partie 1 : Introduction

MOOC Langage C++ INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

1 Objectifs de ce MOOC

Ce MOOC est fait pour des personnes voulant apprendre le langage C++. Il est le fruit des enseignements donnés à l'INSA de Toulouse.

Le langage C++ tient une place toute particulière dans de nombreux domaines. Vous découvrirez dans ce MOOC les différents concepts du C++ : références, constructeurs, destructeurs, surcharge des opérateurs, héritages, notion d'ami, méthode virtuelle, méthode et classe abstraite, généricité avec les patrons, gestion des entrées sorties par des flots, exceptions ou bien encore la bibliothèque STL.

Le livre du langage C au C++ vous permettra en complément d'avoir un ensemble d'exercices corrigés et d'aller plus loin notamment avec la découverte du langage C++ aussi.

2 Historique des paradigmes de programmation

Nous allons nous intéresser à différents paradigmes de programmation ou méthodes de programmation. En général, on leur associe des langages de programmation qui supportent ce paradigme de programmation. Nous ne nous intéresserons qu'aux paradigmes utilisables avec du C ou du C++. Les programmations fonctionnelles et logiques, par exemple, sont volontairement omises.

2.1 Programmation procédurale

Un des tous premiers paradigmes inventés fut la programmation procédurale. On peut la résumer par : **Choisissez les procédures dont vous avez besoin : utilisez les meilleurs algorithmes que vous pouvez trouver.**

L'accent est mis sur l'exécution. Des procédures et des fonctions permettent de réaliser une action. Cette dernière peut nécessiter des arguments et retourner une valeur. **Exemple:**

- fortran (un des plus anciens) ;
- pascal, C, ADA ;
- ....

2.2 Programmation modulaire

Au fil des années, grâce à l'évolution du matériel, les programmes sont devenus de plus en plus gros. Ils sont devenus difficiles à concevoir dans leur totalité. La conception des programmes s'est écartée de la conception des procédures pour se rapprocher de l'organisation des données. Un ensemble de procédures manipulant un ensemble de données est mis dans un **module**. Le paradigme devient :

Choisissez le module dont vous avez besoin ; partitionnez le programme de telle façon que les données soient cachées dans les modules.

Cette méthode est aussi connue sous le nom de **principe d'encapsulation des données**.

2.3 Abstraction de données

La programmation modulaire conduit à la centralisation des données d'un type, sous le contrôle d'un module de gestion de type. Si l'on veut manipuler plusieurs variables gérées par un même module, il faut ajouter une méthode d'identification des variables. La programmation par abstraction de données se focalise sur les types et pas sur les variables. Le principe est :

Choisissez les types dont vous avez besoin : donnez un ensemble complet d'opérations pour chaque type.

Exemple:
ADA, C++, etc.

2.4 Programmation par objet

L'abstraction de données définit une sorte de boîte noire. Une fois définie, on peut difficilement la modifier ou l'adapter à un problème particulier. Le programmation par objet permettra d'**hériter** d'un comportement déjà défini et de l'enrichir simplement. Une autre caractéristique importante des langages objet est leur facilité de représentation de la réalité. Ceci rend plus intuitive l'écriture de programmes. Le principe est donc :

Choisissez les classes dont vous avez besoin pour représenter les objets décrivant votre problème physique ; donnez un ensemble complet d'opérations (appelées méthodes) pour chaque classe ; rendez les points communs explicites à l'aide du mécanisme d'héritage.

Exemple:
EIFFEL, C++, java, etc.

3 Le langage C++

Le C++ a été créé par Bjarne Stoustrup en 1982 (ATT Bell Laboratories)[**?**]. Le C++ va reprendre différentes caractéristiques de langages plus anciens :

- Il reprend la notion de classe de SIMULA-67 et SMALLTALK,
- Il utilise à la base la syntaxe du C d'où il tirera son nom. En C, "++" est l'opérateur d'incrémentation. C++ se veut un meilleur C [**?**],
- Il permet la surcharge, c'est-à-dire qu'un nom identique de fonction peut avoir différentes signatures et codes dans le même programme (algol 68),
- Il copie le mécanisme des exceptions pour la gestion des erreurs de ML, ADA.

