Initiation à la programmation C++

ELC a 11 : Programmation des interfaces graphiques en C++

Emmanuel Dellandréa

emmanuel.dellandrea@ec-lyon.fr

ÉCOLE CENTRALE LYON

Le langage C++

- Historique :
  - Le langage C est le successeur d'un certain nombre de langages développés dans le cadre des Laboratoires Bell.
  - En 1970, Ken Thompson crée le langage B pour réécrire le système UNIX.
  - En 1972, Dennis Ritchie et Ken Thompson créent le langage C, successeur du langage B.
  - C'est un langage général non lié à une machine, mais qui permet un accès facile et efficace au matériel.
  - D. Ritchie a pu réécrire en C 90 % du noyau du système Unix.

Le langage C++

- Historique :
  - En 1978, Kernighan et Ritchie définissent officiellement C dans l'annexe du livre "The C programming language ».
  - Le langage C++ a été créé en 1980 par B. Stroustrup toujours dans les Laboratoires Bell.
  - C++ est une extension de C pour la programmation par objets.
  - En 1998, C++ connaît sa première normalisation ISO.
  - Le standard actuel a été ratifié et publié par l'ISO en septembre 2011.
  - Une mise à jour (mineure) a été publiée en 2017.

Le langage C++

- Caractéristiques :
  - C++ est un langage structuré et modulaire, dans lequel un programme est séparé en fichiers compilables séparément.
  - C++ est un langage efficace qui permet de programmer finement le matériel.
  - C++ est puissant car la souplesse d'expression du langage permet de faire tout ce que l'on désire et d'introduire sans limite des astuces de programmation.
  - C++ permet d’écrire des programmes transportables sur de très nombreuses machines (PC, Mac, stations de travail, serveurs, systèmes embarqués, etc.).

Le langage C++

- Caractéristiques :
  - Chaque compilateur est accompagné d'une bibliothèque standard qui permet d'augmenter encore la portabilité des programmes tout en diminuant l'effort de programmation.
  - L'inconvénient apporté par la grande souplesse d'expression est une sûreté assez faible du langage.

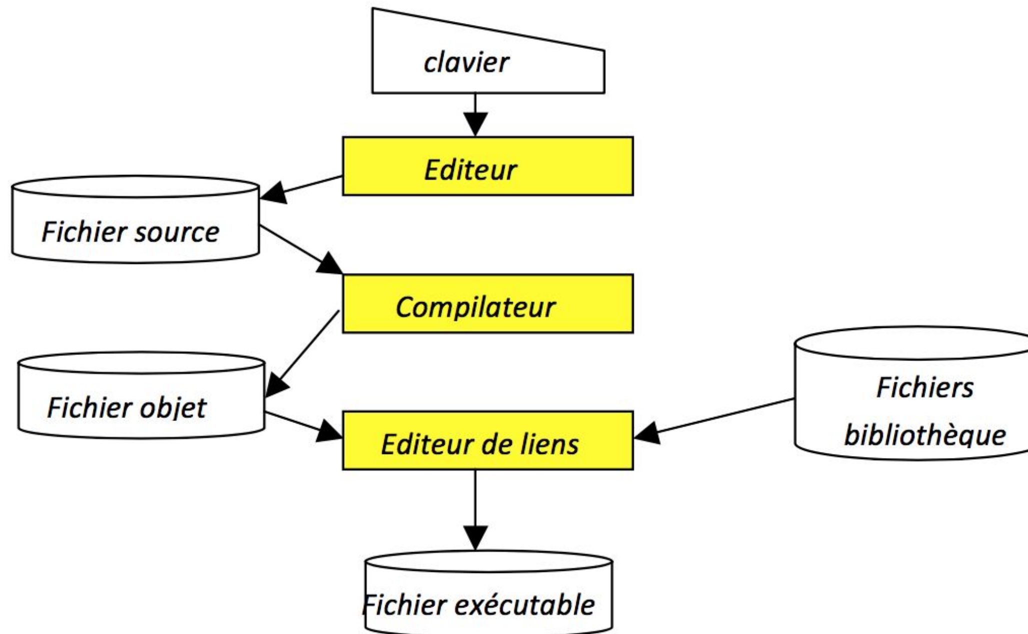
Bases de la programmation

- Déroulement d’un programme
  - Le langage C++ est un langage de haut niveau, indépendant d’une unité centrale particulière.
  - Les instructions C++ sont des lignes de textes éditables avec un éditeur de texte et compréhensibles par un utilisateur humain.
  - Pour mettre en œuvre un programme écrit en langage C++, on doit d’abord :
    - le traduire (le compiler) en langage machine,
    - puis exécuter le programme résultat de la traduction avec l’unité centrale.

