C++

Gilles Maire

2017

Titre du cours

Introduction

Rubriques

- Présentation
- Différences avec le C
- Masquage et surcharge

Présentation

Préambule

- Ce cours n'est pas un cours de C++ à proprement parler puisqu'il n'aborde pas les notions de types de base, les instructions de contrôle ou la définition de fonction du C.
- Il s'adresse à ceux qui connaissent le C et un peu le C++. Si vous ne connaissez pas le C, vous pouvez vousen sortir en connaissant le Javascript ou d'autres languages ressemblants.
- Il propose des exposés, des jeux oraux et des exercices

Pages C++

- Beaucoup d'informations sont disponibles pour compléter cette formation à commencer par la page de Bjarne Stroustrup
- Le site cppreference.com : est la référence en matière de c++, il est en langue anglaise
- Le site fr.cppreference.com : est la traduction du précédent en français

Définitions

- **classe** : une structure C à laquelle on ajoute des fonctions
- **méthodes** : fonctions propres à la classe
- **attributs** : enregistrements ou variables de cette classe
- **déclaration** : déclaration dans un fichier header (.h) des méthodes et des attributs
- **signature** : les types d'arguments passés à la fonction ainsi que sa constance
- **définition** : est l'écriture de la fonction en C++ (.cpp)

Usage

- Dans la famille des compilateurs Gnu :
  - le compilateur **C** est gcc
  - le compilateur **C++** est g++
  - à ne pas confondre avec
    - **gpp** qui est le préprocesseur générique
    - **cpp** le préprocesseur du C
- En C++
  - les fichiers contenant les codes définitions sont les fichiers d'extension cpp
  - les fichiers contenant les déclarations sont les fichiers d'extension h
  - le programme main est contenu généralement dans main.cpp
- En C
  - Les header sont dans les fichiers.h
  - Les sources dans les fichiers .c

Classe vs Structure

- Le but de cet exercice est de constater la similitude entre classe et structure

1. Déclarer un programme C contenant une structure
2. Déclarer un pointeur sur la structure et allouer lui un espace mémoire
3. Déclarer et définir un champ entier dans la structure
4. Dans le programme passer le champ à la valeur 2. Afficher le champ
5. Déclarer une fonction *clear* dans la structure qui met le champ de la structure à zéro par `void clear(){i=0;}`
6. Appeler la fonction clear(). Afficher le champ.
7. Compiler en C et en C++ les différentes étapes. Que constatez-vous ?

Différences avec le C

Entrées et sorties simples

- Les flux d'entrée-sortie sont représentés dans les programmes par les trois variables :
  1. **cin :** le flux standard d'entrée
  2. **cout :** le flux standard de sortie
  3. **cerr :** le flux standard pour la sortie des messages d'erreur

```
#include <iostream>
int main() {
int x=2;
int expression;
std::cin>>expression;// saisie d'une entrée
std::cout <<"expr"<< expression << std::endl; //affichage
return (0);
}
```

Dix lignes cout

- Écrire un programme qui affiche 10 lignes à l'écran.
- Chaque ligne comprendra
  - un numéro incrémental de 1 à 10
  - suivi d'un message fixe

Nouveautés par rapport au C

- Les variables peuvent être déclarées juste avant d'en avoir besoin et non pas en début de bloc
- Les commentaires // sont autorisés ils couvrent la fin de la ligne
- Les types référence : int &a;
- Le mot struct peut ne pas contenir de nom type ou de stucture pour etre compatible avec son associé classe
- Une fonction peut renvoyer une référence => `age("gilles")++;`
- Nous verrons l'utilité pour le passage des paramètres

```
int ages[i];
int &age(char * nom) { ... return ages[i] ;}
```

Nouveaux Types

- Le type bool existe en C++
- long long : depuis le C++11 sur 64 bits quelle que soit l'architecture
- les autres types varient en fonction des compilateurs ou des architectures
- wchar_t[] std::wstring : adressent les caractères Unicode
- Notation longue avec L pour les long long et les wchar :
  - L"Ceci est une chaîne de wchar_t."
  - 2.3e5L

Paramètres de fonctions par référence

- Références en paramètres de fonction (dans cpp et h) passage par valeur

```
type mafonction( int &a , char &b) {
 a=9;
}
```

- L'appel se fait par mafonction(i,j);
- Références sur des données constantes

```
type mafonction( const int &a ) {
int w=a;
a=b ;// Erreur du compilateur
}
```

- Rappel C :
- La déclaration en C se faisait par

```
type mafonction( int *a , char *b) {
 *a=9;
}
```

Classement entiers

1 - Écrire une méthode qui prend deux paramètres entiers en entrée et renvoie ces deux paramètres entiers classés (le premier paramètre le plus petit)
2 - On écrira la même méthode deux fois l'une avec passage des arguments par pointeur l'autre avec passage par référence.

Masquage et surcharge

Paramètres par défaut

- On peut spécifier des paramètres par défaut pour les fonctions avec la syntaxe suivante et ce **dans le fichier h**

```
type mafonction( type1 t1, type2 t2, type3 t3=val1,
      type4 t4=val4);
```

- **NB:** Les affectations doivent être effectuées sur les derniers arguments
- L'appel de la fonction pourra se faire sous l'une des formes :

```
mafonction( t1,t2,t3,t4);
mafonction (t1,t2,t3);
ma fonction (t1,t2);
```

- Dans les deux derniers cas t4 prendra la valeur par défaut

Masquage d'attributs (de variables)

- Le masquage concerne les homonymies d'attributs (de variables d'une classe).
- Une variable est masquée lorsqu'elle est redéfinie.
- Le temps de sa redéfinition, elle n'est plus accessible.
- La variable initiale et la nouvelle variable n'ont rien à voir au niveau emplacement mémoire.

```
1  {
2  int i=3; {
3   int i=8;
4   i++;// i vaut 9
5   } // i vaut 3
6  }
```

Surcharge de fonctions

- On dit qu'une fonction est surchargée quand dans une même classe il existe une autre fonction de même nom.
- On peut surcharger des fonctions si ces fonctions sont différentes :
  - Elles diffèrent par leur type ou leurs nombres de paramètres
  - Deux fonctions de signatures identiques mais l'une étant constante peuvent être surchargées. Dans ce cas d'ambiguïté la fonction non constante est appelée
- On ne peut pas surcharger des fonctions si :
  - Elles ont des arguments identiques mais retournent des types différents

Jeu

Ces paires de fonctions sont-elles surchargeables ?

```
1  void fonction ( int i , char *);
2  void fonction ( char * , int i);
3
4  void fonction ( int i , char *);
5  int fonction ( int i , char *);
6
7  void fonction ( int i, int j);
8  void fonction ( int nombre, int facteur);
9
10 void fonction ( int i , int j );
11 void fonction ( int i , int j = 0);
```

Objet

Rubriques

- Structures et classes
- Déclarations et définitions
- Les constructeurs

Structures et classes

Les structures

```
struct MaStructure{
  int i;
  int j;
};


int main(){
MaStructure objet;
objet.i=2;
return (objet.i);
}
```

Les classes

```
class MaStructure{
public:
  int i;
  int j;
};

int main(){
MaStructure objet;
objet.i=2;
return (objet.i);
}
```

Déclaration dans un fichier .h

```
class Nom
{
public:
    type champ1;
    type champ2;

    type methode1(type argument);
    type methode2(type argument);
}; // ne pas oublier le ;
```

- champ1 et champ2 sont des attributs

Définition dans le fichier cpp

```
#include nom.h
    type Nom::methode1(type argument) {
     //Instuctions où champ1 peut être utilisé
     champ1=2;
     }
    type Nom::methode2(type argument)
    { instructions; }
```

Pointeur this

- Le pointeur **this** est un pointeur sur la classe courante. Il a deux utilisations principales :
  - permettre de passer un pointeur vers la classe courante à une fonction qui demande un argument comme pointeur sur une classe
  - dans les éditeurs avec complétion this-> permet l'affichage de l'ensemble des attributs et des méthodes de la classe.
- Exemple : si Debug est une methode qui demande un pointeur et un niveau de débugage

```
Debug(this,level);
```

