Programmation orientée objet en C++
Licence 3 Informatique

David Genest

Année universitaire 2019-2020

FACULTÉ
DES SCIENCES
UNIVERSITÉ D'ANGERS

Chapitre 1

Introduction

Présentation du cours

De C à C++

Présentation du cours

Programmation orientée objet C++

Objectif du cours

- Approfondir les concepts de la programmation orientée objet. (Déjà vus en Java dans le cours de POO)
- Apprendre le langage C++ (moderne).
- Utiliser des outils d'aide au développement. Environnement de développement, débogueur...

Organisation

- 20h de cours : Concepts de la POO et C++.
- 44h de TP : Programmation C++ et outils de développement.
- + Bases de la programmation d'interfaces graphiques (Qt).

Évaluation

- $\frac{1}{3}$ CC + $\frac{2}{3}$ Examen. CC : 2 TP relevés et notés.

Introduction POO

Programmation Orientée Objet

Paradigme (style) de programmation consistant à assembler des *briques logicielles* (*objets*).
Un *objet* représente un *concept* ou une *entité* du monde.

La plupart des langages de programmation populaires aujourd'hui sont basés sur l'approche objets : Java, C#, Swift, Python, PHP, Javascript... Mais les objets ne sont pas uniquement utilisés pour la programmation...

- Bases de données objets (Oracle)
- Méthodes de conception objets (UML)
- HTML+CSS, (accompagné de Javascript, PHP...)

Objectif premier

Faciliter le développement d'une application robuste.

Introduction POO

Comparaison avec la programmation impérative (C, Pascal, ...)

- **Impératif.** Un problème est décomposé en sous-problèmes, une fonction est écrite pour résoudre chaque problème. Les fonctions s'appellent entre elles en se passant des paramètres.
- **Objets.** On identifie les concepts (ou entités) du problème, ainsi que les relations entre ces concepts. Chacun de ces concepts est ensuite représenté par les données qu'il contient et par les traitements qu'il est capable d'effectuer.

Langage à classes

La plupart des langages de programmation OO sont des langages à classes : une classe définit un « moule » pour un ensemble d'objets qui ont la même structure et fournissent les mêmes traitements.

Introduction C++

C++

- Langage normalisé par l'ISO
- Défini dans les années 1980 (mais a évolué depuis : C++98, C++03, C++11, **C++14**, C++17)
- « Amélioration » de C $\Rightarrow$ facilite l'apprentissage pour quelqu'un qui connaît déjà C. Mais ... faire du C en C++ n'est pas programmer en C++ ! Réutilisation facilitée de code/bibliothèques C
- Doté d'une bibliothèque de classes et algorithmes
- Portable

Utilisé dans Adobe Photoshop, Google Chrome, Microsoft Windows, MS Office, LibreOffice, Edge, Mozilla Firefox, Opera, KDE, Oracle, MySQL, ID Software, Blizzard, EA ...

`http://www.lextrait.com/vincent/implementations.html`

Introduction C++ - Objectifs

- Efficacité (mémoire et rapidité)
  Le langage n'ajoute aucune fonction « cachée » demandant de la mémoire ou des traitements
  → Pas de ramasse-miettes (garbage collector)
- Développement plus rapide
  Utilisation de la bibliothèque standard, Réutilisation du code
- Gestion de la mémoire plus simple
  Gérer les libérations de mémoire (`free`) plus facilement (voire automatiquement), Gestion des chaînes plus simple
- Vérification plus stricte des types
- Avantages de la POO
- Programmation générique, programmation fonctionnelle
- ...

De C à C++

De C à C++

Types et variables

Types de données

- `signed` ; `unsigned`
- `char` ; `short` (`short int`); `int` ; `long` (`long int`); `long long` (`long long int`)
  `float` ; `double` ; `long double`
- `wchar_t` (unicode)
- `bool` → `true` `false`

Types entiers de taille fixée `int8_t, int16_t, uint16_t`...
`http://en.cppreference.com/w/cpp/types/integer`

Types de données

Premier exemple (premierexemple.cc)

```
#include <iostream>

int f(int a, int b) {
  int res(1);
  for (int i=1; i<=b; ++i)
    res *= a;
  return res;
}

int main() {
  int a,b;
  std::cin >> a;
  std::cin >> b;
  std::cout << f(a,b) << std::endl;
  return 0;
}
```

Portée des variables

Une variable est accessible **à partir de sa déclaration dans le bloc** (accolades) **dans lequel elle est déclarée**.