ÉCOLE CENTRALE LYON

Bases de la programmation

- Déroulement d'un programme

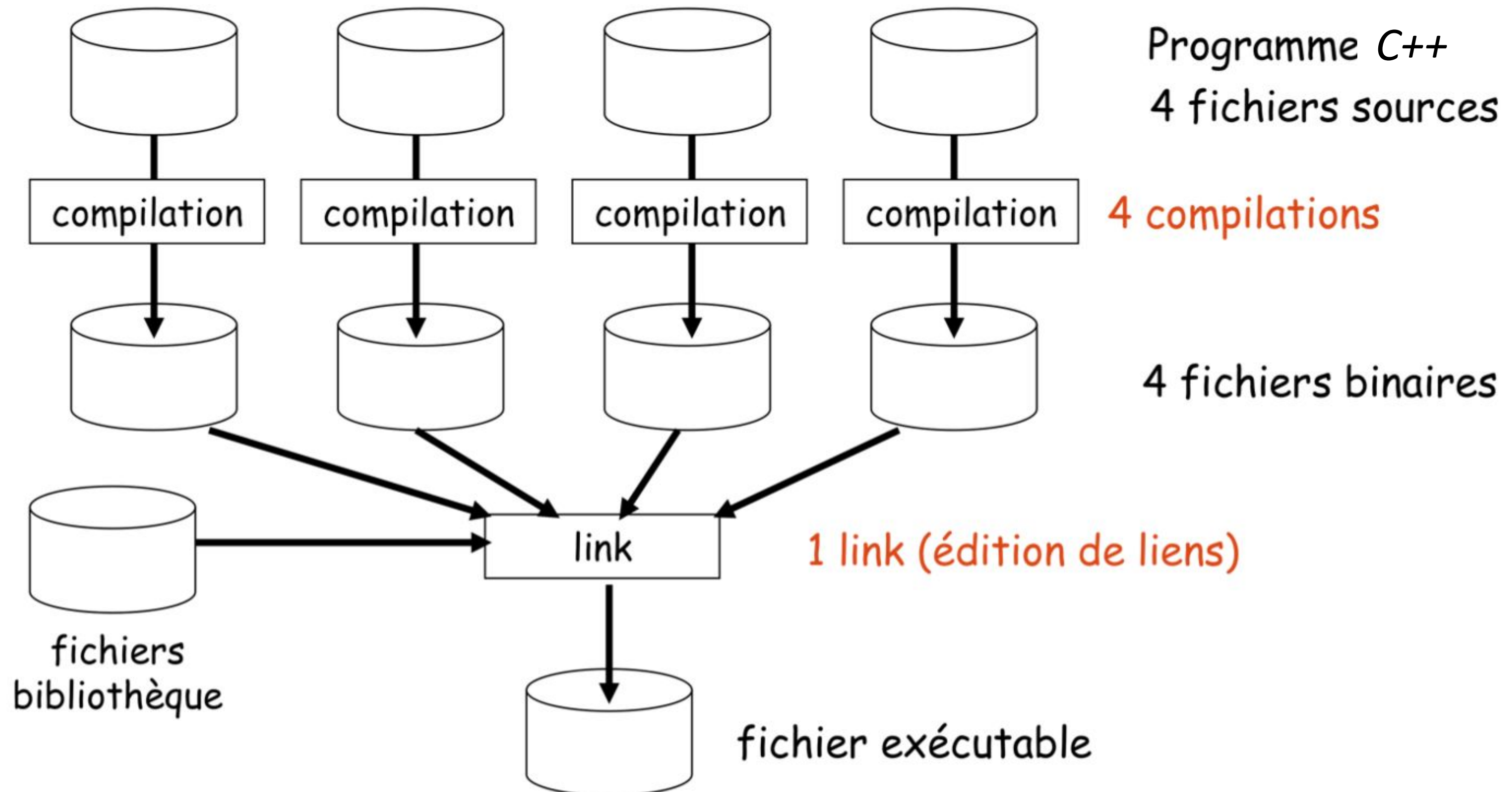


ÉCOLE CENTRALE LYON

Structure d'un programme C++

- Un programme C++ est constitué d'un ensemble de fichiers disque.
- Chaque fichier disque contient des définitions et déclarations de variables, fonctions, classes.
- Un programme C++ contient toujours une fonction main() qui est le point d'entrée du programme.
- Chaque définition de fonction contient l'entête de la fonction (type de retour, nom, arguments typés), des définitions de variables locales et le traitement proprement dit.

Structure d'un programme C++



Exemple d'un programme C++

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

int main()
{
     cout << "Bonjour ECL !" << endl;
     return 0;
}
```

- Explications (1/5) :
  - La directive de compilation `#include <iostream>` permet d'inclure les prototypes des différentes classes contenues dans la bibliothèque standard iostream.
  - Cette bibliothèque contient la définition de cout qui permet entre autre d'afficher des messages à l'écran.

Exemple d'un programme C++

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

int main()
{
     cout << "Bonjour ECL !" << endl;
     return 0;
}
```

- Explications (2/5) :
  - Utilisation de l'espace de nommage standard (`std`)
  - Un espace de nommage peut être vu comme un ensemble d'identifiants C++ (types, classes, variables etc.).
  - cout fait partie de l'espace de nommage std.
  - Cela évite d'utiliser le nom complet `std::cout` .

