Programmation Orientée Objet

Mise en oeuvre en C++

2 - Introduire de nouveaux types

Julien Yves ROLLAND (julien.rolland@univ-fcomte.fr)

Laboratoire de Mathématiques de Besançon
Université de Franche-Comté

2.1

Buts et objectifs

Buts et objectifs de ce chapitre.

1. Introduire les différents types définis par l'utilisateur ;
2. Présenter la notion d'objet et les classes en C++ ;
3. Définir les membres d'une classe et leurs accès ;
4. Détailler les fonctions essentielles d'une classe ;
5. Permettre l'utilisation des classes dans un contexte non orienté objets

2.2

Table des matières

2.3

1 UDT : Types Définis par l'Utilisateur

2.4

User Defined Types

Définition
En complément des *types fondamentaux* immédiatement exploitables, le langage permet l'introduction de nouveaux types. Ils sont construits à partir des types fondamentaux et nécessitent une définition avant d'être employés.

Ils sont répartis en catégories :

— Les énumérations "enum", une séquence ordonnée de constantes ;
— Les énumérations nommées "enum class", une énumération associée à un espace de nom ;
— Les structures " struct ", séquence d'éléments de type arbitraire ;
— Les unions "union", une structure ne contenant à un moment donné qu'un seul de ses membres ;
— Les classes " class ", un type complet regroupant des données et des fonctions.

2.5

1.1 Énumération

2.6

Les énumérations

Définition
Les énumérations enum et enum class sont des séquences de valeurs d'entier, spécifiée par l'utilisateur. Certaines de ces valeurs (souvent toutes) peuvent être nommées, elles sont alors appelées "énumérateurs".

```
1 enum Couleur{Bleu, Vert, Rouge};
2 enum class Forme{Triangle,
3                  Rectangle,
4                  Trapeze};
```

En pratique
Les énumérations sont principalement utilisées pour introduire des constantes nommées dans un code en lieu et place de constantes entières arbitraires.

2.7

Les énumération - suite

Représentation
Une énumération est représentée par un type entier signé ou non, chaque énumérateur possède une valeur.

```
1 enum Couleur : int {Bleu, Vert, Rouge};
2 enum class Forme : char {Triangle,
3                          Rectangle,
4                          Trapeze};
```

En pratique
Le type par défaut est int, il est très rarement modifié.

Valeurs
Les valeurs sont relatives (précédentes + 1), peuvent être spécifiquement définies et la première vaut 0 par défaut.

```
1 enum class Mois{
2   jan=1, feb, mar, avr, mai, jui,
3   juil=20, aou, sep, oct, nov, dec
4   };
```

2.8

Énumérations - encore

Différences enum et enum class

— L'enum class regroupe ses identificateurs dans un espace de nom et ses valeurs ne sont pas implicitement convertibles.

```
enum class Mois{
  jan=1, feb, mar, avr, mai, jui,
  juil, aou, sep, oct, nov, dec
};

Mois m = 1;            // Err int->Mois
Mois m1{1};            // Idem (sauf c++17)
int m2 = mai;          // Err scope + type
int m3 = Mois::nov;    // Err Mois->int

Mois actuel = Mois::oct;
Mois m4 = Mois(2);
int m5 = int(Mois::mar);
```

2.9

Énumérations - fin

Différences enum et enum class

— Un enum intègre ses identificateurs dans la portée de l'enum et ses valeurs sont implicitement convertibles vers des types entiers.

```
enum Mois{
  jan=1, feb, mar, avr, mai, jui,
  juil, aou, sep, oct, nov, dec
};

Mois m = 1;            // Err int->Mois
Mois m1{1};            // Idem
int m2 = mai;
int m3 = Mois::nov;

Mois actuel = Mois::oct;
Mois m4 = Mois(2);
int m5 = int(mar);
```