- Une variable globale est déclarée à l'extérieur de tout bloc.
  Il est toutefois fortement conseillé de ne pas utiliser de variables globales.
- Cas particulier d'une déclaration dans un `for` :
  La variable est accessible uniquement dans les instructions répétées par le `for`.

Portée des variables

Portée des variables (porteevariables.cc)

```
#include <iostream>
signed short varglobale;
int main() {
  varglobale=5;
  unsigned int varlocale=10; // initialisation
  while (varglobale >= 0) {
    int v(6);
    varglobale += v;
    varlocale--;
  }
  std::cout << varglobale << "\n";
  return 0;
}
```

Déclaration et initialisation des variables

Multiple déclarations

```
int a, b, c; // équivalent à
int a; int b; int c;
```

Attention

```
int* a, b;
```

`a` est un pointeur sur un entier, `b` est un entier.

Initialisation C / Initialisation C++

```
int a=0; // équivalent à
int a(0);
```

Dans les deux cas, il s'agit d'une initialisation (et non une affectation).

Initialisation : donner une valeur initiale à une nouvelle variable.

Chaînes de caractères

Le type (classe) `std::string` fournit les opérations courantes sur les chaînes de façon bien plus simple que les « chaînes » C.

- Déclaration et initialisation
```
std::string s1;
std::string s2a("valinit"), s2b="valinit";
std::string s3a(s1), s3b=s1;
```
- Affectation `=`
- Comparaison `== != <=` …
- Accès à un caractère `s[i]`
- Lecture au clavier `std::cin >> s;`
- Affichage `std::cout << s;`
- Longueur d'une chaîne `s.length()`
- Concaténation `+ +=`

`http://en.cppreference.com/w/cpp/string/basic_string`

Tableaux C

Exemple

```
int t1[4];
int t2[4]={1,5,7,9};
int t3[]={12,42,23};
int t2d[4][2];
t2d[2][1] = 12;
t2d[4][2] = 23;  // Attention !
```

Attention

Un tableau est passé **par pointeur** à une fonction.

Une fonction ne peut pas connaître la taille d'un tableau.

```
int maximum(int tab[], int taille) { ... }
std::cout << maximum(t1,4);
void fnct(int tab[]) { tab[1]=5; }
fnct(t1); std::cout << t1[1];
```

Tableaux C++ `std::array`

Le (modèle de) type (classe) `std::array` fournit les opérations de base d'un tableau (d'éléments de même type) de façon plus simple et fiable qu'un tableau C.

- n'est pas converti de façon implicite en pointeur.
- peut être passé par valeur $\rightarrow$ Copie.
- peut être affecté.
- « connaît » sa taille.
- peut détecter un accès à un indice invalide.

`http://en.cppreference.com/w/cpp/container/array`

Tableaux C++ `std::array`

array (array.cc)

```
#include <iostream>
#include <array>
int max(std::array<int,5> t) {
  int m(t[0]);
  for (std::size_t i=1; i<5; ++i)
    if (t[i]>m)
      m=t[i];
  return m;
}
int main() {
  std::array<int,5> tab {3,7,1,2,6};
  std::cout << max(tab) << "\n";
  std::array<int,5> tab2(tab);
  tab2[2]=12;
  tab2.at(5)=15; // Erreur !
  tab=tab2;
  return 0;
}
```

Tableaux C++ `std::array`

- L'indice d'un tableau est de type `std::size_t` ou `std::array<...>::size_type`.
- `[]` ne fait aucune vérification, alors que `at()` lève une exception.

```
terminate called after throwing an instance of
'std::out_of_range'
what():  array::at: __n (which is 5) >= _Nm
(which is 5)
```

- `std::max_element` calcule le maximum d'un conteneur.

Boucle for d'intervalle

Syntaxe for d'intervalle

```
for(typevar nomvar : intervalle)
```

Un intervalle peut être un `std::initializer_list` (un ensemble de valeurs de même type entre accolades séparées par des virgules), un `std::array` (ou toute autre classe disposant de méthodes `begin` et `end`).

Le type de la variable peut être une valeur ou une référence.

