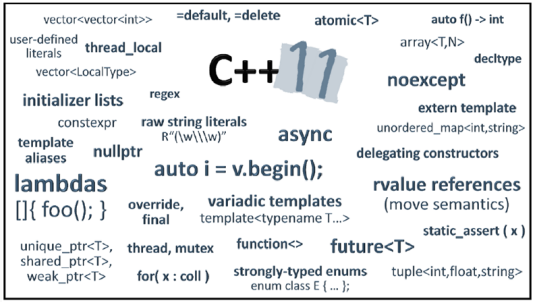
Programmation objet avancée en C++ moderne

Éléments de base et outils d'abstraction.

Mathias Paulin
IRIT-CNRS Université Paul Sabatier

Année universitaire 2016-2017



1. Introduction
2. Éléments de bases du langage
   - Bonjour Monde !
   - Types, déclaration, objets et valeurs
   - Pointeurs, tableaux et références
   - Structure, union et énumération
   - Instructions et structures de contrôle
3. Les mécanismes d'abstraction
   - Classes
   - Construction, destruction, copie et déplacement
   - Surcharge d'opérateurs
   - Gestion dynamique des erreurs

Ce cours n'est pas ...

- Un cours d'algorithmique ...
  - Savoir faire supposé acquis..
- Un cours de conception et programmation objet ...
  - Concepts supposés connus.
- Un cours d'introduction

Ce cours concerne ...

- Les éléments de langage en C++ moderne (C++11 - C++14).
- La programmation d'applications efficaces et maintenables.
- La meta-programmation en C++

Documentation de référence

- cppreference.com : référence en ligne complète pour le C++ moderne
  - existe en Français mais version anglaise vivement conseillée.
- Standard ISO (pour votre curiosité)
  - pour C++11 et C++14, accessibles depuis cppreference.com
- The C++ programming language (4th edition - C++11)- Bjarne Stroustrup
  - Livre de référence par le créateur du langage C++,
  - Livre ayant fortement inspiré ce cours.
- Effective Modern C++ - Scott Meyers
  - Livre orienté vers les bonnes pratiques du C++ moderne ... à lire

Éléments de bases du langage

Bonjour Monde !

Le langage C++ est un héritier du langage C et partage de nombreux concepts avec le C.

Attention

Le C++ n'est pas du C avec une couche objet. C'est un langage à part entière.

Anatomie d'un programme C++

```
#include <iostream>
#include <string>
/*
  Commentaires long
*/
int main() {
  std::string name;
  // commentaire court
  std::cin >> name;
  std::cout << "Good morning " << name << std::endl;
}
```

Éléments de bases du langage

Types, déclaration, objets et valeurs

Une déclaration introduit un identificateur dans le programme.
Elle spécifie un TYPE pour l'identificateur.

- Un TYPE définit un ensemble de valeurs possibles et d'opérations possibles sur un objet.
- Un OBJET est une zone mémoire qui contient une valeur d'un certain type.
- Une VALEUR est un ensemble de bits interprété selon un type.
- Une VARIABLE est un objet nommé.

Types prédéfinis :
Types scalaires : `bool`, `char`, `int`, `float`, ...

Type `void`

Types pointeurs, tableaux et références : `type*`, `type[]`, `type&`, `type&&`

Structure d'une déclaration.

Définie précisément par la grammaire du C++, une déclaration peut se résumer à la structure :

- préfixe optionnel (ex. static, virtual, ...),
- type de base (ex. float, const int, ...),
- déclarateur avec nom optionnel (ex. lenom, tableau[7], *(*)[], ...)
- suffixe optionnel (ex. const, noexcept, ...)
- initialisation ou corps optionnel (ex. ={7,5,3}, { return x; }, ...)

Exemple de déclaration

```
const char* kings[] = { "Antigonus", "Seleucus", "Ptolemy" };
```

Portée et visibilité

Une déclaration introduit un nom dans une zone de visibilité (*scope*).

- Local scope : visibilité locale au bloc de déclaration (block délimité par { }).
- Class scope : visibilité à l'intérieur d'une classe ou structure de déclaration.
- Namespace scope : visibilité à l'intérieur d'un espace de nom de déclaration.
- Global scope : déclaration à l'extérieur d'un namespace, classe ou fonction, visibilité globale.
- Statement scope : Déclaration dans une structure de contrôle (partie entre () de for, while, ...), visibilité dans la structure de contrôle.

Opérateur de portée

Les identificateurs peuvent se masquer, l'utilisation de l'opérateur de visibilité `::` permet de résoudre les problèmes de masquage.

Opérations :

Permettent de définir des combinaisons appropriées de valeurs :

Arithmétique :

- Unaires : +, -
- Binaire : +, -, *, /, %

Comparaisons :

- ==, !=
- <, >, <=, >=

Le C++ réalise toutes les conversions nécessaires sur les types pour que les opérations d'affectation et d'arithmétiques soient compatibles.

