Cours 2 - C++ pour les mathématiques appliquées

Concept d'encapsulation

La dernière fois ... -

Généralités sur le `C++`

... aujourd'hui

- Concept d'objet.
- Et quelques éléments de programmation procédurale.

Introduction à la notion d'objet en C++

Introduction

- `C++` supporte des paradigmes multiples : programmation procédurale, programmation objet, programmation évènementielle et programmation générique.
- Il est possible de n'utiliser qu'un seul paradigme pour écrire un code `C++` : ce choix se fait souvent au détriment de la maintenabilité et de l'élégance.

Objet en C++ (1)

Le supporte tous les éléments de la programmation objet :

- **Encapsulation des données** : les données peuvent avoir des données de visibilités différentes (publique, privée ou protégée). Avec le `C++` , l'encapsulation est une possibilité, elle n'est jamais forcée.
- **Abstraction** : mécanisme permettant de réduire le niveau de détail d'un code. On regroupe les classes selon des caractéristiques communes. Une abstraction bien conçue est généralement simple et s'utilise facilement.

Objet en C++ (2)

- **Héritage** : les propriétés et les fonctionnalités d'une classe existante peuvent être transmise par définition à une autre classe. Les principaux concepts sont l'extensibilité et la réutilisabilité.
- **Polymorphisme** : capacité de manipuler à l'exécution des objets en fonction de leur type et de leur utilisation. Le polymorphisme peut-être obtenu soit à la compilation en utilisant les surcharges d'opérateurs et de fonctions, soit à l'exécution en utilisant des fonctions virtuelles.

Pointeurs et références

Pointeurs : retour sur la définition

- Un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné.
- Exemple : `double *y` est un pointeur vers un `double` : contient l'adresse en mémoire où est stocké un `double`.
- Par défaut, le pointeur n'est pas initialisé : son contenu est celui de l'adresse mémoire avant que celui ne soit créé. Un pointeur non initialisé peut pointer sur une zone mémoire du système actuellement attribué à l'OS.
- Un pointeur doit nécessairement être typé.

Pointeurs (2)

- Le compilateur sait que l'objet créé est un pointeur : il ne lui alloue pas de mémoire, et il connaît le nombre de blocs mémoire qui suivent le bloc pointé.
- Pour accéder à l'adresse d'une variable on utilise l'opérateur `&`.

```
double  x =  4;
double *y = &x;
```

Passage d'argument : par pointeur

Modifier les valeurs sans faire de copie.

```
#include<iostream>
using namespace std::cout;
void xpy(int *x, int *y){
  (*x) += (*y); }
int xpy2(int *x, int *y){
  return (*x) + (*y); }
int xpy3(int x, int y){
  return x+y; }
int main() {
  int a = 5, b = 6;
  cout<<xpy3(a,b)<<endl;
  cout<<xpy2(&a,&b)<<endl;
  xpy(&a,&b);
  cout<<a<<endl;
  return 0;
}
```

A quoi ça sert?

- Permettent de manipuler de façon simple des données pouvant être volumineuses et complexes (passage de paramètres pour les fonctions).
- Permettent de créer des tableaux dont chacun des éléments est de taille différente.
- Permettent d'ajouter des éléments à un tableau en cours d'utilisation.
- Permettent de créer des chaînes.

Différence entre pointeur et référence

- On ne peut pas référencer une référence après sa définition.
- Une fois la référence créée, elle ne peut pas être réaffectée `std::reference_wrapper`.
- Une référence ne peut pas être définie comme : une référence se rapporte toujours à un objet.
- Une référence ne peut pas être non-initialisée : elle ne peut pas être réinitialisée, elle doit être initialisée dès sa création.
- Pour les classes, une référence s'initialise dans la liste d'initialisation.
- Les références peuvent également être utilisées pour le type de retour: cela permet d'éviter la copie d'objet.
- Tant que la référence existe, l'objet retourné existe.

Pointeur *this

- Stocke l'adresse mémoire de la classe instanciée.
- Permet l'accès par pointeur aux variables pour les fonctions pointeurs de la classe.
- N'est pas compté dans la taille de l'objet (`sizeof`).
- N'est pas accessible par les fonctions membres statiques.
- N'est pas modifiable.
- Permet d'identifier plus facilement les données membres des données externes.
- Permet d'enchaîner les appels de fonctions.
- Très utile dans le cas de surcharge d'opérateurs.