Exemple d'un programme C++

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

int main()
{
     cout << "Bonjour ECL !" << endl;
     return 0;
}
```

- Explications (3/5) :
  - Tout programme en C++ commence par l'exécution de la fonction `main` et se finit lorsque la fonction main est terminée.
  - La fonction `main` peut être vue comme le point d'entrée de tout programme en C++.
  - Elle renvoie un entier (code de retour)

ÉCOLE CENTRALE LYON

Exemple d'un programme C++

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

int main()
{
	cout << "Bonjour ECL !" << endl;
	return 0;
}
```

- Explications (4/5) :
  - L'objet `cout` permet d'envoyer des caractères vers le flux de sortie standard du programme (l'écran).
  - En utilisant l'opérateur `<<`, on peut écrire une chaîne de caractères à l'écran.
  - L'instruction `cout << "Bonjour ECL";` affiche donc le message *Bonjour ECL !* à l'écran.

ÉCOLE CENTRALE LYON

Exemple d'un programme C++

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

int main()
{
     cout << "Bonjour ECL !" << endl;
     return 0;
}
```

- Explications (5/5) :
  - L'instruction `return 0;` indique que la fonction `main` est terminée et que tout s'est bien passé.

ÉCOLE CENTRALE LYON

Ecriture d'un programme

- Commentaires :
  - Multi lignes : texte encadré par les caractères `/*` et `*/`
  - Sur une seule ligne : texte introduit par le symbole `//`

```
/* Un commentaire écrit
sur deux lignes */

cout << "Hello" << endl; // Commentaire écrit sur une seule ligne
```

- Instructions :
  - Une instruction C++ peut s'écrire sur plusieurs lignes et se termine par le caractère `;`

```
int x, y;
x = 3 +
2 - 7; y = 32;
```

Types de données

- La notion de type de donnée s’applique à une variable, une constante ou une fonction.
- Un type de donnée caractérise :
  - Les valeurs que peut prendre la donnée.
  - Les opérations applicables à cette donnée.
  - La taille et le codage de la donnée.

Types de données

- Le fait d'indiquer un type pour les données permet d'effectuer des vérifications à la compilation et éventuellement à l'exécution.
- Selon le niveau de vérification effectuée, on a un langage dit faiblement ou fortement typé.
- Le langage C++ est un langage fortement typé avec des vérifications lors de la compilation.

Types de données

- Types de bases / Types construits
  - En C++ on distingue les types de base ou type prédéfinis et les types construits généralement structurés.
  - Les types de base sont des types scalaires : une variable ne correspond qu'à une seule information.
  - Les types construits sont des types structurés : une variable correspond à une collection d'informations.
  - Les types de base correspondent à des informations entières, réelles, caractères et booléennes.

Types de données

- Principaux types de base :
  - Entier (int, long)
  - Caractère (char)
  - Booléen (bool)
  - Réel (float, double)

Variables

- Une variable est un emplacement mémoire défini par un nom (identificateur).
- Un variable est obligatoirement typée en C++ ce qui permet certains contrôles par le compilateur de l'utilisation des variables.
- Une variable doit être définie avant toute utilisation.

Variables

- Syntaxe de définition de variables :

```
type nom_de_variable;
type nom_de_variable = valeur_initiale;
```

- Exemples :

```
int x;                // x variable entière à valeur inconnue
double y=3.14;        // y variable réelle initialisée à 3.14
int u=2, v, w=6;      // u entier valant 2, v entier a valeur
                      // inconnue, w entier valant 6
bool b = true;        // b variable booléenne initialisée à true
```

ÉCOLE CENTRALE LYON

Opérateurs

- Opérateurs arithmétiques : `+, -, *, /, %, ++, --`

```
1.0/4.0    → 0.25          opération effectuée en réel
1/4        → 0             opération effectuée en entier
```

```
11%3 →  2       le reste de la division entière de 11 par 3 vaut 2
```

```
i = 3;
i++;                        // i vaut maintenant 4
i--;                        // I vaut de nouveau 3
```

- Opérateurs de relation : `<, >, <=, >=, ==, !=`

```
if(a==b)x=3;       // si a est égal à b alors x prend la valeur 3
if(x!=0)y=8;       // si x est different de 0 alors y vaut 8
```

Opérateurs

- Opérateurs logiques :
  - && : ET logique
  - || : OU inclusif logique
  - ! : Négation logique

```
if(a>3 && b!=0) . . . . .        // si a>3 ET b≠0 alors . . . . . .
```

- Opérateur d'affectation :

