# TP n°1 – Bases du C++

L'objectif de ce TP est de voir les bases d'un programme C++. Les bases objets du langage JAVA sont supposées acquises. L'énoncé est volontairement détaillé afin de vous faire bénéficier d'un "aide-mémoire" sur quelques aspects importants du C++. Les tâches à accomplir sont surlignées.

## I. La compilation et l'exécution

Vous devez disposer d'une plateforme de tests opérationnelle pour écrire et tester du C++. Pour cela, vous devez avoir installé sur vos machines un **compilateur** tel que GNU Compiler (MinGW sous Windows), Microsoft Visual C++, Bordland C++… Pour une application, la compilation au sens large du terme permet de générer un fichier exécutable (.exe sous Windows). D'autres types de compilations (génération de bibliothèques par exemple) sont possibles, mais ne seront pas utilisées dans ce module. La compilation permet de convertir vos fichiers de code (.cpp, .h…) en fichiers exécutables. Elle comprend plusieurs étapes qui sont détaillées sur la figure 1.

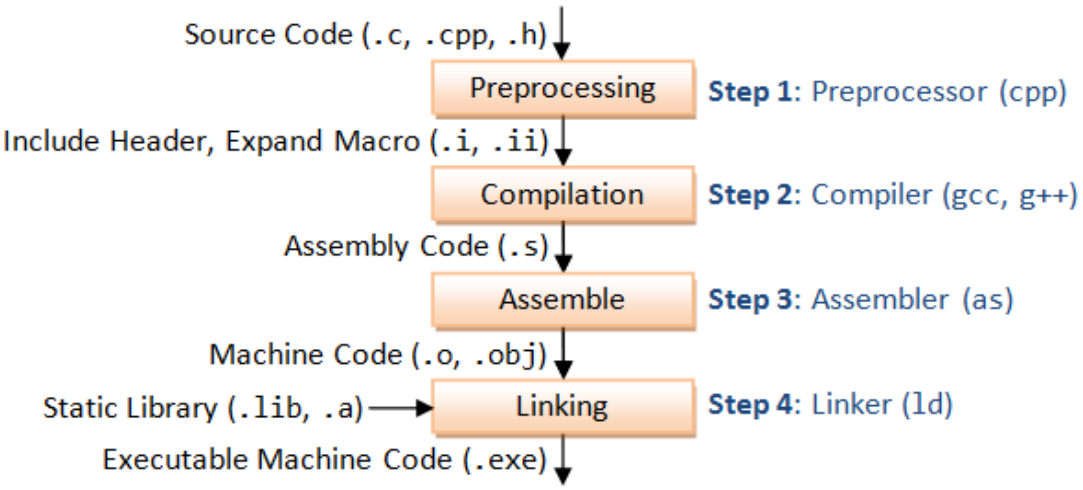


*Figure 1 : étapes de compilation d'un programme C++*

Un code C++ utilise souvent des **bibliothèques** existantes (Qt, wxWidgets, OpenGL, CUDA, OpenCV, CURL, Boost…). L'utilisation de bibliothèques existantes nécessite d'intervenir à la fois sur son code source (inclure les fichiers .h permettant d'utiliser des méthodes/classes de ces bibliothèques), mais aussi sur la configuration du compilateur et plus précisément du linker (étape 4 de la Figure 1) pour lui indiquer quelles ressources des bibliothèques externes (fichiers .a et .lib) doivent être incluses et/ou liées à l'exécutable généré.

Assurez-vous de pouvoir compiler et exécutez un code C++ sur votre plateforme de test en créant un fichier HelloWorld.cpp de la façon suivante :

```
/* fichier HelloWorld.cpp */
// commentaire sur une ligne
#include <iostream>
int main()
{
    std::cout << "Hello world !" << std::endl;
    return 0;
}
```

## II. La structure d'un programme C++

Un programme C++ contient souvent plusieurs fichiers sources. Ils portent souvent les extensions .h, .hpp, .cpp, ou .cxx. On a l'habitude d'associer à un fichier .h un fichier .cpp. Le .h contient les déclarations (méthodes, visibilités, attributs…) et le .cpp contient le code source de chaque méthode. D'autres stratégies sont possibles (tout mettre dans un fichier .h, mixer .h et .cpp…), mais la structure présentée ici vous permettra d'avoir une architecture claire et fonctionnelle.

Exemple d'architecture avec 3 fichiers : monfichier.h, monfichier.cpp et principal.cpp

```
//Fichier monfichier.h
//définition de condition du préprocesseur
#ifndef MONFICHIER_H
#define MONFICHIER_H

#include <iostream> //inclusion pour entrees/sorties (cout)
#include <string> //inclusion pour cout
using namespace std; //espace de nom std

class A { //définition du nom de la classe

  public : // visibilite
    A(); // constructeur par defaut
   ~A(); // destructeur
    string getChaine(); // methode getter sur l'attribut chaine

  private : //visibilite
    string chaine; // attribut
};

#endif
```

```
//Fichier monfichier.cpp
#include "Monfichier.h"
A::A(){
   this->chaine = "";
}
A::~A(){
}
string A::getChaine(){
```

```
   return this->chaine;
}
```

```
//Fichier principal.cpp
#include <iostream>
#include "Monfichier.h"
int main()
{
   A * a = new A();
   std::cout << "Valeur de chaine " << a->getChaine();
   return 0;
}
```

Côté syntaxe du langage, tout ce que vous avez vu en langage C fonctionnera en C++ (boucles, if…). Vous retrouvez ensuite le concept de **classes** vu en JAVA avec l'utilisation de mots-clés similaires (new, this…). De la même façon, les classes sont composées de **constructeurs**, de **méthodes** et d'**attributs**. Tous ont une **visibilité** (public, private, protected) spécifiée dans le fichier .h.