Déclarations et définitions

Méthodes

- Les méthodes sont les procédures que l'on connaît en C, ramenées à un contexte objet.
- Elles sont *déclarées* dans des fichiers header (.h) à l'intérieur d'une classe

```
// dans le fichier MaClasse.h
class MaClasse {
public:
    type mafonction(type1 var, type2 var);
};
```

- Elles sont *définies* dans le fichier cpp

```
// dans le fichier MaClasse.cpp
type MaClasse::mafonction ( type1 var, type2 ){
 return (variable_du_type_du_retour);}
```

Organisation fréquente

- fichier.h

```
class MaJolieClasse {
    int attribut1;
    int attribut2;
    int methode(int i);
};
```

- fichier.cpp

```
#include "fichier.h"
#include <iostream>
int MaJolieClasse::methode (int i){
  attribut1=i;
  std::cout<<i<<std::endl;
}
```

Organisation fréquente (suite)

```
//fichier main.cpp
#include "fichier.h"
int main(){
    MaJolieClasse p;
    p.methode(2);
    return(2);
}
```

- Sous linux on compile cela par

```
g++ fichier.cpp main.cpp -o resultat
```

ou

```
g++ -c -o main.o main.cpp ; g++ -c -o fic.o fichier.cpp
g++ fic.o main.o -o resultat
```

Méthode compacte

```
//fichier main.cpp
#include <iostream>

type MaClasse::mafonction ( type1 var, type2 ){
 return (variable_du_type_du_retour);
}

int MaJolieClasse::methode (int i){
  std::cout<<i<<std::endl;
}
int main(){
MaJolieClasse p;
p.methode(2);
return(2);
}
```

Les constructeurs

Définitions des constructeurs

- Les constructeurs servent à allouer l'espace mémoire nécessaire à l'utilisation de la classe.
- Ils portent le nom de la classe et servent aux initialisations
- Les contructeurs et les destruteurs sont les **seules** méthodes sans type de sortie
- Les constructeurs et les destructeurs ne sont pas héritables même s'ils sont exécutés de facto par les classes filles
- Les constructeurs peuvent être surchargés, c'est à dire que plusieurs constructeurs avec des signatures différentes peuvent exister pour une même classe
- Les constructeurs sont appelés par le mot clé **new** ce qui alloue la mémoire référencée par un pointeur
- Les constructeurs sont également appelés implicitement sur des allocations de variables (Maclasse elem)

Exemples d'utilisation

- Chaque fois qu'un constructeur est appelé on dit qu'objet est **instancié**
- Voici quelques instanciations

```
MaClasse *p = new MaClasse();
MaClasse *p = new MaClasse; // identique
MaClasse p ; // appelle un constructeur
MaClasse *p = new MaClasse(this,contexte);
```

Types de constructeurs

- Le constructeur par défaut est celui qui ne possède pas de type en entrée
- Il existe trois autres types de constructeur
  1. Le constructeur par copie
  2. Le constructeur par transtypage ou de conversion
  3. Le constructeur à arguments multiples
- Le constructeur synthétisé est un constructeur fourni par le compilateur en l'absence de constructeur (défaut, copie)

Classe entier

- Définir une classe dotée d'un membre entier et d'une méthode print() qui permet d'afficher la valeur du nombre membre.

Constructeur par copie

- Quand on affecte un objet à un autre ou que l'on passe un objet en paramètre de procédure, l'objet passé n'est pas l'objet initial mais l'objet en cours de vie.
- On alloue dans ce cas l'objet par un constructeur de copie synthétisé ou explicitement défini.
- Le premier paramètre du constructeur par copie est de type `const Classe &`
- Si celui-ci copie les variables comprises dans la classe il ne copie pas les éventuels pointeurs qui pointent alors sur les mêmes adresses !
- On peut donc être amené à en écrire un plus fin que celui synthétisé.
- Il se déclare comme suit :

```
maclasse ( const mclasse &source) ;
```

Constructeurs par transtypage

- Un constructeur par transtypage est un constructeur qui admet un premier argument qui n'est pas une référence sur la classe mais d'un type quelconque.
- Exemple :

```
classe maclasse {
  maclasse(int i );
}
```

- Ce type de constructeur permet d'initialiser un des attributs de la classe de type entier.
- Appel :

```
maclasse elem(2);
maclasse elem = 2;
```

Points clés

Rubriques

- Initialiation de variables
- Initialisation des constructeurs
- Constance

Initialiation de variables

Initialisation de variables

```
int Nom1 ; Nom1 = valeur ;
int Nom2 = valeur ;
int Nom3 (valeur) ; // équivalent
int Nom4(valeur),Nom5(valeur);
int &Nom6(valeur) // référence Nom6 vaut valeur
int Nom7(); // HORREUR
int const Nom8 (valeur) ; // ne peut changer de valeur
```

Exemples réels :

```
int i=3;
int j(2);
```

Constructeur Copie

- Il s'agit de mettre en évidence la faiblesse du constructeur de copie synthétisé par un cas d'école
- Écrire un programme contenant une classe
- Cette classe contient un attribut entier, un pointeur sur cet attribut ainsi qu'une méthode qui imprime le contenu du pointeur.
- Utilisez le constructeur de copie synthétisé pour copier un deuxième objet
- changer la valeur de l'attribut entier
- Sur quoi pointe le pointeur de l'objet pointé ?
- Corriger le problème avec un constructeur de copie convenable

Usage du mot clé explicit

- Si le mot clé `explicit` est utilisé devant la déclaration du constructeur de transtypage, la notation `maclasse elem=2` ne sera plus autorisée
- L'usage du mot `explicit` a pour but d'interdire les cast implicites qui pourrait découler de la notation précédente
- L'usage du mot clé `explicit` n'interdit pas la notation en `elem(2)` de l'exemple précédent
- Il autorise les casts explicites
  - soit par la notation (type)
  - soit par la notation `static_cast<type>` qui est identique

Constructeurs à arguments multiples

- Un constructeur à arguments multiples permet d'initialiser plusieurs attributs
- Il dispose donc d'une déclaration et d'une définitions comprenant plusieurs arguments
- La déclaration peut se faire de la façon suivante `Maclasse p(2,3,4);`
- Par contre on ne peut pas utiliser de forme avec le signe = comme dans le cas d'un constructeur par transtypage.
- L'usage du mot clé `explicit` interdit ici uniquement les cas de cast implicite puisque les initialisation par transtypage sont sans objet.

Constructeur

Pour la classe QCombobox trouver le type de constructeur appelé dans chacun des cas

```
1 QCombobox *p;
2 QCombobox comb;
3 QCombobox *p = new QCombobox ( "titre");
4 QCombobox *p = new QCombobox ( "titre1","titre2");
5 QCombobox comb="titre";
6 Affiche (comb);
```

Initialisation des constructeurs

Liste d'initialisations

- Il existe un moyen de définir des membres à l'initialisation par un procédé appelé liste d'initialisations
- Soit la classe :

```
class Maclasse{
    Maclasse();
    int m_attribut1;
    int m_attribut2; } ;
```

- Plutôt que de définir :

```
MaClasse::Maclasse() { m_attribut1=1; m_attibut2=2;}
```

- On peut utiliser :

```
Maclassse::Maclasse(): m_attibut2(2), m_attribut1(1)
{}
```

Remarques sur les listes d'initialisation

- , est le séparateur,
- pas de ; à la fin de la liste !
- on ne peut utiliser cette liste d'initialisation que pour les constructeurs et nullement pour d'autres méthodes
- l'ordre des évaluations ne suit pas l'ordre décrit par les , mais l'ordre de déclaration des arguments
- on peut utiliser des valeurs entre parenthèses correspondant à des attributs dans l'ordre de déclaration

```
// on appelle essai(3) => j=3,i=4,k=4
essai::essai (int j) : k(i),i(j+1)
```

Liste d'initialisation avec constructeur

- Cette notation fait également partie de la liste d'initialisation

```
ClasseFille::ClasseFille(int i):ClasseMere(i)
{
}
```

- Lorsqu'un constructeur d'une classe fille est appelé, le constructeur de la classe mère est appelé en premier lieu et par défaut c'est le constructeur sans argument.
- La notation `ClasseMere(i)` signifie l'appel du constructeur de la classe mère avec l'argument i, c'est à dire que dans ce cas là, on force l'appel d'un constructeur de transtypage.