```
x=5; z=2;                        // x vaut 5, z vaut 2
toto=3*x-z;                      // toto vaut 13
```

ÉCOLE CENTRALE LYON

Expressions et instructions

- Expressions :
  - Ensembles ordonnés d'opérateurs, de constantes et de variables : 3*x+toto/5

- Instructions : 3 formes d'instructions
  - Instructions expressions
  - Instructions de contrôle
  - Instructions bloc

Instructions

- Instructions expressions : expressions terminées par ;

```
y=2*x+5;                          // instruction expression
```

- Instructions de contrôle :
  - Modifient l'enchaînement des instructions d'un programme (choix, itération, saut).
  - Correspondent en général aux instructions if, switch, for, while, break …

Instructions

- Instructions bloc :
  - Groupement de plusieurs instructions de manière à ce quelles ne forment qu'une seule instruction.
  - On utilise cette technique quand la syntaxe exige une seule instruction.
  - Le groupe d'instructions est encadré d'accolades pour qu'il soit considéré comme une seule instruction.

```
if(a>b)x=2;          // x=2 ; instruction unique contrôlée par le if
```

```
if(a>b)              // plusieurs instructions contrôlées par le if
{                    // qui ne peut en contrôler qu'une seule
    x=2;
    y=3;
}
```

ÉCOLE CENTRALE LYON

Portée des variables

- Une variable définie à l'intérieur d'une paire d'accolades { } est dite locale, elle n'est accessible que dans le bloc { } où elle est définie.
- Une variable définie à l'extérieur de toutes accolades, donc de toutes fonctions est dite globale, elle est accessible par toutes les fonctions définies dans le même fichier, après l'emplacement de la définition de la variable.

Entrées / sorties de base

- Utilisation de la bibliothèque `iostream` contenant les variables et opérateurs d'entrée / sortie

- Variables :
  - `cin` : flot d'entrée par défaut (clavier)
  - `cout` : flot de sortie par défaut (écran)
- Opérateurs :
  - `>>` : opérateur de lecture
  - `<<` : opérateur d'écriture

Entrées / sorties de base

- Exemple

```
#include <iostream> // Inclusion de la bibliothèque d’E/S
using namespace std; // Utilisation de l’espace de nom standard

int main()
{
     int x=2; double y=3.14;
     cout << "Bonjour"; // écriture d’une constante chaine de caractères
     cout << 'A'; // écriture d’une constante caractère

     cout << x; // écriture d’une variable entière
     cout << y; // écriture d’une variable réelle

     cin >> x; // lecture d’un nombre au clavier, stockage dans x
     cin >> y; // lecture d’un nombre au clavier, stockage dans y

     return 0;
}
```

Entrées / sorties de base

- Enchaînement d'entrées / sorties
  - Si on doit écrire plusieurs informations sur l'écran, il n'est pas nécessaire d'utiliser plusieurs instructions d'écriture, on peut utiliser une seule instruction enchainant plusieurs écritures.

```
cout<<"Resultat : "<<x<< "  "<<y ;   // enchaînement de 3 écritures
```

  - Même principe pour la lecture

```
int x,y ;
cin>>x>>y ;       // lecture de deux nombres, 1er stocké dans x,
                  // 2eme stocké dans y
```

  - Retour à la ligne : endl

```
cout<<x<<endl;       // écrit la valeur de x puis passe à la ligne
```

Instructions de contrôle

- Choix simple :
  - Syntaxe : if (condition) instruction
  - Exemple :

```
if (a > b) x = a - b;
```

  - Remarque : une instruction `if` ne contrôle qu'une seule instruction. S'il y a plusieurs instructions à contrôler, il faut utiliser un bloc `{ }`

```
if (a > b)
{
    x = a - b;
    y = a * b;
    z = x + y;
}
```

Instructions de contrôle

- Alternative :
  - Syntaxe : if (condition) instruction1 else instruction2
  - Exemple :

```
if (a > b)
     x = a - b;
else
     x = b - a;
```

  - Remarque : une instruction `if else` ne contrôle qu'une seule instruction. S'il y a plusieurs instructions à contrôler, il faut utiliser un bloc `{ }`

ÉCOLE CENTRALE LYON

Instructions de contrôle

- Itérations
  - Les instructions d'itération permettent de faire des traitements répétitifs sous la forme de boucle.
  - Il existe trois formes de boucle en C++, certaines sont plus adaptées que d'autres à des situations particulières mais on peut toujours réaliser une boucle avec l'une quelconque des trois formes.
- Trois formes classiques de boucles :
  - tant que (condition) instruction
  - répéter instruction tant que (condition)
  - pour variable dans ensemble_de_valeurs instruction

Instructions de contrôle

- Boucle `while` (tant que)
  - Syntaxe :

```
while(condition)instruction
```

  - Exemple :

```
int s, i, n;
s = 0, i = 0;
cin >> n;
while (i < n)
{
     i = i + 1;
     s = s + i;
}
```

  - Remarque : la condition est toujours évaluée avant chaque itération

Instructions de contrôle

- Boucle `do while` (répéter tant que)
  - Syntaxe :

```
do instruction while(condition);
```

  - Exemple :

```
int s, i, n;
s = 0, i = 0;
cin >> n;
do
{
     i = i + 1;
     s = s + i;
}
while (i < n);
```

  - Remarque : la condition est évaluée après chaque itération

Instructions de contrôle

- Boucle `for` (pour variable dans ensemble)
  - Une boucle `for` contrôle le nombre d'itérations avec une variable qui est initialisée au début puis vérifiée et incrémentée à chaque itération.