Opérateur d'affectation

$=$ est l'opérateur d'affectation.

$a = b$ copie le contenu de l'objet b dans l'objet a.

Opérateur d'initialisation

$\{\}$ est l'opérateur d'initialisation.

$Ta\{b\}$ construit et initialise un objet a de type T avec la valeur b

Exemple d'affectation/initialisation

```
double d1 = 2.3;
double d2 {2.3};
```

```
X a1 {v};      // C++11
X a2 (v);      // C++98
X a3 = {v};    // C99
X a4 = v;      // C K\&R
```

L'utilisation de la liste d'initialisation permet d'éviter les erreurs de conversion :

```
int i1 = 2.3;  // OK, i1 vaut 2
int i2 {2.3};  // Erreur : perte d'information par conversion
```

Règles et préconisations pour l'initialisation

- Un objet dont la visibilité est globale, namespace, ou déclaré comme static est initialisé par l'opérateur {} approprié au type.
- Un objet local et un objet créé sur le tas (alloué dynamiquement) n'est pas initialisé sauf s'il dispose d'un constructeur par défaut.
- Une constante DOIT être explicitement initialisée.
- Une variable peut ne pas être initialisée dans de rares circonstances.

Ne déclarer un nom que lorsque vous avez une valeur à lui affecter.

Initialisation de types utilisateurs

Les types utilisateurs doivent être définis de telle sorte qu'ils soient implicitement initialisés (constructeur par défaut).

Initializer list

L'opérateur d'initialisation { } peut être utilisé pour initialiser des objets plus complexes que les scalaires. On appelle *liste d'initialisation* la liste, délimitée par { }, des valeurs à utiliser pour initialiser un objet complexe.

Exemple de liste d'initialisation

```
int a[] {1, 2, 3, 4};
Struct S {int x; string s };
S s {1, "Hello"};
```

```
complex z1 {1, pi};
complex z2(1, pi); // construction
complex z3();      // fonction
```

Déduction de type et liste d'initialisation

Attention à la déduction de type. L'utilisation de la déclaration `auto` avec une liste d'initialisation ne permet pas toujours d'inférer le type (`std::initializer_list<T>`).

Le C++ offre la possibilité de déduire le type à partir d'une expression à travers deux mécanismes :

- `auto` pour déduire le type d'un objet à partir de son initialisation.
- `decltype(expr)` pour déduire le type d'une expression plus complexe qu'une initialisation comme le type de retour d'une fonction.

Limites de la déduction de type

`auto` et `decltype(expr)` ne permettent que la déduction d'un type déjà connu par le compilateur au moment de leurs évaluations.

Spécificateur de type auto

Remplace le type lorsque la déclaration d'une variable fait appel à un initialiseur.

```
int i1 = 123;
```

```
auto i2 = 123;
```

L'utilisation d'**auto** permet d'éviter des arrondis potentiellement involontaires.

```
char v1 = 1234; // 1234 est un int
auto v2 = 1234; // v2 est un int
```

L'utilisation d'**auto** est surtout utile dans les expressions complexes :

```
template<class T> void f1(std::vector<T>& arg) {
for (std::vector<T>::iterator p = arg.begin(); p!=arg.end(); ++p)
  p = 7;
for (auto p = arg.begin(); p!=arg.end(); ++p)
  p = 7;
}
```

Spécificateur de type `decltype`

Déduction de type sans initialiseur. Utile pour les fonctions et la programmation générique.

Calcul de la somme de 2 matrices génériques

Quel est le type des éléments d'une matrice qui est la somme de deux matrices dont les éléments sont de type T et de type U (potentiellement différents) ?

Spécificateur de type decltype

Déduction de type sans initialiseur. Utile pour les fonctions et la programmation générique.

Calcul de la somme de 2 matrices génériques

Quel est le type des éléments d'une matrice qui est la somme de deux matrices dont les éléments sont de type T et de type U (potentiellement différents) ?

```
template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b) -> Matrix<decltype(T{}+U{})> {
Matrix<decltype(T{}+U{})> res;
for (int i=0; i!=a.rows(); ++i)
  for (int j=0; j!=a.cols(); ++j)
    res(i,j) += a(i,j) + b(i,j);
return res;
}
```

Notion simple et la plus fondamentale d'un objet : Zone contigüe de stockage. Besoin de pouvoir nommer un objet indépendamment de sa construction (déclaration, allocation dynamique, temporaire ...)

lvalue

lvalue = nom pouvant apparaître en partie gauche d'une affectation.

Une lvalue doit pouvoir être identifiée par un nom, un pointeur ou une référence.

Attention les lvalue déclarées constantes ne sont pas être modifiables.

rvalue

rvalue = tout ce qui n'est pas une lvalue.