Boucle for d'intervalle

Boucle for d'intervalle (array2.cc)

```
#include <iostream>
#include <array>

void afficher(std::array<int,5> t) {
  for (int a : t)
    std::cout << a << " ";
  std::cout << "\n";
}

int main() {
  std::array<int,5> tab {3,7,1,2,6};
  for (int a : {3,7,1,2,6})
    std::cout << a << "\n";
  for (int a : tab)
    a *= 2; // Attention, ne fait rien !
  afficher(tab);
  return 0;
}
```

`struct` et `enum`

`typedef` n'est plus utile lors de la déclaration de **`struct`** ou **`enum`**.

Une **`struct`** peut être initialisée à sa déclaration en passant entre accolades `{}` une valeur pour chacun des champs de la structure (dans l'ordre de déclaration).

Usage du typedef (typedef.cc)

```
#include <string>
typedef unsigned int Annee;
enum EtatPersonne {
  MAJEUR,
  MINEUR
};
struct Personne {
  std::string nom;
  std::string prenom;
  EtatPersonne etat;
  Annee naissance;
};
int main() {
  Personne p; // et non struct Personne p
  p.nom = "Durand"; p.etat = MINEUR;
  Personne p2 { "Dupont", "Joe", MAJEUR, 1990 };
  return 0;
}
```

using

using peut être utilisé à la place de **typedef** avec une syntaxe plus naturelle.

Exemple

```
using Annee = unsigned int;
using Tableau = std::array<std::string, 10>;
```

enum

Les types énumérés de C (**enum**) sont utilisables en C++ mais conservent leurs défauts.

enums C (enums1.cc)

```
enum Couleur { Rouge, Vert, Bleu };
enum EtatPersonne { Majeur, Mineur };
int main() {
  Couleur c(Rouge);
  if (c == Majeur) { }; //seulement un avertissement
  int d(c); // même pas un avertissement
  return 0;
}
```

On préfèrera les énumérations fortement typées **enum class** à **enum** : plus sûrs, car ne sont pas convertis silencieusement en **int**.

Énumérations fortement typées : `enum class`

enum class (enums2.cc)

```
enum class Couleur { Rouge, Vert, Bleu };
enum class EtatPersonne { Majeur, Mineur };
int main() {
  Couleur c(Couleur::Rouge);
  if (c == EtatPersonne::Majeur) { }; // Erreur
  int d(c); // Erreur
  return 0;
}
```

De plus les valeurs énumérées ne « polluent » pas l'espace des noms, et doivent être préfixées par le nom de l'énumération.

Arguments par défaut

Les *derniers* arguments d'une fonction peuvent avoir une **valeur par défaut** lors de la déclaration de la fonction.
Si les arguments correspondants ne sont pas passés lors d'un appel, la valeur par défaut est prise en compte.

Exemple

```
void f(int a=2, int b=3) {
  std::cout << a << " " << b << "\n";
}
...
f();    // 2 3
f(5);   // 5 3
f(5,7); // 5 7
```

De C à C++

Allocation dynamique

Allocation dynamique en C

Syntaxe en C

- Allocation simple

```
int * i = (int *)(malloc(sizeof(int)));
```

- Allocation d'une zone

```
int * t = (int *)(malloc(sizeof(int)*taille));
```

- Libération

```
free(t);
```

Allocation dynamique en C++

Syntaxe en C++

- Allocation simple

```
int * i = new int;
```

```
int * i = new int(3);
```
Fixer la valeur initiale

- Allocation d'une zone

```
int * t = new int[taille];
```

- Libération simple

```
delete i;
```

- Libération d'une zone

```
delete []i;
```

Allocation dynamique en C++

Attention

- Tout ce qui est alloué doit être libéré.
- Tout ce qui est alloué par `new ...[]` doit être libéré par `delete []`.
- Ne pas mélanger `new`/`delete` avec `malloc`/`free`.

Utiliser l'allocation dynamique uniquement quand elle est requise.

C++ n'est pas Java, l'allocation dynamique n'est requise que dans certains cas.

C++ n'est pas Java, si on alloue dynamiquement des objets, il faut les libérer explicitement.

De C à C++

`const`

const

Le mot-clef **const** spécifie que la valeur d'une variable est constante.

Exemple

```
const int i(4); // équivalent à
const int i=4; // équivalent à
int const i=4;

i = 5; // erreur de compilation
```

Une (variable) constante (mais pas une variable (non déclarée **const**)) peut être utilisée pour spécifier la taille d'un tableau.
Souvent on marque syntaxiquement les constantes (écriture en majuscules, première lettre en majuscule, préfixe particulier tel que c_ ...)

