La langage C++

M4105C

Sylvain Jubertie
sylvain.jubertie@univ-orleans.fr

Organisation du module

8 semaines

- Semaines 1-3 : Cours/TD/TP
- Semaine 4 : Cours/TD/Examen en séance de TP
- Semaines 5-7 : Cours/TD/TP
- Semaine 8 : Cours/TD/Examen en séance de TP

Évaluation surprise

Suivant attitude des étudiants. . .

Historique

Langage développé par Bjarne Stroustrup au début des années 1980 comme extension du C : *C with classes*, langage standardisé

Caractéristiques

Langage multi-paradigmes

- programmation impérative
- programmation objet
- programmation générique

Domaines

- embarqué : Arduino, micro-controleurs
- systèmes d'exploitation : Windows ! (noyau en C)
- jeux vidéo : OpenGL, Vulkan, moteurs physiques
- calcul : banque, finance, simulations scientifiques, ...
- ...

Ressources en ligne

- isocpp.org : standard C++
- cppreference.com : API
- Coliru : compilateur C++ en ligne
- Godbolt : compilateur C++ en ligne + asm

hello.cpp

```
#include <iostream>

int main()
{
  std::cout << "Hello C++" << std::endl;
  return 0;
}
```

Compilation avec GCC

```
g++ -o hello hello.cpp
```

Exécution

```
./hello
```

Types primitifs

- types entiers signés : `char`, `short`, `int`, `long`, ...
- types entiers non signés : préfixe `unsigned`
- booléen : `bool`
- types flottants : `float`, `double`
- pointeurs
- constantes : modificateur `const`

Initialisation des variables de types primitifs

1. Initialisation par affectation :

```
int a = 5;
float f = 1.5f;
```

2. Initialisation par constructeur :

```
int a( 5 );
float f( 1.5f );
```

3. Initialisation uniforme (C++11) :

```
int a{ 5 };
float f{ 1.5f };
```

Pointeurs

Comme en C, mais C++11 a introduit un pointeur nul typé pour éviter certains écueils du C :

```
int a = 5;
int * p_i = &a;
int * p_j = nullptr;
```

Références

Les références permettent de contenir les adresses de variables, comme les pointeurs, mais obligatoirement initialisées à partir d'une instance ou d'une fonction.
Une fois initialisée, une référence ne peut être changée pour référencer un autre objet donc elle est toujours constante et le modificateur **const** n'est donc pas utilisé.
La syntaxe pour accéder à l'instance référée est la même que pour accéder à l'instance.

```
int a = 5;
int & r_a = &a;
r_a += 3;
std::cout << r_a << std::endl;
```

Modificateur **static**

Le mot clé **static** devant une variable permet de modifier :

- la portée d'une variable globale ou d'une fonction à l'unité de traduction (translation unit) i.e. au fichier `.cpp` : le symbole n'est pas exporté dans la table des symboles.
- le stockage d'une variable locale d'une fonction : la variable locale est stockée globalement mais sa portée est toujours locale.

Exemple

```
static int a = 123; // Stockage global mais portée limitée au fichier.
static void fct0() { ... } // Portée limitée au fichier.
void fct1()
{
  static int i = 0; // Stockage global, initialisée à 0, portée locale.
}
```

Tableaux statiques (pile)

Déclaration :

```
elementtype tabname[ tabsize ];
```

Initialisation des tableaux

```
int t0[ 10 ]; // 10 entiers, valeurs non définies
int t1[ 4 ] = {}; // 0, 0, 0, 0
float t2[] = { 1.0f, 2.0f, 3.0f, 4.0f }; // float[4]
float t3[ 4 ] = { 1.0f }; // 1.0f, 0.0f, ...
```

Tableaux et bibliothèque standard

- Privilégier l'utilisation de `std::array` (cf. section Bibliothèque standard) à la place des tableaux statiques C.
- Les tableaux dynamiques sont présentés dans la sous-section *Gestion mémoire dynamique*.