```
for(expr1;expr2;expr3)instruction
                                        // expr1 = initialisation
                                        // expr2 = condition de maintien
                                        //expr3 = incrémentation
```

  - Remarques :
    - expr1, expr2 et expr3 sont des expressions C++
    - instruction est une instruction C++ quelconque (de forme bloc { } si on veut en avoir plusieurs)
    - La condition est évaluée avant chaque itération

Instructions de contrôle

- Boucle `for` (pour variable dans ensemble)
  - Exemple :

```
int s, p, i, n;
s = 0;
p = 1;
cin >> n;
for (i=0; i < n; i++)
{
    s = s + i;
    p = p * i;
}
```

Instructions de contrôle

- Choix de boucle :
  - Si le nombre d'itérations est connu a priori , alors on choisit une boucle `for` (`for( i = 0; i<10; i=i+1) ...`)
  - Si le nombre d'itérations est inconnu a priori, et qu'il y a une possibilité de nombre d'itérations nul, alors on choisit une boucle `while`
  - Une boucle `for` avec expr1 et expr3 vides peut toujours remplacer une boucle `while`
  - La forme `do while` de boucle est superflue, mais elle est utilisable si son expression est plus naturelle pour le programmeur.

Types de données structurés

- Les types de base sont des types scalaires prédéfinis (int, double, char, bool, etc.) mais l'utilisateur a la possibilité de définir des nouveaux types pour créer ensuite des variables de ce type.
- Les types définis par l'utilisateur sont généralement des types non scalaires (type structurés) qui correspondent à la représentation d'une collection d'informations.
- Quelques types structurés :
  - Structures (`struct`)
  - Vecteurs (`vector`)
  - Chaînes de caractères (`string`)

Type Structure

- Une structure (`struct`) est un type de données non scalaire, hétérogène (les composants d'une structure sont de types différents) et statique.
- Les composants d'une structure s'appellent les membres de la structure.
- Chaque composant est accessible à partir de son nom qui correspond à la distance du membre depuis le début de la structure.

Type Structure

- Définition d’un type structure :

```
struct nom_type
{
    type_membre nom_membre;
     - - - - - - - - - -
     type_membre nom_membre;
};
```

```
struct Article
{
        string nom;
        double prix;;
};
```

- Définition d’une variable de type `struct`

```
struct Article      // définition du type Article
{
    string nom;
    double prix;
};
Article a;          // définition d'une variable a de type Article
```

Type Structure

- Accès aux membres :
  - Syntaxe : nom_variable.nom_membre

```
Article a;                    // a variable de type Article
a.nom="crayon";               // modification du membre nom
a.prix=1.1;                   // modification du membre prix
```

Type Vector

- Un vecteur (`vector`) est un type de donnée non scalaire, homogène (les composants d'une structure sont de même type) et dynamique (sa taille peut changer au cours de l'exécution du programme).
- Ils sont introduits par la librairie standard du C++ (STL).
- Définition d'une variable de type `vector` pour des entiers :

```
#include <vector>
using namespace std;
int main()
{
    vector<int> v(20); // v variable vecteur contenant initialement
                       // 20 éléments entiers mais dont la taille pourra
                       // changer pendant l'exécution du programme
    ...
}
```

Type Vector

- Accès à un élément d'un vecteur :

```
vector<int> v(20);
v[0] = 7; // le premier élément de v vaut 7
v[7] = 12;
v[19] = 8; // le dernier élément de v vaut 8
```

- Accroissement de la taille d'un vecteur :

```
vector<double> v;
v.push_back(2.1);
v.push_back(18.7);  // v contient maintenant 2 éléments
```

- Accès à la taille d'un vecteur :

```
vector<double> v(50);
int taille = v.size(); // taille vaut 50
```

Type Vector

- Vecteur à plusieurs dimensions :
  - Il est possible de définir des variables `vector` à plusieurs indices que l'on utilise avec autant de paires de crochets ([ ]) qu'il y a d'indices.
  - Exemple :

```
int m = 3, n = 4;
vector<int> ligne(n,0);
vector<vector<int>> mat(m,ligne);
mat[1][3] = 7;
for (int i=0; i<m; i++)
{
     for (int j=0; j<n; j++)
          {
          cout << mat[i][j] << " ";
     }
     cout << endl;
}
```

Résultat →

