Programmation C++ moderne

généricité, nouveautés

Gérard Rozsavolgyi

roza@univ-orleans.fr

Jan 2022

Langage C++

généricité, puissance

2022

Introduction

Slides en pdf

- Slides
- Slides 16/9
- Slides 16/10

Le langage C++

M4105C Gérard Rozsavolgyi, basé sur l'ancien cours de sylvain.jubertie. Plan :

1. Bases
2. Généricité
3. Bibliothèque standard
4. Héritage/Polymorphisme
5. Métaprogrammation
6. Outils complémentaires : Boost, CMake
7. Futurs standards

Organisation du module

8 semaines :

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

Historique

- Langage C, Dennis Ritchie dans les années 1970 - ATT
- Langage développé par Bjarne Stroustrup (ATT) au début des années 1980 comme extension du C : `C with classes`, langage standardisé
- Beaucoup d'évolutions récentes : C++11, C++14, C++17, C++2x

Domaines d'utilisation

- embarqué : Arduino, micro-controleurs
- systèmes d'exploitation : Windows ! (noyau en C mais bcp de 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.org : compilateur C++ en ligne + asm
- Boost librairies complémentaires Boost.

Les bases

hello.cpp

```cpp
#include <iostream>
int main()
{
  std :: cout << "Hello C++" << std :: endl ;
  return 0;
}
```

Compilation avec GCC

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

Exécution :

```
./hello
```

Compilation avec C++17 ou C++20

```
g++ -std=c++20 hello.cpp -o hello
```

Compilations avec options

```
g++ -std=c++20 -O2 -Wall -pedantic -pthread -o main main.cpp
```

Avec affichage des warnings et violations des règles du standard utilisé, et usage de la librairie pthreads.

Compilation avec make

`make hello`

(fichier unique)

Dans un Makefile

Ajouter :

```
CXXFLAGS += -std=c++20
```

Avec CMake

Dans CMakeList.txt, mettre en premier :

```
set(CMAKE_CXX_STANDARD 20)
```

hello un peu plus sophistiqué

```
#include <iostream>
#include <string>
#include <vector>
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& vect)
{
    for (auto& elt : vect){ os << elt << ' '; }
    return os;
}
int main()
{
    std::vector<std::string> vect = {"Hello", "from", "GCC", __VERSION__, "!"
```

Initialisation des variables de types primitifs

1. Initialisation par affectation
2. Initialisation par constructeur
3. Initialisation uniforme (C++11)

Différentes initialisations

```
// par affectation
int a = 5;
float f = 1.5f;

// par constructeur :
int a( 5 );
float f(1.5f);

// initialisation uniforme (à partir de 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   pi = &a;
int   pj = 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 toujours constante et le modificateur `const` n'a pas à être utilisé.
- La syntaxe pour accéder à l'instance référée est la même que pour accéder à l'instance.

exemples de références

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

Modifieur static

Le mot clé static devant une variable permet de modifier :

1. 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.
2. le stockage d'une variable locale d'une fonction : la variable locale est stockée globalement mais sa portée est toujours locale.

Exemple d'usge de static

```
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.
}
```

La bibliothèque standard std

Contient beaucoup de choses pour manipuler les objets fondamentaux :

- `std::string`
- `std::iostream`
- `std::array` etc.

Types primitifs

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

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 , 4.0 f }; // oat[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 utilisent une Gestion mémoire dynamique.

Chaînes de caractères

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

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

Chronométrer une fonction en C++ avec std::chrono

```cpp
#include <iostream>
#include <chrono>

long fibo(unsigned n){ if (n < 2) return n; return fibo(n-1) + fibo(n-2); }

int main()
{
    auto start = std::chrono::steady_clock::now();
    std::cout << "f(42) = " << fibonacci(42) << '\n';
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed_seconds = end-start;
    std::cout << "elapsed time: " << elapsed_seconds.count() << "s\n";
}
```

std::format

Par exemple pour afficher proprement une date. (sous Windows uniquemement pour le moment ...)

```
auto now = std::chrono::floor<std::chrono::milliseconds>(std::chrono::system_c
std::cout << std::format("{0:%d.%m.%Y %H:%M:%S}", now);
```

ou

```
auto now = std::chrono::system_clock::now();
std::cout << std::format("{0:%d.%m.%Y %H:%M:%S}", std::chrono::floor<std::chro
```

on peut aussi utiliser le raccourci `%T = %H:%M:%S`

Fonctions en C++

Les fonctions C++ ressemblent à celles du C avec quelques particularités ...

Fonctions

- Semblables à celles du 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.

Exemples de passage de paramètres

```
void fct( int a ) { ... } // Passage par copie.
void fct( int  pa ) { ... } // 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 de différentes manières.
- types simples - pointeurs - références

Exemples d'usages de const

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

Types complexes

Deux méthodes de définitions possibles :

1. struct
2. class

Struct

- Déclaration similaire au C.
- Par défaut attributs accessibles publiquement.

```
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 publics : public.
- Sinon nécessite des constructeurs/accesseurs pour manipuler les attributs.
- Accès aux attributs publics comme pour une structure.

Champs privés

Attributs privés (par défaut mais bien de l'expliciter)

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

Attributs publics

Exemple : attributs publics

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

Attributs publics et privés

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

Encapsulation - vie d'une instance

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

Constructeurs

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

Exemples de constructeurs

