Programmation C++ Avancée

Session 1 – Types et Fonctions

Joel Falcou Guillaume Melquiond

Laboratoire de Recherche en Informatique

Un peu d'histoire...

C++ à travers les âges

- 1978 : Bjarne Stroustrup travaille sur Simula67 qui s'avère peu efficace
- 1980 : Démarrage du projet *C with Classes*, du C orienté objet compilé via `CFront`
- 1983 : *C with Classes* devient C++
- 1985 : Parution de *The* C++ *Programming Language*
- 1998 : Première normalisation de C++ : ISO/IEC 14882:1998
- 2005 : Parution du C++ *Technical Report 1* qui deviendra C++0x
- 2011 : Ratification de C++0x sous le nom C++11 : ISO/IEC 14882:2011
- 2014 : Ratification de C++14
- 2017 : Ratification de C++17
- 2020, ... : Prochaines *milestones* du langage

C++ et C – Une histoire de famille

C++ comme héritier du C

- C++ est un sur-ensemble de C
- C++ tente de minimiser les discordances
- Possibilité de conserver une compatibilité binaire

C++, ce fils rebelle

- C++ évolue afin de délaisser les éléments fondamentaux de C
- Changement drastique du modèle de programmation
- Vers un langage quasi-fonctionnel plus qu'objet et impératif

Rappels

Types fondamentaux

Types numériques

- Entiers : `(signed/unsigned) char`, `(unsigned) short`, `(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`, `std:int16_t`, `std:uint64_t`
- Flottants IEEE-754 : `float`, `double`
- Booléens : `bool` de valeur *true* ou *false*

```
char   c = 'e';
short  s = -32000;
int    i = 154515;

unsigned char  uc = 0xFF;
unsigned short us = 65535;
unsigned int   ui = 2563489542;
```

Types fondamentaux

Types numériques

- Entiers : `(signed/unsigned) char`, `(unsigned) short`, `(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`, `std:int16_t`, `std:uint64_t`
- Flottants IEEE-754 : `float`, `double`
- Booléens : `bool` de valeur *true* ou *false*

```
std::int8_t    c = 'e';
std::uint16_t  s = -32000;
std::uint32_t  i = 0xDEADBEEF;
std::int64_t   l = 0xC01DBABEF0011337L;

std::ptrdiff_t p = 5;
std::size_t    sz = sizeof(l);
std::uintptr_t a = reinterpret_cast<std::uintptr_t>(&p);
```

Types fondamentaux

Types numériques

- Entiers : `(signed/unsigned) char`, `(unsigned) short`, `(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`, `std:int16_t`, `std:uint64_t`
- Flottants IEEE-754 : `float`, `double`
- Booléens : `bool` de valeur *true* ou *false*

```
float  f = 1.45f;
double d = 1.45;

float  sf = 3.6e-4f;
double sd = -9e-25;

float  xf = 0x12.34p10f;
double xd = -0x0.01p-30;
```

Types fondamentaux

Types numériques

- Entiers : `(signed/unsigned) char`, `(unsigned) short`, `(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`, `std:int16_t`, `std:uint64_t`
- Flottants IEEE-754 : `float`, `double`
- Booléens : `bool` de valeur *true* ou *false*

```
bool t = true;
bool f = false;

bool x = t || (!f && t);

if (x) std::cout << "x est vrai" << std::endl;
```

Types fondamentaux

Qualificateurs de types

- **const** exprime la constance d'une valeur
- * indique un type **pointeur**
- & indique un type **référence**

```
float f = 8;
float const cf = 7;

f = 9;
cf = 10; // ERROR: assignment of read-only variable 'cf'
```

Types fondamentaux

Qualificateurs de types

- **const** exprime la constance d'une valeur
- ***** indique un type **pointeur**
- **&** indique un type **référence**

```
int i = 1337, j = 42;
int *pn = nullptr, *pi = &i;

*pi = j;
pn  = pi;
```

Types fondamentaux

Qualificateurs de types

- **const** exprime la constance d'une valeur
- * indique un type **pointeur**
- & indique un type **référence**

```
// Exercice : Donnez le type des variables ci-dessous

int                 v1 = 1;
int*                v2 = nullptr;
int const           v3 = 1;
int const*          v4 = nullptr;
int* const          v5 = nullptr;
int const * const   v6 = nullptr;
```

Types fondamentaux

Qualificateurs de types

- **const** exprime la constance d'une valeur
- * indique un type **pointeur**
- & indique un type **référence**

```
int &rn; // ERROR: reference not initialized
int i = 1337, j = 42;
int &pi = i, &pj = j;

pi = -42;
pj = pi;
```

Types et variables

Chaîne de caractères

- Représentée par le type standard `std::string`
- Accessible via `#include <string>`
- Comparable, copiable, concaténable, redimensionnable
- Convertible depuis/vers les chaînes C

```
std::string empty;
std::string cstyle = "some text";
std::string repeat(9,'*');
std::string copy = cstyle;

repeat[5] = 'X';
copy[copy.size()-1] = '!';
```