En vous basant sur cette architecture, créez la classe Individu contenant :

- Trois attributs protégés : prenon, nom (de type string) et âge (de type int);
- Un constructeur par défaut initialisant les valeurs des 3 attributs à des valeurs que vous choisirez;
- Un constructeur prenant 3 paramètres permettant d'initialiser les 3 attributs;
- 3 méthodes "getter" (**encapsulation**) permettant de retourner les valeurs des attributs;
- 3 méthodes "setter" (**encapsulation**) permettant de changer les valeurs des attributs;

Testez l'instanciation de plusieurs individus dans votre méthode main de la façon suivante :

```
int main()
{
   Individu i1;
   Individu * i2 = new Individu();
   std::cout << "Individu i1 : " << i1.getPrenom() << " " << i1.getNom() << " a
" << i1.getAge() << " ans" << std::endl;
   i2->setNom("Faudeil");
   std::cout << "Individu i2 : " << i2->getPrenom() << " " << i2->getNom() << "
a " << i2->getAge() << " ans" << std::endl;
   return 0;
}
```

Dans ce code, testez aussi votre constructeur utilisant 3 paramètres.

Notez les deux façons d'instancier un individu : soit de façon statique, soit de façon dynamique en utilisant un pointeur et le mot clé new. Un espace mémoire permettant de stocker les informations d'un objet de type Individu est crée et réservé lors de l'utilisation de new. i2 dans l'exemple précédent désigne un **pointeur** sur cet espace mémoire. Que l'on fasse une instanciation statique ou dynamique, un constructeur de la classe Individu est appelé. Ici, c'est le constructeur par défaut, car il n'y a pas de paramètres. i1 est une variable de type Individu et i2 est une variable de type pointeur sur un Individu.

Si on veut libérer l'espace mémoire alloué dynamiquement pour i2, il faut faire :

```
delete i2;
```

Cette instruction fait appel au destructeur de la classe Individu et exécute les instructions contenues dans ce destructeur. Tant que le delete n'est pas appelé ou que le programme n'est pas terminé, l'espace mémoire reste réservé, il faut donc faire bien attention à bien gérer la mémoire en C++ ! Ceci est d'autant plus vrai lorsqu'on utilise des new dans des boucles, il est très facile de saturer la mémoire et de faire planter l'application. Une allocation statique ne nécessite pas de libération mémoire, le ménage est fait à la sortie du bloc. Attention donc à bien maitriser la portée et la durée de vie du bloc dans votre programme.

**Lorsque vous utilisez une variable de type pointeur sur un objet, il faut utiliser -> pour appeler une méthode applicable sur la variable. Dans les autres cas, il faut utiliser le .**

*Remarque* : dans l'exemple précédent, au lieu de `i2->getPrenom()` , on aurait pu écrire `*i2.getPrenom()`. Les 2 notations sont totalement équivalentes.

# III. Héritage

Un concept fondamental de la POO (Programmation Orienté Objet) est celui de l'héritage : il consiste à faire bénéficier à une classe fille de caractéristiques d'une ou de plusieurs classes mères en les spécialisant. Cela permet de faire facilement de la programmation sans avoir à réinventer la roue à chaque fois. On peut utiliser par exemple une classe permettant de gérer la gestion de la souris et redéfinir notre propre classe en héritant afin de spécifier son comportant lors du clic sur son bouton gauche.

Faites hériter la classe Etudiant de la classe Individu en utilisant la syntaxe suivante lors de la déclaration de la classe dans le .h :

```
#include "Individu.h" //ne pas oublier cette inclusion !
class Etudiant : public Individu {
…
}
```

Créer une classe Etudiant comportant :

- Un attribut formation qui est un pointeur sur un tableau de chaines de caractères.

```
string * formation;
```

  Ce tableau contient 5 cases (à initialiser dans le constructeur) correspondant au nom des formations suivies ("CIR1", "CIR2", "CSI1", "M1 TBM", "ITII5"…). Si la formation n'a pas été suivie, la case contient la chaine vide;
- Un constructeur prenant les paramètres nom, prénom et âge de l'étudiant. Les attributs correspondants de la classe Individu sont initialisés à ces valeurs. Le constructeur initialise aussi le tableau pointé par l'attribut formation;

- Une méthode afficheFormation() permettant d'afficher le contenu des cases de l'attribut formation;
- Une méthode setFormation(int annee, string nom) permettant de modifier la chaine de caractère correspondant à la formation de l'année indiquée.