Une rvalue peut être déplacée en mémoire vers une autre zone, laissant la zone originale dans un état valide mais au contenu non spécifié.

Une rvalue permet d'éviter les copies/recopies de données temporaires.

Durée de vie d'un objet : de la fin de l'exécution de son constructeur au début de l'exécution de son destructeur.

Classification des objets par leur durée de vie

- Automatic : sans précision par le programme, objet créé à sa déclaration et détruit lors de sa fin de visibilité. Objets alloués sur la pile.
- Static : objets déclarés avec une visibilité globale ou namespace ou objets `static` déclarés dans une fonction ou une classe. Créés et initialisés une seule fois, durée de vie du programme. Objets alloués sur le tas.
- Free store : objets créés et détruits dynamiquement par les opérateurs `new` et `delete`. Contrôle direct de la durée de vie. Objets alloués sur le tas.
- Temporary : objets temporaires, résultats intermédiaires de calcul. Durée de vie en fonction de leur rôle. Ces objets sont automatiques.

Possibilité de définir des alias (ou synonymes) de type pour faciliter l'écriture.

Utilité de l'aliasing de type

- Le nom original est trop long, compliqué ou illisible pour les programmeurs.
- Une technique de programmation nécessite que différents types aient le même nom dans un contexte précis. (programmation générique)
- Un type spécifique est défini en un seul endroit pour faciliter la maintenance.

Exemples

```
template<typename T> using Vector = std::vector<T, My_allocator<T>>;
template<class T> class container {
    using value_type = T; // every container has a value_type
    // ...
};
using int32_t = int;
```

Éléments de bases du langage

Pointeurs, tableaux et références

Définition fondamentale d'un objet : zone mémoire de stockage.

en C++, chaque objet à une identité, i.e, il réside à une adresse mémoire spécifique.

Connaissant son adresse et son type, il est possible d'accéder à tout objet du système via :

- Les pointeurs
  - Peuvent désigner différents objets à différents moment.
  - Peuvent être nuls.
- Les références
  - Ne désignent qu'un seul objet.
  - Ne peuvent pas être nulles.

Définition, particularités et opérations principales

- Pour un type T, T* est le type *pointeur vers T*.
- Initialisation
  - Opérateur & (adresse de)
  - Allocation dynamique (opérateur `new`)
- Dé-référencement, indirection (opérateur préfixe *)
  - Opération fondamentale sur les pointeurs, accès au contenu de l'adresse.
- Constante `nullptr`
  - Littéral commun à tous les pointeurs, représente le pointeur nul.
  - Remplace la macro NULL ou la valeur $0L$
- Type `void *`
  - Pointeur "non typé"
  - Utilisation non sure et limitée.

Définition, particularités et opérations principales

- Pour un type T, T[size] est le type *tableau de size éléments de type T*.
- size : doit être une expression constante (évaluable à la compilation)
- Bornes d'indices
  - De 0 à size-1
  - Pas de vérification de bornes
- Création/destruction dynamique
  - Opérateurs new[] et delete []
- Tableaux et pointeurs
  - Nom = pointeur vers 1er élément
  - Passage en tant que pointeur en paramètre.
- Structure de bas niveaux, à limiter à la programmation de structures d'encapsulation de haut niveau.

Définition, particularités et opérations principales

Relation étroite entre références et pointeurs :

- Une référence est un alias pour un objet existant
  - traditionnellement, son adresse.
- Une référence permet de manipuler les objets efficacement
  - Passage de paramètre peu coûteux.

Différence fondamentale entre référence et pointeurs

- Utilisation comme un objet
  - Même syntaxe.
- Pas de référence nulle
  - Référence toujours l'objet à partir duquel elle est créée.

Utilisation des références

Principalement pour spécifier les arguments et valeurs de retours d'une fonction.
Obligatoire pour les opérateurs surchargés.

Différents types de références

- lvalue reference : pour un objet dont on veut changer la valeur.
- const reference : pour un objet dont on ne veut pas changer la valeur.
- rvalue reference : pour un objet dont la valeur n'importe plus une fois utilisée (objet temporaire)

Notation

Dans un nom de type, la notation X& signifie "référence à X".

Cette notation définit une référence à une lvalue.

```
int var {1};
int& r {var};  // r and var now refer to the same int
int& rr {1};   // ERROR : 1 is not a lvalue
int x=r;       // x becomes 1
r = 2;         // var becomes 2
```

Déclaration et initialisation

Une référence doit être initialisée à sa déclaration.

```
int var {1};
int& r1 {var};   // OK, r1 initialized
int& r2;         // ERROR : initializer missing
extern int& r3;  // OK, r3 initialized elsewhere
```

Operations

Aucune opération ne s'applique sur une référence.

```
int var {0};
int& rr {var};
++rr;              // var is incremented
int* pp {&rr};    // pp points to var, not to rr
```

Référence et définition de paramètre de fonction

Modification éventuelle du paramètre par la fonction.

```
void increment(int &x) {
++x;
}
void f() {
int x {1};
increment(x);   // x becomes 2
}
```

Références et retour de fonction

L'appel de la fonction peut apparaitre à droite et à gauche d'une affectation.