Types et variables

Chaîne de caractères

- Représentée par le type standard `std::string`
- Accessible via `#include <string>`
- Comparable, copiable, concaténable, redimensionnable
- Convertible depuis/vers les chaînes C

```
if (copy == cstyle)
  std::cout << "s1 et s2 sont identiques\n";
else
  std::cout << "s1 et s2 sont differentes\n";
```

Types et variables

Chaîne de caractères

- Représentée par le type standard `std::string`
- Accessible via `#include <string>`
- Comparable, copiable, concaténable, redimensionnable
- Convertible depuis/vers les chaînes C

```
x = repeat + repeat;
std::cout << x << "\n";
```

Types et variables

Tableau dynamique

- Représenté par le type standard `std::vector<T>`
- Accessible via `#include <vector>`
- Comparable, copiable, redimensionnable
- Optimisé pour gérer tout type de contenu

```
std::vector<int> empty;
std::vector<int> data(3);
std::vector<int> values = {1,2,3,4,5,6,7};

for (std::size_t i = 0; i != values.size(); ++i)
  data.push_back(values[i]);

data.resize(15);
```

Types et variables

Tableau statique

- Représenté par le type standard `std::array<T,N>`
- Accessible via `#include <array>`
- Propose une sémantique de valeur
- Binairement équivalent à `T[N]`

```
std::array<float,7> data;
std::array<float,7> values = {1,2,3,4,5,6,7};

for (std::size_t i = 0; i != values.size(); ++i)
  data[i] = 3.f * values[i];

values = data;
```

Types et variables

Tableau statique

- Représenté par le type standard `std::array<T,N>`
- Accessible via `#include <array>`
- Propose une sémantique de valeur
- Binairement équivalent à `T[N]`

```
typedef std::array<float,3> vec3;

vec3 translate(vec3 const &p, vec3 const &v) {
  vec3 that = p;
  for (std::size_t i = 0; i != that.size(); ++i)
    that[i] += v[i];
  return that;
}
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des **enum**
- C++ permet la spécification du support de type

```
#define NORTH         0
#define SOUTH         1
#define EAST          2
#define WEST          3
#define NONE          4

int wind_direction = NONE;
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des **enum**
- C++ permet la spécification du support de type

```
enum wind_directions { NORTH, SOUTH, EAST, WEST, NONE };

wind_directions w = NONE;
wind_directions e = 453; // ERREUR
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des **enum**
- C++ permet la spécification du support de type

```
enum Color    { RED, GREEN, BLUE };
enum Feelings { EXCITED, MOODY, BLUE }; // ERREUR
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des **enum**
- C++ permet la spécification du support de type

```
enum class Color    { RED, GREEN, BLUE };
enum class Feelings { EXCITED, MOODY, BLUE };

Color color = Color::GREEN;
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des **enum**
- C++ permet la spécification du support de type

```
// un entier 8 bits est suffisant ici
enum class Colors : unsigned char
  { RED = 1, GREEN = 2, BLUE = 3 };
```

Types de données abstraits

Paire et tuple

- `std::pair` : deux membres de types quelconques
- `std::tuple` : généralisation de `pair`
- Copiable, assignable, introspectable

```
std::pair<float,int> chu(3.f,5);
float f = chu.first;
int   i = chu.second;
```

Types de données abstraits

Paire et tuple

- `std::pair` : deux membres de types quelconques
- `std::tuple` : généralisation de `pair`
- Copiable, assignable, introspectable

```
std::tuple<double,int,char> bar =
    std::make_tuple(3.1, 14, 'y');
std::get<2>(bar) = 100;

int myint; char mychar;
std::tie(std::ignore, myint, mychar) = bar;

std::cout << "myint : " << myint << "\n";
std::cout << "mychar: " << mychar << "\n";
```

Types de données abstraits

Paire et tuple

- `std::pair` : deux membres de types quelconques
- `std::tuple` : généralisation de `pair`
- Copiable, assignable, introspectable

```
std::tuple<double,int,char> bar =
    std::make_tuple(3.1, 14, 'y');
auto &[mydouble, myint, mychar] = bar; // C++17
std::cout << "myint : " << myint << "\n";
```

Types de données abstraits

Paire et tuple

- `std::pair` : deux membres de types quelconques
- `std::tuple` : généralisation de `pair`
- Copiable, assignable, introspectable

```
std::tuple<int,char> foo(10,'z');
std::get<0>(foo) = std::get<2>(bar);
mychar = std::get<1>(foo);

std::cout << "foo: ";
std::cout << std::get<0>(foo) << ' ';
std::cout << std::get<1>(foo) << '\n';
```

Aspect Impératif

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

Déclaration d'une fonction