2.10

Utilisation des énumérations - 1

```
enum class Couleur{Bleu, Rouge, Vert};
enum Choix{oui, non, bof};
enum Gout{sale, sucre, bof}; // Error

// **** //
void f(Couleur c){
  switch(c){
  case Couleur::Bleu:
    // ...
    break;
  case Couleur::Vert:
    // ...
    break;
  // Rouge ?
  }
}
```

2.11

Utilisation des énumérations - 2

```
enum class Couleur{Bleu, Rouge, Vert};
enum Choix{oui, non, bof};
enum Gout{sale, sucre};

// **** //
void f2(Choix c){
  if(c == 9){/* ... */}     // Compile
  if(c == bof){/* .. */}
  if(c == sale){/* ... */} // Warning
}

void f3(Couleur c)
{
  if(c == 9){/* ... */}    // Error type
  if(c == Couleur::Bleu ){/* .. */}
  if(c == Gout::sucre){/* ... */} // Err typ
}
```

2.12

1.2 Structure

2.13

Les structures

Définition
Tout comme une "C-array" regroupe des éléments de même types, une struct dans sa forme la plus simple regroupe des éléments de types arbitraires. Les éléments d'une struct sont nommées ses "membres".

En pratique
Une struct déclare un nouveau type qui s'apparente aux class que nous verrons plus tard. Son usage principal reste cependant extrêmement limité : Créer un type permettant de regrouper et manipuler plusieurs types ensemble.

```
      struct Date {
        char  jour;
        short mois;
        int   annee;
        };          // Attention au ;
```

2.14

Déclaration et manipulation

Déclaration, initialisation
Une fois déclarée, une struct se manipule comme un type fondamental. Depuis la norme `c++11`, il est possible d'indiquer des valeurs par défaut pour ses membres lors de la déclaration.

Les variables déclarées du type d'une struct sont parfois nommées "instances" de ce type.

Accès aux membres

— On accède aux membres par l'opérateur "." (dot operator).

— Pour un pointeur de struct , il faut utiliser l'opérateur d'indirection de structure "−>".

Si "p" est un pointeur alors "p−>m" est équivalent à "(∗p).m" L'opérateur −> remplace donc l'opérateur d'indirection ∗.

2.15

Exemple de structure

```
struct Date {
  int jour{31};
  int mois{12};
  int annee{2017};
};

int main(){
  Date today{17, 10};

  cout << today.jour << endl;
  cout << today.mois << endl;
  cout << today.annee << endl;

  Date* pt = &today;

  cout << pt->mois << endl;
  cout << (*pt).annee << endl;
}
```

2.16

Union

Définition
Une union est une struct dont tous les membres ont la même adresse. Une union ne contient qu'un seul de ses membres à la fois.

```
union U{
  int x; double d;
};

U a;
a.x = 7; int x1 = a.x;
cout << x1 << endl;  // 7

a.d = 7.7; int x2 = a.x;
cout << x2 << endl;  // -858993459
```

En pratique
Il est déconseillé d'utiliser les union, le gain qu'elles apportent ne compense pas les soucis qu'elles posent à manipuler.

2.17

1.3 Classe

2.18

De la structure à la classe

Définition
Une class est l'évolution d'une struct, elle peut contenir :

- des types fondamentaux ;
- des UDT ;
- des fonctions.

Une instance d'une class est appelée "objet", la class est le plan, l'objet une réalisation.

En pratique
Formellement une struct **est** une class, à un ou deux détails près. Elle peut aussi contenir des fonctions (usage rare).

Appel sur une classe

- On accède aux membres (fonctions comme variables) avec l'opérateur ".";
- Pour un pointeur, on doit utiliser l'opérateur d'indirection de structure "->".

2.19

Vocabulaire de classe

Membres d'une classe
Comme la struct , les éléments d'une class sont nommés membres :

- les **données membres** constituent l'état de la classe, ses variables ;
- les **fonctions membres** représentent les actions possibles sur ce type.

Interface et implémentation

- On nomme **interface** l'ensemble de la déclaration d'une class qui est utilisé par l'utilisateur et auquel il a accès directement.
- On parle d'**implémentation** pour la partie de la déclaration auquel l'utilisateur n'a accès qu'indirectement, via l'interface.

