

**Université Mouloud Mammeri Tizi-ouzou**  
**Faculté d'électronique et d'informatique**  
**Département d'électronique**



---

**Cours C++ et programmation orientée objet**  
**Les instructions de contrôle, tableaux et fonctions**

---

**Mr. ABAINIA**

**Master  $\mu$ Electronique et instrumentation**



# Instructions conditionnelles



Syntaxe: **if (condition) instruction;**  
**else instruction;**

ou:

```
if (condition)  
{  
    instruction;  
    instruction;  
}  
else  
{  
    instruction;  
    instruction;  
}
```

- ✓ Condition **sans point-virgule**
- ✓ **Résultat** de la condition (**1** ou **0**)
- ✓ Inclure des **opérations math.**



```
#include <iostream>
using namespace std;

int main()
{
    int temperature;
    cin>> temperature;

    if(temperature > 100)
    {
        cout<<"Attention la température est élevée !"<<endl;
        cout<<"Le ventilateur est activé..."<<endl;
    }

    return 0;
}
```



```
#include <iostream>
using namespace std;

int main()
{
    int temperature;
    cin>> temperature;

    float humidity;
    cin>>humidity;

    if(temperature > 27 && humidity < 46)
        cout<<"[Risque de sécheresse] Pompe à eau est activée..."<<endl;
    else
        cout<<"Aucun risque !"<<endl;

    return 0;
}
```



```
#include <iostream>
using namespace std;

int main()
{
    int var = 0;

    cout<<"La valeur de var est : "<<var<<endl;

    if( (var = 3*10+1) > 27)
        cout<<"Ceci est fortement déconseillé !!!!!!"<<endl;

    cout<<"La valeur de var est : "<<var<<endl;

    return 0;
}
```



# Cas de plusieurs tests



**Syntaxe:** **if** (condition) instruction;  
**else if** (condition) instruction;  
**else if** (condition) instruction ;  
**else if** (condition) instruction ;  
**else** instruction ;

**Exemple:**

```
if (var == 1)
    cout<<"Un"<<endl;
else if (var == 7)
    cout<<"Sept"<<endl;
else if (var > 7)
    cout<<"Plus grand"<<endl;
else
    cout<<"Indéfinie"<<endl;
```



# Cas de plusieurs tests (Solution plus optimale)



- ❖ Utiliser la liste de choix (**switch**) au lieu de plusieurs tests successifs.
- ❖ Remplacer les tests d'égalité seulement (**==**).
- ❖ Variable de type primitif (int, float, double, char, bool, etc.).
- ❖ **Else** est remplacé par **default**.
- ❖ Chaque test (ou cas) se termine par l'instruction **break**.



## Syntaxe:

```
switch (variable)
{
    case valeur:
        instruction;
        instruction;
        break;

    ....
    case valeur:
        instruction;
        instruction;
        break;

    default:
        instruction;
        instruction;
        break;
}
```



- ❖ Le bloc d'instructions est délimité par **:** et **break**.
- ❖ Pas besoin d'utiliser les accolades **{ }**.
- ❖ Le cas **default** est exécuté si aucun cas est satisfait.
- ❖ L'**absence** du mot **break** signifie que les cas successifs seront **exécutés successivement**.
- ❖ Le **ET logique** est absent contrairement au **OU logique**.



Exemple:

```
switch (variable)
{
    case 1:
        cout<<"un"<<endl;
        break;
    case 2:
        cout<<"deux"<<endl;
        break;
    case 3:
        cout<<"trois"<<endl;
        break;

    default:
        cout<<"indéfini"<<endl;
        break;
}
```



Exemple:

```
switch (variable)
{
    case 1:
        cout<<"un"<<endl;
    case 2:
        cout<<"deux"<<endl;
        break;
    case 3:
        cout<<"trois"<<endl;
        break;

    default:
        cout<<"indéfini"<<endl;
        break;
}
```

Si la valeur de variable est égale à 1 -> le programme affichera 'un' puis 'deux'



Si la valeur de variable est égale à 1 ou 2 -> le programme affichera 'regroupement de tests'