Exemple pour l'implantation d'un container associatif : Map

```
template<class K, class V>
class Map { // a simple map class
public:
  V& operator[](const K& v); // return the value corresponding to the key v
  std::pair<K,V>* begin() { return &elem[0]; }
  std::pair<K,V>* end() { return &elem[0]+elem.size(); }
private:
  std::vector<std::pair<K,V>> elem; // {key,value} pairs
};
```

Remarque le container associatif standard `std::map` est implanté par un arbre Rouge/Noir, pas par un vecteur.

Références et retour de fonction

```
template<class K, class V>
V& Map<K,V>::operator[](const K& k) {
  for (auto& x : elem)
    if (k == x.first)
      return x.second;
  elem.push_back({k,V{}});    // add pair at end
  return elem.back().second; // return the (default) value of the new element
}
```

L'opérateur [] pourra apparaitre en partie droite ou en partie gauche de l'opérateur d'affectation.

Cet opérateur pourra donc être utilisé pour ajouter une paire <clé, élément> à la collection ou pour accéder à un élément à partir de sa clé.

Références et retour de fonction

```
int main() { // count the number of occurrences of each word on input
  Map<std::string,int> buf;
  for (std::string s; std::cin>>s;)
    ++buf[s];  // same as buf[s]=buf[s]+1;
  for (const auto& x : buf)
    std::cout << x.first << ": " << x.second << "——";
}
```

Résultat d'exécution

entrée aa bb bb aa aa bb aa aa

sortie aa : 5 – bb :3

Pourquoi une autre sorte de références ?

- Une référence lvalue non-`const` désigne un objet que l'utilisateur de la référence peut modifier.
- Une référence lvalue `const` désigne un objet constant, non modifiable du point de vue de l'utilisateur de la référence.
- Une référence rvalue désigne un objet temporaire que l'utilisateur de la référence va modifier, sachant que cet objet ne sera plus utilisé ensuite.

Savoir qu'une référence désigne un objet temporaire permet de supprimer des copies/recopies coûteuses et de les remplacer des des opérations de déplacement (move).

Notation

Dans un nom de type, la notation X&& signifie "référence rvalue à X".

Déclaration et initialisation

Une référence rvalue peut être initialisée par une rvalue mais pas par une lvalue.

```
string var {"Cambridge"};
string f();

string& r1 {var};          // lvalue reference, bind r1 to var (an lvalue)
string& r2 {f()};          // lvalue reference, error: f() is an rvalue
string& r3 {"Princeton"}; // lvalue reference, error: cannot bind to temporary

string&& rr1 {f()};       // rvalue reference, fine: bind rr1 to rvalue (a temporary)
string&& rr2 {var};       // rvalue reference, error: var is an lvalue
string&& rr3 {"Oxford"}; // rr3 refers to a temporary holding "Oxford"
```

Utilité des références rvalue

Echange standard des valeurs de deux objets.

```
template<class T>
swap(T& a, T& b) { // "old-style swap"
  T tmp {a};       // now we have two copies of a
  a = b;           // now we have two copies of b
  b = tmp;         // now we have two copies of tmp (aka a)
}
```

- Les valeurs de a, b et tmp doivent être déplacées et non pas copiées.

Utilité des références rvalue

Echange standard des valeurs de deux objets.

```
template<class T>
swap(T& a, T& b) { // "old-style swap"
  T tmp {a};       // now we have two copies of a
  a = b;           // now we have two copies of b
  b = tmp;         // now we have two copies of tmp (aka a)
}
```

- Les valeurs de a, b et tmp doivent être déplacées et non pas copiées.

Echange optimisé des valeurs de deux objets.

```
template<class T>
swap(T& a, T& b) {                // "perfect swap" (almost)
  T tmp {static_cast<T&&>(a)}; // the initialization may write to a
  a = static_cast<T&&>(b);     // the assignment may write to b
  b = static_cast<T&&>(tmp);   // the assignment may write to tmp
}
```

Utilité des références rvalue

Echange optimisé des valeurs de deux objets.

```
template<class T>
swap(T& a, T& b) {                  // "perfect swap" (almost)
  T tmp {static_cast<T&&>(a)}; // the initialization may write to a
  a = static_cast<T&&>(b);      // the assignment may write to b
  b = static_cast<T&&>(tmp);    // the assignment may write to tmp
}
```

- `static_cast<T&&>(x)` transforme ici x en une rvalue.
- Un opérateur optimisé pour les rvalues peut être utilisé pour x.
- Si le type T possède un constructeur ou un opérateur d'affectation par déplacement, il sera utilisé.