Mise en oeuvre constructeur

- Mettre en évidence les différentes remarques évoquées précédemment
- nous définirons 4 attributs

Constance

Constance

- **const :** ce mot clé est utilisé pour rendre le contenu d'une variable non modifiable, ie en lecture seule. Quand ce mot clé est appliqué à une structure, aucun des champs de la structure n'est accessible en écriture.
- **mutable :** ne sert que pour les membres des structures. Il permet de passer outre la constance éventuelle d'une structure pour ce membre. Ainsi, un champ de structure déclaré mutable peut être modifié même si la structure est déclarée const.

Variables statiques

- Se déclare par l'emploi du mot static comme suit

```
static type variable;
```

- Elle peut se déclarer locale à une fonction

```
void f(void)
{
    static int i = 0; /* initialisée une seule fois */
}
```

- Elle peut être globale en dehors de tout bloc
  - On la définit dans un fichier cpp, en dehors de toute méthode via la syntaxe :

```
static type variable=valeur;
static int j = 20;
```

Variable statique utilité

- Dans les fonctions :
  - la variable sera persistante et initialisée qu'une seule fois à la compilation
  - Par contre elle peut varier
- À l'extérieur de toute fonction
  - sa portée sera limitée au seul fichier où elle est déclarée.

Methode et variable statiques

- Déclarer une classe, munie d'une méthode publique
- Dans cette méthode déclarer une variable statique, l'initialiser et l'incrémenter
- Appeler la méthode plusieurs fois depuis le main

Méthodes statiques

- Les méthodes statiques sont déclarées dans dans une classe mais ne partagent pas les données de la classe.
- Elles sont appelées sous la forme Classe::fonction() bien qu'on puisse les appeler via un objet instancié.
- Elles sont déclarées sous la forme affichées ci dessous et leur définition n'utilise pas le mot static.

```
static type fonction();
```

Autres types de classe de stockage

- **auto :** (par défaut) Les variables ont pour portée le bloc d'instructions dans lequel elles ont été crées. Elles ne sont accessibles que dans ce bloc et leur durée de vie est restreinte à ce bloc. Ce mot clé est facultatif.
- **register :** cette classe de stockage permet de créer une variable dont l'emplacement se trouve dans un registre du microprocesseur. Déprécié en C++11
- **volatile :** cette classe de variable sert lors de la programmation système. Elle indique qu'une variable peut être modifiée en arrière-plan par un autre programme (interruption, thread, autre processus, OS). Elle n'est jamais mis en registre.
- **extern :** cette classe est utilisée pour signaler que la variable peut être définie dans un autre fichier. Elle est utilisée dans le cadre de la compilation séparée.

Héritage

Rubriques

- Dérivation
- Encapsulation
- Cas d'école

Dérivation

Dérivation

- Dériver une classe, c'est fabriquer une classe contenant les mêmes propriétés plus de nouvelles
- La classe dérivée est appelée fille ou sous classe
- Une sous classe a plus de membres que la classe mère et non l'inverse (on dit qu'elle est plus large)
- Une classe mère peut avoir plusieurs filles et réciproquement
- Constructeurs, destructeurs, classes amies ne sont pas hérités

Mise en œuvre

- dans un fichier “nommere.h”

```
class NomMere {
public:
    type champ;
    type methode1(type argument);
}; // ne pas oublier le ;
```

- dans un fichier “nomfille.h”

```
class NomFille: public NomMere {
  type NouvelleMethode();
}; // ne pas oublier le ;
```

Classe fille

- La déclaration d'une classe fille se fait dans le fichier de définition au niveau de la classe
- À l'instantiation d'une classe fille, le constructeur de la classe mère est appelé ensuite le constructeur de la classe fille
- À l'instantiation d'un objet qui hérite en n ième niveau d'un objet mère, les constructeurs seront appelé du plus haut niveau vers le plus bas.
- À la désallocation d'un objet, le destructeur de la classe fille est appelé puis le destructeur de la classe mère.

Déclaration de la classe fille

- Dans le fichier header la déclaration `class NomFille: public NomMere` implique l'une des deux stratégies
  - l'inclusion des header de la classe mère par un `#include "mere.h"` en zone d'inclusion
  - l'utilisation du mot `class mere;` en cas de référence croisée, c'est à dire si la mère fait référence à la fille
- Si la classe mère ne dispose que de constructeurs avec paramètres on devra spécifier les paramètres dans la déclaration, ou si on veut forcer l'appel d'un constructeur particulier de la classe mère

```
// dans le cpp de la fille
Fille::Fille(type parametre):Mere(parametre);
```

Constructeur classe fille

Mettre en évidence par des affichages dans les constructeurs mère et fille que lorsqu'on instancie une classe fille, on appelle les constructeurs de tous les parents.

Encapsulation

Héritage multiple

- L'héritage multiple ne désigne pas le phénomène des petites filles ou arrière petites filles d'une classe mère, mais le fait d'être filles de deux classes mères.
- On déclare un héritage multiple comme suit :

```
class Fille : public Mere1 , public Mere2 { ... };
```

- Si une méthode publique provient de deux classes mères différentes, on tombe dans ce cas dans un cas de surcharge interdite avec erreur de compilateur.
- Le cas 1 Mère, 2 filles, 1 petite fille posera des problèmes de compilation quand un membre de petite fille appellera un membre de mère même si aurait pu penser que le compilateur s'en sortirait.

Héritage multiple

```
class Mere {
public: Mere();
    Mere( int i );
    void FonctionMere( int i );
};
class Fille: public Mere {
public: Fille();
    void FonctionFille ( int i );
};
```

Quelles sont les possibilités correctes et incorrectes ?

1. Mere a(10);
2. Mere b; b.FonctionFille();
3. Fille c(12);
4. Fille d; d.FonctionMere();

Définition encapsulation

- L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.
- L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet via trois qualificateurs : private, public, protected qui affectent chaque méthode ou attribut qui peut être :
  - **Private :** accessible qu'à l'intérieur de la classe ou aux fonctions externes déclarées friend
  - **Public :** accessible à l'intérieur et à l'extérieur de la classe
  - **Protected :** comme private mais accessible par les classes dérivées uniquement

Exemple : Mere

```
class Mere
{
public:
  int m_i;
  int m_j;
  void methodeMerePublique();
  Machin();
private:
  int m_k;
  void methodeMerePrivee();
protected:
  void methodeMereProtegee();
}
```

Exemple : Fille

```
class fille : public Mere
{
public:
  int m_i;
  int m_j;
  void methodeMerePublique();
  Machin();
private:
  int m_k;
  void methodeMerePrivee();
  void methodeMereProtegee();
}
```

Jeu mère fille

```
class Mere {
public: int publique;
private: int privee;
protected : int protegee;
};
class Fille : public Mere{
private: int fille;
};
```

Départagez les affirmations vraies et fausses :

1. Dans main.cpp : Mere m ; m.privee=1;
2. Dans main.cpp : Fille f; f.protegee=1;
3. Dans main.cpp : Fille f; f.privee=3;
4. Dans la classe Fille : protegee=1;
5. Dans la classe Fille : privee=3;

Cas d'école

Particularité amusante

Portée d'héritage

- Nous avons vu l'héritage public, il existe l'héritage **privé** et l'héritage **protégé**

```
class Fille : private Mere { ... };
class Fille : protected Mere { ... };
```

- En mode protégé les membres publics de la classe de base deviennent protégés, les autres gardent leur propriété
- En mode privé tous les membres deviennent privés

Redéfinition vs surcharge

- Dans une même classe deux méthodes aux noms identiques sont surchargées si elles ont des signatures différentes
- Dans une classe fille une nouvelle méthode portant le même nom et une signature différente d'une méthode de la classe mère est une nouvelle surcharge
- Par contre si elle ne diffère en rien de la méthode de la classe mère elle la remplace et on parle alors de **redéfinition**
- Les membres redéfinis qu'ils soient méthodes ou attributs restent accessibles depuis la classe fille via le qualificateur `Mere::` devant leur nom

Redéfinition de fonction

- Mettre en évidence un cas de redéfinition de fonction
- Accéder à la méthode redéfinie à partir de la classe fille.
  - avec sa définition mère
  - avec sa définition fille

Méthodes virtuelles

Rubriques

- Polymorphisme
- Classes Friends
- Opérateurs

Polymorphisme

Présentation polymorphisme

- Le polymorphisme consiste à appeler une méthode de la classe fille alors qu'un pointeur adresse une classe mère;
- Ce méchanisme n'est rendu possible que parce qu'une méthode est déclarée virtuelle et qu'elle est présente à la fois dans la classe fille et la classe mère.
- Dans certaines conditions que nous allons mettre en évidence dans l'exercice suivant on pourrait s'attendre à ce que la méthode mère soit appelée, or c'est la méthode fille qui va être appelée.