```
0 0 0 0
0 0 0 7
0 0 0 0
```

Type string

- On est souvent conduit à manipuler par programme des chaînes de caractères pour représenter des informations textuelles (nom, prénom, adresse, …).
- La bibliothèque STL de C++ propose pour cela le type `string`
- Pour l'utiliser, il faut inclure le fichier bibliothèque `string` et annoncer l'utilisation du préfixe par défaut `std` pour

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

Type string

- Définition de variables `string` :

```
string s;                   // s variable string vide
string s1 = "bonjour";     // s2 variable string contenant "bonjour"
```

- Affectation de chaînes :

```
string s1 = "truc";
string s2 = "machin";
s1 = s2;                                   // s1 contient "machin"
```

- Comparaison de chaînes :

```
string s1 = "abracadabra";
string s2 = "zut";
if(s2>s1) cout<<"zut est superieur a abracadabra"<<endl;
```

Type string

- Opérateur d’accès à un caractère d’une chaîne :

```
string s1("bonjour");
cout<<s1[2]<<endl;                     // ecrit n
```

- Taille d’une chaîne :

```
string s1("bonjour");
cout<<"taille de s1 : "<<s1.size()<<endl;              // donne 7
```

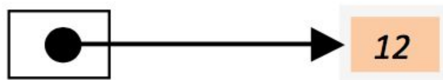
Lecture et écriture globale de chaînes :

```
string s1, s2 = "salut";
cin >> s1;
```

Type pointeur

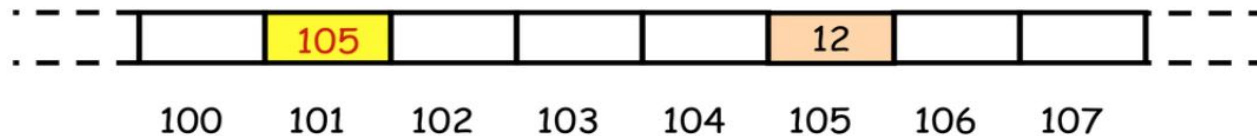
- La valeur d'une variable pointeur est l'adresse d'une autre information

Notation graphique d'un pointeur



pointeur *information pointée*

Représentation en mémoire



La variable pointeur est implantée dans la case mémoire 101, elle contient la valeur 105 qui est l'adresse de la case qui contient 12

Type pointeur

- Définition d'une variable pointeur
  - Syntaxe :

```
type_pointé * variable;
```

  - Exemple :

```
int *p;                     // p pointeur sur int
double *x,*y,z;             // x et y sont des pointeurs sur double
                            // z est un double
```

- Opérateurs concernant les pointeurs :

```
&       fournit l'adresse d'une variable
*       fournit l'élément pointé par une variable pointeur
```

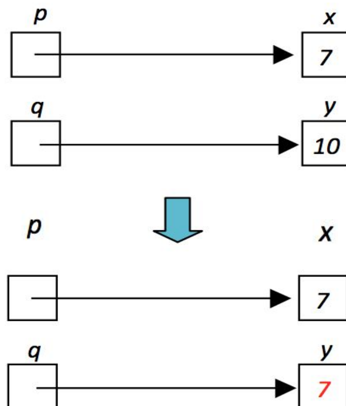
Type pointeur

- Exemple

```
int *pt;                    // pt pointeur sur int
int x = 3, y;               // élément pointé inconnu
pt = &x;                    // pt pointe sur x
y = *pt;                    // y vaut 3 (info pointée par pt)
pt = 0;                     // pt ne pointe sur rien
```

Type pointeur

- Utilisation des pointeurs