Constructeur par copie ou déplacement

Choix automatique en fonction de :

- la disponibilité du constructeur,
- la nature de l'initialisation.

Exemple

```
template<class T> class vector { // ...
  vector(const vector&r); // copy constructor (copy r's representation)
  vector(vector&& r);     // move constructor ("steal" representation from r)
};

  vector<string> s;
  vector<string> s2 {s};          // s is an lvalue, so use copy constructor
  vector<string> s3 {s+"tail"};   // s+"tail" is an rvalue so pick move constructor
```

Utilité des références rvalue

Écriture simplifiée de l'opération swap : utilisation de `std::move(x)`

- `std::move(x)` est identique à `static_cast<T&&>(x)`

```
template<class T>
swap(T& a, T& b) {    // "perfect swap" (almost)
  T tmp {move(a)};    // move from a
  a = move(b);        // move from b
  b = move(tmp);      // move from tmp
}
```

Cela marche-t-il pour `swap(v,vector<int>{1,2,3})` ?

Utilité des références rvalue

Écriture simplifiée de l'opération swap : utilisation de `std::move(x)`

- `std::move(x)` est identique à `static_cast<T&&>(x)`

```
template<class T>
swap(T& a, T& b) {    // "perfect swap" (almost)
  T tmp {move(a)};    // move from a
  a = move(b);        // move from b
  b = move(tmp);      // move from tmp
}
```

Cela marche-t-il pour `swap(v,vector<int>{1,2,3})` ?

Il faut surcharger l'opérateur `swap`

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

Éléments de bases du langage

Structure, union et énumération

Spécificités étudiées par l'usage en TD et TP.

Éléments de bases du langage

Instructions et structures de contrôle

Instructions : vue d'ensemble

- Jeu d'instruction conventionnel en C++.
- Les expressions et les déclarations contiennent toute la sémantique du programme.
- Une déclaration est une instruction.
- Une expression devient une instruction si on y ajoute un ";".

Les instructions définissent l'ordre d'exécution du programme.

Exemple

```
a = b+c; // expression statement
if (a==7)// if-statement
  b = 9; // execute if and only if a==9
```

Un compilateur peut réordonner les instructions tant que le résultat reste identique.

Déclaration d'une variable ⇒ exécution de l'initialiseur lorsque le flot de contrôle passe sur la déclaration.

Moins d'erreur de programmation

- Déclarer une variable lorsque l'on en a besoin ⇒ moins de variables non initialisées.

```
void f(vector<string>& v, int i, const char* p) {
  if (p==nullptr)
    return;
  if (i<0 || v.size()<=i)
    error("bad index");
  string s = v[i];
  if (s == p) {
    // ...
  }
  // ...
}
```

Déclaration d'une variable ⇒ exécution de l'initialiseur lorsque le flot de contrôle passe sur la déclaration.

Essentiel dans de nombreux styles de programmation

- La valeur d'un objet ne change pas après sa création..

Meilleures performances

- Initialisation d'un objet à une valeur correcte et exploitable.

```
void use() {
  string s1;
  // ...
  s1 = "The best is the enemy of the good.";
  // ...
}
```

Ici, initialisation par défaut suivie d'une affectation : 2 opérations alors qu'un suffit.

Formes de la sélection

- `if ( condition ) instruction`
- `if ( condition ) instruction else instruction`
- `switch ( condition ) instruction`

Une `condition` est une `expression` ou une `déclaration`

Si `condition` est une `déclaration`, le nom défini est visible jusque à la fin de l'instruction de sélection.

Instruction `if`

- Sémantique standard.
- Une branche ne peut pas se réduire à une déclaration.

Instruction `switch`

- En général, plus efficace que le `if` (génération d'une table d'indirection)

Déclarations dans les cas

- Déclarations de variables dans le bloc du switch autorisées.
- Saut au delà d'une initialisation interdit.

```
switch (i) {
  case 0:
    int x;          // uninitialized
    int y = 3;      // ERROR : can be bypassed (explicitly initialized)
    string s;       // ERROR : can be bypassed (implicitely initialized)
  case 1:
    ++x;          // ERROR : use of uninitialized object
    ++y;
    s = "nasty!";
}
```

Formes de répétition

- `while ( condition ) instruction`
- `do instruction while ( condition )`
- `for ( for-init-statement`$_{opt}$ `; condition`$_{opt}$ `; expression`$_{opt}$ `) instruction`
- `for ( for-declaration : expression ) instruction`

Un `for-init-statement` est soit une `expression` ou une `déclaration`

Des schémas de répétition plus compliqués peuvent être développés en utilisant des `lambda` et les `lalgorithms`

Répétition range-for

- `for ( for-declaration : expression ) instruction`

Sémantique

- Donne accès à toutes les valeurs d'une séquence
  - Liste d'initialisation, Tableau, Container, ...
  - Toute collection munie des opérateur **begin** et **end**.

```
int sum(vector<int>& v) {
  int s = 0;
  for (int x : v)
    s+=x;
  return s;
}
```

- La variable de contrôle peut être de n'importe quel type et référence.