Le C++ est un langage qui a connu plusieurs évolutions avant d'être normalisé en 1998 par l'ANSI. Il évolua en 2003.

Le C++ est compatible avec le C. Ceci permet de reprendre tous les programmes écrits en C et de profiter de ses avantages :

— Proche des systèmes d'exploitation (écrit en C en général),
— Langage très répandu,
— Existence de nombreuses bibliothèques (communication, calcul, etc),
— Rapidité d'exécution (si on n'utilise pas certains mécanismes),
— Nombre de personnes connaissant le C, d'où un effort limité pour apprendre le C++.

Le C++ ajoute ou modifie principalement les caractéristiques suivantes au C que nous décrirons dans la première partie du cours :

— Gestion plus stricte des types et des constantes,
— Des entrées/sorties plus puissantes,
— Valeurs, références et pointeurs,
— Les fonctions.

Nous verrons ensuite les fonctionnalités apportées par la programmation objet :

— La notion d'objet et de classe,
— Les classes :
  — Les attributs et méthodes,
  — La protection,
  — Les constructeurs et destructeurs,
  — La surcharge des fonctions et des opérateurs.
— Les classes dérivées :
  — L'héritage et l'instanciation,
  — Membre et classe virtuelle,
  — Les "patrons".
— La notion de flot : "stream",
— Gestion des erreurs par le mécanisme des exceptions.

4 Structure d'un programme en C++

Voici un petit programme qui crée une classe helloclass. Cette classe permet d'initialiser un message de bienvenue et fournit une méthode pour l'afficher.

```
// utilisation d'objets d'entrées-sorties : les inclusions
#include <iostream>

// faciliter le nommage
using namespace std;

// déclaration de la classe
class helloclass {
// zone privée
private:
  string machaine;

// zone publique
public:
  // méthode d'initialisation
  helloclass(char *unechaine){
```

```
    machaine=string(unechaine);
  }
  // méthode pour l'affichage
  void rencontre(){
    cout << machaine <<endl;
  }
};
// fin de la classe

// début du programme principal
int main(){
  // instanciation de la classe
  helloclass ambre("hello");
  // appel d'une méthode sur un objet
  ambre.rencontre();
  // fin du programme principal
  return(0);
}
```

Nous constatons donc des similarités entre le C et le C++ mais aussi de nouveaux mots clés, opérateurs et structuration du code. Les programmes écrits en C sont presque totalement compatibles avec le C++. Les quelques modifications mineures servent à améliorer les ambiguïtés du langage C.

5 Premiers pas avec les entrées-sorties en C++

Le C++ définit une nouvelle méthode pour gérer les entrées sorties par rapport au C. Il utilise la notion de flot qui sera décrite dans le chapitre **??**. Les fonctions "printf" ou "scanf" du C sont remplacées par un objet et des opérateurs nouveaux. Pour pouvoir utiliser ce type d'entrées sorties, il faut inclure les instructions : " `#include <iostream> et using namespace std` ". On accède alors à quatre flots d'entrées sorties :

- **cin** pour les entrées,
- **cout** pour les sorties,
- **cerr** pour les erreurs en sortie,
- **clog** pour les sorties de "log".

Deux nouveaux opérateurs sont créés :

- "<<" permet de mettre sur un flot des données, il équivaut à "mettre sur",
- ">>" permet de récupérer des données, il équivaut à "lire à partir de".

Exemple:

```
void main() {
          int x;
          float y = 1.4;
          cout << ``x=?''; // ⟺ en C à printf(``x=?'')
          cin >> x; // ⟺ en C à scanf(``%d'',&x)
          cout << ``\n'' << ``x='' << x << ``,y=''<< y;
          // ⟺ en C à printf(``\n x= %d,y= %f'',x,y)
         }
```

6 Valeur par défaut

Le C++ permet de mettre des valeurs par défaut aux arguments d'une fonction. Il peut y avoir un nombre quelconque de paramètres omis. Toutefois, on doit commencer par la fin des arguments et il ne peut y avoir de trous. Autrement dit, le C++ commence l'identification des paramètres par le premier paramètre et ce jusqu'à ce que la liste des paramètres soit épuisée, à ce moment-là, il passe aux valeurs par défaut données dans la signature de la fonction. Il n'y a pas d'identification de type comme dans d'autres langages (ADA par exemple).