Exemple:

```
switch (variable)
{
    case 1:
    case 2:
        cout<<"regroupement de tests"<<endl;
        break;
    case 3:
        cout<<"trois"<<endl;
        break;
    default:
        cout<<"indéfini"<<endl;
        break;
}
```



## Avantages de switch au lieu de if else ?

- ✓ Plus facile à lire (organisation)
- ✓ Plus efficace en termes d'accessibilité (tableau de donnée)



## Désavantages de switch ?

- ✓ Pas de réels, strings, etc.
- ✓ Pas d'intervalles
- ✓ Pas de conditions



# Boucles



**Il existe trois types de boucle en C++, dont chacun est doté d'une syntaxe différente.**

- **For**
- **While**
- **Do while**

**Pour résoudre les problèmes de répétition et le parcours des tableaux/listes.**



```
for (partie initialisation ; partie test ; partie incrémentation)
```

```
{  
Instructions;  
}
```

Exécuté avant  
d'entrer à la  
boucle

Exécuté au début  
de chaque  
itération de la  
boucle  
(critère d'arrêt)

Exécuté à la fin de  
chaque itération  
de la boucle

## Exemple commun:

```
for (int i=0 ; i<10 ; i++)  
{  
    cout<<i<<endl;  
}
```

## en langage C

```
int i;  
for (i=0 ; i<10 ; i++)  
{  
    printf("%d\n", i);  
}
```

***Le critère d'arrêt ne dépend pas forcément de la variable itérative***



## Exemple :

```
// consommable est une variable booléenne
for (int element=3 ; consommable ; element*=2)
{
    // faire quelques choses
}
```

**ou**

```
// consommable est une variable booléenne
for (int element=3 ; consommable==true ; element*=2)
{
    // faire quelques choses
}
```



## Exemple d'une application réelle (Arduino):

```
// au lieu de faire servo.write(90);  
for (int angle=1; angle<=90 ; angle++)  
{  
    servo.write(angle);  
}
```

servo = un objet (classe) pour contrôler les servo moteurs  
write = une fonction propre de l'objet (une méthode)

Cette manipulation est très pratique pour faire tourner le servo petit à petit et éviter de produire un choc



```
while (condition)  
{  
    Instructions;  
}
```



Vérifier si la condition  
est vraie avant de  
dérouler la boucle

## Exemple

```
while (distance > 0)  
{  
    Avancer(); // fonction qui fait bouger le robot  
    distance = CalculDistance(); // fonction pour calculer la dist  
}
```



```
do
{
    Instructions;
} while (condition);
```



Exécuter le bloc  
d'instructions une fois,  
puis vérifier la condition  
pour répéter le processus

## Exemple

```
do
{
    position = positionGPS(); // lire la position GPS
    distance = CalculDistance(); // calculer la distance
} while (distance > 0) // tant que on n'a pas atteint la destination
```



## Dans quel cas chaque boucle est utilisée?

### **for:**

- ✓ parcourir une liste d'éléments successifs
- ✓ savoir combien de fois le processus se répète au début

### **while :**

- ❖ répéter un processus itératif dépendant d'une condition
- ❖ ignorer combien de fois le processus se répète

### **do while:**

- exécuter le processus au moins une fois
- initialiser les données d'un processus itératif

## Quelques scénarios alternatifs non-optimaux

```
while (condition)
{
    // bla bla...
}
```



```
for (; condition ;)
{
    // bla bla...
}
```

```
for (init ; condition ; incr )
{
    // bla bla...
}
```



```
init
while(condition)
{
    // bla bla...
    incr 
}
```

```
do
{
    // instructions
} while (condition);
```



```
// instructions
while (condition)
{
    // instructions
}
```



## Boucles infinies

```
while (true)
{
    // bla bla...
}
```

```
for (;;)
{
    // bla bla...
}
```

**Utilisées dans les processus itératifs sans fin comme le cas des microcontrôleurs (eg. fonction `loop` d'Arduino)**

## Boucles imbriquées

```
while (cond)
{
    while (cond)
    {
        // bla bla...
    }
}
```

```
for (init ; cond ; incré)
{
    for (init ; cond ; incré)
    {
        // bla bla...
    }
}
```

```
for (init ; cond ; incré)
{
    while (cond)
    {
        // bla bla...
    }
}
```