Répétition `for`

- Sémantique standard, nombreuses adaptations possibles.
- A utiliser pour les répétitions simples
  - Introduire une variable de boucle, tester, mettre à jour la variable.

Répétition `while` et `do`

- Sémantique standard.
- A utiliser pour les répétitions complexes
  - Pas de variable de boucle simple.
  - Mise à jour de la variable de boucle au milieu du corps.

Les mécanismes d’abstraction

Classes

Tour d'horizon

- Une classe définit un type utilisateur.
- Une classe est composée de membres, principalement des attributs et des méthodes.
- Des méthodes membre spécifiques définissent la sémantique de l'initialisation, copie, déplacement et destruction.
- Les membres sont accédés par le sélecteur . pour un objet ou $->$ pour un pointeur.
- Les opérateurs $(+, -, (), [] \ldots)$ peuvent être redéfinis pour une classe.
- Une classe est un espace de nom contenant ses membres.
- Les membres `public` définissent l'interface de la classe, les membres `private` son implantation.
- Une `struct` est une classe dont tous les membres sont publics.

Syntaxe et visibilité des membres

Une classe est définie de la façon suivante :

```
class X {
  // default class visibility members
  int an_attribute;
  float some_method(int p);
public:
  // public members : interface
  X(); // default constructor
  float a_public_method(int) const;
private:
  // private members : implantation
  float another_attribute;
  int one_more_method(const X& p);
};
```

Syntaxe et visibilité des membres

- Visibilité `public` (défaut pour les `struct`) :
  - Les membres `public` sont toujours accessibles par l'utilisateur.
- Visibilité `private` (défaut pour les `class`) :
  - Les membres `private` ne sont accessibles que de l'intérieur de la classe.

Méthodes membres

- Fonctions déclarées dans une `class` ou `struct`
- Appelables uniquement pour un objet donné : `objet.method(param)`
- Définition dans l'espace de nom de la classe :

```
float X::some_method(int p) {
// ...
}
```

Les mécanismes d'abstraction

Construction, destruction, copie et déplacement

Méthodes membres spécifiques

- Gestion du cycle de vie d'un objet.
- Méthodes primordialdes pour l'utilisabilité et la performance

Exemple

```
string ident(string arg) {     // string passed by value (copied into arg)
  return arg;                  // return string (move the value out to a caller)
}
int main () {
  string s1 {"Adams"};         // initialize string (construct in s1).
  s1 = ident(s1);              // copy s1 into ident()
                               // move the result of ident(s1) into s1;
                               // s1's value is "Adams".
  string s2 {"Pratchett"};     // initialize string (construct in s2)
  s1 = s2;                     // copy the value of s2 into s1
                               // both s1 and s2 have the value "Pratchett".
}
```

Fonctions utilisées

- Un constructeur de `string` pour `s1` et `s2`
- Un constructeur par copie de `string` pour le passage de paramètre à `ident`
- Un constructeur par déplacement pour placer le résultat de `ident` dans un objet temporaire
- Une affectation par déplacement pour placer le temporaire dans `s1`
- Une affectation par copie pour copier `s2` dans `s1`
- Un destructeur pour libérer les ressources utilisées par `s1`, `s2` et l'objet temporaire.

Fonctions utilisées

- Un constructeur de `string` pour `s1` et `s2`
- Un constructeur par copie de `string` pour le passage de paramètre à `ident`
- Un constructeur par déplacement pour placer le résultat de `ident` dans un objet temporaire
- Une affectation par déplacement pour placer le temporaire dans `s1`
- Une affectation par copie pour copier `s2` dans `s1`
- Un destructeur pour libérer les ressources utilisées par `s1`, `s2` et l'objet temporaire.

Attention ...

Bien qu'un compilateur-optimiseur puisse supprimer certaines de ces actions, elles peuvent être exécutée et impacter les performances !

Gestion du cycle de vie d'un objet.

Déclaration des méthodes en un bloc cohérent.

```
class X {
public :
  X(Sometype);              // "ordinary "constructor: create an object
  X();                      // default constructor
  X(const X&);              // copy constructor
  X(X&&);                   // move constructor
  X& operator=(const X&); // copy assignment: clean up target and copy
  X& operator=(X&&);        // move assignment: clean up target and move
  ~X();                     // destructor: clean up
  // ...
};
```

Possibilité de définir par défaut ces méthodes et d'en supprimer certaines.

- Mots clés `delete` et `default`

Usage des constructeurs

- Initialisation d'objets et de temporaires à la création.
- Implanter des conversion de type explicites.