Polymorphisme

- Déclarer une classe Mere et une classe Fille
- Définir pour chacune d'entre elles des membres différents
- Déclarer une fonction de même signature dans chaque classe mais qui produit un traitement différent (cout<< différent)
- Déclarer dans la fonction main deux pointeurs sur la classe Mere
- Le premier pointera sur un élément Mère
- Le second sur un élément Fille
- Appeler la fonction de même signature depuis les deux pointeurs
- Que se passe-t-il à l'appel de la fonction de même signature ?

Méthodes virtuelles

- Le mot clé *virtual* devant une déclaration de fonction indique qu'une fonction de même nom est déclarée dans une classe dérivée avec la même signature
- La fonction comme elle est déclarée dans la classe mère existe donc également dans la classe fille. Il y a donc surcharge.
- Mais le mot *virtual* indique que le pointeur de la mère accédera au champ de la fille si l'initialisation a été faite par un pointeur sur la classe mère avec égalité d'un objet sur la classe fille

Polymorphisme effectif

- Reprendre l'exercice précédent et déclarer les fonctions identiques virtuelles.

Contrainte sur les fonctions virtuelles

- La fonction virtuelle de la classe Mere et la fonction de la classe Fille doivent avoir les mêmes paramètres
- La fonction ne doit pas être obligatoirement déclarée virtuelle dans la classe Fille
- Le paramètre de retour peut différer mais doit respecter la contrainte suivante :
  - Le type de retour de la classe fille doit pouvoir être converti en type de retour de la classe fille.
  - La classe du type de retour de la mère doit être accessible depuis la fille
  - Si ces paramètres ne sont pas réunis le compilateur provoquera une erreur.

Fonctions virtuelles pures et classes abstraites

- La fonction virtuelle de la classe mère peut être déclarée

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

- La fonction virtuelle n'a pas de définition elle est appelée **fonction virtuelle pure**
- Cela forcera l'ensemble des classes fille à redéfinir la fonction sous peine d'erreur de compilation
- Cela rendra la classe **abstraite** ce qui veut dire qu'aucun objet ne pourra être instancié avec cette classe mais uniquement avec des classes dérivées.
- Une classe abstraite ne peut donc être utilisée que si elle est dérivée et si ses méthodes virtuelles pures sont définies.

Classes Friends

Fonctions amis

- Une fonction amie d'une classe est une fonction d'une autre classe qui sans faire partie de la classe a le droit d'accéderà tous les attributs publics ou privés de la classe, elle peut être privée ou publique
- La déclaration se fait en ajoutant le terme `friend` devant la déclaration de la fonction :

```
friend type fonction() ;
```

- La définition de la fonction se fera sans le symbole `maclasse::` puisque la fonction ne fait pas partie de la classe.
- Elle a souvent tendance à faire partie d'une autre classe, aussi on peut la trouver avec un définisseur `autreclasse::`

Classes amies

- Une classe amie est une classe qui peut accéder à tous les membres d'une classe.
- Elle est déclarée comme suit :

```
class Maclasse {
....
friend class Amie;
};

class Amie {
};
```

- La relation d'amitié n'est pas transitive
- La classe amie peut utiliser tous les membres même ceux qui sont privés

Opérateurs

Surcharge des opérateurs

- On peut définir ou redefinir les opérateurs à tous les objets
- On ne peut pas définir les opérateurs `. .* :: ?: sizeof`
- On ne peut surcharger les opérateurs que si un des membres de l'opération est du type de la classe
- Une fois surchargés, les opérateurs gardent leur priorité et leur associativité initiale
- Ils perdent leur éventuelle commutativité
- $++$ peut ne pas être lié à $+$
- Pour définir une surcharge sur un opérateur $+$, il faut définir la fonction nommée `operator+`
- L'exemple `operator+` est valable pour tous les opérateurs y compris dans les chapitres suivants.

Liste des opérateurs qui peuvent être surchargés

```
+ ++ +=
- -- -=
* ** *=
/ / /=
= == !=
% %= // modulo
^ // xor bit à bit
|  |= // ou bit à bit
&  &= // & bit à bit
< << <= <<= // <<= décalage bit à bit droit
> >> >= >>=  // >>= décalage bit à bit gauche
~  ~=   //  no bit à bit
||  &&
-> [] () new new[] delete delete[]
```

Surcharge d'un opérateur par une fonction membre

- On définit a+b par la méthode operator+ appliquée à la classe c'est à dire obj1.operator+(obj2)
- +a sera défini par obj.operator+();

```
class Point {
    Point(int x, int y);
public:
    Point operator+( Point) ;
};
Point Point::operator+( Point q){
return Point(x + q.x, y + q.y);
}
Emploi :
Point p, q, r;
r = p + q;
```

Surcharge d'un opérateur par une fonction non membre

- Dans ce cas la fonction `operator+` aura deux arguments pour que obj1 + obj2 soit équivalent à la fonction `operator+( obj1, obj2);`
- Exemple :

```
Point operator+( Point p,  Point q) {
return Point(p.X() + q.X(), p.Y() + q.Y());
}
```

- On peut être amené à utiliser une fonction non membre qui sait faire l'opération, elle est alors est déclarée amie
- On doit choisir : la surcharge d'un opérateur par une fonction membre ou (exclusif) par une fonction non membre !

Surcharge de l'opérateur d'affectation `operator=`

- L'opérateur d'affectation est un peu différent car :
  - un opérateur synthétisé est construit par le compilateur analogue au constructeur de copie
  - il peut être nécessaire de disposer d'un opérateur d'affectation qui ne peut être surchargé que par l'emploi d'une fonction membre

Opérateur de conversion vers un autre type

- On peut disposer d'un opérateur de conversion vers un type du langage ou vers une classe
- Au niveau de la déclaration on utilise la syntaxe `operator type()` avec un ou plusieurs blancs obligatoires entre operator et type()
- L'opérateur doit retourner le type
- **Exemple :**

```
// Déclaration
MaClasse::operator int()
 {
     return m_int;
 }
//utilisation
cout <<objet.operator int()<<endl;
```

Symétrie

- La symétrie consiste à fournir des méthodes qui permettent la symétrie des opérateurs :
  - Si on définit une méthode membre d'ajout via operator+, on permet l'opération A=A+B mais pas A=B+A
  - Stricto sensu l'opérateur operator + ne permet donc pas la définition de méthodes symétriques
- Si on souhaite fournir des méthodes symétriques il faut utiliser une fonction non membre

Heure et opération

- Écrire une classe Heure définissant trois membres Heure, Minute, Seconde
- Créer l'opérateur + ajoutant deux heures d'abord par une fonction membre, mettre en évidence un problème de symétrie.
- Corriger en définissant l'opérateur par une fonction friend

Divers

Rubriques

- Définition
- Nommage

Définition

Les exceptions

- Lorsqu'on traite une erreur elle survient dans une fonction n'ayant pas le pouvoir d'action sur le problème
- C'est le même cas dans des imbrications d'accolades d'où l'on voudrait bien sortir un code d'erreur en court-circuitant le code
- L'exception du C++ permet de court-circuiter la sortie d'un bloc ou de plusieurs niveaux de fonction au moyen d'une instruction *throw*
- L'exception est basée sur trois mots clés :
  1. **throw valeur** : signale l'anomalie en lançant l'exception
  2. **try { ..... }** : fait le test de la sequence entre { }
  3. **catch(type val) { ...}** : conduite à tenir en cas d'anomalie

Exemple d'exception

```
int division(int a,int b)
    {  if (b==0)  throw 0;  else return a/b ;}
void main() {
    try{ division(1,0)  }
    catch(int code) { cerr << Exception << code << endl; }
    catch(...) { cerr<<"Exception inconnue"<<endl ; }
 }
```

- Dans le programme main() on exécute le code situé après try (la fonction division).
- Cette fonction envoie un throw en cas de problème en fonction du type appelé on est dirigé vers le catch correspondant.
- Si la division est faite correctement aucun catch ne sera appelé.

