Programmation Orientée Objet sous C++

Med. AMNAI

Filière SMI - S5

Département d'Informatique

10 novembre 2020

Plan

Plan

Plan

Plan

Plan

Plan

Objectifs du cours

- Acquirir la syntaxe de base du langage C++
- Acquirir les concepts de base de la Programmation Orientée Objet en faisant appel au langage de programmation selon la bibliothéque standard de C++.

Historique du C++

Le Langage C++

- Héritier du Langage C : Le C a été inventé au cours de l'année **1972** dans les laboratoires Bell par Dennis Ritchie et Ken Thompson.
- Début en **1983** : Bjarne Stroustrup développa le C++ afin d'améliorer le C, en lui ajoutant des « classes ».
- En **1998** que le C++ fut normalisé pour la première fois. Une autre norme corrigée fut adoptée en **2003**.

Présentation du C++

Le Langage C++

- Le C++ est une surcouche de C, offrant des possibilités de la POO avec quelques incompatibilités de syntaxe
- Le C Séquentiel : les instructions se suivent et dans un ordre précis et transmises au processeur de la machine dans cet ordre.
- C++ est un langage multiparadigmes. Il respecte à la fois le paradigme objet et séquentiel du langage C.

Qu'est-ce qu'un programme

Programmer le matériel d'un ordinateur, c'est lui fournir :

- Une série d'**instructions** qu'il doit exécuter. Ces instructions sont généralement écrites dans un langage de programmation, puis ;
- Sont **traduites** (avant d'être exécutées) en langage machine (qui est le langage du microprocesseur). Cette traduction s'appelle **compilation** ;

La traduction automatique implique certaines contraintes pour le programmeur :

- Ecrire les instructions en respéctant la syntaxe du langage utilisé,
- Déclarer les données et fonctions qu'il va utiliser .

Organisation d'un programme en C++

Un programme écrit en C++ se compose généralement de plusieurs fichiers-sources. Il y a deux sortes de fichiers sources :

- Ceux qui contiennent effectivement des **instructions** ; leur nom possède l'extension **.cpp** ;
- Ceux qui ne contiennent que des **déclarations** ; leur nom possède l'extension **.h** (signifiant "**header**" ou "**en-tête**"). ;

Remarque : Un fichier **.h** sert à regrouper des déclarations qui sont communes à plusieurs fichiers **.cpp**. Dans un fichier **.cpp** on prévoit l'inclusion automatique des fichiers **.h** qui lui sont néecessaires, grâce aux directives de compilation **#include**.

Organisation d'un programme : Exemple

```
#include<iostream>

using namespace std ;

int main ( ) {

cout << "hello world !" << endl ;

}
```

Démo : Hello.cpp

Définition de Variables

Une variable, comme son nom l'indique, est un espace mémoire dont le contenu peut varier au cours de l'exécution. Elle est atachée à une adresse indiquant où se trouve sa valeur.

- Toute variable doit être définie avant dêtre utilisée ;
- Une définition peut apparaître n'importe où dans un programme.
- Une variable définie en dehors de toute fonction – et de tout espace de nom – est une variable globale).
- Une variable est caractérisée par :
  - son **Nom** : mot composé de lettres ou chiffres, commençant par une lettre,
  - son **Type** précisant la nature de cette variable (nombre entier, caractère, objet etc.),
  - sa **Valeur** qui peut être modifiée à tout instant.

Types de base

1. **vide** : **Void** . Aucune variable ne peut être de ce type.
2. **entiers**, par taille-mémoire croissante :
   - **char**, stocké sur **1** octet ; valeurs (-128 ... 127),
   - **short**, stocké sur **2** octets ; valeurs (-32768 à 32767),
   - **long**, stocké sur **4** octets ; valeurs de -2^{31} à $2^{31} - 1$,
   - **int**, coincide avec **short** ou **long**, selon l'installation.
3. **réels**, par taille-mémoire croissante :
   - **float**, stocké sur **4** octets ; précision : environ 7 chiffres,
   - **double**, stocké sur **8** octets ; précision : environ 15 chiffres,
   - **long double**, stocké sur **10** octets ; précision : environ 18 chiffres.