```
type name( parameter1, parameter2, ...) { statements }
```

- `type` : Type de la valeur retournée par la fonction
- `name` : Identifiant de la fonction
- `parameter*` : Transfert d'information du point d'appel à la fonction
- `statements` : Corps de la fonction, *i.e.* le code effectif de la fonction

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
double addition(double a, double b)
{
  return a + b;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
void decimate(double *a)
{
  *a *= 0.9;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
void decimate(double &a)
{
  a *= 0.9;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
// C++11
auto confuzzle(double a, int &b, float c)
      -> decltype(c/b - b/a)
{
  b = static_cast<int>(c/a);
  return c/b - b/a;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
// C++14
auto confuzzle(double a, int &b, float c)
{
  b = static_cast<int>(c/a);
  return c/b - b/a;
}
```

Fonctions et surcharge

Définition

- Forme de polymorphisme ad-hoc
- En C : une fonction = un symbole
- En C++ : une fonction = une signature
- Une signature = symbole + type des paramètres + qualificateur

Exemples :

- `double f()`
- `double f(int)` – OK
- `double f(double,int)` – OK
- `double f(...)` – OK
- `int f()` – KO

Fonctions génériques

- Généralisation de la surcharge de fonction
- Généralisation/abstraction des types des paramètres
- Déduction automatique des types

```
template<typename T> T min(T const &a, T const &b)
{
  return a<b ? a : b;
}
```

Fonctions génériques

- Généralisation de la surcharge de fonction
- Généralisation/abstraction des types des paramètres
- Déduction automatique des types

```
double a = min(13., 37.);
float b = min(4.f, 3.f);
int c = min(4, 3);
char d = min('e', 'z');
```

Surcharge des opérateurs

Objectifs

- Rendre une classe similaire à un type de base
- Renforcer la sémantique et simplifier la syntaxe
- Attention aux abus !

Syntaxe

- Opérateur unaire membre : `type type::operator!()`
- Opérateur binaire membre : `type type::operator+(type)`
- Opérateur unaire : `type operator-(type)`
- Opérateur binaire : `type operator+(type,type)`

Surcharge des opérateurs

```
struct rational
{
  int  numerator()   const { return num; }
  int  denumerator() const { return denum; }

  rational operator-() const { return {-num, denum}; }

  rational& operator*=(rational const &rhs)
  {
    num   *= rhs.num;
    denum *= rhs.denum;
    return *this;
  }

  int num, denum;
};
```

Surcharge des opérateurs

```
rational operator*(rational const &lhs, rational const &rhs)
{
  rational that = lhs;
  return that *= rhs;
}
```

Règles de résolution

Processus général [1]

- Les variantes du symbole sont recherchées pour créer l'*Overload Set* (Ω).
- Toute variante n'ayant pas le nombre de paramètres adéquat est éliminée pour obtenir le *Viable Set*.
- On recherche dans cet ensemble la *Best Viable Function*.
- On vérifie l'accessibilité et l'unicité de la sélection.

Que faire de tout ça ?

- Comment définir (Ω) ?
- Quels critères pour la *Best Viable Function* ?

[1] C++ *Templates: The Complete Guide* – David Vandevoorde, Nicolai M. Josuttis

Règles de résolution

Construction de Ω

- Ajouter toutes les fonctions non-templates avec le bon nom
- Ajouter les variantes templates une fois la substitution des paramètres templates effectuée avec succès
- Ω est un treillis : les fonctions non-templates dominent les fonctions templates

Sélection de la *Best Viable Function*

- Chaque argument est associé à un Séquence de Conversion Implicite (ICS)
- Chaque argument est trié par rapport à son ICS
- Si un argument n'est pas triable, le compilateur boude

Règles de résolution

Les séquences de conversion implicite

- Séquence standard (SCS)
  - correspondance exacte
  - promotion
  - conversion (numérique ou sous-typage)
- Séquence utilisateur (UDCS), composée de
  - une première séquence standard
  - une conversion définie par l'utilisateur (opérateur ou constructeur)
  - une deuxième séquence standard

  Une UDCS est meilleure qu'une autre si sa seconde SCS est meilleure
- Séquence de conversion par ellipse C

Règles de résolution

```
void f(int)          { cout << "void f(int)\n"; }
void f(char const*) { cout << "void f(char const*)\n"; }
void f(double)       { cout << "void f(double)\n"; }


f(1); f(1.); f("1");  f(1.f);  f('1');
```

Résultats

- f(1) → void f(int)
- f(1.) → void f(double)
- f("1") → void f(char const*)
- f(1.f) → void f(double)
- f('1') → void f(int)

Règles de résolution

```
void f(int)          { cout << "void f(int)\n"; }
void f(char const*) { cout << "void f(char const*)\n"; }
void f(double)       { cout << "void f(double)\n"; }
template<class T> void f(T) { cout << "void f(T)\n"; }

f(1); f(1.); f("1");  f(1.f);  f('1');
```

Résultats

- f(1) → void f(int)
- f(1.) → void f(double)
- f("1") → void f(char const*)
- f(1.f) → void f(T)
- f('1') → void f(T)