Références

- `C++` introduit un nouveau concept : référence = pointeur déguisé
- Syntaxe de déclaration : `Type & Identificateur;`
  - Obligatoirement initialisée à la création
  - Ne voit que la variable pointée (alias)

```
{
  int i=2;
  int &j=i;
  printf ( "i =% i \n" , i ) ;
  printf("j=%i\n",j); //impression d'un int
  int &k ;//Erreur de compilation
}
```

A l'affichage, nous avons `2` et `2` ; `j` pointe sur `i`

Si `i` est modifiée, `j` l'est également

Passage d'une référence

- On peut transmettre une référence dans une fonction

```
void ajoute1(double & x) {x=x + 1;};
int main() {
  double x = -1;
  ajoute1(x);
}
```

- Permet de modifier une variable externe à la fonction. Évite de transmettre un pointeur!

```
void ajoute1(double * x) {*x=(*x) + 1;};

int main() {
  double x = -1;
  ajoute1(&x); /*transmission du pointeur de x*/
}
```

Passage de référence (2)

Attention : faire la distinction entre les deux écritures

- `int  x; &x` pointeur sur un `int`
- `int &x; x` référence sur un `int`

Valeur à gauche

- Une référence est une valeur à gauche. L'objet pointé par la référence est modifiable.
- On peut rendre l'objet pointé non modifiable.

```
void ajoute1(const double & x) {x = x + 1;};

int main() {
  double x = -1;
  ajoute1(x);
}
```

On obtient une erreur de compilation

- Passage par référence est équivalent au passage par pointeur.
- Intérêt pour les gros objets en ajoutant la sécurité.

Retourner une référence

- Dans certains cas, il est nécessaire de retourner une référence

```
double vec[10];
double & element(const int & i) {
  return vec[-1];
}

int main() {
  for(int i=1;i<=10;i++) element(i)=1;
}
```

- `element(i)` est une référence et permet de modifier `vect[i]`.
- `double element(const int &i)` permet seulement de lire `double x = element(i)`

Retour d'une référence

- Il ne faut pas retourner une référence sur une variable locale qui est détruite à la sortie de la fonction

```
double & f(double x)
{
  double y;
  ...
  y = ...;
  return(y);
}

int main()
{
  double z=f(1);
  ...
}
```

- Avertissement avec g++

Les nouvelles références/pointeurs

- `nullptr` : remplace `NULL` .
- `unique_ptr` : attribuer une zone mémoire à un objet.
- `shared_ptr` : partager une zone mémoire entre plusieurs objets.
- `auto` : inférence automatique de type

```cpp
#include <iostream>
int main(){
  int my_array[5] = {1, 2, 3, 4, 5};
  // Boucle automatique sur le contenu d'un tableau
  for (auto&& x : my_array) {
    std::cout<< x << " ";
  }
  std::cout<<std::endl;
  return 0;
}
```

- `&&` : sémantique de déplacement mémoire.

Concept de classe

Notion d'objet

Une classe d'objet est une nouvelle structure définie par l'utilisateur et qui encapsule des membres

- des données
- des fonctions
- des types

Le concepteur doit assurer

- l'auto-consistance
- la robustesse
- la généricité

La conception nécessite une réflexion.

Déclaration

- Un objet est un type de variable `C++` défini par l'utilisateur.
- Déclaration avec `struct` ou `class`
- Opérateur d'accès à un membre : `Objet.membre`
- Pointeur sur un objet : `pObjet->membre`

Déclaration : exemple

```
class Polynome {
  public :
    static bool imprimer();
    int degree;
    double x, y, z;
    void imprimer(ostream &os) {
      std::cout<<"Coeffcients du mônome : "<<x<<" "<<y<<" "<<z;
    }
};

int main() {
  Polynome p; //instanciation d'un objet Polynome
  p.degree = 3;
  p.x = 0; p.y=0; Pp.z=0;
  p.imprimer();
  Polynome *pp = &p; //pointeur sur le Polynome p
  pp->x=1.;
}
```

Allocation

- Lors de l'instanciation d'un objet, il y a allocation automatique de l'espace nécessaire pour stocker les membres. Par exemple, pour la classe `Polynome`, il y a une allocation de 4 octets pour `degree` et 3×16 octets pour `x, y, z`.
- Lorsque l'objet est détruit, il y a libération automatique de l'espace alloué.
- Attention, la destruction ne libère pas l'espace alloué par un pointeur.