```
int x,y :          // x et y int
int *p,*q ;        // p et q pointeurs sur int
x = 7 ;            // p et p pointent sur
y = 10 ;           // n'importe quoi
p = &x ;           // p pointe sur x
q = &y ;           // q pointe sur y
*q = *p;           // elt pointé par q prend
                   // valeur de elt pointé par
                   // p
```

Exemple d'implantation en mémoire

Adresse	Contenu	Variable
1000	7	x
1001	~~10~~ 7	y
1002		
1003		
1004	1000	p
1005	1001	q
1006		

Allocation dynamique

- Il existe une zone de mémoire gérée par le système d’exploitation pour fournir de la place aux structures de données dynamique qui doivent pouvoir s’agrandir.
- Cette « réserve d’octets » permet au système de prêter de l’espace mémoire temporairement et de le récupérer plus tard pour pouvoir le prêter à d’autres programmes.
- Il existe donc un mécanisme géré par le système pour gérer ce prêt de mémoire.

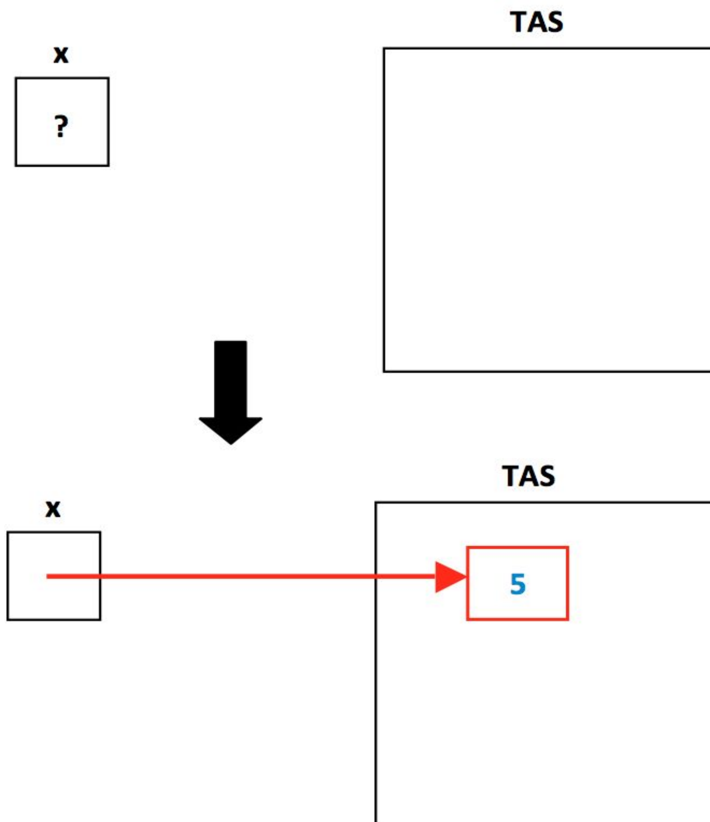
ÉCOLE CENTRALE LYON

Allocation dynamique

- Du point de vue des données, le système dispose d'une part de la zone « réserve d'octets » appelée tas (heap) et d'autre part d'une table qui permet de noter à quel programme a été prêté une zone de mémoire.
- Le système fournit des fonctions d'emprunt et de restitution
  - Emprunt : opérateur `new`
  - Restitution : opérateur `delete`
- A la fin de l'exécution d'un programme, il y a restitution automatique de l'espace emprunté par ce programme

Allocation dynamique

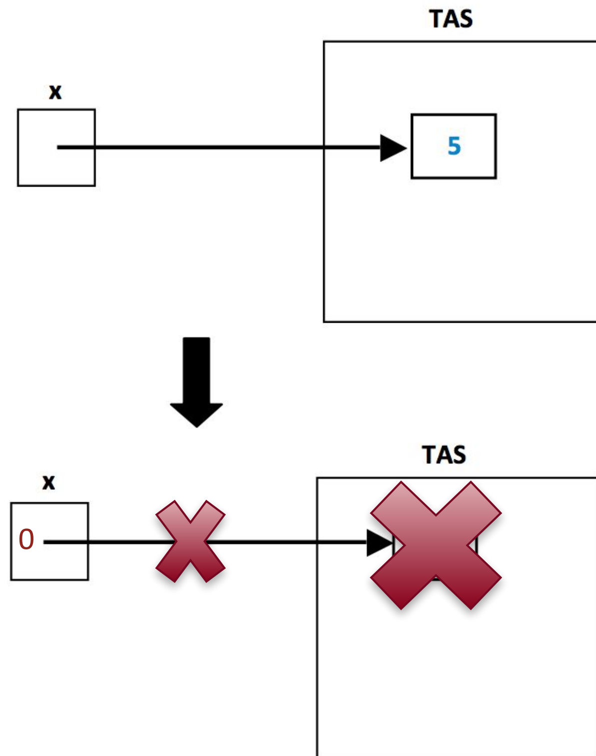
- Exemple de réservation de place pour stocker un int




```
int * x ;        // x pointeur sur int
x = new int ;    // réservation de place pour un int
*x = 5 ;         // dans le TAS
                 // rangement de la valeur 5 dans
                 //l'espace réservé
```

Allocation dynamique

- Exemple de libération d'espace désigné par un pointeur




```
delete x ;
// la zone dans le tas est toujours pointée par x, mais
// elle est notée comme libre dans la table de
// réservation. Elle peut être affectée à un autre
// programme
// x ne doit plus utiliser la zone, sous peine d'erreurs

x = 0 ;
// x ne pointe sur rien, la zone est inaccessible pour x
```

Allocation dynamique

- Les tableaux dynamiques
  - Syntaxe :
    - Allocation :

```
type_pointé *pointeur = new type_pointé[taille];
```

    - Libération

```
delete[] pointeur;
```

Allocation dynamique

- Les tableaux dynamiques
  - Exemple :

```
#include <iostream>
using namespace std;
int main()
{
     int i, taille;
     cout << "Tapez la valeur de taille : ";
     cin >> taille;
     int *t;
     t = new int[taille];
     for (i = 0; i < taille; i++)
          t[i] = i * i;
     for (i = 0; i < taille; i++)
          cout << t[i] << endl;
     delete[] t;
     return 0;
}
```

Résultat →