Chaînes de caractères

Type complexe `std::string` de la bibliothèque standard :

Initialisation

```
std::string s0 = "Hello C++";
std::string s1( "Hello" );
std::string s2{ "Hello" };
```

Fonctions

- comme en C.
- passage par référence en plus : la variable passée par référence est modifiable dans la fonction. Equivalent à un passage par pointeur mais avec une syntaxe plus simple. Une référence ne peut être nulle.

Exemple

```
void fct( int a ) { ... } // Passage par copie.
void fct( int * p_a ) { ... } // Passage par pointeur.
void fct( int & a ) { a = 3; ... } // Passage par référence.
```

const

Le mot clé const peut être appliqué aux arguments d'une fonction.

Exemple

```
void fct( int const a ); // a: copie non modifiable.
void fct( int const * p_a ); // copie du pointeur modifiable
                             // mais valeur pointée non modifiable.
void fct( int * const p_a );
// copie du pointeur non modifiable mais valeur pointée modifiable.
void fct( int const * const p_a ); // rien n'est modifiable.
void fct( int & a ) // Passage par référence, a modifiable.
void fct( int const & a )
  // Passage par référence, a non modifiable.
```

Types complexes

2 méthodes de définitions possibles :

1. `struct`
2. `class`

Structures

Déclaration similaire au C.

Par défaut attributs accessibles publiquement.

Définition

```
struct Student
{
  unsigned int id;
  std::string firstname;
  std::string lastname;
  // ...
}; // Attention au ; à la fin de la définition !
```

Instantiation/Accès

```
Student s0;
s0.id = 3;

Student s1{ 42, "Arthur", "Accroc"};
std::cout << s1.firstname << std::endl;
```

Classes

- Définition similaire à une structure.
- Par défaut attributs privés donc non accessibles de l'extérieur de la classe.
- Possibilité de déclarer les attributs publiques : `public`.
- Sinon nécessite des constructeurs/accesseurs pour manipuler les attributs.
- Accès aux attributs publiques comme pour une structure.

Exemple : attributs privés (par défaut)

```
class Student
{
private: // attributs privés
  unsigned int id;
  std::string firstname;
  std::string lastname;
  // ...
};
```

Exemple : attributs publiques

```
class Student
{
public: // attributs accessibles de l'extérieur de la classe.
  unsigned int id;
  std::string firstname;
  std::string lastname;
  // ...
};
```

Exemple : attributs publiques et privés

```
class Student
{
private:
  unsigned int id;
public:
  std::string firstname;
  std::string lastname;
  // ...
};
```

Encapsulation

Encapsulation = Attributs + Méthodes associés.

Vie d'une instance

1. Instanciation : constructeurs, initialisation des attributs
2. Utilisation : méthodes
3. Destruction : destructeur, préparation avant libération de la mémoire

Constructeurs

- Par défaut : sans arguments.
- Avec arguments.
- Par copie : copie des attributs d'une autre instance.
- Par déplacement (move constructor) : appropriation d'une autre instance.

Exemple

```
class Student
{
public:
  // constructeur par défaut.
  Student() { ... }

  // constructeur avec arguments.
  Student( ... ) { ... }

  // constr. de recopie, on ne modifie pas l'original donc const.
  Student( Student const & s ) { ... }

  // move constr., rvalue reference.
  Student( Student && s ) { ... }
```

Constructeur de recopie

- Si le constructeur de recopie n'est pas définie par défaut, alors un constructeur de recopie par défaut est utilisé : tous les attributs sont copiés (comme pour les structures en C).
- L'instance à copier est passée par référence : `&`.
- Si l'instance à copier n'est pas modifiée (attributs non modifiés) par le constructeur de recopie, alors la déclarer `const`

Appels entre constructeurs

Un constructeur peut en appeler un autre (C++11).