Valeurs explicites

- Les caractères entre apostrophes, correspondant à des entiers de type char. Par exemple : **'A' (= 65), 'a' (= 97), '0' (= 48), ' ' (= 32). (les codes ASCII).**
- chaînes de caractères entre guillemets, pour les affichages. Par exemple : "Au revoir !".
- valeurs entières, par exemple : **123** , **-25000** , **133000000** (ici respectivement de type char, short, long).

Exemple

```cpp
#include <iostream>

int main()

{
    char a = 'x';
    short b = 1 ;
    long c = 2 ;
    int d = 3 ;
    float e = 3.14 ;
    double f = 3.14 ;
    long double g = 3.14;


    std::cout << " char " << sizeof(a) << ": " << a << std::endl;
    std::cout << " short " << sizeof(b) << ": " << b << std::endl;
    std::cout << " long " << sizeof(c) << ": " << c << std::endl;
    std::cout << " int " << sizeof(d) << ": " << d << std::endl;
    std::cout << " float " << sizeof(e) << ": " << e << std::endl;
    std::cout << " double " << sizeof(f) << ": " << f << std::endl;
    std::cout << " long double " << sizeof(g) << ": " << g << std::endl;
}
```

Démo : sizeof.cpp

Déclaration des variables

- **Syntaxe** : <type> <liste de variables> ;
  - *<type>* est un nom de type ou de classe,
  - *<liste de variables>* est un ou plusieurs noms de variables, séparés par des virgules.
- **Exemples** :
  - int i, j, k ; // déclare trois entiers i, j, k
  - float x, y ; // déclare deux réels x, y

Déclaration et Initialisation

- En même temps qu'on déclare une variable, il est possible de lui attribuer une valeur initiale.
  - `int i, j = 0, k;`
  - `float x, y = 1.0;`
- On peut déclarer une constante en ajoutant const devant le nom du type :
  - `const double PI = 3.14159265358979323846;`
  - `const int MAX = 100;`
- Les valeurs des constantes ne peuvent pas être modifiées.

Chaînes de caractères

Il existe une **classe string**, ce n'est un pas un type élémentaire.
Pour l'utiliser, il faut placer en tête du fichier :
`# include <string>`

- `string t ;` définit une chaine vide ;
- `string s(25,'a') ;`
- `string mot = "bonjour" ;`
- `s.size()` représente la longueur de *s*
- `s[i]` est le i-ème caractère de **s (i = 0,1,. . . s.size()-1)**
- `s+t` est une nouvelle chaîne correspondant à la *concaténation* de *s* et *t*.

Tableaux

Pour utiliser la **classe vector**, il faut placer en tête du fichier :

```
# include <vector>
```

- Exemples :
  - vector<int> Tab(100,5) ; //vecteur de 100 éléments valant 5 v.size() = 5
  - vector<int> Tab(50) ; //vecteur de 50 éléments
  - vector<double> T ;
- Structure générale : **vector< type > Nom(n,v)**
  - vector< type > Nom1 = Nom2 ; les valeurs de *Nom2* sont alors recopiées dans *Nom1*.
  - T.size() représente la taille de *T*. **size()** renvoie en fait un entier non signé, son type exact est **vector<type> : :size type**

Exemple

```
#include <iostream>
/*
Déclaration de la fonction somme
*/

double somme(double a, double b);

int main()
{
    int a, b;
    std::cout << "Donnez deux entiers" << std::endl;
    std::cin >> a >> b;
    std::cout << a << " + " << b << "=" << somme(a, b) << std::endl;
    return 0;
}

// somme retourne la somme de a et b deux réels fournis en paramétres.
double somme(double a , double b)
{
return a+b;
}
```

Démo : exp3.cpp

Exemple

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

int main () {
   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   str3 = str1;  // copy str1 into str3
   cout << "str3 : " << str3 << endl;
   cout << "str2 : " << str2 << endl;

   str3 = str1 + str2; // concatenates str1 and str2
   cout << "str1 + str2 : " << str3 << endl;

   len = str3.size(); // total length of str3 after concatenation
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

Démo : string.cpp

Définition, Opérateurs arithmétiques

Définition : En combinant des noms de variables, des opérateurs, des parenthèses et des appels de fonctions, on obtient des expressions. En C++, on appelle expression tout ce qui a une valeur.