Exemple:

```
int f(int i = 1, int j =2){ ... }
....
f(); // ⟺ f(1,2)
f(0); // ⟺ f(0,2)
f(3,4); // ⟺ f(3,4)
```

7 Expansion en ligne

Lors de l'exécution d'un programme, l'appel à une fonction nécessite un certain nombre d'instructions supplémentaires générées par le compilateur (mémorisation de l'adresse de retour, saut en mémoire, gestion des arguments, etc). Dans le cas où l'on désire un programme optimisé en temps d'exécution, ceci est pénalisant. Les compilateurs actuels essaient de réaliser des optimisations. L'expansion en ligne en est une. Ceci revient à recopier le code de la fonction au niveau de l'appel pour éviter le mécanisme d'appel et de retour de fonction. Le C++ avec le mot clé "`inline`" permet d'indiquer au compilateur qu'une expansion en ligne est peut-être possible. L'expansion n'est possible que pour des fonctions simples : sans boucle complexe, sans récursivité, etc. Ce mécanisme remplace les macros-instructions du langage C, sources de nombreuses erreurs (voir annexe **?? ??**). La fonction déclarée `inline` doit être définie et utilisée dans le même fichier.

Exemple:

```
inline int max(int a, int b) {
return ( a > b ? a : b);
}
```

Le paragraphe **?? ??** complètera cette notion d'expansion en ligne dans le cas d'une séparation entre la déclaration et l'implémentation des classes.

7.1 Allocation et libération de mémoire dynamique

Le C++ offre de nouveaux opérateurs par rapport au C pour l'allocation et la désallocation mémoire, respectivement **new** et **delete**. A l'exécution d'un appel de l'opérateur `new`, le système d'exploitation attribue des octets contigus (par l'algorithme du premier accord, "first fit") et renvoie l'adresse du premier octet. C'est donc un tableau d'octets qui est alloué. L'opérateur "new" ajoute une affectation à l'initialisation par rapport au traditionnel "malloc" du langage C. En cas d'échec, une exception "bad_alloc" est générée. La gestion de ces dernières sera vue dans le chapitre **??**.

Exemple:

```
int *tab;
int *p;
char *chaine =new char[7]; // déclaration et allocation d'un tableau
tab = new int[10]; // allocation d'un tableau de 10 entiers
```

```
p = new int (5); // allocation d'un entier et affectation avec la valeur 5
```

L'opérateur "delete" détruit un objet créé par un "new". Attention : il y a un format particulier pour désallouer des tableaux d'objets (ajout de [] après `delete`). On ne peut pas désallouer un pointeur sur une constante (ceci est également vrai en C). **Exemple:**

```
int *tab;
int *p;
int i;
tab = new int[10]; // allocation d'un tableau de 10 entiers
p = new int (5); // allocation d'un entier et affectation
delete p;
delete [] tab; //désallocation du tableau
p = &i; // delete p est INTERDIT
```

L'opérateur "delete" ne peut pas être surchargé.

8 références

Une référence est un "alias", c'est-à-dire un autre nom pour le même objet. Une référence doit être initialisée. Une fois initialisée, elle ne peut plus servir d'alias pour autre chose. Un alias est équivalent à un pointeur constant avec un accès transparent à la valeur pointée. Une référence se déclare en utilisant `&` et doit être initialisée en même temps que sa déclaration.

Exemple:

```
main(){
            int i;
            int &ri =i; // ri est une référence à i
            const int &r2=2; // r2 est une référence à une constante
            ....
            ri=2; // accès transparent à la valeur pointée (pas de *) ⟺ i=2
}
```

Les références sont essentiellement utilisées pour le passage de paramètres en C++. Elles permettront de passer des paramètres d'entrées-sorties sans avoir à manipuler les pointeurs comme en C, ce qui est beaucoup plus élégant.