2.20

Organisation du code

Déclaration
Comme vu précédemment, un fichier en-tête ".h" est utilisé pour regrouper les déclarations.

- Le mot clef "public :" indique la partie traitant l'interface
- Le mot clef " private :" sépare l'implémentation

Par défaut, les membres d'une class sont déclarés private alors que les membres d'une struct sont public .

Définition
De même, les définitions des fonctions membres d'une classe sont regroupées dans un fichier source ".cpp".

Une classe définit son propre espace de nom, il est donc obligatoire de le préciser si on souhaite définir une fonction membre en dehors de la portée de déclaration au format `nom_classe::nom_fonction`.

2.21

Exemple classe Date : Déclaration

Fichier date.h

```
1  #ifndef DATE_H
2  #define DATE_H
3
4  class Date{
5   public:
6    int month();
7    int day();
8    int year();
9
10   void affiche();
11   void reset();
12
13  private:
14   int y_, m_, d_;
15 };
16
17 #endif  // DATE_H
```

2.22

Exemple classe Date : Définition

Fichier date.cpp

```
#include "date.h"
#include <iostream>

int Date::month(){ return m_; }
int Date::day(){ return d_; }
int Date::year(){ return y_; }

void Date::affiche()
{
  std::cout << d_ << "/"
            << m_ << "/"
            << y_ << std::endl;
}

void Date::reset()
{
  d_ = 0; m_ = 0; y_ = 0;
}
```

2.23

Exemple classe Date : Utilisation

Fichier main.cpp

```
#include "date.h"
#include <iostream>

using namespace std;

int main()
{
  Date d;

  cout << d.year() <<endl;
  cout << d.month() <<endl;
  cout << d.day() <<endl;

  d.affiche();
  d.reset(); d.affiche();

  return 0;
}
```

2.24

2 Les classes en pratique

2.25

2.1 Construction

2.26

Comment construire une classe ?

Constructeur

```
  public:
    Date(int y, int m, int d);
```

Le constructeur est la fonction en charge de réserver la mémoire et d'initialiser les données membres.

- Le constructeur ne retourne rien
- Le constructeur doit avoir le même nom que la class
- Le constructeur peut être surchargé

```
Date anniversaire;          // Init error
Date unjour{12, 24, 2007};// Run-time error

Date hier(2017, 10, 16);   // C++98
Date hier{2017, 10, 16};
Date demain = {2017, 10, 18};
Date noel = Date{2017, 12, 24};
```

2.27

Définir des valeurs par défaut

Initialisation par la déclaration ("in-class member initialization")

```
private:
    int y_{0}, m_{0}, d_{0};
```

Constructeur par défaut
Le constructeur par défaut est le constructeur sans argument

```
Date d;        // ou Date d{};
// Appellera le constructeur suivant
Date::Date() {/* ... */}
```

Liste d'initialisation ("member initializer list")
Un constructeur (défaut ou non) peut utiliser une liste d'initialisation. Obligatoire pour une donnée const ou ref "&".

```
Date::Date() : y(2017), m{11} {/* ... */}
```

2.28

Définir des valeurs par défaut - exemples

Fichier date.cpp

```
// Par affectation
  Date::Date(){
    y_ = 1999;
    m_ = 12;
    d_ = 31;
  }

// Mieux : init (attention ordre declaration)
  Date::Date(): y_{1999}, m_{12}, d_{31}
  {}
```

Fichier main.cpp

```
int main()
{
  Date d;
  cout << d.year() <<endl;
}
```

2.29

Quelle initialisation choisir ?

- Préférer la liste d'initialisation dans un constructeur plutôt qu'une série d'affectation.
- Le corps d'un constructeur peut faire des opérations complexes, nécessitant des affectations.
- Les instructions d'un constructeur écrasent les valeurs saisies dans la déclaration.
- L'initialisation en déclaration ("in-class member initialization") n'existe pas en C++98.
- La liste d'initialisation est obligatoire pour les classes dérivées (cf. POO).

