

# Présentation du langage C++ (suite)

IUT Lyon 1

Yosra ZGUIRA

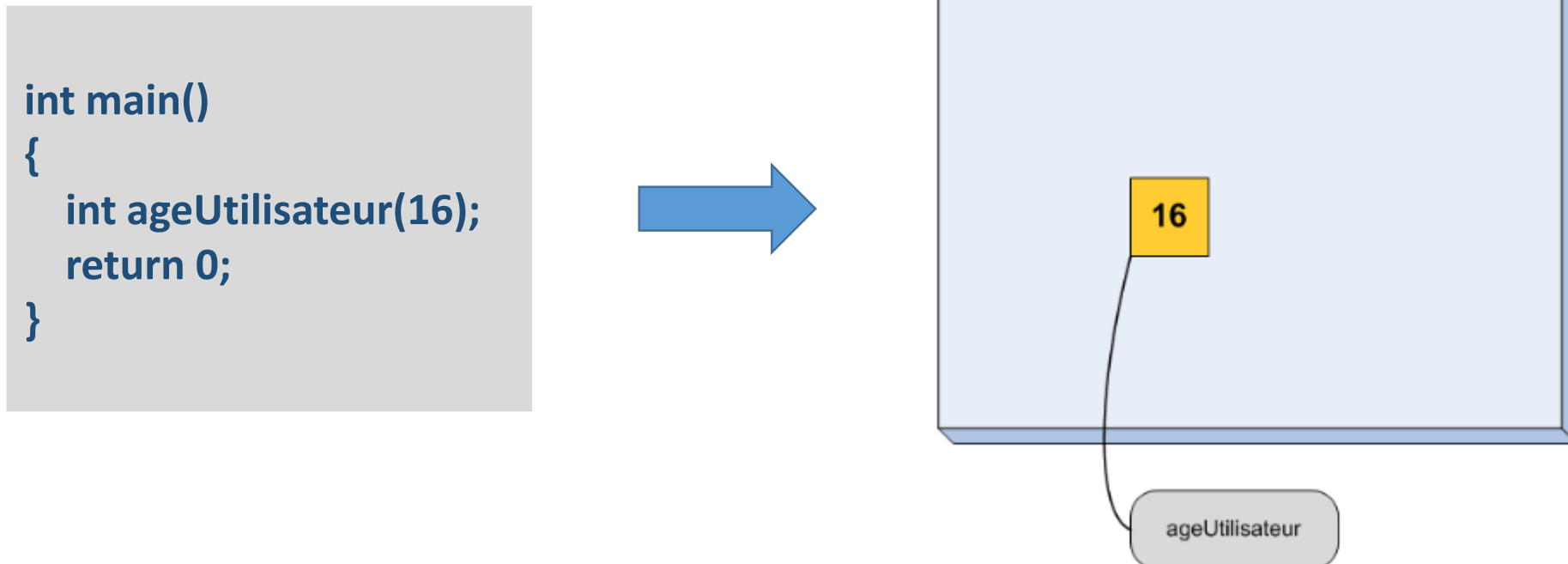
[yosra.zguira@insa-lyon.fr](mailto:yosra.zguira@insa-lyon.fr)

2016 - 2017

## **Les pointeurs en C++**

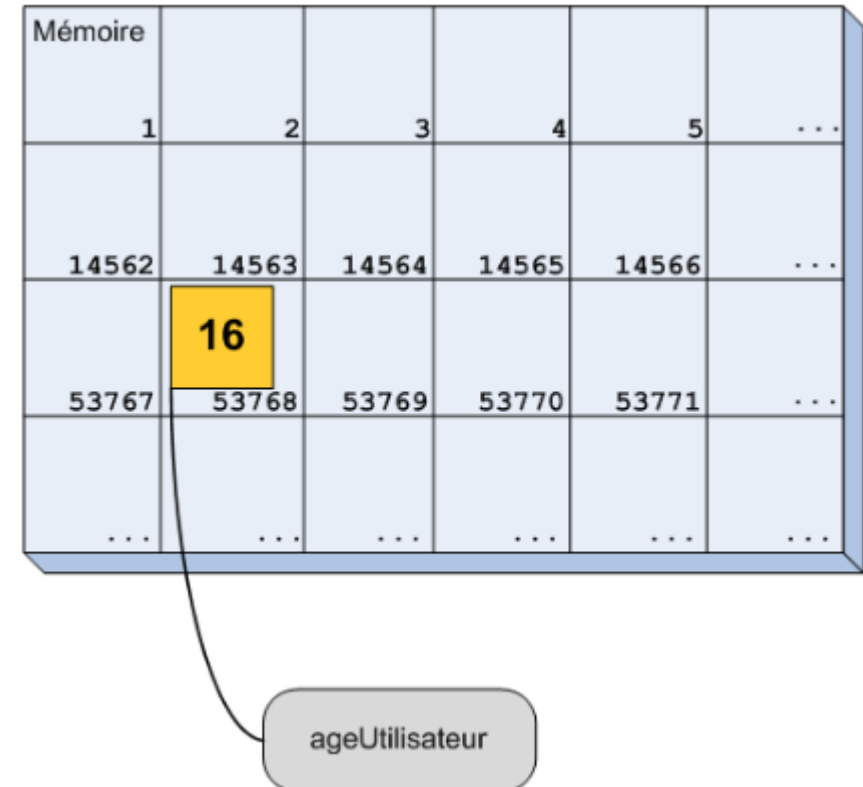
## Rappel (1/3)

- Lors de la déclaration d'une variable, l'ordinateur alloue un emplacement dans sa mémoire où se trouve sa valeur.



## Rappel (2/3)

- La mémoire d'un ordinateur est réellement constituée de « cases ».
- Il y a plusieurs milliards sur un ordinateur récent !
- Chaque case possède un numéro unique, son **adresse**.
- La figure montre toutes les cases de la mémoire avec leurs adresses.
- Notre exemple utilise une seule de ces cases, la **53768**, pour y stocker sa variable.
- **Chaque variable possède une seule adresse et que chaque adresse correspond à une seule variable.**



## Rappel (3/3)

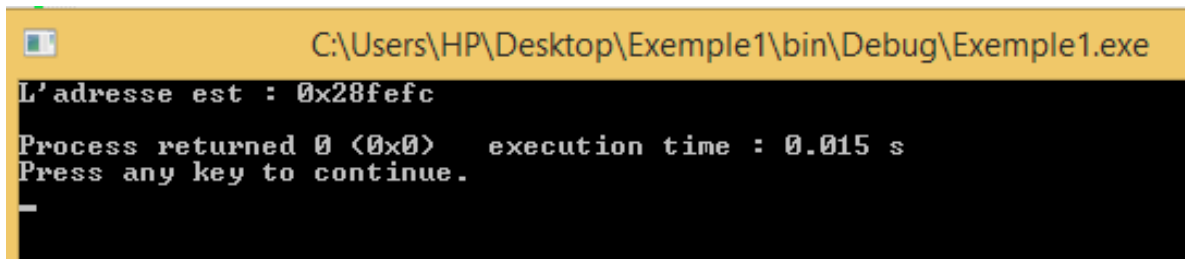
- **L'adresse** est donc un deuxième moyen d'accéder à une variable.
- On peut atteindre la case jaune du schéma par deux chemins différents :
  - ✓ passer par son **nom**: *ageUtilisateur*
  - ✓ accéder à la variable grâce à son **adresse** (son numéro de case). Pour notre exemple, on pourrait dire à l'ordinateur « Affiche moi le contenu de l'adresse 53768 »

# Affichage de l'adresse

- En C++, le symbole pour obtenir l'adresse d'une variable est l'esperluette (&).

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

int main()
{
    int ageUtilisateur(16);
    cout << "L'adresse est : " << &ageUtilisateur << endl; //Affichage de l'adresse de la variable
    return 0;
}
```



Cette adresse est un nombre. Elle est écrite en hexadécimal (en base 16).

# Déclaration d'un pointeur (1/3)

- **Un pointeur est une variable qui contient l'adresse d'une autre variable.**
- Pour déclarer un pointeur il faut, comme pour les variables, deux choses :
  - ✓ un type ;
  - ✓ un nom.
- **Le type d'un pointeur:** il faut indiquer quel est le type de variable dont on veut stocker l'adresse et ajouter une étoile (\*).