Exemple

```
Student( std::string const & firstname,
         std::string const & lastname ) ...

Student( std::string const & firstname,
         std::string const & lastname,
         unsigned short firstinscr )
  : Student( firstname, lastname ) ...
```

Destructeur

- Appelé automatiquement lors de la sortie de la zone de portée de la variable.
- Pas d'argument.
- Utilisé pour modifier des attributs statiques (compteur), libérer des ressources (mémoire, fichier : RAII), ...

Exemple

```
...
  // Destructeur
  ~Student() { ... }
...
```

Exemple

```
...
{  // Début de la zone de portée.
   Student s( ... ); // Instanciation
   std::cout << s.firstname << std::endl;
   ...
}  // Fin de la zone de portée, appel au destructeur  Student
```

Problème

Lorsque la classe contient des pointeurs, descripteurs de fichiers, ...de manière générale des données dynamiques, la copie par défaut effectue uniquement la copie du pointeur, du descripteur de fichier, ...

Pour effectuer une copie complète (*deep copy*), il est obligatoire de surcharger le constructeur de recopie pour effectuer la copie des données pointées.

Exemple

```
class A {
   ...
   int * p_i;
   A() {
      p_i = new int;
   }
};
...
A a;
A b( a ); // b.p_i pointe vers la même donnée que a.p_i
*(a.p_i) = 42;
std::cout << *(b.p_i) << std::endl; // affiche 42...
```

Mais `double free corruption` à la sortie du programme...

Forme de Coplien

Pour effectuer la *deep copy* d'une instance il convient de mettre la classe dans la forme canonique de Coplien qui impose que la classe possède :

- un constructeur par défaut
- un constructeur de recopie
- un destructeur
- un opérateur d'affectation

Méthodes

Définition comme les fonctions.

Attention, l'instance est accessible par le pointeur **this**.

Exemple

```
class Student {
  ...
public:
  void setFirstname( std::string const & firstname ) {
    this->firstname = firstname;
  }
};
```

Méthodes constantes

Les méthodes ne modifiant pas l'état de l'instance doivent être déclarées constantes (pas obligatoire mais...).

Exemple

```
class Student {
  ...
public:
  unsigned int getId() const {
    return id;
  }
};
```

Modificateur **static**

En plus de pouvoir se placer devant des variables et des fonctions (C), le modificateur **static** peut être placé devant :

- les attributs d'une classe,
- les méthodes d'une classe.

Attributs **static**

- Attributs de classe et non d'instance
- Déclaration dans la classe
- Attributs stockés dans segment `data`
- Initialisation obligatoire en dehors de la classe

Exemple

```
class A {
...
  static int n;
};
int A::n = 0;
```

Méthodes static

- Accès à des attributs statiques
- Appel par nom complet

Exemple

```
class A {
public:
  static void fct() { ... }
};
// appel
A::fct();
```

Surchage d'opérateurs

Presque tous les opérateurs peuvent être surchargés : $+$, $-$, $*$, $=$, $[]$, $()$, ...
Certains doivent être définis dans la classe, d'autres à l'extérieur, d'autres indifféremment.

- cppreference.com: operator overloading
- wikipedia.org: operators in C and C++

Cela permet de simplifier l'écriture mais doit être utilisé à bon escient et de manière pertinente.
Dans l'exemple suivant, une classe Matrix permet de définir des matrices.
Pour 2 instances a et b de cette classe, cela a un sens d'écrire a + b pour effectuer la somme des 2 matrices.

operator+ dans la classe

Dans ce cas l'opérateur s'applique entre l'instance courante et celle passée en argument.

```
class Matrix {
   ...
   Matrix operator+( Matrix const & m ) { ... }
   ...
};
```

operator+ externe

```
Matrix operator+( Matrix const & m0,
                  Matrix const & m1 )
{ ... }
```

operator= : opérateur d'affectation

Définition obligatoire dans la classe :

```
class Matrix {
  ...
  Matrix & operator=( Matrix const & m ) {...}
};
```

L'instance courante prend le contenu de son argument et est retournée.
Cela permet d'écrire : a = b = c = ...;

operator<<

Dans le cas de la surchage pour l'affichage, l'opérateur modifie et retourne le flux (std::ostream) passé en paramètre :

```
std::ostream & operator<<( std::ostream & out, Matrix const &
{
   ...
   return out;
}
```

L'opération n'est pas censée modifier l'instance passée en argument donc **const**.
La matrice ne doit pas être passée par copie donc &.

auto : inférence de type (C++11)

Le spécificateur **auto** permet d'attribuer un type en le déduisant automatiquement. Il permet d'obtenir des codes plus génériques.