En pratique
L'initialisation "in-class" permet de définir des valeurs *indépendantes du contexte*.

- Elle allège la surcharge du constructeur : Inutile de répéter les valeurs par défaut des membres.
- Elle alloue aussi une valeur "utilisable" si le constructeur ne la fixe pas (exemple : "0" dans une date).

2.30

Constructeur et conversion

Conversion de type
Un constructeur n'ayant qu'un seul et unique argument définit une opération de conversion du type de l'argument vers le type de la classe.

```
class Complex{
  public:
    Complex(double);
    Complex(double, double);
   };

  Complex z1 = 3.1415;   // Implicit
  Complex z2 = Complex{3.0, 2.5}; // Explicit
```

Mot-clé " explicit " (en déclaration)
Interdit la conversion implicite et force l'utilisation de la forme explicite. Peut être utilisé sur toutes les variantes de constructeur.
calc_module({3.0, 2.5}) vs. calc_module(Complex{3.0, 2.5})

2.31

2.2 Destruction

2.32

Comment détruire des objets

Un constructeur peut avoir besoin d'allouer des ressources, de manipuler la mémoire, de créer des pointeurs, d'ouvrir des fichiers,... Dans ce cas, il est quasiment systématiquement obligatoire de préciser comment libérer ces ressources.

Destructeur

- Le destructeur, comme le constructeur, ne retourne rien
- Il ne prend aucun argument
- Il ne peut pas être surchargé (logique...)
- Son nom est celui de la classe précédé de "~"

```
public:
  ~Date();
```

Le destructeur d'une classe est appelé quand l'objet sort de la portée (ou à l'appel de delete pour un pointeur d'objet).

2.33

2.3 Copie

2.34

Copier une classe

La copie d'une classe est une opération qui apparaît :

- Quand on souhaite construire un objet à partir d'un autre objet de même classe

```
int a{5}; int b{a};
```

- Quand on construit un objet en affectant un objet de même classe

```
int a{5};
int b = a;
```

- Quand on affecte un objet avec les valeurs d'un autre de même classe

```
int a{5}; int b;
b = a;
```

Définition

- Les deux premiers cas sont une construction, ils appellent un constructeur spécifique : constructeur par copie.
- Le troisième cas est une affectation, il appelle l'opérateur d'affectation.

2.35

Prototype des copieurs

Constructeur par copie ("Copy constructor")

— C'est un constructeur, il ne retourne rien

— Il prend l'objet copié par référence (constante ou non)

```
MaClass( const MaClass& autre );
```

Opérateur d'affectation ("Copy assignment operator")

— Opérateur, il retourne une objet (valeur ou référence)

— Il prend l'objet copié de même type (référence, valeur, constant ou non)

```
MaClass& operator=(const MaClass& autre);
```

En pratique

On évitera les autres formes (causes de multiples problèmes).

2.36

Profondeur de copie

Quand on copie un pointeur, quel est la donnée d'intérêt ? L'adresse contenue dans le pointeur ou la valeur de l'objet pointé ?

Terminologie

— Une copie superficielle ("shallow copy") ne copie que l'adresse du pointeur, la donnée n'existe qu'en un seul emplacement mémoire.

— Une copie en profondeur ("deep copy") crée un second objet contenant un duplicata de la donnée initiale.

En pratique

— Les références et les pointeurs servent à manipuler les copies superficielles.

— Constructeurs par copie et opérateur d'affectation prennent en charge la copie en profondeur.

2.37

2.4 Déplacement

2.38

Déplacement d'objet

Il est parfois :

— inutile de conserver l'objet d'origine d'une copie ;

— impossible d'utiliser une simple référence car on souhaite être le propriétaire des données.

