employe	no_employe	no_employe	

- Pour accéder au **nom** de la personne qui a l'index **i** dans le tableau **t**, on écrira : **t[i].nom;**

Structures

Exemple

- Un programme qui définit la structure **complexe**, composée de deux champs de type double, et qui calcule la norme d'un **nombre complexe** :

```
#include <stdio.h>
#include <math.h>
struct complexe
{
    double reelle;
    double imaginaire;
};
main()
{
    struct complexe z;
    double norme;
    ...
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
    printf("norme de (%f + i %f) = %f \n", z.reelle, z.imaginaire, norme);
}
```

Structures

L'instruction typedef

- L'instruction **typedef** permet d'éviter d'écrire le mot-clé **struct** à chaque définition d'une nouvelle **variable** de type **structure**.
- **typedef** sert à créer un **alias de structure** (un équivalent de structure), qui doit apparaître avant la définition de la structure.
- **Exemple** : une structure « **Coordonnees** » stockant les coordonnées d'un point :

```
typedef struct Coordonnees Coordonnees;  
struct Coordonnees  
{  
    int x;  
    int y;  
};
```

OU

```
typedef struct Coordonnees  
{  
    int x;  
    int y;  
} Coordonnees;
```

- **typedef** : indique que nous allons créer un **alias de structure**.
- **struct Coordonnees** : nom de la structure dont on va créer un alias.
- **Coordonnees** : nom de l'alias de structure (c à d : un équivalent).

Structures

L'instruction typedef

- **Exemple** : En utilisant **typedef**, on aura plus besoin de mettre le mot **struct** à chaque définition de variable de type **Coordonnees** :

```
#include <stdio.h>
typedef struct Coordonnees Coordonnees;
struct Coordonnees
{
    int x;
    int y;
};
int main()
{
    Coordonnees point1, point2 ;
    point1.x=10;
    point1.y=22;
    printf("les coordonees du point1 sont : x=%i y=%i", point1.x, point1.y);
    return 0;
}
```

Structures

Pointeurs vers une structure

- On déclare une **variable de type pointeur** vers une **structure** de la manière suivante :

```
typedef struct personne personne;  
struct personne  
{  
    char nom[30];  
    char prenom[30];  
    int no_employe;  
};  
personne *p;
```

- On peut alors affecter au **pointeur p** des adresses de **struct personne** :

```
int main() {  
    personne pers; //pers est une variable de type struct personne  
    personne *p; //p est un pointeur vers une struct personne  
    p = &pers;  
}
```


Structures

Accès aux éléments d'une structure pointée

- Soit **p** un pointeur vers une structure **personne** :

```
typedef struct personne personne;  
struct personne  
{  
    char nom[30];  
    char prenom[30];  
    int no_employe;  
};  
personne *p;
```

- Pour accéder à un **membre** de la structure pointée par **p**, on utilise la syntaxe suivante : **p -> nom;** (équivalent de : **(*p).nom;**)
- **Exemple** : Soit **p** un pointeur vers la structure **personne**.

pour affecter une valeur au membre **no_employe** de la structure pointée par **p**, on peut écrire :

p -> no_employe = 13456;

Structures

Passage de structures en paramètre de fonctions

- Il est possible de faire passer une **structure** comme **paramètre** d'une fonction.
- **Exemple** : On considère la structure **date** suivante :

```
typedef struct date date;  
struct date  
{  
    int jour, mois, annee;  
};
```

→ une fonction de comparaison de deux dates pourra s'écrire :

```
int cmp_date(date d1, date d2)  
{  
    if (d1.annee > d2.annee)  
        .....;  
    if (d1.annee < d2.annee)  
        .....;  
    ...  
}  
.....  
date d1, d2;  
int c = cmp_date(d1, d2);
```

Structures

Allocation et libération d'espace pour les structures

□ Allocation d'espace : fonction malloc :

- La fonction **malloc** admet un paramètre qui est la taille en octets de l'élément désiré : une **structure**.
- La fonction **malloc** renvoie un pointeur vers l'espace alloué (la structure créée).

□ Exemple :

```
#include <stdlib.h>
typedef struct personne personne;
struct personne
{
    char nom[30];
    char prenom[30];
    int no_employe;
};
personne *p;
p = malloc (sizeof(personne));
```

Structures

Allocation et libération d'espace pour les structures

□ Libération d'espace : fonction free :

- On libère l'espace allouée par **malloc** au moyen de la fonction **free()**.
- La fonction **free()** admet un seul paramètre : un pointeur précédemment rendu par un appel à **malloc**.

□ Exemple :

```
#include <stdlib.h>
typedef struct personne personne;
struct personne
{
    char nom[30];
    char prenom[30];
    int no_employe;
};
personne *p;
p = malloc (sizeof(personne));
... //utilisation de la structure allouée
free(p);
```