Allocation : exemple

```
class Polynome {
  public :
    int degree;
    double *coeffs_;
};

int main() {
  Polynome p; // instanciation d'un objet Polynome
  p.degree=3;
  p.coeffs_ = new double[p.degree];
}
```

Protection des attributs

Un membre d'une classe peut-être déclaré

- `public` accessible partout, par tout le monde
- `private` accessible seulement par la classe et dans la classe
- `protected` accessible dans la classe et par des relations d'héritage.

Protection des attributs : exemple

```
class Polynome {
  public :
    int degree;
  private :
    double *coeffs_;
  protected :
    bool active;
};

int main() {
  Polynome p; // instanciation d'un objet Point
  p.degree=3;
  p.coeffs_ = new double[p.degree]; // Erreur
}
```

Fonctions membres

- On peut déclarer dans une classe autant de fonctions que l'on souhaite.
- S'il n'y a pas de protection spéciale, ces fonctions ont accès à toutes les données de la classe
- On peut renvoyer une autoréférence
- `this` est le pointeur sur l'objet en cours. `*this` est l'objet lui même.

Fonctions membres : exemple

```
void zero() {
  for(int i=1; i<=p.degree; i++)
    this->coeffs_[i]= 0;
}

int main() {
  Polynome p;
  p.zero();
}
```

struct et class

On peut, dans certains cas, `struct` utiliser au lieu de `class`

- utiliser une classe si il y a un invariant; utiliser une structure si les attributs peuvent évoluer indépendamment.
- représenter la distinction entre une interface et une implémentation à l'aide d'une `class`.
- Utiliser une plutôt `class` qu'une `struct` si un membre est non public.
- `struct` est conseillé pour les types simples.

Structure

```
class Complex {
  public:
    double & reel(){return x_;}
    double & imag(){return y_;}
    double module() {
      return sqrt(x_*x_+y_*y_);
    }
  private :
    double x_, y_;
}
```

```
struct complex {
  double x,y;
  double module() {
    return sqrt(x*x+y*y);
  }
}
```

Structuration du code : déclaration

Afin de rendre le code plus lisible et modulable, il est souvent préférable de séparer la déclaration de la classe de sa définition.

```
#include "polynome.h"
Polynome::Polynome(int d, double v) {
  degree = d;
  coeff_= new double[degree];
  for(int i=0; i<degree; i++)
    coeffs_[i] = val;
}

Polynome::Polynome(const Polynome &p) {
  degree = p.degree;
  if (degree=0) return;
  coeffs_ = new double[degree];
  for (int i=0;i<degree; i++)
    coeffs_[i] = p.coeffs_[i];
}
```

L'implémentation dans l'entête correspond à un implicite.

Structuration du code : définition

Cette séparation permet également d'améliorer l'abstraction.

```
#once Polynome

class Polynome {
    double *coeffs_;
  public:
    int degree;
    Polynome(int d, double v=0);
    Polynome(const Polynome &p);
};
```

Déclarations anticipées

Les classes peuvent être déclarées en avance.

```
class_keyword name;
```

- C'est une déclaration incomplète de type. Il sera défini plus tard.
- Les types incomplets peuvent être utilisés pour les références et pointeurs.
- Utilisation
    - éviter les définitions circulaires.
    - réduire les temps de compilation des inclusions transitives.

Déclaration anticipée : exemple

En tête

```
class TypeInfo;
class Vecteur;
class Polynome{
    Vecteur & membre;
    void TypeInfo();
}
```

Implémentation

```
#include "Vecteur.h"

/* Implémentation */
```

Utilisation de const

- `const` permet de protéger une variable en écriture
- En C++, si un objet `const` ayant comme portée le fichier n'est pas explicitement défini comme `extern` , alors il sera visible uniquement dans le fichier. Les lignes suivantes sont équivalentes

```
const int            i = 1;
static const int     k = 3;
```

- En C++, `const` a quelques utilisations supplémentaires
  - pointeur sur un double constant `const double * p`
  - pointeur constant sur un double `double * const p`
  - référence et objet pointé sont constants `const double & p`