`const` et pointeurs

Attention à la position du `const`

- Pointeur variable sur des caractères variables `char * c;`
- Pointeur constant sur des caractères variables `char * const c;`
  `c = c1;` est interdit. `*c='a'; c[1]='b';` sont autorisés.
- Pointeur variable sur des caractères constants
  `const char * c;` (ou `char const * c;`)
  `c = c1;` est autorisé. `*c='a'; c[1]='b';` sont interdits.
- Pointeur constant sur des caractères constants
  `const char * const c;` ou (`char const * const c;`)
  `c = c1; *c='a'; c[1]='b';` sont interdits.

Moyen mnémotechnique : lire à l'envers. « Un pointeur sur un caractère constant », « Un constant pointeur sur un caractère »...

De C à C++

Références

Références

Une référence est une variable synonyme (qui porte le même nom) qu'une autre variable.

Syntaxe (Déclaration d'une référence)

```
type & nomvar(var_référencée);
```

La variable référencée doit obligatoirement être précisée au moment de la *déclaration*.

Exemple

```
int i;
int & j(i);

i=5;
std::cout << j; // 5
j=3;
std::cout << i; // 3
```

Références

Les références sont principalement utilisées pour le passage par référence d'un argument à une fonction. Dans ce cas, on ne doit pas préciser la variable référencée.

Exemple

```
void f(int & a, int & b) {
  a=2; b=3; }
...
int c,d;
f(c,d);
std::cout << c << " " << d; // 2 3
```

Simplifie dans la plupart des cas le « passage par pointeur » de C.

Passage par valeur / référence

Passer par valeur une variable provoque la copie de cette variable.

```
int maximum(std::array<int, 1000000> t);
```

Cette copie est gênante dans le cas de structures de données de grande taille : coût de la copie.

Solution. Passer le paramètre par référence.

```
int maximum(std::array<int, 1000000> & t);
```

Problème. La fonction peut modifier le paramètre.

- Bug difficile à trouver.
- La signature de la fonction doit être une documentation.

Passage par valeur / référence

Un passage par référence constante évite de faire une copie, mais empêche de modifier l'objet passé.

```
int maximum(std::array<int, 1000000> const & t);
```

Le passage par référence constante doit être préféré à un passage par valeur pour les types composés (struct, conteneurs, classes).

Passage par valeur / référence

Passage par référence (maximum.cc)

```
#include <iostream>
#include <array>
#include <algorithm>
const int Tailletab(10000);
using tableau=std::array<int, Tailletab>;
// int maximum(tableau t, std::size_t n=Tailletab-1) {
// provoque une erreur de segmentation
int maximum(tableau const & t, std::size_t n=Tailletab-1) {
  if (n == 0)
    return t[n];
  else
    return std::max(maximum(t, n-1), t[n]);
}
int main() {
  tableau t;
  for (int & init : t) init = random() % 1000;
  std::cout << maximum(t) << "\n";
  return 0;
}
```

De C à C++

Inférence de type

Inférence de type : `auto`

L'utilisation du mot-clef **`auto`** permet d'éviter de préciser le type d'une variable locale quand cette variable est initialisée à la déclaration.

Attention

`auto` n'est pas un type dont les variables peuvent contenir n'importe quelle valeur.
Le type d'une variable déclarée **`auto`** est déterminé au moment de la compilation en fonction de sa valeur initiale, et ne peut être modifié.

Exemple

```
auto f; //declaration of 'auto f' has no initializer
std::array<int,10> tab { ....};
auto premier(tab[0]);
premier = "ch"; //invalid conversion from 'const char*' to 'int'
auto t2(tab);
auto max(maximum(tab));
```

auto

`auto` est souvent utilisé dans des parcours de structures de données.

`auto` et boucle `for` d'intervalle (auto.cc)

```
using tableau=std::array<int,5>;

void afficher(tableau const & t) {
  for (auto a : t)
    std::cout << a << " ";
  std::cout << "\n";
}
```

Quel est le type inféré par le compilateur ?