Copie et déplacement d'objets

Cinq situations d'utilisation du constructeur par copie ou déplacement

- Source d'une affectation
- Initialisation d'un objet
- Argument de fonction
- Valeur de retour d'une fonction
- Exception

Usage des constructeurs

- Initialisation d'objets et de temporaires à la création.
- Implanter des conversion de type explicites.

Copie et déplacement d'objets

Cinq situations d'utilisation du constructeur par copie ou déplacement

- Source d'une affectation
- Initialisation d'un objet
- Argument de fonction
- Valeur de retour d'une fonction
- Exception

Génération par le compilateur

Génération possible de ces opérateurs par le compilateur, sauf pour le constructeur ordinaire.

Sans constructeurs

Utilisation des constructeurs par défaut pour :

- Initialiser membre à membre, par copie ou par défaut.

Exemple

```
struct Work {
  string author;
  string name;
  int year;
};

Work s9 { "Beethoven",
          "Symphony No. 9 in D minor, Op. 125; Choral",
          1824
};                                  // memberwise initialization
Work currently_playing { s9 };      // copy initialization
Work none {};                       // default initialization
```

Avec constructeurs

Attention : attribut non public, constructeur obligatoire.

Constructeurs par défaut

Constructeur ordinaire défini, constructeur par défaut non généré automatiquement.

Exemple 1

```
struct S1 {
  int a,b;     // no constructor, only public members
};

S1 x11(1,2);  // error: no constructor
S1 x12 {1,2}; // OK: memberwise initialization
S1 x13(1);    // error: no constructor
S1 x14 {1};   // OK: x14.b becomes 0
```

Avec constructeurs

Attention : attribut non public, constructeur obligatoire.

Constructeurs par défaut

Constructeur ordinaire défini, constructeur par défaut non généré automatiquement.

Exemple 2

```
struct S2 {
  int a,b;
  S2(int a = 0, int b = 0) : a{aa}, b{bb} {} // constructor
};

S2 x21(1,2);  // OK: use constructor
S2 x22 {1,2}; // OK: use constructor
S2 x23(1);    // OK: use constructor and one default argument
S2 x24 {1};   // OK: use constructor and one default argument
```

Initialisation des attributs

Un constructeur DOIT initialiser les attributs membre, éventuellement par défaut.

Liste d'initialisation

```
struct S2 {
  int a,b;
  S2(int a = 0, int b = 0) : a{aa}, b{bb} {} // constructor
};
```

Une liste d'initialisation apparait après le profil du constructeur :

- Débute par " :"
- Attributs dans l'ordre de déclaration dans la classe
- Séparés par ","

Initialisation par affectation possible mais à éviter !

Sémantique

Transferer une valeur entre deux objets a et b peut se faire de 2 façons :

- Copie : sens conventionel de x=y. x et y ont la même valeur après l'affectation que y avant.
- Déplacement : après déplacement, x a la valeur de y et y est dans un état "vide"

Les différences fondamentales sont :

- La copie doit éventuellement acquérir des ressources (allocation mémoire), pas le déplacement.
- La copie peut lever une exception, pas le déplacement
- Le déplacement est bien plus efficace que la copie

Attention : Lors de la programmation d'un opérateur de déplacement, laisser l'objet dans un état valide, i.e. pouvant être détruit.

Copie

La copie d'une classe X est définie par deux opérateurs :

- Constructeur par copie : `X(const X&)`
- Affectation par copie : `X& operator=(const X&)`

Déplacement

Le déplacement d'une classe X est définie par deux opérateurs :

- Constructeur par déplacement : `X(const X&&)`
- Affectation par déplacement : `X& operator=(const X&&)`

Attention

Ne pas oublier de copier un attribut lors de l'implantation de ces opérateurs.

Les mécanismes d’abstraction

Surcharge d’opérateurs

Notion d'opérateur

Notations conventionnelle pour rationaliser l'expression d'une idée, d'un concept, d'un traitement...

L'expression "x+y*z" est plus rapide à analyser que "multiplier y par z et ajouter le résultat à x".

Comme tout langage de programmation, le C++ offre un ensemble d'opérateurs portant sur les types de base.

Problème : Peu de programmes se contentent des types de base pour réaliser leur action.

Solution : Permettre la définition des opérateurs sur les types utilisateur.

Opérateurs conventionnels

Majoritairement des opérateurs sur des types numériques (arithmétique).

Opérateurs de manipulation d'objets et de fonction (->, [], (), ...)

Opérateurs définissables

+	–	*	/	%	^	&
|	~	!	=	<	>	+=
–=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	––	–>*	,
–>	[]	()	new	new[]	delete	delete[]

Opérateurs comme fonction

Les opérateurs s'implémentent comme des fonction dont le nom est composé de `operator` suivi du (ou des) symboles de l'opérateur.