***Typiques pour les tableaux multidimensionnels et structures complexes (eg. arbres, graphes, etc.)***



## Instructions spéciales utilisées dans les boucles

**break** : pour sortir immédiatement d'une boucle

**continue** : pour abandonner l'itération en cours

*n'affecte pas les boucles parentes (seulement la courante)*



## Exemple:

```
bool intrusion = false;
while(true)
{
    intrusion = detecterMouvement();
    if(intrusion) break;
    else cout<<"Aucune intrusion "<<endl;
}
```

## Exemple:

```
for (int index=0 ; index <= 5 ; index++)
{
    if(index == 3) continue;
    cout<<index<<endl;
}
```

## Affichage:

Aucune intrusion  
Aucune intrusion  
Aucune intrusion  
Aucune intrusion  
Aucune intrusion  
....

## Affichage:

0  
1  
2  
4  
5

il manque le 3

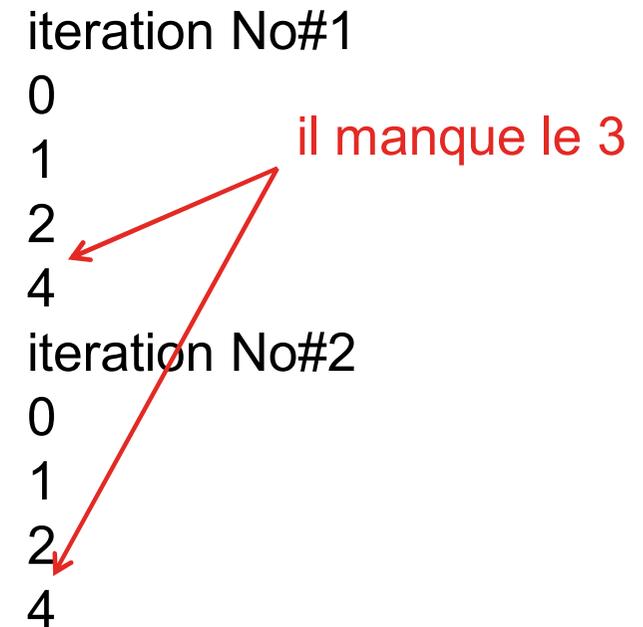
## Exemple:

```
for (int iteration=1 ; iteration <= 2 ; iteration++)  
{  
    cout<<"iteration No #"<<iteration<<endl;  
  
    for (int index=0 ; index <= 4 ; index++)  
    {  
        if(index == 3) continue;  
        cout<<index<<endl;  
    }  
}
```

## Affichage:

```
iteration No#1  
0  
1  
2  
4  
iteration No#2  
0  
1  
2  
4
```

il manque le 3





## Exemple:

```
for (int iteration=1 ; iteration <= 2 ; iteration++)  
{  
    cout<<"iteration No #"<<iteration<<endl;  
  
    for (int index=0 ; index <= 4 ; index++)  
    {  
        if(index == 3) break;  
        cout<<index<<endl;  
    }  
}
```

## Affichage:

iteration No#1

0

1

2

iteration No#2

0

1

2



# Tableaux



- Un tableau est une série d'éléments de **même type** enregistrés dans un espace de **mémoire contigu**
- Le **même identifiant** pour tous les éléments du tableau
- La même déclaration d'une variable en spécifiant la taille
- Par défaut toutes les cases sont initialisées à **zéro**

**Deux types** de tableaux:

Statique  
(taille fixe)

Dynamique  
(taille variable)

## Syntaxe (tableau statique):

```
type nom_tableau [taille];  
type nom_tableau [taille] [taille];  
type nom_tableau [taille] [taille] [taille];  
type nom_tableau [taille] [taille] [taille] [taille];
```

## Exemple :

```
char nom_tableau [50];  
short nom_tableau [4] [10];  
double nom_tableau [30] [5] [11];  
int nom_tableau [3] [5] [9] [2];
```



## Initialisation explicite du contenu du tableau:

```
short nom_tableau [4] = {10, 5, 3, 50};
```

```
char nom_tableau [7] = {'b', 'o', 'n', 'j', 'o', 'u', 'r'};
```

```
char nom_tableau [7] = "bonjour";
```