```
int *pointeur;
```

- Cela déclare un **pointeur** qui peut contenir l'adresse d'une variable de type **int**.

## Déclaration d'un pointeur (2/3)



- On peut également écrire:

```
int* pointeur;
```

- Cette notation a un léger inconvénient, c'est qu'elle ne permet pas de déclarer plusieurs pointeurs sur la même ligne, comme ceci :

```
int* pointeur1, pointeur2, pointeur3;
```

➔ Seul ***pointeur1*** sera un pointeur, les deux autres variables seront des entiers tout à fait standard.



# Déclaration d'un pointeur (3/3)

- **Il faut toujours déclarer les pointeurs en les initialisant à l'adresse 0.**
- L'adresse **0** n'existe pas car la première case de la mémoire avait l'adresse 1.

➔ Car lors de la déclaration d'un pointeur de cette manière:

```
int *pointeur;
```

aucune adresse est connue. C'est une situation très dangereuse.

- ➔ Si vous essayez d'utiliser le pointeur, vous ne savez pas quelle case de la mémoire vous manipulez. Cela peut être n'importe quelle case, par exemple celle qui contient votre mot de passe Windows ...
- ➔ Il ne faut donc *jamais* déclarer un pointeur sans lui donner d'adresse.

```
int *pointeur(0);  
double *pointeurA(0);  
unsigned int *pointeurB(0);  
string *pointeurC(0);  
vector<int> *pointeurD(0);  
int const *pointeurE(0);
```

# Stockage d'une adresse (1/2)

```
int main()
{
    int ageUtilisateur(16);    //Une variable de type int
    int *ptr(0);               //Un pointeur pouvant contenir l'adresse d'un nombre entier

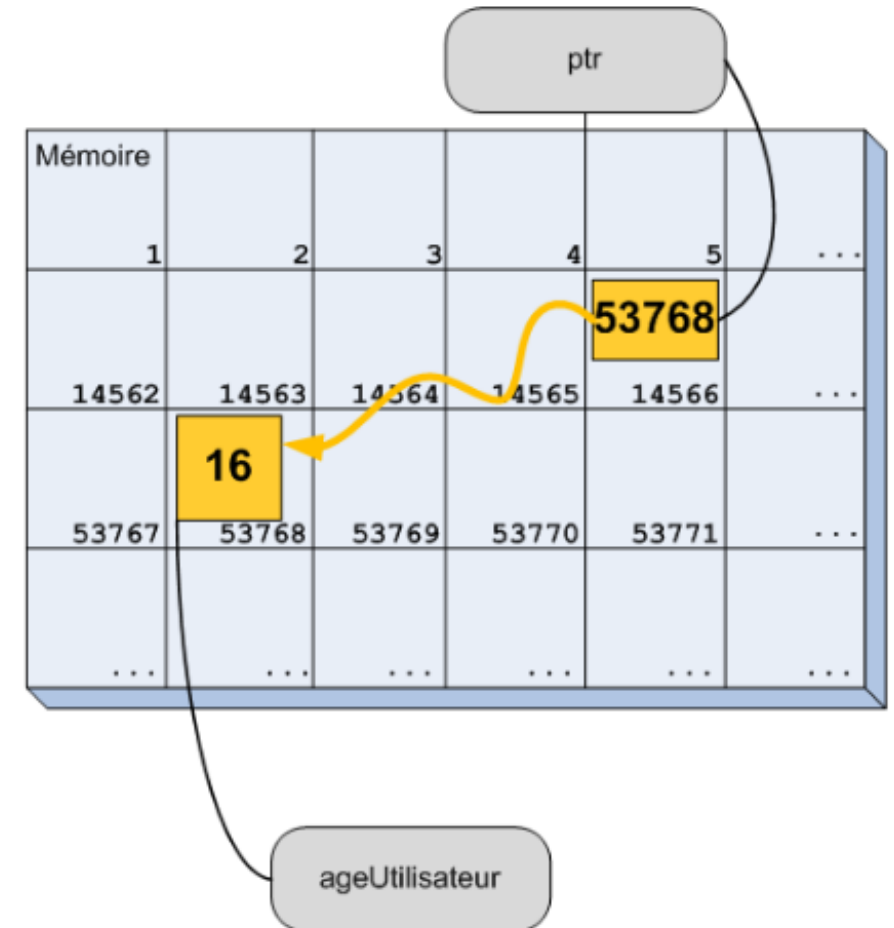
    ptr = &ageUtilisateur;    //On met l'adresse de 'ageUtilisateur' dans le pointeur 'ptr'

    return 0;
}
```

- La ligne **ptr = &ageUtilisateur;** écrit l'adresse de la variable **ageUtilisateur** dans le pointeur **ptr**.  
→ On dit alors que le pointeur **ptr** pointe sur **ageUtilisateur**.

## Stockage d'une adresse (2/2)

- La mémoire contient sa grille de cases et la variable *ageUtilisateur* dans la case n°**53768**.
- Dans la case mémoire n°**14566**, il y a une variable nommée *ptr* qui a pour valeur l'adresse **53768**, qui est l'adresse de la variable *ageUtilisateur*.



# Affichage d'une adresse

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

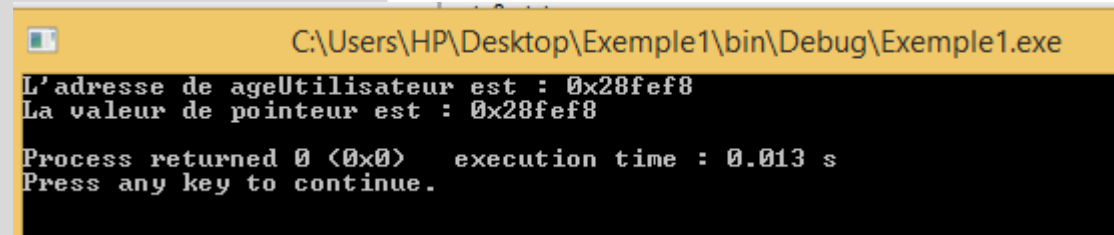
int main()
{
    int ageUtilisateur(16);
    int *ptr(0);

    ptr = &ageUtilisateur;

    cout << "L'adresse de ageUtilisateur est : " << &ageUtilisateur << endl;
    cout << "La valeur de pointeur est : " << ptr << endl;

    return 0;
}
```

La valeur du pointeur est donc bien l'adresse de la variable pointée.



```
C:\Users\HP\Desktop\Exemple1\bin\Debug\Exemple1.exe
L'adresse de ageUtilisateur est : 0x28fef8
La valeur de pointeur est : 0x28fef8

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

# Accès à la valeur pointée (1/2)

- Les pointeurs permettent d'accéder à une variable sans passer par son nom.
- Il faut utiliser l'étoile (\*) sur le pointeur pour afficher la valeur de la variable pointée.

```
int main()
{
    int ageUtilisateur(16);
    int *ptr(0);

    ptr= &ageUtilisateur;

    cout << "La valeur est : " << *ptr << endl;

    return 0;
}
```

## Accès à la valeur pointée (2/2)

- En faisant `cout << *ptr`, le programme effectue les étapes suivantes :
  1. Aller dans la case mémoire nommée `ptr` ;
  2. Lire la valeur enregistrée ;
  3. « Suivre la flèche » pour aller à l'adresse pointée ;
  4. Lire la valeur stockée dans la case ;
  5. Afficher cette valeur : qui sera 16.



- Pour une variable **int nombre** :
  - ✓ **nombre** permet d'accéder à la **valeur** de la variable ;
  - ✓ **&nombre** permet d'accéder à **l'adresse** de la variable.
- Sur un pointeur **int \*pointeur** :
  - ✓ **pointeur** permet d'accéder à la **valeur du pointeur**, c'est-à-dire à l'adresse de la variable pointée ;
  - ✓ **\*pointeur** permet d'accéder à la **valeur de la variable pointée**.

## **L'allocation dynamique en C++**



# Allocation d'un espace mémoire (1/3)

## ▪ La gestion automatique de la mémoire

Lors de la déclaration d'une variable, le programme effectue deux étapes :

- ✓ Il demande à l'ordinateur de lui fournir une zone dans la mémoire.  
→ **allocation** de la mémoire.

- ✓ Il remplit cette case avec la valeur fournie.  
→ **initialisation** de la variable.

- En C++, pour demander manuellement une case dans la mémoire:

  - utiliser l'opérateur **new**.

  - **new** demande une case à l'ordinateur et renvoie un pointeur pointant vers cette case.

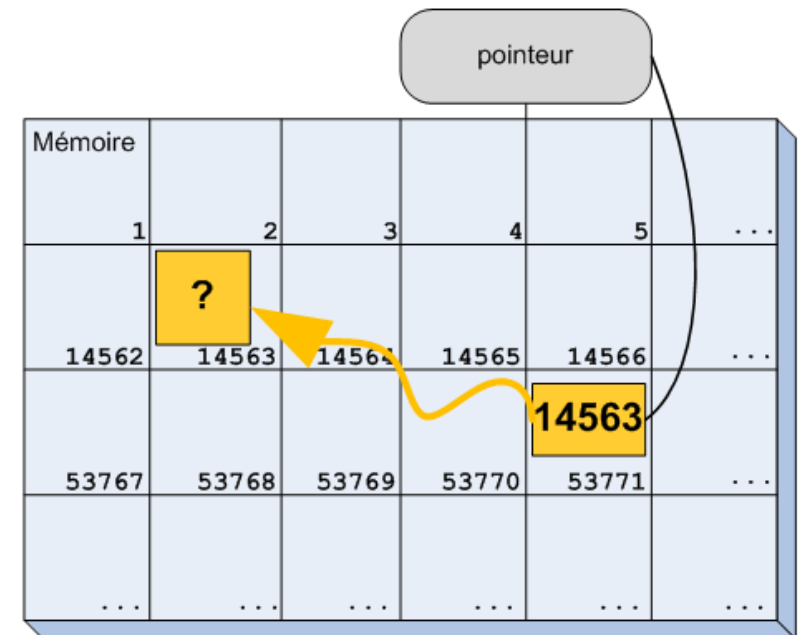
- Il suffit de faire suivre le mot clé **new** du type de la variable à allouer.