```
Tapez la valeur
de taille : 4
0
1
4
9
```

ÉCOLE CENTRALE LYON

Fonctions

- Dans un programme, les sous-programmes permettent d'enregistrer et d'exécuter des séquences de codes paramétrables.
- En C++, les sous programmes sont appelés fonctions

Fonctions

- La définition d'une fonction consiste à fournir :
  - la définition du type de valeur de retour
  - la définition du nom de la fonction
  - la définition des arguments typés de la fonction
  - la définition du traitement réalisé par la fonction (instructions)
  - une instruction éventuelle de renvoi de la valeur de retour (`return`)

Fonctions

- Définition d'une fonction : exemple

```
double moyenne(int x, int y)          // x, y arguments formels
{
   double m;
   m = (x + y)/2.0;
   return m;                          // renvoi de la valeur de retour
}
```

- `double` est le type de la valeur de retour de la fonction `moyenne`
- Les arguments formels sont `x` et `y`, de type `int`, ils sont passés par copie (par défaut)
- `m` est une variable locale à la fonction (utilisable seulement à l'intérieur de la fonction)
- Le traitement réalisé par la fonction est défini entre `{ }`
- Cette fonction indique qu'elle renvoie un `double`, ce qui correspond à l'instruction `return m ;`

Fonctions

- Appel d'une fonction

```
int i=5, j=7;
double z;
z = 3*moyenne(i,j); // i,j arguments d'appel
```

- Remarque : les arguments formels `x` et `y` sont remplacés lors de l'appel par les arguments réels `i` et `j`.

Fonctions

- Fonction sans valeur de retour :
  - Une fonction peut effectuer un travail sans renvoyer une valeur, c'est le cas, par exemple, d'une fonction qui affiche une valeur sur l'écran.
  - Dans ce cas, on indique que le type de retour est `void` (vide), dans la définition et il n'y a pas d'instruction `return`.

```
void affiche(int x)
{
    cout<< "x vaut : "<<x<<endl;
}
```

Fonctions

- Passage d'arguments :
  - Par copie (passage par défaut)
  - Par adresse (utilisation de pointeurs)
  - Par référence

Fonctions

- Passage d'arguments par copie
  - C'est le mode de passage par défaut
  - La fonction ne peut pas modifier les arguments
  - Elle ne peut modifier que des copies qui disparaissent au retour de la fonction

```
int f(int x)
{
    x=2*x;
    return 10+x;
}
. . . . . . . .
int i=5,j;
j=f(i);
cout<<j<<endl;                          // j vaut 20, calculé, renvoyé
cout<<i<<endl;                          // i vaut 5, inchangé
```

Fonctions

- Passage d’arguments par adresse
  - Les arguments sont des pointeurs
  - La fonction peut donc accéder aux valeurs désignées par les pointeurs, et donc les modifier

```
void triple(int *px)
{
     *px = *px * 3;
}
...
int i = 5;
int *pi = &i;
triple(pi);
cout << i << endl; // i vaut 15
```

Fonctions

- Passage d'arguments par référence
  - Les arguments sont des références (utilisant le symbole &)
  - La fonction peut accéder aux valeurs des arguments d'entrée et donc les modifier
  - le mode de passage se note dans la définition de la fonction, l'appel de la fonction est noté avec la même syntaxe que dans un passage par copie

```
void triple(int &x)
{
    x = x * 3;
}
...
int i = 5;
triple(i);
cout << i << endl; // i vaut 15
```

Fonctions

- Définition avant appel
  - Une fonction doit être connue par le compilateur avant que l'on puisse l'appeler.
  - Ceci conduit à placer la définition d'une fonction avant la définition d'une fonction qui l'appelle.

```
double moyenne(int x, int y)          // definition de la fonction
{ return (x+y)/2.0;}

int main()
{
int i = 4,j=8,k;
k = moyenne(i,j);                     // appel de la fonction
cout<<k<<endl;
return 0;
}
```

Fonctions

- Utilisation de déclarations de fonctions
  - Dans le cas d'un programme constitué de nombreuses fonctions, il est judicieux de fournir d'abord les déclarations de toutes les fonctions, dans l'ordre alphabétique.
  - Ensuite il faut fournir la liste des définitions de fonctions, par ordre alphabétique.

```
double moyenne(int x, int y);  // déclaration de la fonction moyenne
```

  - Une déclaration s'appelle aussi prototype ou en tête de la fonction

Fonctions

- Utilisation de déclarations de fonctions : exemple

```
double moyenne(int int);          // déclaration de la fonction moyenne
int main()
{
int i = 4,j=8,k;
k = moyenne(i,j);                 // appel de la fonction
cout<<k<<endl;
return 0;
}
double moyenne(int x, int y)      // definition de la fonction
{return (x+y)/2.0;}
```

ÉCOLE
CENTRALE LYON

36 av. Guy de Collongue
69134 Écully cedex
T + 33 (0)4 72 18 60 00
www.ec-lyon.fr