## Initialisation explicite du contenu du tableau:

```
short nom_tableau [ ] = {10, 5, 3, 50};
```

```
char nom_tableau [ ] = "bonjour";
```

**Le compilateur déduit la taille du tableau en comptant le nombre d'éléments de la liste**

**Lorsque on ne sait pas exactement la taille de la liste (*notamment la chaîne de caractères*)**



## Initialisation explicite du contenu du tableau:

~~short nom\_tableau [ ] [ ] = { {5, 3}, {2, 54}, {15, 13} };~~

(Fausse)

Spécifier la taille des  
autres dimensions



short nom\_tableau [ ] [2] = { {5, 3}, {2, 54}, {15, 13} };

(Correcte)

## Accès à une case spécifique du tableau:

`nom_tableau [indice]`

`nom_tableau [indice_ligne][indice_column]`

`nom_tableau [indice_1][indice_2][indice_3]`

### Exemple:

`nom_tableau[0]`

`nom_tableau[0][7]`

`nom_tableau[8][5][2]`

### Exemple réel:

`nom_tableau[0] = 13.6;`

`cout<<nom_tableau[0]<<endl;`

`cin>>nom_tableau[0];`

## Accès à tous les éléments du tableau:

nom\_tableau [0]  
nom\_tableau [1]  
nom\_tableau [2]  
nom\_tableau [3]  
....



```
for(int indice=0; indice < taille; indice++)  
{  
    nom_tableau [indice];  
}
```



# Pointeurs



- Variable comme les variables standards
- Contient l'adresse physique d'une autre variable
- Doit être déclarée avant d'être utilisée

**type** \* **nom**;

n'importe quel  
type valide

étoile de  
multiplication

**Les pointeurs sont indispensables dans les grands projets**



- **Variable dynamique (occupation/libération d'espace)**
- **Tableau dynamique avec une taille variable**
- **Opérations arithmétiques**
- **Pointeur de pointeur**



- Utiliser l'opérateur **new** pour allouer un espace mémoire
- Utiliser l'opérateur **delete** pour libérer l'espace alloué
- Utiliser la valeur **NULL** pour initialiser le pointeur à zéro

## Exemple 1:

```
int temperature = 30;  
int *ptr_value = &temperature;
```

```
cout<<temperature<<endl;           30  
cout<<ptr_value<<endl;             0x28ff44;  
cout<<*ptr_value<<endl;           30
```



**Pointeurs sont déconseillés dans les simples programmes comme celui-ci**



## Exemple 2:

```
int *ptr_temperature = new int;  
ptr_temperature = 30;  
  
cout<< ptr_temperature <<endl;
```

## Exemple 3:

```
int *tab_dyn = new int [10];  
tab_dyn[0] = 10;  
...  
tab_dyn[9] = 52;  
...  
delete tab_dyn;
```

**Pointeurs  
sont  
déconseillés  
dans les  
simples  
programmes  
comme  
l'exemple 2**



## Fréquentes erreurs commises (tableaux dynamiques) ?

- Utilisation d'une variable pointeur **sans allocation (new)**
- Réutilisation d'un pointeur **sans réallocation** après **suppression**
- Allocation d'une **taille inférieure** à la taille requise
- Réutilisation (réallocation) d'un pointeur **sans suppression**



## Astuces de sécurité

- Initialiser le pointeur à **NULL** après suppression

```
delete ptr;  
ptr = NULL;
```

- Ajouter des crochets **[ ]** pour supprimer un tableau

```
delete [ ] ptr;  
ptr = NULL;
```

- Tester si le pointeur **ne pointe pas** sur le vide avant de le supprimer

```
if(ptr != NULL)  
{  
    delete ptr;  
    ptr = NULL;  
}
```



## Exemple 1:

```
int *tab_dyn;  
...  
...  
for (int index=0; index<10; index++)  
    cin>>tab_dyn[index];
```

**Erreur (bug)**

```
int *tab_dyn;  
...  
tab_dyn = new int [10];  
for (int index=0; index<10; index++)  
    cin>>tab_dyn[index];
```