`int` (et non `const int`, `int &` ou `int const &`)

Vérifions cela...

auto

auto (auto.cc)

```
void doubler(tableau const & t) {
  for (auto a : t)
    a *= 2;
}
```

Ne provoque pas d'erreur de compilation… Mais ne fait rien : `a` est une variable locale de type `int`. Le fait que `t` soit passé par référence constante n'est pas la cause :

auto (auto.cc)

```
void doubler(tableau & t) {
  for (auto a : t)
    a *= 2;
}
```

Même cause, mêmes effets : `a` est un `int`.

auto

Il est possible de combiner `auto` avec `const` et/ou `&`.

auto (auto.cc)

```
void doubler(tableau const & t) {
  for (auto & a : t)
    a *= 2;
}
```

On demande au compilateur que `a` soit une référence…

Le compilateur infère donc qu'il est de type `int const &`. Il y a donc une erreur de compilation `assignment of read-only reference 'a'`.

auto

auto (auto.cc)

```
void doubler(tableau & t) {
  for (auto & a : t)
    a *= 2;
}
```

Fait ce que l'on attend.
Utiliser `auto const &` permet de spécifier qu'une référence constante est attendue (pour éviter une copie : utile quand la copie est coûteuse).

Attention. Une initialisation à partir d'un ensemble de valeurs (`auto t{1,2,3};`) ne crée pas un `std::array` mais un `std:initializer_list`.

De C à C++

Conversion de type

Conversion de type

Syntaxe en C

```
(nouveautype)expression
```

Exemple

```
int main() {
  int a(2), b(3);
  float f(a/b);
  std::cout << f;
  return 0;
}
```

- Affiche 0
- Forcer la division sur les réels

  ```
  float f(((float)a)/b);
  ```
- Affiche `0.666667`

Conversion de type

Inconvénient de la syntaxe C : **Peu visible** (syntaxe) alors que dangereux.

Syntaxe en C++

- `static_cast<nouveautype>(expression)`
  Force la conversion de type de l'expression dans le nouveau type
  ```
  float f(static_cast<float>(a)/b);
  ```
- `reinterpret_cast<nouveautype_ptr>(expression_ptr)`
  Force la conversion de type dans le cas de pointeurs
  ```
  void * p(nullptr);
  int* r(reinterpret_cast<int *>(p));
  ```

- Les conversions sont plus visibles.
- Deux opérateurs différents pour deux opérations différentes (il y en a deux autres)

De C à C++

Compilation

Compilation

- Comme en C, le code est séparé en un fichier d'entêtes (`.hh`) et un fichier d'implantation (`.cc`).
- L'exécution commence par la fonction
  `int main(int argc, char * argv[])` ou
  `int main()`
- Compilation avec GNU C++
  - `g++ -Wall -std=c++14 fichier.cc`
    compilation et édition de liens, `fichier.cc` doit contenir un `main`
  - `g++ -Wall -std=c++14 -c fichier.cc`
    compilation seule, crée `fichier.o`
  - `g++ -Wall fichier1.o fichier2.o -o fichierexe`
    édition de liens

Compilation avec CMake

Problème. Dans un projet de grande taille, il faut compiler un grand nombre de fichiers sources (ceux qui sont nécessaires) et faire l'édition de liens à chaque test.

Solution. Écrire un `Makefile`. Fastidieux.

Solution. Utiliser un générateur de `Makefile` (ou d'autres systèmes de construction).
`http://www.cmake.org`

Configuration d'un projet : `CMakeLists.txt`

(CMakeLists.txt)

```
1  cmake_minimum_required(VERSION 3.1.0)
2  project(nomprojet CXX)
3
4  set(CMAKE_CXX_STANDARD 14)
5  set(CMAKE_CXX_STANDARD_REQUIRED on)
6  set(CMAKE_CXX_EXTENSIONS off)
7
8  if(CMAKE_CXX_COMPILER_ID MATCHES "GNU")
9    add_compile_options(-Wall -Wpedantic)
10 endif()
11
12 add_executable(nomexecutable source1.cc source2.cc)
```

Compilation avec CMake

- Lignes 1-2 : Entête commun de tous les projets C++. Penser à changer le nom du projet.
- Lignes 4-6 : Compilation en mode C++14 sans extensions du compilateur.
- Lignes 8-10 : Compilation avec les avertissements.
- Ligne 12 : Une cible exécutable : nom de l'exécutable (sans extension) suivi des noms des fichiers sources à compiler.

Compilation avec CMake

- Création d'un répertoire de construction
  `cd /tmp ; mkdir build ; cd build`
- Génération du `Makefile`
  dans le répertoire `build` :
  `cmake` *nom du répertoire contenant CMakeLists.txt*
- Construction
  `make`

CMake est intégré dans de nombreux environnements de développement (QtCreator, KDevelop, etc.) et permet de générer des projets pour Linux, Windows, MacOS.