Remarques sur les exceptions

- On peut trouver un autre *try* dans l'exécution du code *try*
- Les différents *catch* permettent un aiguillage en fonction du type
- Les *throw* appellent un catch correspondant au type de l'argument du *throw*
- Les trois points de suspension du catch correspondent aux autres cas non traités ; ils ne peuvent être positionnés que sur le dernier catch.
- On s'abstient d'utiliser les mécanismes d'exception dans un contexte temps réel car le code encadré par le try est plus lent qu'un code normal.
- Si l'exception n'est pas traité dans une procédure, elle est remontée à la procédure appelante jusqu'à tomber sur un catch ou sur le programme main qui exécutera un terminate.

Attraper une exception

- Lorsque l'exécution d'une instruction lance une exception, un gestionnaire catch ayant un argument compatible avec l'exception est recherché dans les fonctions actives de la plus récemment appelée à la plus anciennement appelée
- On recherche un type T2 compatible avec le type T1 de l'exception.
- Un type compatible est :
  - soit strictement égal
  - T1 et T2 pointent vers une classe de base et sa classe dérivée accessible
- Les casts ne sont pas admis c'est à dire qu'un catch(float) n'attrape pas un int

Exemple exception

```
main(){
try
{                          // Bloc pouvant generer une exception
            Analyse a;
            a.LectureChiffre();
        }
catch
(UneClasse) {          // Capter et traiter l'erreur
            cerr << "Erreur, entier invalide\n";
        }
(UnType) {
cerr<<"Autre erreur\n";
}
}
```

- le trow appellera un objet du type (UneClasse) qui provoquera le catch (UneClasse)
- S'il appelle un objet de type (UnType) le catch (UnType sera appelé)

Exception

Écrire une classe *chiffre* qui comporte une exception permettant de tester si un entier est un chiffre

La classe exception et sa dérivée stdexcept

- On utilise en général une classe dérivant de la classe standard std::exception qui dispose d'une fonction membre what() qui renvoie une description de l'exception.
- On prendra par exemple la classe stdexcept qui définit les classes logic_error, domain_error, invalid_argument, length_error, out_of_range etc ..
- L'exemple suivant envoie brutalement un *throw* dans le *try*

```
#include <iostream>
#include <stdexcept>
int main() { try  { throw std::logic_error( "Essai" ); }
    catch ( const std::exception & e ) {
            std::cerr << e.what();
    }
}
```

Nommage

Espace de nommage

- Un espace de nommage permet de regrouper plusieurs déclarations de classes ou de membres dans un groupe nommé.
- Il permet ainsi des regroupements logiques d'informations.
- Il permet aussi de différencier des classes de même nom provenant de deux librairies en leur attribuant des espaces de nommage différents. Ce cas s'appelle résolution des noms.
- Une fois un nom donné, celui-ci permet d'accéder aux membres par l'opérateur de portée ::
- Le mécanisme permet de nommer les méthodes ou attributs depuis l'extérieur d'une classe, il ne s'adresse donc qu'aux membres publics.

Déclaration d'un nommage

- Un nommage peut également concerner une simple fonction

```
namespace A {
    void f(int);
}
```

- L'utilisation sera faite sous l'une des 3 formes :

```
using  A::f;
f(12);

A::f(12);

using namespace A;
f(12);
```

Utilisation d'un nommage

- Pour le nommage

```
namespace identifiant{
    class Machin {
    ....
    };
}
```

- on utilisera l'un des formes

```
using namespace identifiant ;
Machin i;
Identifiant::Machin j;
```

Exemples connus

- Un exemple connu d'espace de nom est `std` qui regroupe cin, cout et cerr.
- On peut indiquer que le nom de l'espace de nommage est implicite et n'a pas besoin d'être mentionné à chaque usage par l'instruction :

```
using namespace std;
```

Ceci permet de remplacer la notation

```
std::cout <<"HelloWorld"<<std::endl;
```

par

```
cout<<"Hello Word"<<endl;
```

Exclusion de méthodes d'un namespace

Pour exclure une méthode d'un namespace, il suffit de déclarer le namespace individuellement dans le fichier .h chacun des
méthodes et d'exclure la ou les méthodes concernées.

```
class Nom {
  namespace Client::Nom() {};
   char *Activite();
   // activité ne fait pas partie du namespace
  namespace  Client::Ville() {} ;
};
```

Espace de nommage anonyme

- Un espace de nommage peut être défini sans nom avec la syntaxe suivante : `namespace {... }`
- Dans ce cas là il est anonyme et on ne peut plus l'appeler sauf dans le fichier dans lequel il a été déclaré
- Ce mécanisme garantit que dans tout l'espace de nommage anonyme on n'aura pas deux fois la même variable déclarée

Templates

Rubriques

- Présentation
- Plus loin
- Foncteurs

Présentation

Présentation des Templates

- Les fonctions template sont donc des fonctions qui peuvent travailler sur des objets dont le type est un type générique (c'est-à-dire un type quelconque), ou qui peuvent êtres paramétrés par une constante de type intégral. (un type intégral désigne bool, char, signed char, unsigned char, char16_t, char32_t, wchar_t, short, int, long, long long, unsigned short, unsigned int, unsigned long, unsigned long long)
- Les classes template sont des classes qui contiennent des membres dont le type est générique ou qui dépendent d'un paramètre intégral.
- En général, la génération du code a lieu lors d'une opération au cours de laquelle les types génériques sont remplacés par des vrais types et les paramètres de type intégral prennent leur valeur.
- Cette opération s'appelle l'instanciation des template. Elle a lieu lorsqu'on utilise la fonction ou la classe template pour la première fois.
- Les types réels à utiliser à la place des types génériques sont déterminés lors de cette première utilisation par le compilateur, soit implicitement à partir du contexte d'utilisation du template, soit par les paramètres donnés explicitement par le programmeur.

Définition d'un type générique

- Les template qui sont des types génériques sont déclarés par la syntaxe suivante :

```
template <class|typename nom[=type][, class|typename nom[=type] [...]>
```

- nom est le nom que l'on donne au type générique dans cette déclaration on le note généralementT, U, V etc .
- Le mot clé class a ici exactement la signification de « type ». Il peut d'ailleurs être remplacé le mot clé typename.
- La même déclaration peut être utilisée pour déclarer un nombre arbitraire de types génériques, en les séparant par des virgules.
- Les paramètres template qui sont des types peuvent prendre des valeurs par défaut, en faisant suivre le nom du paramètre d'un signe égal et de la valeur. La valeur par défaut doit être un type déjà déclaré.

Définition de classes template

- La déclaration et la définition d'une classe template se font comme celles d'une fonction template : elles doivent être précédées de la déclaration template des types génériques.

```
template <paramètres_template>
class|struct|union nom;
```

- Dans la définition des classes template, si les méthodes de la classe ne sont pas définies dans la déclaration de la classe, elles devront elles aussi être déclarées template :

```
template <paramètres_template>
type classe<paramètres>::nom(paramètres_méthode)
{
    ...
}
```

Patrons de fonction

- Un patron de fonction est une sorte de “fonction potentielle” : c’est un canevas, un plan à partir duquel le compilateur est capable, en fonction des besoins, de générer plusieurs fonctions.
- La fonction utilisera le mot clé template suivi des paramètres template et utilisera les types définis comme s’il s’agissait de types existants:
- **Exemple** : cette fonction pourra être utilisée sur les types qui comprennent l’opérateur >
- Pas de référence pour les paramètres des fonctions en dessous de C++17

```
template<class  T>
T maximum( T arg1 , T arg2){
    return (arg1 > arg2) ? arg1 : arg2 ;
}
```

Utilisation des templates

```
1  #include <iostream>
2  using namespace std;
3  template<typename T>
4      T maximum( T arg1 , T arg2) {
5          return (arg1 > arg2) ? arg1 : arg2 ;
6      }
7  main()
8  {
9     int i=3;int j=2;
10    char a='a';  char b='b';
11    cout<< maximum(i,j) <<":" << maximum(a,b)  <<":"<<maximum(3,4) <<endl;
12 }
```

Piège sur les pointeurs

- Imaginons dans notre exemple précédent l'instanciation suivante :

```
1  const char * s1 = "toto";
2  const char * s2 = "titi";
3  cout << Maximum( s1, s2 ) << endl;
```

- c'est bien les adresses que nous comparons et le plus grand n'est pas toto mais titi puisque son adresse est plus élevé !