Ajoutez une méthode afficheInfos() à la classe Individu permettant d'afficher nom, prénom et âge d'un individu sur la sortie standard.

Comme vous l'avez vu en JAVA, toutes les méthodes de la classe Individu sont applicables sur un objet de type Etudiant.

Dans votre main, instanciez un objet de type Etudiant et appliquez-lui les méthodes setFormation(int, string), afficheFormation() et la méthode afficheInfos() de la classe Individu.

*Remarques :*

- on peut ajouter le mot clé **`virtual`** lors de la déclaration de la méthode `afficheInfos()` dans la classe Individu

```
virtual void afficheInfos();
```

En faisant un `Individu * i = new Etudiant();` et en appelant la méthode `i->afficheInfo()`, c'est la méthode de la classe Etudiant qui est appelée. Sans le mot clé virtual, c'est la méthode de la classe Individu qui le sera. La subtilité parait superflue si vous n'êtes pas familier avec la POO mais l'utilisation de virtual permet de résoudre parfois des comportements bien étranges ! L'utilisation du mot clé est donc recommandée lorsque vous désirez redéfinir la méthode dans les classes filles.

- on peut ajouter en plus du virtual, un "=0" lors de la déclaration de la méthode afficheInfos() dans la classe Individu

```
virtual void afficheInfos() = 0;
```

Dans ce cas, la méthode est dite **virtuelle pure** et la classe associée est dite **abstraite**. Le code de la méthode afficheInfos() **ne doit pas** être écrit dans le fichier .cpp de la classe Individu mais il devra **obligatoirement** être écrit dans une méthode afficheInfos() des classes filles à la classe Individu. Le fait d'avoir une méthode virtuelle pure impose donc de redéfinir cette méthode dans les classes filles. De plus, **une classe mère qui possède une méthode virtuelle pure ne peut pas être instanciée,** car une de ces méthodes n'est pas explicitement écrite, c'est pour cette raison que l'on parle de classe abstraite. Ce type de classe permet de définir un **comportement générique** pour ses classes filles.

## IV. Polymorphisme d'héritage

Derrière ce nom compliqué se cache un concept simple. Réécrivez une méthode afficheInfos() dans la classe Etudiant et faites en sortes d'afficher toutes les infos d'un étudiant. Testez cette méthode. Vous remarquez alors que en appelant la méthode afficheInfos() sur un objet de type Etudiant, la méthode afficheInfos() nouvellement définie dans la classe Etudiant est appelée et non plus celle de la classe mère. C'est ce qu'on appelle le **polymorphisme d'héritage**.

Vous pouvez aussi par exemple ajouter un paramètre dans la méthode afficheInfos() de la classe fille. Dans ce cas, on parle de **polymorphisme paramétrique** (= méthode avec un même nom, mais des paramètres différents).

## V. Jouons avec les pointeurs

Dans votre main, créez un nouvel individu de la façon suivante (on suppose que i2 est l'individu de l'exercice II) :

```
Individu * petitNouveau = i2;
```

Changez des données (nom, prénom) de l'individu petitNouveau et regardez ce qu'il se passe pour l'individu i2. Pourquoi ?

Dans votre main, créez un nouvel individu de la façon suivante :

```
Individu petitpetitNouveau = *i2;
```

Ce que vous venez de faire s'appelle un **déréferencement** : on accède au contenu de ce qui est pointé. Pour appeler des méthodes sur l'objet petitpetitNouveau, il faut donc modifier la syntaxe du code C++.

Effectuez une allocation statique d'un nouvel individu :

```
Individu is1("Chuck","Norris",53);
```

Vous pouvez tout à fait créer un pointeur sur cet objet. Pour cela, vous devez utiliser la syntaxe suivante :

```
Individu * is2 = &is1;
```

Le pointeur is2 est initialisé à l'adresse mémoire de is1 grâce à l'opérateur &. Un pointeur est donc une variable contenant l'adresse de l'objet pointé. Testez la création d'un pointeur pointant sur l'adresse de l'individu is1 et faites afficher les informations lui correspondant.

## VI. Constructeur de copie

Si vous ne le redéfinissez pas, un constructeur de copie (comme pour le constructeur par défaut et pour le destructeur) est automatiquement créé lors de la compilation de votre code. Pour la classe Etudiant, ce constructeur possède la syntaxe suivante :

```
Etudiant(const Etudiant & e);
```

Il sert à copier l'objet à partir d'un autre objet, et c'est donc lui qui est appelé lorsque vous faites :

```
Etudiant e1("Chuck", "Norris", 53);
Etudiant e2 = e1; // appel du constructeur de copie
Etudiant e3(e1); // appel du constructeur de copie
```

Créez un constructeur de copie dans votre classe Etudiant et faites en sorte que tous les paramètres de l'étudiant (nom, prénom, année et aussi formation) soient bien dupliqués en mémoire. Une modification sur l'un ne doit plus entrainer de modification sur l'autre ! Vérifier notamment bien ce qu'il se passe pour votre pointeur *formation* et le contenu pointé…