Opérateurs binaires

Deux implantations possibles :

- Fonction membre à 1 argument
  - `X X::operator+(X x);`
- Fonction globale à 2 argument
  - `X operator+(X x, X y);`

`a + b` peut s'interpréter comme `a.operator+(b)` ou `::operator+(a, b)`

Opérateurs unaires

Qu'ils soient préfixés ou post-fixés, deux implantations possibles :

- Fonction membre à 0 argument
  - `X X::operator-();`
- Fonction globale à 1 argument
  - `X operator-(X x);`

  `-a` peut s'interpréter comme `a.operator-()` ou `::operator-(a)`
- Différentiation prefix/postfix
  - Opérateur préfixe : `X X::operator++();` ou `X operator++(X& x);`
  - Opérateur postfixe : `X X::operator++(int);` ou `X operator++(X& x, int);`

Sémantique prédéfinie

Respecter le plus possible la sémantique prédéfinie des opérateurs.

Définition du profil des opérateurs

Respecter une notation conventionnelle de l'opérateur.

- Choix pour les arguments
  - Passage par valeur
    - Limiter aux objets de petite taille.
  - Passage par référence
    - Constante ou non selon la nature de l'opérateur.
- Choix pour la valeur de retour
  - Pas de pointeurs ou de référence vers un nouvel objet !
    - Crash et fuite mémoire assurés.
  - Valeur ou référence
    - Par valeur si constructeur par déplacement.
    - Par référence si le premier paramètre de l'opérateur est modifié.

Exemple : Matrice (simple et incomplète)

```
class Matrix {
  Dimension dim;
  std::vector<double> values;
public :
  Matrix(Dimension d, std::initializer_list<double> v);
  Matrix() = default;
  // default copy and move constructors generated by the compiler
  double operator()(int r, int c) const;
  double& operator()(int r, int c);
  Matrix& operator+=(const Matrix& a);

  friend std::ostream& operator<<(std::ostream &os, const Matrix& m);
  friend std::istream& operator>>(std::istream &is, Matrix& m);
};

Matrix operator+(const Matrix &a, const Matrix&b);
```

Les mécanismes d’abstraction

Gestion dynamique des erreurs

Exceptions

Une exception permet d'obtenir une information sur une erreur survenue en un endroit et qui sera gérée ailleurs.

Une fonction ne pouvant résoudre un problème lève une exception (throw).

Approche try-catch

```
class Some_error {
};

int do_task() {.
  if (/* could perform the task */)
    return result;
  else
    throw Some_error{};
}
```

Exceptions

Une exception permet d'obtenir une information sur une erreur survenue en un endroit et qui sera gérée ailleurs.

Une fonction pouvant traiter le problème gère l'exception (try-catch).

Approche try-catch

```
void taskmaster() {
  try {
    auto result = do_task();
    // use result
  } catch (Some_error) {
    // failure to do_task: handle problem
  }
}
```

Exemple : Matrice (simple et incomplète)

```
struct Dimension{
  int r;
  int c;
};

Matrix::Matrix(Dimension d,
               std::initializer_list<double> v) : dim{d}, values{v} {
}
```

Exemple : Matrice (simple et incomplète)

```
double Matrix::operator()(int r, int c) const {
  return values[(r-1)*dim.c+(c-1)];
}

double &Matrix::operator()(int r, int c) {
  return values[(r-1)*dim.c+(c-1)];
}

Matrix& Matrix::operator+=(const Matrix& a){
  for(int i=0; i<values.size(); ++i)
    values[i]+=a.values[i];
  return *this;
}
```

Exemple : Matrice (simple et incomplète)

```
Matrix operator+(const Matrix& a, const Matrix& b){
  Matrix tmp {a};
  tmp+=b;
  return tmp;
}

std::ostream& operator<<(std::ostream& os, const Matrix& m) {
  for(int r=1; r<=m.dim.r; ++r) {
    os << "| ";
    for (int c=1; c<=m.dim.c; ++c)
      os << m(r, c) << ' ';
    os << "|\n";
  }
  return os;
}
```

Exemple : Matrice (simple et incomplète)

```
int main(){
  Matrix m1{{2, 3}, {1, 2, 3, 4, 5, 6}};
  Matrix m2{{2, 3}, {7, 8, 9, 10, 11, 12}};

  std::cout << m1  << std::endl << m2 << std::endl;

  Matrix m3 = m1 + m2;
  std::cout << m3 << std::endl;

  m1(2,1) = 9;
  std::cout << m1 << std::endl;
}
```

Exemple : Matrice (simple et incomplète)

```
$ ./a.out
| 1 2 3 |
| 4 5 6 |

| 7 8 9 |
| 10 11 12 |

| 8 10 12 |
| 14 16 18 |

| 1 2 3 |
| 9 5 6 |
```

Questions ?