Exemple:

```
#include <iostream>
using namespace std;
int ordonner (int a, int b) {
            int prov;
            if (a > b) {
                        prov = a; a=b; b=prov;
            }
            cout << "min dans la fonction = " << a << ‘‘,’’;
            return a;
}
main () {
            int x = 2, y = 1, min = ordonner (x,y);
            //affiche: min dans la fonction = 1,
            cout << "min = " << min
```

```
        << " x = " << x << " y = " << y << "\n";
        //affiche: min = 1 x = 2 y = 1
}
```

Ce programme affiche : '`min dans la fonction = 1, min = 1 x = 2 y = 1`' signifiant que le traitement est correct à l'intérieur du corps de la fonction (i.e. '`a`' contient bien la plus petite valeur), mais les valeurs sont fausses à l'extérieur ('`x`' ne contient pas la plus petite valeur). L'échange des valeurs des variables du programme principal n'a pas été transmis par la fonction `ordonner`. La fonction a travaillé sur les copies de `x` et `y`.

Exemple:

Le programme précédent peut alors s'écrire en C++ :

```
#include <iostream>
using namespace std;
int ordonner (int &a, int &b) {
        int prov;
        if (a > b) {prov = a; a=b; b=prov;}
        cout << "min dans la fonction = " << a << ‘‘,’’;
        return a;
}
main () {
        int x = 2, y = 1, min = ordonner (x,y);
        //affiche: min dans la fonction = 1
        cout << "min = " << min
        << " x = " << x << " y = " << y << "\n";
        //affiche: min = 1 x = 1 y = 2
}
```

En C, pour faire fonctionner la fonction `ordonner` comme ci-dessus, il aurait fallu utiliser des pointeurs. Ce mode référencé permet d'éviter de manipuler des pointeurs explicitement. Le fait que le paramètre soit transmis par référence devient transparent pour l'utilisateur de la fonction.

Remarques:

Une fonction peut aussi renvoyer une référence en retour. On peut dans ce cas trouver dans le membre de gauche d'une affectation l'appel à une fonction.

Le programme ci-dessous :

- comptabilise l'occurrence des lettres minuscules de 'a' à 'z' et de tout autre caractère d'une chaîne,
- met le résultat dans le tableau d'entier CptAlph. L'indice 0 compte tous les caractères en dehors des lettres de 'a' à 'z'. Les indices de 1 à 26 comptent les occurrences des lettres de 'a' à 'z',
- utilise la fonction `find` qui renvoie une référence sur la case du tableau où l'on comptabilise les occurrences du caractère passé en paramètre,
- incrémente le retour avec l'opérateur ++ afin de comptabiliser le nombre de caractères.

Exemple:

```
// comptage du nombre d’occurences de ’a’ à ’z’ dans un mot
#include <iostream>
using namespace std;
const int MAX_STRING = 255;
int CptAlph[27]; // 0: autre lettre, 1..26 correspond à ’a’..’z’
char chaine[MAX_STRING];
// fonction retournant la référence au compteur de cette lettre
int & find (char c){
```

```
            if ((c<'a') || (c>'z')) return CptAlph[0];
            return CptAlph[c-'a'+ 1];
}
main(){
            int i;
            cin >> chaine;
            for(i=0;i< strlen(chaine);i++) find(chaine[i])++;
            cout << "\n autre:" << CptAlph[0];
            for (i=1;i<27;i++) cout << "\n"<< char('a'+i-1) << ":"
            << CptAlph[i];
}
```

Un des avantages du passage par référence est d'éviter une recopie des arguments d'appel. Ceci est nécessaire par exemple lorsque l'on manipule des tableaux volumineux. L'inconvénient est alors que l'on peut modifier à l'intérieur de la fonction la valeur du tableau dans le cas des tableaux constants. Ceci peut ne pas être souhaitable. Le C++ offre une possibilité pour éviter cela. Il suffit de déclarer l'argument comme une référence sur une constante.

Exemple:

```
struct tab {
            int t[100000];
}
int GetMax(const tab & t);
```

Lors de l'utilisation des références, il n'y a en général pas de conversion implicite. Les types passés en paramètre doivent correspondre exactement. Il y a éventuellement conversion des paramètres dans le cas où la référence est déclarée sur une constante.