**Solution**

## Exemple 2:

```
int *tab_dyn = new int[10];
```

```
for (int index=0; index<10; index++)  
    cin>>tab_dyn[index];
```

```
if(tab_dyn != NULL)  
{  
    delete [ ] tab_dyn;  
    tab_dyn = NULL;  
}
```

```
for (int index=0; index<20; index++)  
    cin>>tab_dyn[index];
```

**Erreur (bug)**

```
int *tab_dyn = new int[10];

for (int index=0; index<10; index++)
    cin>>tab_dyn[index];

if(tab_dyn != NULL)
{
    delete [ ] tab_dyn;
    tab_dyn = NULL;
}

tab_dyn = new int[20];
for (int index=0; index<20; index++)
    cin>>tab_dyn[index];
```

## Solution



### Exemple 3:

```
int *tab_dyn = new int[10];  
...  
...  
for (int index=0; index<100; index++)  
    cin>>tab_dyn[index];
```

**Erreur (bug)**

```
int *tab_dyn = new int [100];  
...  
...  
for (int index=0; index<100; index++)  
    cin>>tab_dyn[index];
```

**Solution**

## Exemple 4:

```
// maStruct est une structure de donnée  
// de taille 10Mb (Mega byte)
```

```
maStruct *tab_dyn = new int[50];  
for (int index=0; index<50; index++)  
{  
    // effectuer des opérations  
}
```

```
tab_dyn = new int[100];  
for (int index=0; index<100; index++)  
{  
    // effectuer des opérations  
}
```

**500Mb alloués**

**+**

**1Gb alloués**

**Total =  
1,5Gb alloués**



```
// maStruct est de taille 10Mb
```

```
maStruct *tab_dyn = new int[50];  
for (int index=0; index<50; index++)  
{  
    // effectuer des opérations  
}  
  
if(tab_dyn != NULL)  
{  
    delete [ ] tab_dyn;  
    tab_dyn = NULL;  
}  
  
tab_dyn = new int[100];  
for (int index=0; index<100; index++)  
{  
    // effectuer des opérations  
}
```

500Mb alloués



500Mb libérés



100Mb alloués



**Total =  
100Mb alloués**

## Pointeur de pointeur (typiquement une matrice)

```
int ** matrice;
```

```
matrice = new * int [300];
```

```
for(int ligne=0; ligne < 300; ligne++)  
{  
    matrice[ligne] = new int [100];  
}
```

**initialisation**

---

```
for(int ligne=0; ligne < 300; ligne++)  
{  
    delete [ ] matrice[ligne];  
    matrice[ligne] = NULL;  
}  
delete [ ] matrice;  
matrice = NULL;
```

**suppression**



## Autres méthodes d'allocation de mémoire

**malloc (taille) : allocation sans initialisation**

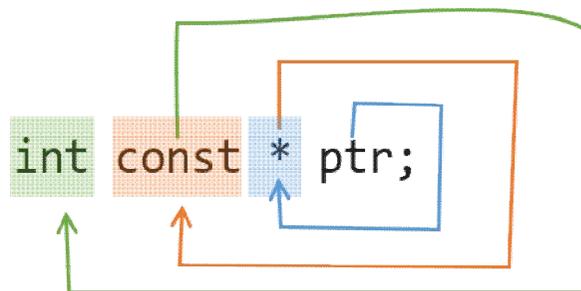
**calloc (taille) : allocation + initialisation à 0**

**Vieilles méthodes utilisées en C (déconseillées)**

## Différentes déclarations des pointeurs ?

`int *`  
`const int *`  
`int const *`  
`int * const`  
`int const * const`  
Etc.

Il faut suivre Clockwise/Spiral Rule pour interpréter la déclaration



`ptr` is a `pointer` to `const int`

Figure tirée du site *stackoverflow*



**ils existent des pointeurs intelligents qui offrent plus de  
sécurité**  
**ex.: unique\_ptr, shared\_ptr, auto\_ptr, etc.**

# Références



- alias d'une variable qui **partage** la **même adresse**
- ne peut pas être initialisée avec **NULL**
- **ne peut pas** changer l'objet de référence
- **doit être initialisée** au moment de la déclaration