Surcharge avec const

- La surcharge de méthodes avec `const` introduit des comportements différents
  - une variable **lvalue** non constante préfère la méthode non `const`
  - une variable **lvalue** `const` ne peut utiliser qu'une méthode `const`

```
class Polynome {
    int getA() {return 1;};
    int getA() const {return 2};
    int getB() const {return getA()};
    int getC() {return getA()};
};
Foo &foo = /* ... */;
const Foo &cfoo = /* ... */;
```

- `foo.getA()==1, foo.getB() == 2, foo.getC() == 3`
- `cfoo.getA()==2, cfoo.getB() == 2, foo.getC() == erreur`

Qualificateur de référence

Les méthodes peuvent avoir une sémantique normale ou de rvalue. On utilise, selon les cas, les opérateurs `&` ou `&&`.

```
struct Polynome{
    std::string s;
    Polynome(std::string t="anonyme") :s{t}();
    void nom()& {
      std::cout<< s << "Instance normale" <<std::endl;}
    void nom()&&{
      std::cout<< s << "Instance temporaire"<<std::endl;}
};
Polynome p{"nul"};
p.nom(); // Instance normale
polynome{}.nom(); // Instance temporaire
```

La surcharge avec une référence est contaminante pour l'ensemble des surcharges.

Constructeur/Destructeur

Constructeur

- Il est fortement conseillé d'initialiser un objet à l'aide d'un constructeur `class` (arguments)
- Un constructeur est généralement déclaré `public`

Constructeur : exemple

```
class Polynome {
  public :
    unsigned int degree;
    Polynome(int d); // declaration du constructeur
  private :
    double *coeffs_;
};

Polynome::Polynome(int d) {       // implementation externe
  degree = d;
  coeffs_ = new double[degree];   // allocation
  for(int i=0;i<degree;i++) coeffs_[i] = 0;
}
```

Surcharge de constructeur

```
class Polynome {
  double *coeffs_;                    // private par défaut
public :
  unsigned int degree;
  Polynome(int d, double v=0);        // constructeur (degree, val)
  Polynome(const Polynome & p);       // constructeur par copie
};
Polynome::Polynome(unsigned int d, double val) {    // degree, val
  degree=d ;
  if (degree == 0) return ;
    coeffs_=new double[degree];
  for(int i=0;i<degree; i++) coeffs_[i]=val;
}
Polynome::Polynome(const Polynome & p) {            // par copie
  degree=p.degree;
  if (degree == 0) return ;
    coeffs_=new double[degree];
  for(int i=0;i<degree;i++) coeffs_[i]=p.coeffs_[i]; //recopie
}
```

Constructeur par copie

- Par défaut, un constructeur par recopie est toujours créé. Mais ce constructeur se contente de recopier les membres bit à bit.
- Il ne se préoccupe pas des zones mémoires allouées par l'utilisateur.
- Le constructeur par recopie est invoqué implicitement lors du transfert d'objet comme argument d'entrée ou de retour.
- `P=R` appelle le constructeur par copie.

Constructeur : utilisation

```
int main() {
  Polynome p1;                      // creation d'un Polynome par défaut
  Polynome p2=Polynome(3,0);        // creation d'un Polynome par valeurs
  Polynome p3=p1;                   // creation d'un Polynome par recopie
  Polynome *pp = new Polynome(3); // pointeur sur un Polynome
}
```

Appel de constructeur

- Lorsqu'un objet est initialisé, un constructeur approprié est appelé.
- Les arguments de la liste d'initialisation sont fournis au constructeur.
- Syntaxe : `class_type identifiant(arguments` ; ou `class_type identifiant{arguments};`
- Les différentes formes d'initialisation
  - défaut : `Polynome f;`
  - avec valeur : `Polynome {};`
  - directe : `Polynome f(42);`
  - liste : `Polynome f{42};`
  - copie : `Polynome f=g;`

Destructeur

- Lorsque un objet est détruit, il libère l'espace qu'il a créé pour stocker des membres mais pas l'espace alloué par l'utilisateur. On peut, à l'aide d'un destructeur libérer cet espace mémoire.

```
class Polynome {
  double *coeffs_;
public:
  int degree;

  Polynome(int d);
  ~Polynome();
}

Polynome::~Polynome() {
  delete [] coeffs_;
}
```

- Un destructeur n'a jamais d'argument et est public.

Conclusion

A retenir

- Gestion manuelle de la mémoire.
- Encapsulation des données.
- Protection des données.

La prochaine fois

- Opérateurs
- Héritage