```
int *pointeur(0);  
  
pointeur = new int;
```

## Allocation d'un espace mémoire (2/3)

- La deuxième ligne demande une case mémoire pouvant stocker un entier et l'adresse de cette case est stockée dans le pointeur.
- Il y a deux cases mémoires utilisées:
  - ✓ la case **14563** qui contient une variable de type *int* non initialisée ;
  - ✓ la case **53771** qui contient un pointeur pointant sur la variable.

```
int *pointeur(0);  
  
pointeur = new int;
```



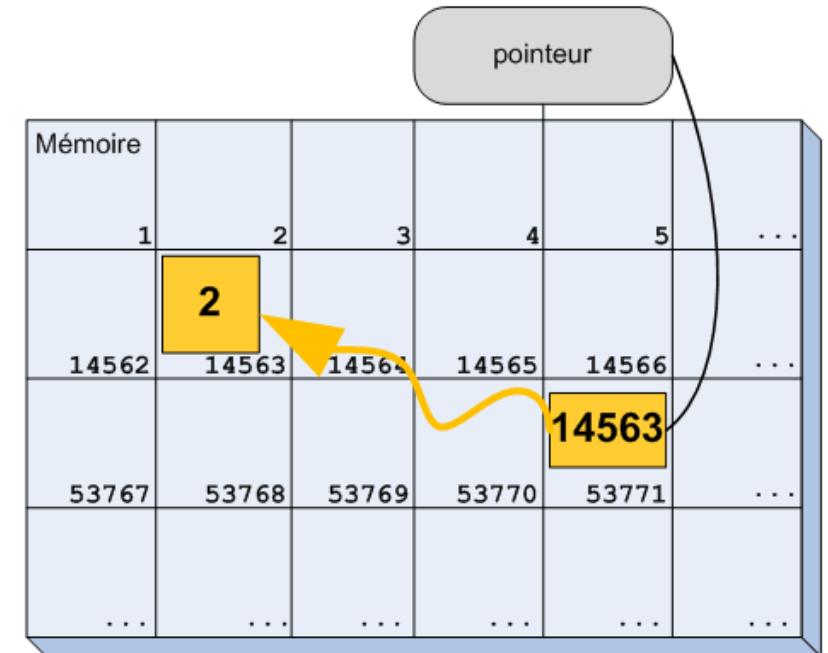
# Allocation d'un espace mémoire (3/3)

- Lorsque la variable est allouée manuellement  
→ il faut y accéder par le **pointeur**.

```
int *pointeur(0);
```

```
pointeur = new int;
```

```
*pointeur = 2; //On accède à la case mémoire pour en  
               modifier la valeur
```



- La case **14563** est donc remplie.

# Libération de la mémoire (1/2)

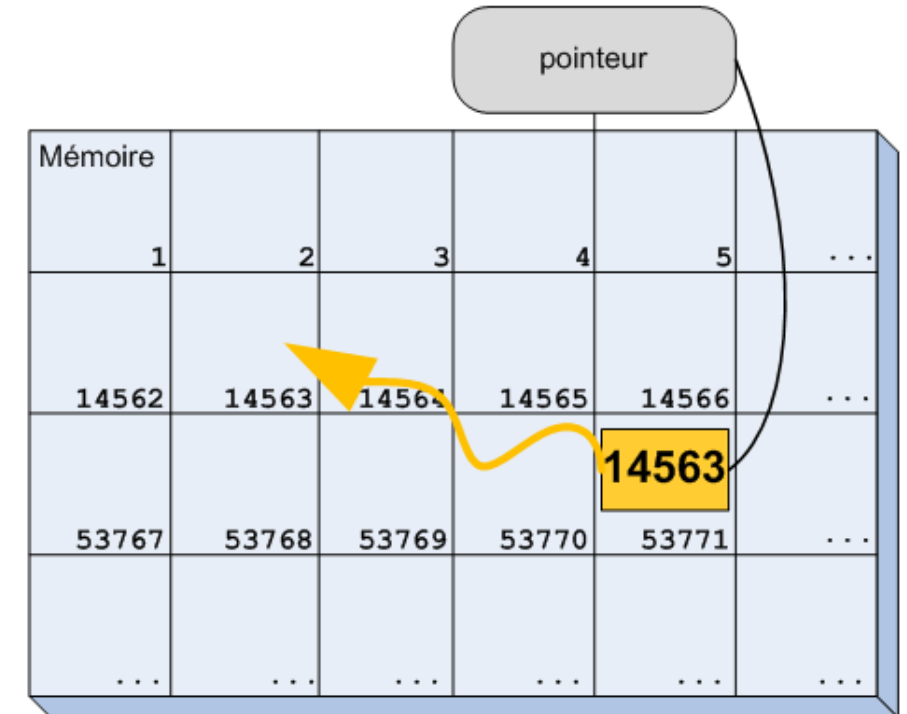
- En C++, la restitution de la mémoire se fait via l'opérateur **delete**.

```
int *pointeur(0);  
  
pointeur = new int;  
  
delete pointeur;      //On libère la case mémoire
```

- La case est rendue à l'ordinateur qui pourra la réutiliser.
- **Mais**, le pointeur existe toujours et il pointe toujours sur la case !

## Libération de la mémoire (2/2)

- La flèche pointe vers une case qui n'appartient au pointeur.
  - Si la case est réutilisé par un autre programme, le pointeur risque de modifier son contenu.
- ➔ Il est donc essentiel de supprimer cette « flèche » en mettant le pointeur à l'**adresse 0**.



```
int *pointeur(0);  
pointeur = new int;
```

```
delete pointeur;    //libération de la case mémoire  
pointeur = 0;       //le pointeur ne pointe plus vers rien
```

# new[] et delete[]

- Les opérateurs **new[]** et **delete[]** sont utilisés pour allouer et restituer la mémoire pour les types tableaux.
- La syntaxe de **delete[]** est la même que celle de **delete**.
- L'opérateur **new[]** nécessite de donner la taille du tableau à allouer.

```
int *Tableau=new int[10000]; // créer un tableau de 10000 entiers
```

- La destruction du tableau:

```
delete[] Tableau;
```

## **Les références en C++**

# Présentation des références

- Une **référence** est un alias (synonyme) d'une autre variable.
  - ➔ C'est à dire qu'utiliser la variable, ou une référence à cette variable est équivalent.
  - ➔ Ce qui signifie que l'on peut modifier le contenu de la variable en utilisant une référence.
- Les références ressemblent beaucoup aux pointeurs.
- Elles ont été créées pour simplifier l'utilisation des pointeurs.



# Les références à l'intérieur d'une fonction (1/3)

- Pour la création d'une référence, il faut utiliser le symbole **&** dans la déclaration :

```
type &identificateur = variable;
```

ou

```
type &identificateur (variable);
```

## Les références à l'intérieur d'une fonction (2/3)

- Pour la création d'une référence, il faut utiliser le symbole **&** dans la déclaration :

```
type &identificateur = variable;
```

ou

```
type &identificateur (variable);
```

Dans une **déclaration**, le symbole **&** signifie "*Je veux créer une référence*".

Sinon, le symbole **&** signifie "*Je veux obtenir l'adresse de cette variable*".

# Les références à l'intérieur d'une fonction (3/3)

- Exemple:

```
int &referenceSurNbrEnfants;
```

- Il y a deux règles:

✓ Règle 1: une référence doit être initialisée dès sa déclaration.

✓ Règle 2: une fois initialisée, une référence ne peut plus être modifiée.

- Exemple:

```
int nbr = 2;    // Déclaration de la variable nbr
```

```
int &referenceSurNbrEnfants = nbr; // Déclaration et initialisation d'une référence sur la variable nbr
```

# Utilisation de la référence (1/2)

- **A quoi servent les références ?**

- ➔ Les références permettent de simplifier l'écriture des programmes pour éviter au maximum les erreurs.
- ➔ Pas besoin de mettre une étoile \* devant la variable pour dire qu'on veut obtenir la valeur.
- ➔ Les références sont principalement utilisées pour passer des paramètres aux fonctions.

# Utilisation de la référence (2/2)

- Exemple:

```
int main()
{
    int nbr = 2;
    int &referenceSurNbrEnfants = nbr;

    cout << referenceSurNbrEnfants << endl;
    cout << nbr << endl;

    referenceSurNbrEnfants = 3;

    cout << referenceSurNbrEnfants << endl;
    cout << nbr << endl;

    return 0;
}
```



2  
2  
3  
3

Cet exemple affiche la variable **nbr**, la modifie, et la réaffiche, le tout **en passant par une référence !**



Toute autre affectation de la référence modifie en fait la variable référencée.

Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

# Les références vers des structures

- Pour accéder à un élément d'une structure , il faut utiliser le symbole point "." et non le symbole flèche "->".

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

int main()
{
    Coordonnees point;
    Coordonnees &referenceSurPoint = point;
```

```
referenceSurPoint.x = 6;
referenceSurPoint.y = 8;

cout << "x : " << referenceSurPoint.x << endl;
cout << "y : " << referenceSurPoint.y << endl;

return 0;
}
```

# Les références et les fonctions (1/2)

- Les références sont principalement utilisées pour passer des paramètres aux fonctions.
- Exemple:

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

void remiseAZero(Coordonnees &pointAModifier);
```

## Les références et les fonctions (2/2)

```
int main()
{
    Coordonnees point;

    remiseAZero(point);    // Pas besoin d'indiquer l'adresse de point avec un & lors de l'appel
                           // éviter d'avoir à taper des symboles en plus pour minimiser les erreurs

    return 0;
}

void remiseAZero(Coordonnees &pointAModifier) // La fonction indique qu'elle récupère une référence
{
                                           // La référence s'utilise exactement comme une variable

    pointAModifier.x = 0;
    pointAModifier.y = 0;
}
```



# Différence entre les références et les pointeurs

- Les références et les pointeurs sont étroitement liés.
- En effet, une variable et ses différentes références ont la même adresse, puisqu'elles permettent d'accéder à un même objet.
  - ➔ Utiliser une référence pour manipuler un objet revient donc exactement au même que de manipuler un pointeur constant contenant l'adresse de cet objet.
  - ➔ **Les références permettent simplement d'obtenir le même résultat que les pointeurs, mais avec une plus grande facilité d'écriture.**

- Exemples:

### Exemple avec pointeur

```
int main()
{
    int nbr = 2;
    int *pointeurSurNbr = &nbr;

    cout << *pointeurSurNbr;

    *pointeurSurNbr = 3;

    cout << *pointeurSurNbr;

    return 0;
}
```

### Exemple avec référence

```
int main()
{
    int nbr = 2;
    int &referenceSurNbr = nbr;

    cout << referenceSurNbr ;

    referenceSurNbr = 3;

    cout << referenceSurNbr ;

    return 0;
}
```

- Minimiser les risques d'erreur dans nos programmes.
- ➔ Pas besoin d'utiliser le symbole **\*** à chaque fois qu'on veut accéder à la variable nbr.
- Les références sont là pour simplifier l'écriture du code source.

## **Les tableaux en C++**

# Présentation des tableaux

- Les **tableaux** sont des structures de données constituées d'un certain nombre d'éléments de même type.
- L'accès à un élément du tableau se fait en indiquant son indice entre crochets (**indice de 0 à nombre\_d\_éléments-1**).
- Il permet le stockage de plusieurs variables de même type.

# Les tableaux statiques (1/4)

- **Un tableau statique:** sa taille reste inchangée et est fixée dans le code source.
- La déclaration d'un tableau se fait comme suit:

```
type  Nom_Tableau[taille];
```

- Exemple:

```
int  const  tailleTableau(10);           //Déclaration de la taille du tableau  
  
double  Tableau_Moyenne[tailleTableau];
```

## Les tableaux statiques (2/4)

### ❑ Accès aux éléments d'un tableau:

- Pour accéder à une case d'un tableau, on utilise la syntaxe suivante:

```
nomDuTableau[numeroDeLaCase];
```

- La première case possède le numéro 0 et pas 1 !
- Exemple: pour accéder et modifier la quatrième case du tableau "Tableau\_Moyenne":

```
Tableau_Moyenne[3] = 2,5;
```

# Les tableaux statiques (3/4)

## ❑ Parcours d'un tableau:

- Exemple:

```
int const  tailleTableau(3);  
double  Tableau_Moyenne[tailleTableau];  
  
Tableau_Moyenne[0] = 16,5;           //Remplissage de la première case  
Tableau_Moyenne[1] = 12,4;           //Remplissage de la deuxième case  
Tableau_Moyenne[2] = 19,75;          //Remplissage de la troisième case  
  
for(int i(0); i<tailleTableau; ++i)  
{  
    cout << Tableau_Moyenne[i] << endl;  
}
```

# Les tableaux statiques (4/4)

## ❑ Les tableaux et les fonctions:

- Les fonctions ne peuvent pas renvoyer un tableau statique.
- Exemple: une fonction qui calcule la moyenne des valeurs d'un tableau.

```
double Calcul_Moyenne(double tableau[], int tailleTableau)
{
    double moyenne(0);

    for(int i(0); i<tailleTableau; ++i)
    {
        moyenne += tableau[i];           //On additionne toutes les valeurs
    }
    moyenne /= tailleTableau;

    return moyenne;
}
```



# Les tableaux dynamiques (1/6)

- **Un tableau dynamique** est un tableau dont le nombre de cases peut varier au cours de l'exécution du programme.
- Il permet d'ajuster la taille du tableau au besoin du programmeur.

## ❑ Déclaration d'un tableau dynamique:

```
vector<Type> Nom(tailleTableau);
```

- On écrit la taille du tableau entre parenthèses et non entre crochets.

## Les tableaux dynamiques (2/6)

- Exemple:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> tableau(10);

    return 0;
}
```

# Les tableaux dynamiques (3/6)

## ❑ Remplissage du tableau dynamique:

- En C++, il est possible de:
  - ✓ remplir directement toutes les cases du tableau en ajoutant un *deuxième argument* entre les parenthèses :

```
vector<int> tableau(10, 2); //Crée un tableau de 10 entiers valant tous 2
```

- ✓ déclarer un tableau sans cases en ne mettant pas de parenthèses du tout :

```
vector<int> tableau; //Crée un tableau de 0 entiers
```

- ➔ La taille des tableaux dynamiques peut varier.
- ➔ On peut donc ajouter des cases par la suite !

## Les tableaux dynamiques (4/6)

### ❑ Accès aux éléments des tableaux dynamiques:

- L'accès aux éléments des tableaux dynamiques est exactement identique aux tableaux statiques.

### ❑ Ajout des cases à la fin d'un tableau dynamique:

- Il faut utiliser la fonction **push\_back()**.

```
vector<int> tableau(4,2);    //Un tableau de 4 entiers valant tous 2
```

```
tableau.push_back(6);      //On ajoute une 5ème case au tableau qui contient la valeur 6
```

# Les tableaux dynamiques (5/6)

## ❑ Suppression de la dernière case d'un tableau dynamique:

- Il faut utiliser la fonction **pop\_back()**:

```
vector<int> tableau(4,2);    //Un tableau de 4 entiers valant tous 2  
  
tableau.pop_back();
```

## ❑ Taille d'un tableau dynamique:

- **size()** permet de récupérer un entier correspondant au nombre d'éléments d'un tableau.

```
vector<int> tableau(4,2);    //Un tableau de 4 entiers valant tous 2  
  
int taille = tableau.size();
```

# Les tableaux dynamiques (6/6)

## ❑ Les tableaux dynamiques et les fonctions:

- Le passage d'un tableau dynamique en argument à une fonction est beaucoup plus simple que pour les tableaux statiques.
- Il faut mettre **vector<type>** en argument.
- Il n'y a pas besoin d'ajouter un deuxième argument pour la taille du tableau, grâce à la fonction **size()**.

```
#include <vector>
```

```
void fonctionAvecTableau(std::vector<int>& tableau);
```

# Les tableaux multidimensionnels (1/5)

- Les tableaux multidimensionnels sont des tableaux qui contiennent des tableaux.
- Ses variables sont arrangées selon des axes X et Y et pas uniquement selon un seul axe.