**type** &nom = nom\_variable;



## Exemple:

```
int temperature = 30;  
int &temp = temperature;
```

```
cout<< temperature <<endl; // valeur affichée est 30  
cout<< temp <<endl; // valeur affichée est 30
```

```
temp += 10;
```

```
cout<< temperature <<endl; // valeur affichée est 40  
cout<< temp <<endl; // valeur affichée est 40
```

```
temperature += 10;
```

```
cout<< temperature <<endl; // valeur affichée est 50  
cout<< temp <<endl; // valeur affichée est 50
```



## Usage des références vs usage des pointeurs

- **Pointeurs sont très utiles lors de la manipulation des tableaux et des objets volumineux (mémoire dynamique)**
- **Références sont très utiles dans les fonctions (paramètres et types de retour)**



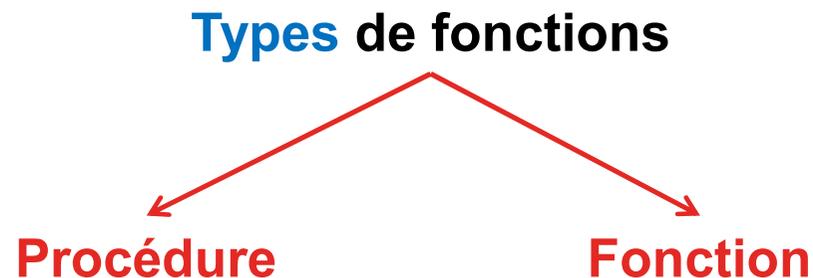
# Fonctions



Une fonction est un programme contenant un **ensemble d'instructions**.

**But = découpage** du programme en petits éléments **réutilisables**

**But = réduction** des processus (bloc d'instructions) **redondants**



## Procédure vs Fonction ?

**Procédure** = sans type et ne renvoie rien (type = void)

**Fonction** = contient un type et renvoie une valeur/tableau/objet

```
type nom_fonction( )  
{  
    // instructions  
  
    return valeur;  
}
```

```
type nom_fonction( paramètres )  
{  
    // instructions  
  
    return valeur;  
}
```



```
void affichage()  
{  
    for(int compteur=0; compteur < 100; compteur++)  
    {  
        cout<<"punition à l'école primaire..."<<endl;  
    }  
}
```

```
void affichage()  
{  
    int compteur=0;  
    while(compteur < 100)  
    {  
        cout<<"punition à l'école primaire..."<<endl;  
        compteur++;  
    }  
}
```

## Exemple 1

Affichage  
de la  
phrase  
100 fois



```
void affichage(int cpt)
{
    for(int compteur = 0; compteur < cpt; compteur++)
    {
        cout<<"punition à l'école primaire..."<<endl;
    }
}
```

```
void affichage(int cpt)
{
    int compteur = 0;
    while(compteur < cpt)
    {
        cout<<"punition à l'école primaire..."<<endl;
        compteur++;
    }
}
```

## Exemple 2

Affichage  
de la  
phrase  
plusieurs  
fois

```
int add(int a , int b)
{
    int resultat = a + b;

    return resultat;
}
```

```
int add(int a , int b)
{
    return (a + b);
}
```

## Exemple 3

**Fonction  
pour  
calculer  
la somme**



## Particularités des fonctions? (1)

- On peut déclarer **plusieurs** fonctions dont le **même nom**.
- On peut déclarer la **même** fonction avec **différents paramètres**.
- On peut déclarer la **même** fonction avec **différents types de retours**.
- On peut **renvoyer** un type **différent** de types des **paramètres**.
- On peut **mélanger** différents **types** de **paramètres**.

```
int add(int a , int b)
{
    ...
}
```

```
float add(int a , short b , float b)
{
    ...
}
```

```
int add(int a , int b , int c)
{
    ...
}
```

```
float add(int a , int b)
{
    ...
}
```

```
float add(float a , float b)
{
    ...
}
```



## Particularités des fonctions? (2)

- On **doit définir** la fonction **avant** de l'utiliser.
- On peut **passer** une **fonction** comme **paramètre** de la **même** fonction.
- On peut **passer** une **fonction** comme **paramètre** d'une **autre** fonction.
- On **ne doit pas définir** la fonction à **l'intérieur** d'une autre.
- On peut passer des **pointeurs** et des **références** comme **paramètres**.