- **Opérateurs arithmétiques**
  - + : addition ;
  - - : soustraction ;
  - * : multiplication ;
  - / : division. Attention : entre deux entiers, donne le quotient entier.
  - % : entre deux entiers, donne le reste modulo. ;
- **Exemples** :
  - 19.0 / 5.0 vaut 3.8,
  - 19 / 5 vaut 3,
  - 19 % 5 vaut 4.

Les règles de priorité

- Dans les expressions, les règles de priorité sont les règles usuelles des mathématiciens. Par exemple, l'expression 5 + 3 * 2 a pour valeur 11 et non 16. En cas de doute, il est conseillé de mettre des parenthèses.
- ++ : incrémentation. Si **i** est de type entier, les expressions **i**++ et ++**i** ont toutes deux pour valeur la valeur de **i**. Mais elles ont également un effet de bord qui est :
  - pour la première, d'ajouter ensuite 1 à la valeur de i (post-incrémentation),
  - pour la seconde, d'ajouter d'abord 1 à la valeur de i (pré-incrémentation).
- – : décrémentation. **i**– et –**i** fonctionnent comme **i**++ et ++**i**, mais retranchent 1 à la valeur de i au lieu d'ajouter 1.

Opérateurs d'affectation

- **= (égal)** : sous forme de *<variable> = <expression>*
  l'<expression> est d'abord évaluée ; cette valeur donne la valeur de l'expression d'affectation, la <variable> reçcoit ensuite cette valeur.
  - **Exemples** : i = 1
  - i = j = k = 1 (vaut 1 et donne à i, j et k la valeur 1)

- **+=, -=, *=, /=, %=**
  sous forme de *<variable> <opérateur>= < expression>*
  - L'expression est équivalente à :
    <variable> = <variable> <opérateur> <expression>
  - **Exemple**, l'expression i += 3 équivaut à i = i + 3.

Conversion de type

Une expression d'affectation peut provoquer une conversion de type.

- **Exemple** : supposons déclarés int i ; float x ;
- Alors, si **i** vaut **3**, l'expression **x = i** donne à **x** la valeur **3.0** (*conversion entier -> réel*).
- Inversement, si **x** vaut **4.21**, l'expression **i = x** donne à **i** la valeur **4**, partie entière de **x** (*conversion réel -> entier*).
- On peut également provoquer une conversion de type grâce à l'opérateur **()** (*type casting*). Avec **x** comme ci-dessus, l'expression **(int)x** est de type **int** et sa valeur est la partie entière de **x**.

Opérateurs d'entrées-sorties

Ce sont les opérateurs « et », utilisés en conjonction avec des objets prédéfinis **cout** et **cin** déclarés dans <**iostream.h**> (*ne pas oublier la directive #include <iostream.h> en début de fichier*).

- **Formes** :
  - **cout « <expression>** : affichage à l'écran la valeur de *<expression>*,
  - **cin » <variable>** : lecture au clavier la valeur de *<variable>*.

Opérateurs d'entrées-sorties : Afficher à l'écran

- **Syntaxe** : cout « expr1 « . . . « exprn ;
  - Cette instruction affiche *expr1* puis *expr2*. . .
  - Afficher un saut de ligne se fait au moyen de **cout « endl**.
  - Exemple :
    - int i=45 ;
    - cout « "la valeur de i est " « i « endl ;

Opérateurs d'entrées-sorties : Lire au clavier

- **Syntaxe** : cin > > var1 > > . . . > > varn ; ;
  - Cette instruction lit (au clavier) des valeurs et les affecte à *var1* puis *var2* . . .
  - Les caractères tapés au clavier sont enregistrés dans le buffer. Les espaces, les tabulations et les fins de lignes sont des séparateurs.