## ❑ Déclaration d'un tableau multidimensionnelle:

```
type  nomTableau[tailleX][tailleY];
```

- Exemple:

```
int  tableau[3][4];    //Un tableau d'entiers positifs à deux dimensions (3 lignes, 4 colonnes)
```

## Les tableaux multidimensionnels (2/5)

- ❑ Accès aux éléments d'un tableau multidimensionnelle:

```
int tableau[3][4];
```

Tableau[0][0]	Tableau[0][1]	Tableau[0][2]	Tableau[0][3]
Tableau[1][0]	Tableau[1][1]	Tableau[1][2]	Tableau[1][3]
Tableau[2][0]	Tableau[2][1]	Tableau[2][2]	Tableau[2][3]



# Les tableaux multidimensionnels (3/5)

## ❑ Initialiser les éléments:

- Il y a plusieurs manières d'initialiser un tableau multidimensionnel.

### 1. Initialisation individuelle de chaque élément:

```
tableau [0][0] = 6;
```

```
tableau [0][1] = 8;
```

```
...
```

# Les tableaux multidimensionnels (4/5)

## 2. Initialisation grâce à des boucles:

- Par exemple les éléments de `tableau[3][4]` pourront être initialisés à 0 par les instructions suivantes :

```
int i,j;

for (i=0; i<2; i++) {

    for (j=0; j<3; j++) {

        tableau[i][j] = 0;
    }
}
```

# Les tableaux multidimensionnels (5/5)

## 3. Initialisation à la définition:

```
type  Nom_du_tableau [Taille1][Taille2]...[TailleN] = {a1, a2, ... aN};
```

- Les valeurs sont attribuées aux éléments successifs en incrémentant d'abord les indices de droite, c'est-à-dire pour un tableau à 2 dimensions : [0][0], [0][1], [0][2] ... puis [1][0] etc.

# Les tableaux de caractères (1/3)

- Les chaînes de caractères sont en fait des tableaux !
- Un tableau de **char** constitue une chaîne de caractères.

```
char chaine[15] = "BONJOUR";
```

- L'exemple est équivalent à cette écriture:

```
char chaine[20] = {'B','O','N','J','O','U','R','\0'};
```

➔ Le dernier caractère est nul pour indiquer la fin de la chaîne de caractères.

## Les tableaux de caractères (2/3)

- Les fonctions de manipulation de chaîne de caractères sont les mêmes que dans le langage C.
- Elles portent un nom commençant par **str** (**String**).
- **strlen**: retourne la longueur de la chaîne source (sans le caractère nul final).

```
int  strlen(const char* source)
```

- **strcpy**: copie la chaîne source dans le tableau pointé par *dest*.

```
void  strcpy(char* dest, const char* source)
```

## Les tableaux de caractères (3/3)

- **size()**: connaître le nombre de lettres dans une chaîne de caractères.
- **push\_back()**: ajouter des lettres à la fin de la chaîne de caractères, comme avec **vector**.
- Contrairement aux tableaux, on peut ajouter plusieurs lettres d'un coup en utilisant le **+=**.

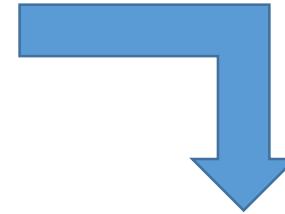
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string prenom("Albert");
    string nom("Einstein");

    string nomComplet;           // Déclaration d'une chaîne vide
    nomComplet += prenom;        // Ajout du prénom à la chaîne vide
    nomComplet += " ";
    nomComplet += nom;           //Ajout du nom de famille

    cout << "Votre nom complet est:" << nomComplet << "." << endl;

    return 0;
}
```



Votre nom complet est: Albert Einstein.

## **Les fonctions en C++**



# Présentation des fonctions

- Le but des **fonctions** est de **découper son programme en petits éléments réutilisables**.
- Une fonction pourra appeler d'autres fonctions et ainsi de suite.
- Une fonction peut même s'appeler elle-même : on parle de **fonctions récursives**.

## ❑ Avantages:

- Découper son programme en fonctions permet de *s'organiser*.
- Partager plus facilement le travail entre plusieurs développeurs travaillant sur le même programme.

# Prototype d'une fonction (1/2)

- **Le prototype d'une fonction:**

- ✓ précise le **nom de la fonction**
- ✓ donne le **type de la valeur de retour de la fonction** (*void* quand il n'y a pas de retour)
- ✓ donne les **types des paramètres éventuels**.

```
type nomFonction(arguments)
```

## Prototype d'une fonction (2/2)

- Exemple:

```
int fonction(int x,int y);    // prototype de la fonction
```

- Le rôle d'un prototype n'est pas de définir les instructions de la fonction, mais donner sa **signature**.
- On les trouve principalement dans les fichiers d'en-tête (extension **.h**).

# Définition d'une fonction

- Le premier élément est le **type de retour**. Il permet d'indiquer le type de variable renvoyée par la fonction.
- Le deuxième élément est le **nom de la fonction**.
- Entre les parenthèses, il y a la liste des arguments de la fonction.

Ce sont les données avec lesquelles la fonction va travailler.

Il peut y avoir un argument, plusieurs arguments ou aucun argument (comme pour main()).

```
type nomFonction(arguments)
{
    //Instructions effectuées par la fonction
}
```

# Portée des variables (1/3)

## ❑ Variables locales:

- **Une variable locale** est déclarée dans le corps d'une fonction.
- Elle n'est accessible que depuis cette fonction dès sa déclaration et disparaît une fois que l'on sort du plus petit bloc { } contenant la déclaration de la variable.

```
int n;  
cout << "Donner un nombre";  
cin >> n;  
  
if (n > 0) {  
    int x = 0; // x est une variable locale  
} else {  
    int x = 1;  
}  
  
int b = x; // ne compile pas , x n'existe plus !
```

## Portée des variables (2/3)

### ❑ Variables globales:

- **Une variable globale** a pour portée tout le programme.
- Elle est accessible à toutes les fonctions et ne disparaît qu'avec le fin de l'exécution du programme.

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

int b;    // variable globale

int somme(int x, int y) {
    int a;    // variable locale à la fonction
    a = x+y;
    return a;
}
```

## Portée des variables (3/3)

- Il est tout à fait possible d'avoir deux variables ayant le même nom pour autant qu'elles soient déclarées dans des fonctions différentes.

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

double carre(double x)
{
    double nombre;
    nombre = x*x;
    return nombre;
}
```

```
int main()
{
    double nombre, carreNombre;
    cout << "Entrez un nombre : ";
    cin >> nombre;

    carreNombre = carre(nombre);

    cout << "Le carre de " << nombre << " est " << carreNombre << endl;
    return 0;
}
```

# Passage par valeur (1/3)

- La valeur de l'expression passée en paramètre est copiée dans une **variable locale**.
  - ➔ C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.
- Si l'expression passée en paramètre est une variable, son contenu est copié dans la variable locale.
- Aucune modification de la variable locale dans la fonction appelée ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière.
- Le **C** ne permet de faire que des passages par valeur.



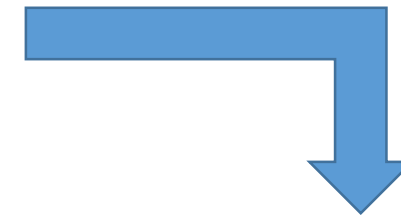
## Passage par valeur (2/3)

- Exemple:

```
int ajouteDeux(int a)
{
    a+=2;
    return a;
}

int main()
{
    int nombre(4), resultat;
    resultat = ajouteDeux(nombre);

    cout << "Le nombre original vaut : " << nombre << endl;
    cout << "Le resultat vaut : " << resultat << endl;
    return 0;
}
```

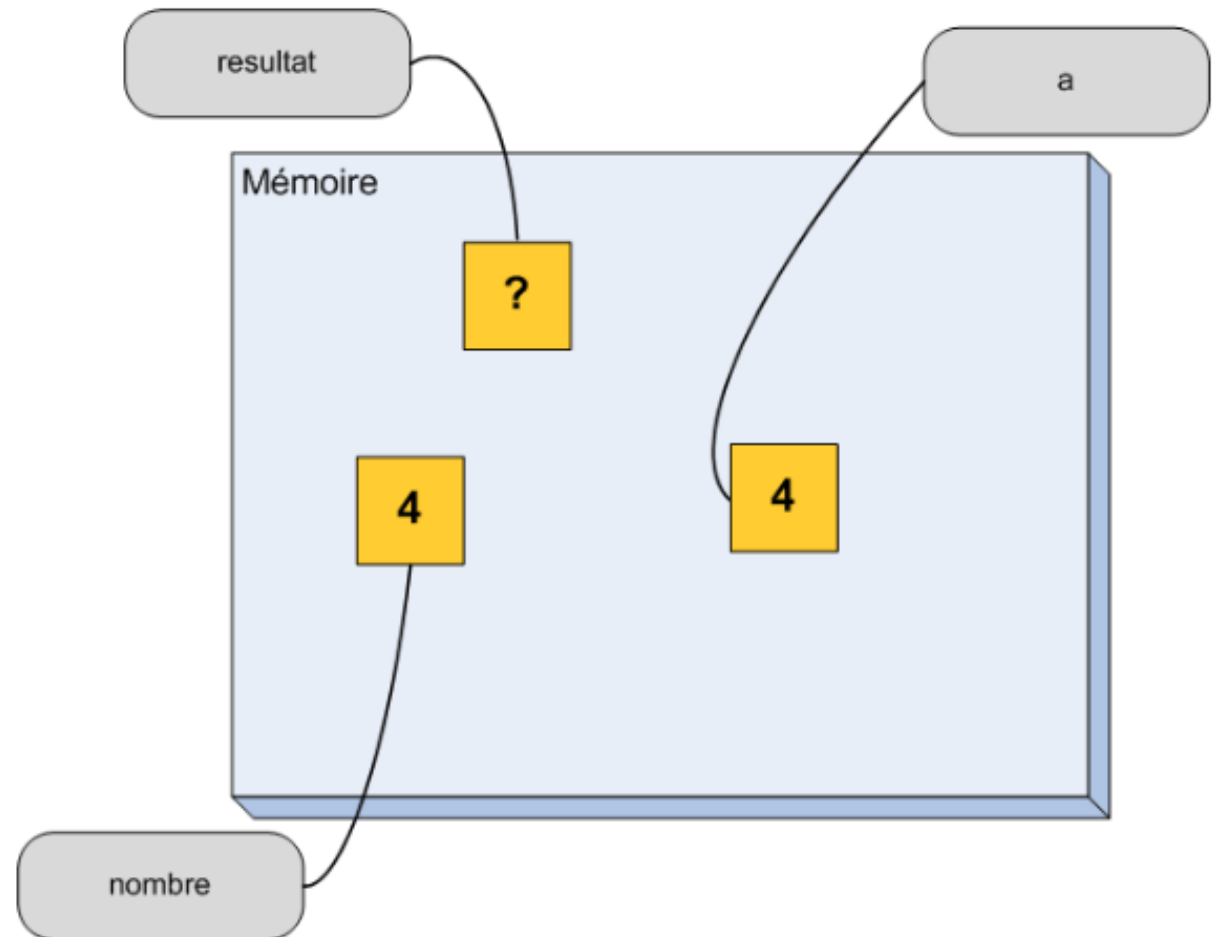


Le nombre original vaut : 4

Le resultat vaut : 6

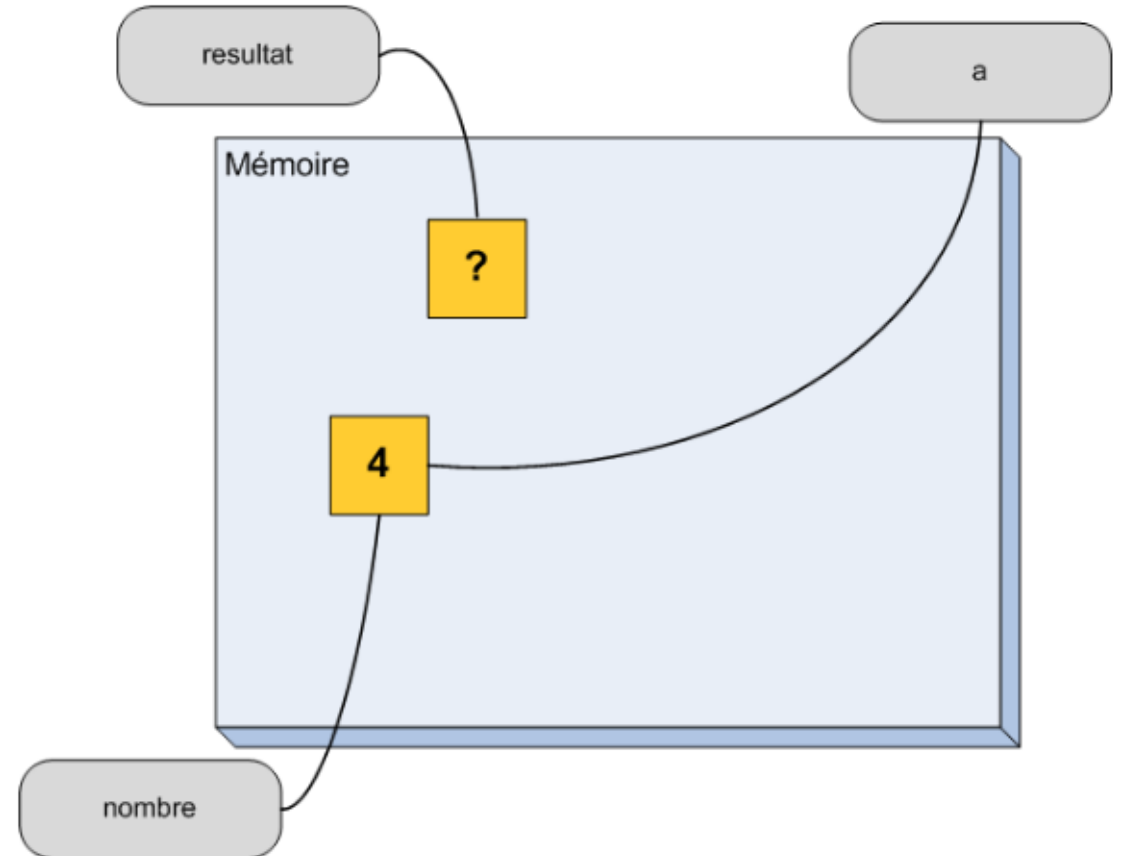
## Passage par valeur (3/3)

- La variable nombre est copiée dans une **nouvelle case mémoire**.
- On dit que l'argument **a** est passé **par valeur**.
- La variable nombre reste inchangée.



# Passage par référence (1/3)

- Il est possible d'ajouter une « *deuxième étiquette* » à la variable **nombre** à l'intérieur de la fonction au lieu de copier la valeur de **nombre** dans la variable **a**.
  - ➔ Utiliser une **référence** comme argument de la fonction.
- La variable **a** et la variable **nombre** sont confondues.
  - ➔ On dit que l'argument **a** est passé par référence.



## Passage par référence (2/3)

- Exemple:

```
int ajouteDeux(int& a)
{
    a+=2;
    return a;
}

int main()
{
    int nombre(4), resultat;
    resultat = ajouteDeux(nombre);

    cout << "Le nombre original vaut : " << nombre << endl;
    cout << "Le resultat vaut : " << resultat << endl;
    return 0;
}
```



Le nombre original vaut : 6

Le resultat vaut : 6

Comme **a** et la variable **nombre** correspondent à la même case mémoire, faire **a+=2** a modifié la valeur de **nombre** !

# Passage par référence (3/3)

## ➤ Utilité du passage par référence:

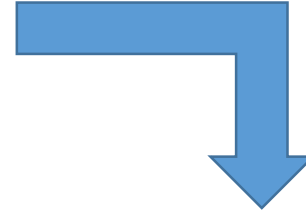
- Prenons par exemple, l'exemple classique de la fonction **echangeVariables()**.
- C'est une fonction qui échange les valeurs de ses deux arguments.