**// définition de la fonction add**

```
int add(int a , int b)
{
    return (a + b);
}
```

**// appel de la fonction add**

```
int res = add(3 , 9); // la variable res contiendra 12 (3+9)
```

```
int res_2 = add(3 , res); // la variable res_2 contiendra 15 (3+12)
```

```
int res_3 = add(3 , add(3 , 9) ); // la variable res_3 contiendra 15 (3+12)
```



```
int res_3 = add(3 , add(3 , 9) ); // la variable res_3 contiendra 15 (3+12)
```

**add(3 , 9)**

**add(3 , 9)**

Exécuté en premier lieu et renvoie un résultat



add(3 , **12** )

add(3 , 9) est **remplacée** par le résultat **12**



Exécuter la fonction add(3 , 12 ) et renvoyer le résultat



Enregistrer le résultat dans la variable res\_3

## Les limites ?

❖ On ne peut renvoyer qu'une seule valeur.

```
return valeur ou variable
```

❖ On ne peut pas utiliser la fonction avant de la définir.

```
add(13 , 20);
```

```
int add(int a , int b)  
{  
    return (a + b);  
}
```

## Les solutions ?

❖ **On ne peut renvoyer qu'une seule valeur.**

On utilise les types composés pour renvoyer plusieurs valeurs.

❖ **On ne peut pas utiliser la fonction avant de la définir.**

On déclare un prototype de la fonction au début et on la définit après.

## Particularités des fonctions? (3)

➤ La modification des valeurs des paramètres n'est pas prise en considération après l'exécution de la fonction.

solution



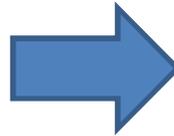
➤ Paramètres en pointeurs ou références.

```
void fonc(int a)
{
    a++;
    cout<<a;
}

int variable = 10;
fonc(variable); // affiche11
cout<<variable; // affiche10
```

```
void fonc(int a)
{
    a++;
    cout<<a;
}
```

```
int variable = 10;
fonc(variable); // affiche11
cout<<variable; // affiche10
```

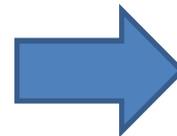


```
void fonc(int & a)
{
    a++;
    cout<<a;
}
```

```
int variable = 10;
fonc(variable); // affiche11
cout<<variable; // affiche11
```

```
void fonc(int * a)
{
    *a++; // bug
    cout<<*a;
}
```

```
int variable = 10;
fonc(variable); // erreur
cout<<variable; // affiche11
```



```
void fonc(int * a)
{
    (*a)++;
    cout<<*a;
}
```

```
int variable = 10;
fonc(&variable); // affiche11
cout<<variable; // affiche11
```



# Prototypes de fonctions



- ❖ On définit **seulement l'entête** de la fonction **sans implémentation**.
- ❖ On met le **point virgule** après la **déclaration** de l'entête.
- ❖ On **implémente** la fonction indépendamment.
- ❖ Le **prototype et l'entête** doivent avoir les **mêmes types et paramètres**.

On ne met pas les noms  
des paramètres

```
int add(int a , int b);
```

```
...
```

```
int add(int a , int b)
```

```
{
```

```
    ...
```

```
}
```

Astuce



```
int add(int , int );
```

```
...
```

```
int add(int a , int b)
```

```
{
```

```
    ...
```

```
}
```