Dans ces cas, il est nécessaire de "déplacer" la donnée plutôt que la copier. De façon similaire à la copie, il existe donc :

Constructeur par déplacement ("Move constructor")

```
MaClass( MaClass&& autre );
```

Affectation par déplacement ("Move assignment")

```
MaClass& operator=(MaClass&& autre);
```

Pas de const pour l'argument, forcément...

2.39

Copie ou déplacement ?

Quand une voiture est vendue, on ne construit pas à l'identique une nouvelle voiture pour ensuite détruire la précédente ? On donne simplement les clefs au nouveau propriétaire. C'est la différence entre les opérations de copie et de déplacement.

C'est exactement la logique qu'emploie le compilateur : L'objet copié va-t-il être immédiatement détruit ?

L'opération de déplacement est utilisé :

— Dans une instruction return

— Avec la fonction std :: move (#include < utility >)

— Dans l'implémentation des conteneurs de la STL

En pratique

Cette notion n'est qu'une optimisation de la notion de copie et n'apparaît qu'au moment où la performance est recherchée. Quand le déplacement n'est pas défini, une copie a lieu.

2.40

2.5 Génération "par défaut"

2.41

Les opérations nécessaires (?) et suffisantes (?)

Les opérations essentielles [1]

- Constructeur par défaut (sans argument)
- Constructeur par copie
- Opérateur d'affectation par copie
- Constructeur par déplacement
- Opérateur d'affectation par déplacement
- Destructeur

Générateurs "par défaut" ("special member functions")
Si ces fonctions ne sont pas déclarées par l'utilisateur alors le compilateur génère automatiquement ces fonctions. Le comportement est très basique et ne gère - entre autre - pas la copie en profondeur.

2.42

Génération de base

Cas de dés-activation des fonctions "par défaut"

- La déclaration d'un constructeur (n'importe lequel) supprime le constructeur par défaut "par défaut".
- Si une des opérations de déplacement ou une des opérations de copie est déclarée, l'ensemble n'est pas généré.

Exceptions des copies

- Les déplacements sont liés à la génération, pas les copies
- Si un destructeur est déclaré, les opérations de copie "par défaut" sont générées. Une alerte est levée.

Activation, Dés-activation explicite
Dans la déclaration des prototypes des fonctions "par défaut" :

- "= default ;" génère automatiquement de la fonction
- "= delete ;" supprime la génération de la fonction

2.43

Conseils

- Considérer le constructeur, les affectations et le destructeur comme un trio : Si l'un est spécifié, le générateur "par défaut" n'est certainement pas compétent.
- Si un constructeur demande l'utilisation d'une ressource, il est certainement obligatoire de la libérer avec un destructeur.
- Privilégier liste d'initialisation aux affectations multiples.
- Si des valeurs par défaut sont nécessaires, commencer par une initialisation dans la déclaration et laisser le constructeur par défaut "par défaut".
- Le constructeur "par défaut" n'initialise pas les types de base et appelle le constructeur par défaut des objets membres.
- Ne pas trop se préoccuper du déplacement, la copie sert souvent de remplacement.
- (débutant) Être explicite sur les générateurs "par défaut".

2.44

3 Compléments

2.45

3.1 Mot clef const dans une classe

2.46

Qui est constant ? L'objet, la fonction, la donnée ?

Fonction membre constante
Une fonction membre constante déclare qu'elle ne modifie pas les données membres de la classe.

```
1 void affiche() const;
2 void Date::affiche() const {/* ... */}
```

1. Lien vers une note de blog récapitulative, norme c++11

Objet constant
Les fonctions membres constantes sont les seules autorisées sur un objet const.

```
const Date hier;
hier.affiche();
hier.reset();       // Error
```

En pratique
Si une fonction membre ne change pas l'état d'une classe, utiliser systématiquement const.

2.47

3.2 Gestion de la mémoire

2.48

Le retour du pointeur

Nous avons déjà vu la définition d'un pointeur : Il contient une adresse et permet de manipuler par indirection le contenu de la variable. Nous avons vu son intérêt dans la manipulation de structure de donnée basique comme le *C-array* : Inutile de manipuler toutes la structure, lourde en mémoire, un pointeur suffit.