```
void echangeVariables(int& x, int& y)
```

```
{  
    int aux(x);  
    x = y;  
    y = aux;  
}
```

```
int main()
```

```
{  
    int x(4), y(6);  
  
    cout << " x vaut " << x << " et y vaut " << y << endl;  
  
    echangeVariables(x,y);  
  
    cout << " x vaut " << x << " et y vaut " << y << endl;  
    return 0;  
}
```




x vaut 4 et y vaut 6

x vaut 6 et y vaut 4

- Les valeurs des deux variables ont été échangées.
- Si l'on n'utilisait pas un **passage par référence**, ce seraient alors des **copies** des arguments qui seraient échangées, et non les vrais arguments.

➔ Cette fonction serait donc inutile.

# Le passage par référence constante (1/2)

- L'avantage du **passage par référence** est qu'aucune copie n'est effectuée.
  - Si on a une fonction recevant en argument une chaîne de caractères de type string qui a une grande taille:
    - ➔ Copier la chaîne prend du temps et inutile
    - ➔ Utiliser un **passage par référence** (aucune copie n'est effectuée)  
*Inconvénient:* autoriser la **modification** de l'argument
- 
- La solution est d'utiliser **un passage par référence constante**.
  - On évite la copie en utilisant une référence et l'on empêche la modification de l'argument en le déclarant **constant**.

## Le passage par référence constante (2/2)

- Exemple:

```
void fonction1(string texte);           //Implique une copie coûteuse de 'texte'
{
}

void fonction2(string& texte);          //Implique que la fonction peut modifier 'texte'
{
}

void fonction1(string const& texte);    //Pas de copie et pas de modification possible
{
}
```



# Les valeurs par défaut des paramètres (1/6)

- En C++, on peut donner des **valeurs par défaut** à certains paramètres des fonctions.  
→ **Pas d'obligation d'indiquer à chaque fois tous les paramètres lorsqu'on appelle une fonction.**
- Utiliser les paramètres par défauts peut s'avérer utile lorsque vous ne connaissez pas la valeur de certains paramètres alors qu'une fonction les attend.

- **Exemple:**

Un exemple sans valeurs par défaut.

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

// Prototype de la fonction
int CalculSecondes(int heures, int minutes, int secondes);

int main()
{
```

## Les valeurs par défaut des paramètres (2/6)

```
cout << CalculSecondes(2, 14, 32) << endl;

return 0;
}

// Définition de la fonction
int CalculSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}
```

## Les valeurs par défaut des paramètres (3/6)

- Pour rendre la valeur des heures **obligatoire** et les valeurs des minutes et des secondes **facultatifs**, on modifie le prototype de la fonction:

```
int CalculSecondes(int heures, int minutes = 0, int secondes = 0);
```

- La nouveauté en C++, on peut supprimer des paramètres lors de l'appel de la fonction:

```
cout << CalculSecondes(2) << endl;
```

➔ **Le compilateur lit les paramètres de gauche à droite.**

➔ Comme il n'y en a qu'un et que seules les heures sont obligatoires, il devine que la valeur "2" correspond à un nombre d'heures.

## Les valeurs par défaut des paramètres (4/6)

```
cout << CalculeCondes(2,14) << endl;
```

➔ Correct

➔ On indique les heures et les minutes

```
cout << CalculSecondes(2,,32) << endl;
```

➔ **Non correct**

➔ On indique les heures et les secondes.

➔ Le compilateur analyse les paramètres de gauche à droite.

➔ Le premier correspondra forcément aux heures, le second aux minutes et le troisième aux secondes.

## Les valeurs par défaut des paramètres (5/6)

- Puisqu'on ne peut pas sauter des paramètres même s'ils sont facultatifs:

```
cout << CalculSecondes(2, 0, 32) << endl;
```

- Pour rendre les heures facultatifs:

```
int CalculSecondes(int secondes, int minutes, int heures = 0);
```

➔ Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres (à droite).

# Les valeurs par défaut des paramètres (6/6)

- On peut rendre tous les paramètres facultatifs:

```
int CalculSecondes(int heures = 0, int minutes = 0, int secondes = 0);
```

- L'appel à cette fonction:

```
cout << CalculSecondes() << endl;
```

# La surcharge des fonctions (1/3)

- **La surcharge des fonctions** est une technique permettant de créer plusieurs fonctions ayant le même nom.
- Chaque fonction est constituée de 3 éléments:
  - ✓ Un type de retour
  - ✓ Un nom
  - ✓ Une liste de paramètres
- **En C**, le compilateur se basait sur le nom.
  - ➔ Si deux fonctions avaient le **même nom**, la compilation plantait !
  - ➔ L'identification est faite sur le nom.

## La surcharge des fonctions (2/3)

- En C++, le compilateur se base sur le **nom** et les **paramètres** !
  - ➔ On peut avoir deux fonctions avec le même nom, à condition que celles-ci reçoivent des paramètres différents.
- Le **nom** et les **paramètres** de la fonction constituent la **signature de la fonction**.
- Le type de retour n'est pas pris en compte pour identifier la fonction.
- Ce qui peut varier deux fonctions ayant le même nom:
  - ✓ Le nombre de paramètres
  - ✓ Le type de chacun de ces paramètres



# La surcharge des fonctions (3/3)

- Exemple:

```
int somme(int nb1, int nb2)
{
    return nb1 + nb2;
}

double somme(double nb1, double nb2)
{
    return nb1 + nb2;
}
```

➔ Ce sont des fonctions surchargées

## **Les fichiers en C++**

# Généralités sur les flux (1/2)

- Les entrées et sorties en C++ se font toujours par l'intermédiaires de **flux** (ou flots), qu'on peut considérer comme des canaux:
  - ✓ recevant de l'information dans le cas d'un flux de sortie
  - ✓ fournissant l'information dans le cas d'un flux d'entrée
- Le flux de sortie standard est **cout**.
  - ➔ Ce flot est connecté par défaut à **l'écran**.
- Le flot d'entrée standard est **cin**.
  - ➔ Ce flot est connecté par défaut au **clavier**.

## Généralités sur les flux (2/2)

- Les opérateurs de manipulation des flots, qui permettent le transfert de l'information sont:
  - ✓ `<<` qui permet l'écriture sur un **flot de sortie**
  - ✓ `>>` qui permet la lecture sur un **flot d'entrée**
- Dans cette section, nous allons donc parler **des flux vers les fichiers**.
- Pour les fichiers, il faut inclure l'entête **fstream**:

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

# Généralités sur les fichiers

- **La règle générale pour créer un fichier est la suivante :**
  - ✓ il faut l'ouvrir en écriture.
  - ✓ on **écrit** des données dans le fichier.
  - ✓ on ferme le fichier.
- **Pour lire des données écrites dans un fichier :**
  - ✓ on l'ouvre en lecture.
  - ✓ on **lit** les données en provenance du fichier.
  - ✓ on ferme le fichier.

# Écriture dans un fichier (1/7)

## ❑ Ouvrir un fichier en écriture:

- Un flux est une variable dont le *type* serait **ofstream** et dont la *valeur* serait le chemin d'accès du fichier à lire.
- Comme pour les variables, les règles à suivre pour le choix du nom du flux sont:
  - ✓ les noms des flux sont constitués de lettres, de chiffres et du tiret-bas \_ uniquement ;
  - ✓ le premier caractère doit être une lettre (majuscule ou minuscule) ;
  - ✓ pas d'accents ;
  - ✓ pas d'espaces dans le nom.

## Ecriture dans un fichier (2/7)

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

int main()
{
    ofstream monFlux("C:/C++/Fichiers/monFichier.txt");
    //Déclaration d'un flux permettant d'écrire dans le fichier
    return 0;
}
```

- Le chemin d'accès au fichier peut prendre les deux formes suivantes:

- ✓ **Un chemin absolu**, c'est-à-dire montrant l'emplacement du fichier depuis la racine du disque.

Par exemple: ("C:/C++/Fichiers/monFichier.txt");

- ✓ **Un chemin relatif**, c'est-à-dire montrant l'emplacement du fichier depuis l'endroit où se situe le programme sur le disque.

Par exemple : ("Fichiers/scores.txt") si mon programme se situe dans le dossier ("C:/C++/").

**Si le fichier n'existait pas, le programme le créerait automatiquement !**

## Ecriture dans un fichier (3/7)

- Le plus souvent, le nom du fichier est contenu dans une chaîne de caractères **string**.
- Dans ce cas, il faut utiliser la fonction **c\_str()** lors de l'ouverture du fichier.

```
string const nomFichier("C:/C++/Fichiers/monFichier.txt");  
  
ofstream monFlux(nomFichier.c_str());  
//Déclaration d'un flux permettant d'écrire dans un fichier.
```



## Écriture dans un fichier (4/7)

- Exemple:

```
ofstream monFlux(" C:/C++/Fichiers/monFichier.txt");  
    //On essaye d'ouvrir le fichier  
  
if(monFlux)           //On teste si tout est OK  
{  
    //Tout est OK, on peut utiliser le fichier  
}  
else  
{  
    cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;  
}
```

- Il faut toujours **tester** si tout s'est bien passé car des problèmes peuvent survenir lors de l'ouverture d'un fichier (disque dur plein par exemple).

## Ecriture dans un fichier (5/7)

### ❑ Ecrire dans un flux:

- Les chevrons (<<) permettent d'envoyer des informations dans un flux.

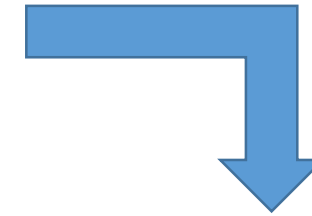
```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    string const nomFichier("C:/C++/Fichiers/monFichier.txt ");
    ofstream monFlux(nomFichier.c_str());

    if(monFlux)
    {
```

## Ecriture dans un fichier (6/7)

```
monFlux << "Bonjour, je suis une phrase écrite dans un fichier." << endl;  
  
monFlux << 24.75 << endl;  
  
int age(20);  
  
    monFlux << "J'ai " << age << " ans." << endl;  
}  
else  
{  
    cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;  
}  
return 0;  
}
```



Bonjour, je suis une phrase écrite dans un fichier."

24,75

J'ai 20 ans.

## Ecriture dans un fichier (7/7)

### ❑ Ecrire à la fin d'un fichier:

- Pour ne pas écraser des données déjà écrites dans un fichier existant, il faut écrire à la fin du fichier.