# Fonctions récursives

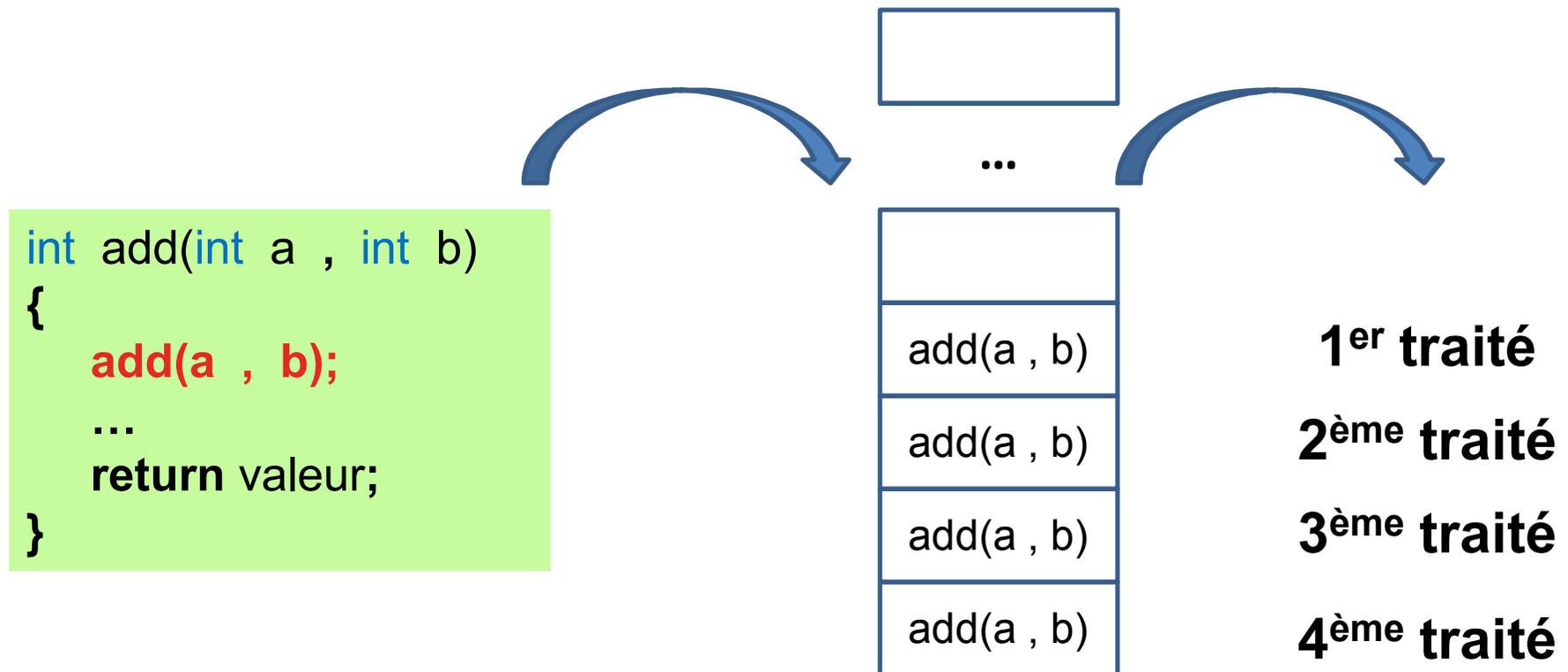
Une **fonction récursive** est une fonction qui fait **appel à elle même**

```
int add(int a , int b)
{
    add(a , b);
    ...
    return valeur;
}
```

### Inconvénients:

- ❖ Elle doit être conditionnée pour sortir de la boucle infinie
- ❖ Risque de remplir la pile (mémoire)

## Dernier entré premier sorti (Last-in First-out) LiFo



```
void affichage(int N)
{
    N--;

    if(N > 0) affichage(N);

    cout<<"Numéro " <<N<<endl;
}
```

```
affichage(5);
```

```
Numéro 0
Numéro 1
Numéro 2
Numéro 3
Numéro 4
Numéro 5
```

## Exemple 1

### Fonction récursive d'affichage d'une liste de nombres ordonnés



```
void affichage(int N)
{
    N--;

    if(N >= 0) affichage(N);

    cout<<"Numéro " <<N<<endl;
}
```

```
affichage(5);
```

```
Numéro -1
Numéro 0
Numéro 1
Numéro 2
Numéro 3
Numéro 4
Numéro 5
```

## Exemple 2

### Fonction récursive d'affichage d'une liste de nombres ordonnés

```
int factorielle(int N)
{
    int fact = N;

    if(N > 0)
    {
        fact *= factorielle(N-1);
    }

    return fact;
}
```

## Exemple 3

Fonction pour  
calculer la  
factorielle d'un  
nombre entier

$$N! = N * (N-1) * (N-2) * (N-3) * (N-4) * \dots * 1$$



# Prochain cours

## Structures et Programmation Orientée Objet