Exemple

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


int main()
{
    int a;
    a = 0;
    cout << "a vaut : " << a << endl;
    a = 5;
    cout << "a vaut à present : " << a << endl;

}
```

Démo : exp1.cpp

Exemple

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

void test(int j)
{
    cout<<j<< endl ;
}

int main()
{
    int i =20 ;
    cout << "bonjour" << endl ;
    test(i) ;
}
```

Démo : exp2.cpp

Exemple

```
#include <iostream>
/*
Déclaration de la fonction somme
*/

double somme(double a, double b);

int main()
{
    int a, b;
    std::cout << "Donnez deux entiers" << std::endl;
    std::cin >> a >> b;
    std::cout << a << " + " << b << "=" << somme(a, b) << std::endl;
    return 0;
}

// somme retourne la somme de a et b deux réels fournis en paramétres.
double somme(double a , double b)
{
return a+b;
}
```

Démo : exp3.cpp

Structures de contrôle : IF, FOR, While, Do..While

- **Syntaxe** :
- **if** (*expr*) *instr*
- **if** (*expr*) *instr1* **else** *instr2*
- **for** (*expr1* ;*expr2* ;*expr3*) *instr*
- **while** (*expr*) *instr*
- **do** *instr* **while** (*expr*) ;

Définition de fonction

type nom(liste des arguments) corps **Syntaxe** :

- *type* est le type du résultat de la fonction.(void si il s'agit d'une procédure)
- *La liste des arguments* (paramètres formels) : *type1 p1, ..., typen pn*
- Le *corps* décrit les instructions à effectuer. Le corps utilise ses propres variables locales, les éventuelles variables globales et les paramètres formels.
- Si une fonction renvoie un résultat, il doit y avoir au moins une instruction return expr ; (return le cas d'une procédure).

Exemple

```
#include<iostream>
#include<vector>
using namespace std;
vector<int> SaisieTab(){
    int taille ;
    cout << " Entrer une taille : " ;
    cin >> taille ;
    vector<int> res(taille,0) ;

    for (int i=0 ; i< taille ; i++) {
    cout << " val = " ;
    cin >> res[i] ;
}
return res ;
}
```

```
void AfficherTab(vector<int> T){
    for (int i=0 ; i< T.size() ; i++)
    cout << T[i] << " " ;
}

int main(){
    vector<int> tab;
    tab=SaisieTab();
    AfficherTab(tab);
}
```

Démo : fct1.cpp

Passage par valeur

Par défaut, les paramètres d'une fonction sont initialisés par une copie des valeurs des paramètres réels.

Exemple

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

void permut(int a,int b){
    int aux ;
    cout<<" fct  : avant a = "<< a << " b = "<< b << endl;
    aux = b ;
    b = a ;
    a = aux ;
    cout<<" fct  : apres a = "<< a << " b = "<< b << endl;
}
int main() {
    int x,y ;
    x= 10 ;
    y= 20;
    cout<<" main : avant x = "<< x << " y = "<< y << endl;
    permut(x,y) ;
    cout<<" main : apres x = "<< x << " y = "<< y << endl;
}
```

Démo : parvaleur.cpp

Passage par adresse

- Pour modifier la valeur d'un paramètre réel dans une fonction, il faut passer ce paramètre par **adresse (&)**.
- Une référence sur une variable est un synonyme (par copie d'adresse) de cette variable, c'est-à-dire une autre manière de désigner le même emplacement de la mémoire.

Exemple

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

void permut(int* a,int* b){
    int aux ;
    cout<<" fct  : avant &a : "<< &a << " " << *a << " &b : " << &b << " " << *b << endl;
    aux = *b ;
    *b = *a ;
    *a = aux ;
    cout<<" fct  : apres &a : "<< &a << " " << *a << " &b : " << &b << " " << *b << endl;
}
int main() {
    int x,y ;
    x= 10 ;
    y= 20;
    cout<<" main : avant &x : "<< &x << " " << x <<" &y : "<< &y << " " << y << endl;
    permut(&x,&y) ;
    cout<<" main : apres &x : "<< &x << " " << x <<" &y : "<< &y << " " << y << endl;
}
```

Démo : paradresse.cpp