9 Opérateur de portée en C++

Le C++ offre un nouvel opérateur " : :" permettant d'accéder à des noms normalement hors de portée. " : :' permet d'accéder au nom global qui aurait pu être masqué.

Exemple:

```
int x;
void f()
{
            int x = 1; // masque la variable x globale
            ::x = 2; // affectation de la variable x globale
}
```

Remarque:

Il est impossible d'accéder à une variable locale masquée. Cet opérateur sera manipulé de nouveau par la suite.

10 Les espaces de noms

10.1 Concept

Les espaces de noms ont été introduits pour permettre de lever toute ambiguïté sur les symboles (nom de fonction, classe, objet, variable, etc) utilisés dans un programme. La définition d'un espace de

noms se fait avec l'instruction `namespace` suivie du nom choisi. L'accès en dehors de l'espace de noms se fait en utilisant l'opérateur de portée. **Exemple:**

```
namespace plancomplexe {
  int c;
  struct complexe{
      int r,i;
    };
} // fin de l'espace de nom plancomplexe

namespace espacecomplexe {
  float c;
  struct complexe{
      int r,i;
    };
} // fin de l'espace de nom espacecomplexe

int main(){
  plancomplexe::complexe C1;
  espacecomplexe::complexe C2;
  plancomplexe::c=2;
  espacecomplexe::c=3.3;
  return 0;
}
```

Dans son utilisation classique, les différents espaces de noms seront dans des fichiers séparés, utilisés via la primitive d'inclusion `include`.

La déclaration des espaces se fait au niveau global. Il est interdit par exemple de définir un nouvel espace de noms à l'intérieur d'une fonction. Par contre on peut imbriquer les espaces de noms.

On peut aussi déclarer un espace de noms en plusieurs fois, ou bien encore en ne donnant que la spécification.

Exemple:

```
namespace plancomplexe {
   ...
}
namespace espacecomplexe {
   ...
   namespace couple{
      int x,y;
      ...
   }
}
namespace plancomplex {
   ...
}
   ...
espacecomplexe::couple::x=2;
```

10.2 Instruction using

Afin d'alléger l'écriture d'un programme en C++, on dispose de l'instruction `using` qui va permettre de s'approprier tout ou partie d'un espace de noms.

Exemple:

appropriation partielle

```
namespace plancomplexe {
  int c;
  struct complexe{
      int r,i;
    };
} // fin de l'espace de noms plancomplexe
using plancomplexe::c; // à partir de maintenant inutile
               // d'écrire plancomplexe::c,
               //c tout seul suffit
int main(){
  plancomplexe::complexe C1;
  c=3; // simplification de l'écriture
  return 0;
}
```

Exemple:
appropriation totale

```
namespace plancomplexe {
  int c;
  struct complexe{
      int r,i;
    };
} // fin de l'espace de noms plancomplexe

using namespace plancomplexe; // à partir de maintenant
          // on a accès à tout ce que
          // contient l'espace planccomplexe
          // sans utiliser l'opérateur de portée
int main(){
  complexe C1;
  c=3;
  return 0;
}
```

Si des ambiguïtés existent dans le code source, une erreur sera générée à la compilation si on utilise effectivement deux symboles ambigus.

L'instruction using est transitive :

```
namespace A {
  int n;
  double x;
} // fin de l'espace de noms A

namespace B{
  using namespace A;
  int y;
} // fin de l'espace de noms B
using namespace B;
n=3; // equivalent à A::n
```

10.3 Autres caractéristiques

On peut définir des alias afin de simplifier l'usage des espaces de noms. **Exemple:**

```
namespace court un_espace_de_noms_avec_un_nom_long;
using namespace court; // plus simple
```

On a la possibilité de définir des espaces anonymes. Dans ce cas, on ne peut pas utiliser à l'extérieur de l'espace anonyme ce qui est défini à l'intérieur. Ceci est utilisé à la place de `static` quand on veut limiter la visibilité de symboles au fichier qui les définit. **Exemple:**

```
namespace {
 int x;
  .....
}
// impossible d'accéder à x
```