Exemple

```
int a = 5;
int b = a;
float x = fct();
```

Le compilateur peut déterminer le type de `b` en fonction de celui de `a`, ou le type de `x` en fonction du type de retour de la fonction `fct`.

```
auto b = a;
auto x = fct();
```

Par contre on ne peut pas écrire : **auto** `x`;

Remarques

auto déduit uniquement le type, pas les modificateurs :

```
auto i = 5; // i de type int.
auto const j = 3; // j de type const int
auto & k = &i; // k de type int &
```

Autres utilisations

auto peut aussi déduire le type de retour d'une fonction :

```
auto fct() { return 5; }
```

decltype : récupération de types

Le spécificateur decltype(expr) permet de récupérer le type d'une expression donnée en argument.

```
int fct() { ... }

decltype( fct() ) x; // int x;

int tab[ 10 ];
decltype( tab[ 0 ] ) e; // int e;
```

Structures conditionnelles : if

Comme en C :

```
if( cond )
{
   ...
}
else
{
   ...
}
```

if constexpr : C++17

Lorsque l'expression peut être évaluée à la compilation, seul le code du bloc **if** ou **else** est généré par le compilateur :

```
if constexpr( const_expr ) {...}
```

Exemple

```
if constexpr( sizeof(int) == 4 )
```

Structures conditionnelles : **switch**

Comme en C :

```
switch( value )
{
   case 0: ...; break;
   case 1: ...; break;
   ...
   default: ...
}
```

Boucles : for

Comme en C :

```
for( init ; stop_cond ; op )
{
   ...
}
```

Exemple

```
for( std::size_t i = 0 ; i < 10 ; ++i )
{
   ...
}
```

Range-based loops depuis C++11

```
int tab[ 5 ] = { ... };

// par réf : on modifie le contenu.
for( int & i: tab ) { // ou : auto & i: tab
   i = std::rand() % 10;
}

for( int i: tab ) { // ou auto i: tab
   std::cout << i << std::endl;
}
```

Rappel de C

Allocation/désallocation dans le tas :

- Allocation : malloc
- Désallocation : free

Inconvénient : taille donnée en octets pour l'allocation.

C++

- Allocation : **new**
- Désallocation : **delete**

Exemple

```
int * p_i = new int; // allocation d'un int dans le tas.
*p_i = 4;
...
delete p_i;

int * p_tab = new int[100]; // allocation de 100 int.
p_tab[ 10 ] = 123;
...
delete [] p_tab;
```

Problèmes

- fuites mémoire : mémoire jamais désallouée.
- erreurs de segmentation : accès ou suppression de pointeurs invalides.

Erreurs communes

```
// Fuite mémoire.
int * t0 = new int[ 100 ];
t0 = new int[ 100 ];

// Erreur segmentation.
int * t1; // Valeur indéfinie !
t1[ 10 ] = 5;
```

Solutions

- *smart pointers* pour remplacer les pointeurs.
- conteneurs standards pour masquer l'allocation dynamique.

smart pointers en C++11

Les *smart pointers* (pointeurs intelligents) ajoutent un aspect sémantique aux pointeurs et s'occupent de la gestion des pointeurs et de la libération de la mémoire en gardant la syntaxe des pointeurs.