Patron de fonction à l'intérieur d'une classe

```
class StringBuilder
{
public:
    template<typename T>
    void Append( const T & t )  {
        this->oss << t;
    }

    std::string GetString() const   {
        return this->oss.str();
    }
private:
    std::ostringstream oss;
};
```

Plus loin

Instanciation implicite et explicite

- Dans notre exemple précédent nous avons supposé que les deux types étaient les mêmes.
- L'instruction `maximum (4,3)` est claire pour le compilateur, c'est une instanciation implicite
- L'instruction `maximum(3.2,2)` aurait provoqué une erreur de compilation
- Pour lever cette erreur nous aurions pu forcer le type avec l'instruction suivante `Maximum<double>(3.2,2)` ou même `Maximum(3.2,(double)2)`
- L'emploi du symbole `<double>` est une instanciation explicite

Spécialisation

On peut spécialiser un template pour certains types donnés avec utilisation du symbole <>

```
template <typename T>
void QuiSuisJe( const T & x )  {
    std::cout << "Je ne sais pas" << std::endl;
}
template <>
void QuiSuisJe<int>( const int & x )  {
    std::cout << "Je suis un int" << std::endl;
}
template <>
void QuiSuisJe<MaClasse>( const MaClasse & x )  {
    std::cout << "Je suis un MaClasse" << std::endl;
}
```

Patrons de classe

```
template <typename montype> class
template <typename T, typename V =int>
```

- On utilise dans les programmes le ou les types génériques dérivés
- Ligne 2, on peut définir un type par défaut ici *int*, par défaut le type sera int
- Il est interdit déclarer un membre template virtuel `template<typename T> virtual bool machin( int x);`

Exemple

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

À L'utilisation nous aurons :

```
mypair<int> myobject (115, 36);
mypair<double> myfloats (3.0, 2.18);
```

Template sur les opérateurs

Un opérateur peut être surchargé comme le montre l'exemple suivant :

```
template<typename T>
bool operator==(const String<T>& s1, const String<T>& s2)
{
if (s1.size()!=s2.siz e()) return false;
for (auto i = 0; i!=s1.size(); ++i)
if (s1[i]!=s2[i]) return false;
return true;
}
```

Performances

- Les templates sont un moyen d'écrire du code générique
- Quelques règles d'implémentation cachées nécessitent d'explorer la documentation ou de recherche les messages d'erreur via Google
- Donc le codage n'est pas très simple.
- En terme de compilation, l'utilisation des templates (et pas la définition du code avec des templates) prend plus de temps
- Pendant l'exécution les programmes ne sont pas impactés en terme de performance.
- La taille des programmes par contre augmente puisqu'on a duplication du code en fonction de chaque instanciation

Swap en template

- Définir une fonction template swap(T i, T j) qui swappe deux objets
- L'appeler avec deux entiers
- L'appeler avec deux élements d'une classe définie

Foncteurs

Foncteurs

- Un foncteur est la surcharge non plus d'un opérateur classique mais de l'opérateur ()
- Pour surcharger l'opérateur () on procède à la surcharge de l'opérateur `operator()()` qui se démarque de `operateur()` pour ne pas apporter de confusion avec un constructeur.
- **Exemple :**

```
#include <iostream>
class absValue {
public:
    float operator()(float f) { return f > 0 ? f: -f;}
};
int main( ) {
    float f = -3.8x;
    absValue aObj;  float abs_f = aObj(f);
    std::cout << "f = " << f << "  => " << abs_f << std::endl;
}
```

Itérateurs

A quoi servent les foncteurs ?

- On aurait pu utiliser des constructeurs d'affectation plutôt que d'utiliser la notion de foncteur.
- Or nous le verrons dans l'étude la STL notamment sur les vecteurs, la fonction `foreach` est utilisable comme suit avec un foncteur en dernier argument

```
vector<int> vect;
for (int i=1; i<10; ++i) {vect.push_back(i);};
for_each (vect.begin(), vect.end(), print_it);
```

Cas d'utilisation

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first,
       InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;
}
```

Programmation

Rubriques

- Conseils
- RTTI
- Pointeurs intelligents

Conseils

Les singletons

- Le singleton est une technique permettant de répondre aux deux besoins suivants :
  1. assurer une instance unique d'elle-même visant à protéger des informations tout le long de l'exécution du programme
  2. empêcher à d'autres programmeurs du projet d'en créer d'autres.
- Elle est utilisée par exemple pour garder des pointeurs sur des données globales : la langue de l'application, la connexion avec la base de données, la version du programme

Méthode de déploiement

- Déclarer une classe dont le constructeur est privé ce qui empêche toute nouvelle instanciation à l'extérieur de la classe
- Déclarer son destructeur privé afin de prévenir toute destruction de l'objet instancié
- Déclarer une variable membre pointeur statique vers cette classe, accessible par tous les objets

```
class Singleton {
private:
    Singleton (): m_version ("1.0") { }
    ~Singleton () { }
    char[200] m_version;
    static Singleton *m_singleton;
public:
  int getVersion () { return m_version; }
  // Fonctions de création et destruction du singleton
  static Singleton *getInstance () {
    if (m_singleton == NULL){m_singleton=new Singleton;}
    else { std::cout << " existe déjà !" <<std::endl;}
    return m_singleton; }
};
```

Programmation robuste

- Constructeur de copie : écrire un constructeur de copie dès que des pointeurs sur des objets externes existent dans la classe
- Opérateur d'affectation d'objet : même remarque
- Destructeur virtuel : on est sûr de bien détruire la classe fille et non la classe mère si on l'accède depuis un pointeur
- Utilisation des données constantes : en paramètre des fonctions et en regard des méthodes qui ne modifient pas l'objet permet de s'aider du compilateur pour être sûr que les objets ne sont pas modifiés

RTTI

Définition

- **RTTI** est l'acronyme de Real Time Type Information
- C'est une notion qui permet de déterminer le type d'un objet non plus à la compilation mais dynamiquement au moment de l'exécution du programme.
- La fonction maîtresse du RTTI est une fonction de cast dynamique `dynamic_cast` qui prend deux arguments :
  - le premier est le type passé entre les signes < et >
  - le second un pointeur sur ce type
- Si le dynamic cast aboutit, cela veut dire qu'au moment de l'exécution le type de l'objet en deuxième argument est bien du type entre crochet, sinon le pointeur NULL est renvoyé

Exemple pratique

- Imaginons la procédure `onClick ( widget *p)` appelée avec l'objet graphiqué cliqué
- Imaginons que les classes widget ont des classes dérivées : comme `bouton`, `combobox`, `lineedit`
- La séquence suivante permettra de déterminer quel type de widget a été appelé

```
onClick ( widget *p) {
if (  bouton * pb=dynamic_cast<bouton *>(p)) {
    // pb pointe sur un bouton
}
else {
    // pb est nul car on a pas cliqué sur  un bouton
}
}
```

Règles à respecter

- On ne peut faire un `dynamic_cast` que sur un membre accessible c'est à dire public, le cast dynamique respectant les portées privées ou protégées.
- Le cast dynamique suppose que le pointeur en argument porte sur une classe polymorphique c'est à dire que la classe pointée possède des fonctions virtuelles. En effet, sans polymporphisme, le cast ne peux savoir si le type donné est valide ou pas.
- Le cast peut se faire sur des classes hérités plusieurs fois avec les mêmes contraintes.
- Dans les cas d'héritage en diamant, quand il y a ambiguïté le résultat est NULL

TypeId

Syntaxe : `typeid(type)` ou `typeid(expression)` - Le résultat est une valeur de type `const type_info&` où *type_info* est une classe de la bibliothèque standard comportant au moins les membres suivants :