Son rôle en C++ ne se limite ni à cette facilité ni au *C-array* : il permet de généraliser cet usage en permettant l'allocation et la gestion de la mémoire.

Un objet manipulé par pointeur :

— Peut profiter de l'allocation dynamique [2] : La mémoire réservée n'est pas limitée à la taille de la pile ($\approx$ 8Mo)
— Réduit son coût de manipulation (une adresse)
— N'a plus son cycle de vie lié à sa portée de création
— Permet de profiter du polymorphisme (cf. POO)

2.49

Exemple

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

int main(){
  char buffer[1024*1024*8]; // segfault

  return 0;
}
```

Passer par un pointeur permet l'opération :

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

int main(){
  char* buffer = new char[1024*1024*8];

  return 0;
}
```

2.50

3.3 Gestion dynamique de la mémoire

2.51

Les opérateurs de mémoire : Réserver, retourner

Allocation dynamique : new

```
MonType* ptype  = new MonType;
MonType* ptype2 = new MonType[5];
```

L'opérateur new (ou new[]) appelle le constructeur adéquate en fonction des arguments passés (ici constructeur par défaut) et retourne un pointeur.

Dés-allocation : delete

```
delete   ptype;
delete[] ptype2;
```

L'utilisateur est responsable de la dés-allocation et risque de fuite de mémoire.

2.52

2. L'allocation dynamique se fait sur le tas, et plus spécifiquement dans un espace nommé "free store".

Exemple

```
double* calc(int res_size, int max){
      double* p   = new double[max];
      double* res = new double[res_size];
      // Gros calcul
      delete[] p;
      return res;
}

double* res = calc(100, 100);
delete[] res;
```

Commentaire

— On peut créer une structure de donnée dans une fonction puis la "sortir".

— À chaque new, il apparaît un delete.

— Contrairement au *C-array* "dégénéré" en pointeur, l'allocation dynamique conserve une borne de début et de fin indépendantes de la portée, d'où delete [].

2.53

Commentaires et précisions

Précisions sur l'opérateur delete

— L'opérateur delete appliqué à un pointeur de classe appelle le destructeur de la classe.

— Il est dangereux d'appeler delete sur un pointeur un seconde fois

— Il n'y a aucun danger à appeler delete sur " nullptr "

Il est donc courant, une fois un objet détruit mais pour un pointeur non détruit de fixer sa valeur à " nullptr ".

Erreurs d'allocation

L'opérateur new gère les erreurs. Si l'allocation échoue, on peut "capter" l'erreur :

— À l'aide des exceptions (std :: bad_alloc)

— À l'aide du mot-clé "(std :: nothrow)" (#include <new>) qui fixe alors la valeur à nullptr

2.54

3.4 Permission des accès

2.55

Accès aux membres

Depuis l'extérieur

— "objet.membre" : membre de l'interface (public), on manipule un objet.

— "pointeur−>membre" : membre de l'interface (public), on manipule un pointeur d'objet.

— Toute l'implémentation (private) est interdite.

Depuis la classe

Pour les fonctions membres de la classe :

— Tous les membres la classe sont accessibles par leur nom.

— Les membres d'autres instances de la même classe (par exemple un argument de copie) aussi.

— Uniquement les membres public pour les autres classes.

Pointeur this (explicite)

Dans une fonction membre, pointe sur l'instance en cours.

2.56

Exemple

```
1  // Acceuil contient une Date en public
2  class Date, Acceuil;
3
4  Date::Date(const Date& i): y_{i.y_}
5  {
6    m_ = i.m_;
7    this->d_ = i.day();
8  }
9
10 void Acceuil::lancement(){
11   myDate.d_ += 1 ;                  //Erreur
12   myDate.affiche();
13 }
14
15 int main(){
16   Date* test = new Date;
17   Date date;
18   cout << test->month() << endl;
19   date.d_ = 2 * test->day();        // Erreur
20   }
```

2.57