3 types sont disponibles :

- unique_ptr : pour un pointeur qui ne doit pas être partagé,
- shared_ptr : pour un pointeur partagé,
- weak_ptr : pour un pointeur temporaire sur un shared_pointer.

unique_ptr

Le pointeur ne peut être partagé ou accessible à partir de plusieurs liens.
La propriété est transmise par la fonction std::move qui "déplace" le pointeur.

```
unique_ptr< Student > p_s0( new Student( ... ) );
// ou : auto p_s0 = std::make_unique¡ Student ¿( ... );
std::cout << *p_s0 << std::endl;
//auto p_s1 = p_s0; // Pas de constructeur de recopie.

// p_s1 prend la propriété, p_s0 devient invalide.
unique_ptr< Student > p_s1( std::move( p_s0 ) );
// std::cout ¡¡ *p_s0 ¡¡ std::endl; // Erreur segmentation.

p_s1->setFirstName( "Hector" );
```

shared_ptr

Le pointeur peut être partagé mais l'instance pointée n'est pas copié.
Un compteur contient le nombre de pointeurs sur l'instance qui est détruite lorsque le compteur est égal à 0.

```
shared_ptr< Student > p_s0( new Student( ... ) );
// ou : auto p_s0 = std::make_shared¡ Student ¿( ... );
std::cout << *p_s0 << std::endl;
auto p_s1 = p_s0;
std::cout << p_s1.use_count() << std::endl;
p_s1->setFirstName( "Hector" );
std::cout << *p_s0 << std::endl;
std::cout << *p_s1 << std::endl;
```

Templates

La généricité consiste à développer un code ou une structure de données avec des types non explicitement définis i.e. génériques.
La généricité peut s'appliquer aux :

- fonctions, par exemple std :: max,
- classes, par exemple std :: vector.

On peut parler alors de méta-fonctions ou de méta-classes qui vont être instanciées et produire respectivement des fonctions ou des classes.
Ces méta-fonctions et méta-classes peuvent être paramètrées par des types primitifs ou complexes et par des valeurs entières.

Fonctions génériques

```
template < typename T >
T fct( T & v )
{
   return v + v; // suppose que l'op. + est défini sur le type.
}
```

Instantiation

```
int a = 3;
auto b = fct( a ); // ou fct<int >( a );
```

Spécialisation

Si un type particulier nécessite un traitement spécifique, il est possible de spécialiser un template.

Exemple

```
template <>
float fct<float>( float & v )
{
  return v * 3.0f;
}
```

Classes génériques

```
template < typename T >
class Matrix
{
private:
  std::vector< T > m;
  std::size_t dimx, dimy;
public:
  ...
  T operator()( std::size_t i, std::size_t j ) const
  {
    return m[ j * dimx + i ];
  }
}
```

Standard Template Library

Concepts :

- Conteneurs
- Itérateurs
- Algorithmes

Conteneurs

- std :: array
- std :: vector
- std :: deque
- std :: forward_list
- std :: list
- std :: set
- std :: map
- ...

Itérateurs

- Généralisation des pointeurs pour le parcours/manipulation des conteneurs
- Récupération à partir des conteneurs
- Interface pointeur

Algorithmes

- Manipulation des conteneurs
- Traitement, recherche, dénombrement, tri, ...

Exemple

```
std::vector< float > v( 100 );
std::generate_n( v.begin(), v.end(), std::rand );
std::sort( v.begin(), v.end() );
```

Foncteurs

- Type complexe + surcharge de l'opérateur ()
- Ressemble à un appel de fonction
- Possibilité de stocker un état interne

Exemple simple

```
class A {
  void operator() const {
    std::cout << "Hello functor\n";
  }
};
// Utilisation
A a;
a();
```

Exemple : Compteur

```
class counter {
  std::size_t cpt;
  counter(): cpt( 0 ) {}
  auto operator() { return cpt++; }
};
```

Exemple : Générateur

```
struct generator {
  generator() {
    std::srand(...);
  }
  auto operator() const { return std::rand(); }
};
```

Lambdas

- Fonction anonyme
- Utile pour passer en paramètre d'un algorithme