```
class type_info {
public:
const char *name() const;
int operator==(const type_info&) const;
int operator!=(const type_info&) const;
int before(const type_info&) const;
...
};
```

Pointeurs intelligents

Problématique des pointeurs

- Les pointeurs intelligents (smart pointer ou managed pointer) visent à corriger 4 problèmes que l'ont rencontre avec les pointeurs classiques (appelés pointeurs nus)
  1. Oubli ou impossibilité de désallouer de la mémoire allouée par un pointeur
  2. Double désallocation de mémoire allouée par un pointeur
  3. Accéder à une zone mémoire référencée par un pointeur invalide
  4. On ne sait pas trop ce que fera le destructeur au moment de la désallocation
- Les pointeurs intelligents font partie du language C++11
- Ils ne sont pas nécessaires pour ceux qui utilisent des libraires munies de *garbage collector* comme Qt
- Ils sont parfois appelés limited garbage collectors.

Présentation des pointeurs intelligents

- Ils se nomment *shared_ptr*, *weak_ptr* et *unique_ptr*
- Il sont accessibles via un `#include <memory>` via l'espace de nom *std*
- Ils sont la base de la RAII (Resource Acquisition Is Initialization) ; le principe de RAII est de donner la propriété de toute ressource allouée par le tas, par exemple la mémoire allouée dynamiquement ou les handles d'objet système, à un objet alloué par la pile dont le destructeur contient le code de suppression ou de libération de la ressource ainsi que le code de nettoyage associé.
- Un pointeur intelligent est donc une classe contenant un pointeur réel ainsi que des informations complémentaires notamment le code de son destructeur
- Suivant l'implémentation de votre compilateur vous pouvez avoir à passer l'option `-std=c++11` à g++

unique_ptr

- Remplace le pointeur *auto_ptr* qui est devenu obsolète
- Comme son nom l'indique, il gère de manière unique une ressource. Il ne peut être copié ou passé en paramètre de fonction par valeur, ni utilisé dans un algorithme STL de recopie.
- Il peut juste être déplacé vers un autre objet ** de sorte que quand le deuxième objet possède l'unique le premier ne le possède plus.
- On peut bien sûr utiliser plusieurs pointeurs ** dans un programme, mais un seul sur un objet.
- **Syntaxe :** `template <class T> class std::;`

Exemple de unique_pointer

```
#include <iostream>
#include <memory>

int main ()
{
  std::unique_ptr<int> p1 ( new int(4)) ;
  std::unique_ptr<int> p2 ( new int (2));
  std::unique_ptr<int> p3 =p2  ; // Erreur de compilation
  std::unique_ptr<int> p4= move(p2);
 }
```

shared_ptr

- Il est conçu pour des scénarios dans lesquels plusieurs propriétaires peuvent devoir gérer la durée de vie de l'objet en mémoire
- On peut le copier, le passer en valeur ou l'assigner à d'autres instance **
- À chacune de ces copies, un compteur interne est incrémenté
- Chaque fois qu'un objet passe hors de portée, le compteur est décrémenté
- Quand le compte est à zéro le pointeur est libéré
- On peut libérer le pointeur par un appel à la fonction membre *reset*. La fonction reset avec argument permet également d'allouer un nouveau pointeur.
- **Syntaxe :** `template <class T> class std::;`

Exemple de shared pointer

```
#include <iostream>
#include <memory>


int main ()
{
  std::shared_ptr<int> p1 ( new int(4)) ;
  std::shared_ptr<int> p2 ( new int(3));
  std::shared_ptr<int> p3=p1;
  std::cout <<p1.use_count()<<"\t"
            <<p2.use_count()<<"\t"
            <<p3.use_count()<<std::endl;
  // 2  1   2
}
```

weak_ptr

- *std::* est un pointeur qui prend une référence sur un objet tenu par un pointeur partagé sans augmenter le compteur
- On évite d'utiliser un ** sauf quand on tombe sur des cas de références cycliques
- On va voir l'exemple de la page suivante un cas d'utilisation avec une illustration d'une fuite mémoire sur des références cycliques

```
#include <iostream>
#include <memory>
using namespace std;
class B;
class A{ public:   shared_ptr<B> myB; };
class B {  public:   shared_ptr<A> myA;  };
main(){
shared_ptr<A> a (new A); shared_ptr<B> b ( new B);
<A> aprime =a;
cout << a.use_count() << ", " << b.use_count() << endl;
a->myB = b;
cout << a.use_count() << ", " << b.use_count() << endl;
b->myA = a;
cout << a.use_count() << ", " << b.use_count() << endl;
```

Avantages et inconvénients des pointeurs intelligents

- **Avantages** :
  - Permet dans des programmes imbriqués d'éviter les pertes de mémoires ou les destructions double de ressources.
  - Est plus robuste dans les mécanismes d'exceptions
- **Inconvénients** :
  - Alourdit le code
  - À ne pas utiliser avec un environnement qui possède un garbage collector
  - Continuer à utiliser les pointeurs dans des petites portions de code
  - Comme tous les types génériques, le procédé est lent à la compilation
  - Névite pas tous les problèmes comme le montrait le code précédent

La STL

Rubriques

- Les types
- Conteneurs
- Algorithmes

Les types

Présentation

- La Standard Template Library implémente un grand nombre de classes template décrivant des containers génériques pour le langage C++
- Elle fournit en outre des algorithmes qui permettent de manipuler aisément ces containers (pour les initialiser, rechercher des valeurs, ...).
- La STL introduit également le concept d'iterator qui permet de parcourir très facilement un container en s'affranchissant complètement de la manière dont il est implémenté.
- Les concepts développés dans la STL sont étendus par la librairie boost qui fournit elle même des améliorations aux futures version de la STL

Les types de base (non STL)

- Les booléens :
  - **bool :** peut prendre la valeur true ou false
  - **Déclaration** : `bool fin = false; bool continue= a>b;`
- les caractères :
  - **char :** par défaut sur 8 bits
  - **signed char :** signé avec des valeurs positives ou négatives
  - **unsigned char:** non signés c'est à dire positifs des ASCII 0 à 255
  - **wchar_t :** caractères unicodes
  - **char16_t :** caractères UTF16
  - **char32_t :** caractères UTF32
  - **Exemples :**

```
char a='a'; (int)c=97;
```

Les types de base : les entiers et les flottants (non STL)

- Entiers : ce sont les entiers sous leurs différentes représentation
  - **int, signed int, unsigned int,short int,long int ,long long int** : ces types sont variables en fonction du processeur
  - `#include <cstdint>` permet de disposer des types int64_t,uint_fast16_t,
  - 3 est un int, 3U est un unsigned int, 3L est un long integer, 0xOFUL est un unsigned Long, 2ULL est unsigned Long Long,
- Flottants :
  - **double :** 1.23 .23 0.23 1. 1.0 1.2e10 1.23e-15
  - **float :** 3.14159265f 2.0f 2.997925F 2.9e-3f (suffixés par f ou F)
  - **long double :** 3.14159265L 2.0L 0.997925L 2.9e-3L (suffixés par l ou L)

Les types de base : les string (std)

- Les string font partie de la librairie std, on les utilse donc avec la directive `using namespace std;`
- Ne plus utiliser les char[10] ou `char *`
- Les string définissent un type chaînes de caractères avec des méthodes associées, il ne faut pas confondre :
  - les string du C obtenus par `#include <string.h>`
  - les string du C++ obtenus par `#include <string>`
- On trouvera dans la référence à l'adresse http://www.cplusplus.com/reference/string/string/
- Nous verrons un peu plus loin l'utilisation des itérateurs

Quelques utilisations des string