```
ofstream monFlux("C:/C++/Fichiers/monFichier.txt ", ios::app);
```

- **app** est un raccourci pour *append*, qui indique qu'il faut ajouter à la fin.

# Lire un fichier (1/8)

## ❑ Ouvrir un fichier en lecture:

- Utiliser un **ifstream** au lieu d'un *ofstream*.
- Tester l'ouverture, afin d'éviter les erreurs.

```
ifstream monFlux("C:/C++/Fichiers/monFichier.txt ");

if(monFlux)
{
    //Tout est prêt pour la lecture.
}
else
{
    cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << endl;
}
```

## Lire un fichier (2/8)

- Il y a trois manières différentes de lire un fichier :
  - ✓ **Ligne par ligne**, en utilisant **getline()** ;
  - ✓ **Mot par mot**, en utilisant les chevrons **>>** ;
  - ✓ **Caractère par caractère**, en utilisant **get()**.

## Lire un fichier (3/8)

### ❑ Lire ligne par ligne:

- Cette méthode permet de récupérer une ligne entière et de la stocker dans une chaîne de caractères.

```
string ligne;
```

```
getline(monFlux, ligne);           //On lit une ligne complète
```

## Lire un fichier (4/8)

### ❑ Lire mot par mot:

```
double nombre;  
monFlux >> nombre;    //Lit un nombre à virgule depuis le fichier  
  
string mot;  
monFlux >> mot;        //Lit un mot depuis le fichier
```

- Cette méthode lit ce qui se trouve entre l'endroit où l'on se situe dans le fichier et l'espace suivant.
- Ce qui est lu est traduit en *double*, *int* ou *string* selon le type de variable dans lequel on écrit.



## Lire un fichier (5/8)

### ❑ Lire caractère par caractère:

```
char a;  
monFlux.get(a);
```

- Ce code lit une seule lettre et la stocke dans la variable **a**.

## Lire un fichier (6/8)

### ❑ Lire un fichier en entier:

- **getline()** permet de savoir si l'on peut continuer à lire en renvoyant un **bool**.
- Elle renvoie **true** si la lecture peut continuer.
- Elle renvoie **false** si on arrive à la fin du fichier ou qu'il y a une erreur.

# Lire un fichier (7/8)

- Exemple:

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

int main()
{
    ifstream fichier("C:/C++/Fichiers/monFichier.txt ");

    if(fichier)
    {

        string ligne;           //Une variable pour stocker les lignes lues
```

## Lire un fichier (8/8)

```
while(getline(fichier, ligne))           //Tant qu'on n'est pas à la fin, on lit
{
    cout << ligne << endl;               //Et on l'affiche dans la console
}
else
{
    cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << endl;
}

return 0;
}
```

## Fermer un fichier (1/2)

- Les fichiers ouverts sont automatiquement refermés lorsque l'on sort du bloc où le flux est déclaré.
- Exemple:

```
void fonction()
{
    ofstream flux("C:/C++/Fichiers/monFichier.txt ");           //Le fichier est ouvert

    ....

} //Lorsque l'on sort du bloc, le fichier est automatiquement refermé
```

## Fermer un fichier (2/2)

- Pour fermer le fichier avant sa fermeture automatique, il faut utiliser la fonction **close()** des flux.

```
void fonction()
{
    ofstream flux("C:/C++/Fichiers/monFichier.txt ");           //Le fichier est ouvert

    //Utilisation du fichier

    flux.close();                                                //On referme le fichier
                                                                //On ne peut plus écrire dans le fichier à partir d'ici
}
```

# Retarder l'ouverture d'un fichier

- Il est possible de retarder l'ouverture d'un fichier après la déclaration du flux en utilisant la fonction **open()**.

```
void fonction()
{
    ofstream flux;                                //Un flux sans fichier associé

    flux.open("C:/C++/Fichiers/monFichier.txt ");

    //Utilisation du fichier

    flux.close();                                  //On referme le fichier
}
```

# Le curseur dans le fichier (1/4)

- Lorsqu'on ouvre un fichier, le curseur est placé tout au début.
- Il existe des moyens de se déplacer dans un fichier afin de lire uniquement les parties qui nous intéressent réellement.

## 1. Connaitre la position du curseur:

- Il existe deux fonctions qui permettent de savoir à quel caractère du fichier on se situe.
  - ✓ pour les flux entrant (**ifstream**): **tellg()**
  - ✓ pour les flux sortant (**ofstream**): **tellp()**

```
int position = fichier.tellp();
```



## Le curseur dans le fichier (2/4)

### 2. Se déplacer dans le fichier:

- Ils existent deux fonctions qui permettent le déplacement dans le fichier:
  - ✓ pour les flux entrant (**ifstream**): **seekg()**
  - ✓ pour les flux sortant (**ofstream**): **seekp()**
- Ces fonctions reçoivent deux arguments : **une position** dans le fichier et **un nombre de caractères** à ajouter à cette position.

```
flux.seekp(nombreCaracteres, position);
```

## Le curseur dans le fichier (3/4)

- Les trois positions possibles sont :

- ✓ le début du fichier : **ios::beg**

- ✓ la fin du fichier : **ios::end**

- ✓ la position actuelle : **ios::cur**

- Exemple:

```
flux.seekp(10, ios::beg);
```

➔ Se placer 10 caractères après le début du fichier

## Le curseur dans le fichier (4/4)

### ❑ Connaître la taille d'un fichier:

- Se déplacer à la fin du fichier
- Récupérer la position qui correspond à la taille du fichier.
- Exemple:

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

int main()
{
    ifstream fichier(" C:/C++/Fichiers/monFichier.txt ");           //ouverture du fichier
    fichier.seekg(0, ios::end);                                       //se déplacer à la fin du fichier

    int taille;
    taille = fichier.tellg();                                         //récupérer la position qui correspond à la taille du fichier

    cout << "Taille du fichier : " << taille << " octets." << endl;

    return 0;
}
```

## **Les structures en C++**

# Présentation des structures (1/2)

- Les **structures** permettent de regrouper plusieurs variables dans une même entité. Ainsi il est possible de construire de nouveaux types plus complexes.
- Les objets contenus dans la structure sont appelés **champs de la structure**.
- Les structures sont déclarés comme suit:

```
struct Nom_Structure {  
  
    type_champ1  
    Nom_Champ1;  
    type_champ2  
    Nom_Champ2;  
    type_champ3  
    Nom_Champ3;  
  
    ... };
```

## Présentation des structures (2/2)

- Les données peuvent être de n'importe quel type hormis le type de la structure dans laquelle elles se trouvent.

```
struct MaStructure {  
  
    int nbrEnfants;  
    char NbrEnfants;  
    struct MaStructure StructEnfant;  
  
};
```



- La structure suivante est **incorrecte**.
- Le type de donnée *struct MaStructure* n'est pas autorisé

# Définition d'une variable structurée

- La définition d'une variable structurée est une opération qui consiste à créer une variable ayant comme type celui d'une structure.

```
struct  Nom_Structure  Nom_Variable_Structuree;
```

- Exemple:

```
struct Personne {
```

```
    int Age;  
    char Sexe;
```

```
};
```



```
struct  Personne  Romain,Stephane,Fabrice,Alexis;
```

➔ Définition de 4 variables structurées.



# Accès aux champs d'une variable structurée

- Chaque variable de type structure possède des champs repérés avec des noms uniques.
- Pour accéder aux champs d'une structure on utilise l'opérateur de champ (un simple point) placé entre le nom de la variable structurée et le nom du champ.

```
Nom_Variable.Nom_Champ;
```

- Exemple:

```
Alexis.Age = 20;
```

```
Alexis.Sexe = 'Masculin';
```

# Tableaux de structures

- Il suffit de créer un tableau dont le type est celui de la structure et de le repérer par un nom de variable:

```
Nom_Structure  Nom_Tableau[Nb_Elements];
```

- Chaque élément du tableau représente une structure du type que l'on a défini.
- Le tableau suivant (nommé Tab) contient 6 variables structurées de type struct Personne.
- Exemple:

```
Personne  Tab[6];
```