```
1  std::string  Nom="Dupont";
2  Nom=+"marcem";
3  int NbCar=Nom.size();
4  Nom[NbCar-1]='l';
5  Nom.at(6)='M';
6  if ( ! Nom.empty() ) {}
```

1. Initialisation et dimensionnement automatique
2. Concaténation
3. Accès à la taille
4. Accès au dernier élément.
5. Idem par la fonction **at**
6. Teste si la chaîne est vide

Les conteneurs : vector (std)

- Les vecteurs sont des éléments contigus en mémoire à la différence des list que nous verrons plus loin.
- Ressemblent aux tableaux, mais sans déclaration de taille préliminaires
- Les redimensionnement sont automatiques mais si un a un vecteur le 100 Mo on aura une réallocation de 100 Mo sur certaines tâches.
- Nous pouvons fabriquer un vecteur de tout type
- Obtenus par `#include <vector>`
- On trouvera dans la référence à l'adresse http://www.cplusplus.com/reference/vector
- Nous verrons la notion d'itérateur plus loin

Quelques utilisations du vector

```
1  std::vector<int> a(5,10);
2  a.push_back(4);
3  a.push_back(0);
4  int nb=a.size();
5  int size=a.capacity();
```

1. Déclaration du vecteur avec 5 entiers à la valeur 10
2. Ajout de 4 à la fin avec réallocation de tout le bloc de mémoire !)
3. Ajout de 0 à la fin
4. Nombre d'éléments
5. Taille mémoire occupée par le vecteur

Conteneurs

Les itérateurs

- permettent de balayer une liste, un vecteur, un string et tous les types composites de la STL
- Chaque fois qu'un itérateur est présent on peut l'accéder via la syntaxe `Objet::iterator variableiterator` Par exemple pour les vecteur que nous venons de voir l'itérateur sera `std::vector<int>::iterator it`
- Nous disposons d'itérateurs du type
  - **begin** qui permet de référencer le premier élément, sur certains conteneurs begin+1 est le suivant.
  - **end** qui permet de référencer le dernier élément. Une fois (sur le dernier élément ne pas faire `*it` !
  - l'opérateur ++ permet de référencer l'élément suivant (+=n le nième)
  - le contenu de l'itérateur sera `*iterator` (ce n'est pas un pointeur valeur=&iterator est interdit)

Les itérateurs : exemple

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main ()
5  {
6    vector<int> myvector;
7    for (int i=1; i<=5; i++) myvector.push_back(i);
8    for (vector<int>::iterator it = myvector.begin() ;
9         it != myvector.end();
10        ++it)
11      cout << ' ' << *it;
12   return 0;
13 }
```

Quelques remarques complémentaires sur les itérateurs

- Certains itérateurs se retrouvent dans tous les conteneurs,
- Mais tous les conteneurs n'implémentent pas les mêmes itérateurs
- L'itérateur end() référence un élément vide.
- *rbegin* et *rend* permettent de balayer du dernier élément vers le premier mais on utilise toujours l'incrément ++it pour itérer.
- en ajoutant c devant begin, ou rbegin ainsi que devant end et rend, on force l'itérateur à pointer sur un élément non modifiable par `*it`.

Les conteneurs : list (std)

- Les listes gèrent des éléments non contigus en mémoire, ce qui veut dire que l'ajout ultérieur ne réallouera qu'un élément.
- Nous pouvons fabriquer une liste de tout type
- Obtenu par un `#include <list>`
- La documentation de référence est à l'adresse http://www.cplusplus.com/reference/list
- Remarque : les conteneurs list, vector (ainsi que array et deque) sont appelés conteneurs de séquences

Quelques utilisations des List

```
use namespace std;
list <string> liste;
liste.push_back("Gilles");
liste.push_front("Albert");
liste.push_back("Zoé");
liste.sort();
liste.reverse();
for ( list<string>::iterator it= liste.begin();
       it != liste.end();
        it++)
  cout <<*it<<endl;
```

Conteneurs associatifs : Map

- On appelle un conteneur associatif, un conteneur indexé par un type en général chaînes de caractères, c'est à dire où l'on accède aux éléments non plus par leur position, mais par une clé.
- Un Map permet d'accéder ainsi aux éléments : `typedef pair<const Key, T> value_type;`
- L'itérateur est un peu plus compliqué :
  - on utilise encore les opérateurs begin et end (ainsi que leurs dérivés constant ou reverse)
  - on utilise l'incrémentation ++
  - par contre `*it`, n'a pas de sens ici puisque nous avons deux partie la clé et la valeur. On y accède par `it->first` et `it->second`

Exemple de conteneur associatif Map

```
#include <iostream>
#include <map>
#include <string>

using namespace std;
int main ()
{
  map<std::string,int> Age;
  Age["Pierre"] = 37;
  Age["Marie"] = 65;
  Age["Jennifer"] = 24;

  for (map<string,int>::iterator it=Age.begin();
       it!=Age.end(); ++it)
   cout << it->first <<" => "<< it->second<< endl;
}
```

Autres conteneurs

- **array :** éléments contigues en mémoire taille connue à la compilation (C++11)
- **queue :** file fifo
- **stack :** file lifo
- **deque :** idem vector mais avec *push_front* en plus
- **priority_queue :** file ordonnée (C++11)
- **unordered_set:** table de hashage (C++11)
- **unordered_map:** (C++11)

Algorithmes

Algorithmes

- Les algorithmes s'appuient sur les itérateurs, ce sont des facilités d'écriture
- On les trouve dans la documentation de référence en premier item de la partie *other*
- Ces algorithmes sont classés par catégorie, nous allons en étudier quelques uns.
  - ceux qui ne modifient pas les conteneurs
  - ceux qui peuvent les modifier
  - ceux appelé partitions
  - ceux spécialisés dans le tri
  - ceux spécialisés dans la recherche
  - ceux spécialisés dans les fusions de conteneurs
  - ceux dans la gestion des min/max
  - dans les comparaisons lexicographiques

Fill : remplissage

- **Usage :**

```
template <class Iterator, class T>
void fill (Iterator first, Iterator last, const T& val);
```

- remplit avec *val* la plage allant de *first* à *end*.
- **Exemple :**

```
std::vector<int> vect(8); // vect: 0 0 0 0 0 0 0 0
std::fill (myvector.begin(),vect.begin()+4,5);
//vect: 5 5 5 5 0 0 0
```

Swap : échange

- **Usage :**

```
template <class T> void swap ( T& a, T& b )
```

- échange les deux variables
- **Exemple :**

```
int x=10, y=20;  // x:10 y:20
std::swap(x,y);  // x:20 y:10
std::vector<int> foo (4,x), bar (6,y);
// foo:4*20 bar:6*10
std::swap(foo,bar);
// foo:6x10 bar:4x20
```

Copy : copie

- **Usage :**

```
template <class InputIt, class OutputIt>
  OutputIt copy (InputIt first, InputIt last,
  OutputIt result);
```

- copie une variable dans l'autre
- **Exemple :**

```
int init[]={10,20,30,40,50,60,70};
std::vector<int> vect(7);
std::copy ( init, init+7, myvector.begin() );
```

Rotate : réorganisation par rotation

- **Usage :**

```
template <class ForwardIt>
  void rotate (ForwardIt premier, ForwardIt milieu,
               ForwardIt dernier);
```

- effectue une rotation de sorte que l'élément pointé par *milieu* devienne le premier
- **Exemple :**

```
std::vector<int> vect;
for (int i=1; i<10; ++i) vect.push_back(i);
// 1 2 3 4 5 6 7 8 9
std::rotate(vect.begin(),vect.begin()+3,vect.end());
// 4 5 6 7 8 9 1 2 3
```

Opération sur les ensembles

- Le set est un conteneur d'éléments ordonnés et uniques
- Il permet des recherches très rapides par dichotomie
- Le set est organisé en mémoire sous forme d'un arbre binaire
- Sa documentation se trouve à l'adresse
  http://www.cplusplus.com/reference/set/set

Exemple d'utilisation

```
1  #include <iostream>
2  #include <set>
3  using namespace std;
4  int main(){
5  set <int> S;
6  S.insert(9); S.insert(10);S.insert(10);
7  S.insert(1);S.insert(2);S.insert(14);
8  S.insert(-1);
9  for ( set<int>::iterator it = S.begin();
10       it!=S.end(); ++it)
11       cout<<*it<<endl;
12 }
```

6 On insère deux fois l'élément 10.
7 Les éléments sortent ordonnés et uniques