```
class Student
{
public:
// constructeur par défaut.
Student() { ... }
// constructeur avec arguments.
Student( ... ) { ... }
// constr. de recopie, on ne modi e pas l'original donc const.
Student( Student const & s ) { ... }
// move constr., rvalue reference.
Student( Student && s ) { ... }
```

Constructeur de recopie

1. Si le constructeur de recopie n'est pas défini par défaut, alors un constructeur de recopie par défaut est utilisé : tous les attributs sont copiés (comme pour les structures en C).
2. L'instance à copier est passée par référence : &.
3. Si l'instance à copier n'est pas modifiée (attributs non modifiés) par le constructeur de recopie, alors on doit 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 ) ...
```

Destructeurs

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

Exemple de destructeur

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

Exemple d'appel :

```
{ // 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
```

new et delete

- Les mots-clefs `new` et `delete` sont utilisés pour allouer et libérer de la mémoire dans le contexte du C++.
- On peut donc les utiliser au lieu des traditionnels `malloc` et `free`du C. (ne pas mélanger …)
- delete appelle le destructeur et désalloue la mémoire

exemple new/delete

```
int *a = new int;
delete a;

int *b = new int[5];
delete [] b;
```

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 de constructeur recopie

```
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 ; // a che 42...
```

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

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

1. Définition comme les fonctions.
2. 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 ;
};}
```

Modifieur static

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

1. les attributs d'une classe,
2. 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 static

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

Méthodes static

Accès à des attributs statiques : appel par nom complet

```
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.

Cf. :

- 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 (attention aux progs illisibles !!!).`

Exemple Matrix

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 )
{ ... }
```

Surcharge affection

- operator = : opérateur d'affectation
- définition obligatoire dans la classe

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

Surcharge affectation (suite)

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 d'où le `const`.
- La matrice ne doit pas être passée par copie d'où le `&`.

Déclaration des constructeurs/méthodes/...

- Dans la classe (.hpp).
- En dehors de la classe mais dans le fichier .hpp.
- Dans le fichier .cpp.
- Dans la classe elle-même

exemple de déclaration de constructeur dans la classe

Dans un .cpp :

```
class C {
}; void fct () { ... }
En dehors de la classe, dans le .hpp
class C {
void fct ();
};
void C:: fct () { ... }
```

Définition de constructeur dans le .hpp :

```
class C {
void fct ();
};
```

Déclaration dans le .cpp

```
#include <....hpp>
void C:: fct () { ... }
```

Inférences de types

1. `auto` : inférence de type (C++11)
2. Le spécificateur `auto` permet d'attribuer un type en le déduisant automatiquement.
3. Il permet d'obtenir des codes plus génériques.

Exemple inférence de type

```
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 &
```

Autre utilisation de auto

`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 et boucles

- Le `if`est comme en C mais il y a des subtilités ...
- On peut y ajouter le `switch` pour aiguillage.
- Le `for`peut se faire classiquement mais pas mal d'autres choses possibilités (range based loops, auto, etc.)

if

Comme en C :

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

if C++17

`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 )
```

Structure conditionnelle : 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 modi e 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 ;
}
```

Affichage/Saisie console

On a déjà aperçu ces objets permettant de représenter les flux fondamentaux du système.

Objets globaux

```
std::cin \\ entrée standard.
std::cout \\ sortie standard.
std::cerr \\ sortie erreur.
```

Exemple cin et cout

```
int a;
std::cin >> a;
std::cout << a << std::endl ;
```

Fichiers

- Lecture/Ecriture
- mode texte : manipulation de caractères, mots, lignes.
- mode binaire : manipulation d'octets.

```
#include <fstream>
std::ifstream, std::ofstream
```

Fichiers texte

```
// mode texte
std::ofstream ofs( "output.txt" );
ofs << "Hello!\n";
std::ifstream ifs ( "input.txt" );
std::string s ;
ifs >> s;
std::cout << s << std::endl ;
```

Exemple complet

```
#include<fstream>
#include<iostream>

int main(){
    std::cout<<"Ecriture dans fic.tx :"<<std::endl;
    std::ofstream ofs("fic.txt");
    ofs << "Hello le monde !\n";
    ofs.close();
    std::cout<<"Relecture de fic.txt" <<std::endl;
    std::ifstream ifs("fic.txt");
    std::string s;
    ifs >> s;
    ifs.close();
    std::cout << s << std::endl;
}
```

Fichiers binaires

```
//Mode binaire
std::ofstream ofs( "output.bin" );
int a = 5;
// Cast de pointeurs : de int* vers char*.
ofs.write( reinterpret cast< char  >( &a ),
sizeof( int ) );
std :: ifstream ifs ( "input.bin");
if s.read ( reinterpret cast< char  >( &a ), sizeof(int) );
std :: cout << a << std :: endl ;
```

Problèmes de mémoire

L'un des points délicats en C comme en C++

Allocation mémoire en C

- Allocation/désallocation dans le tas :
- Allocation : malloc
- Désallocation : free
- Inconvénient : taille donnée en octets pour l'allocation.

Allocation en C++

- Allocation : new
- Désallocation : delete

Exemple :

```
int   pi = new int ; // allocation d'un int dans le tas.
pi = 4;
. . .
delete pi ;
int   ptab = new int [100]; // allocation de 100 int.
ptab [ 10 ] = 123;
. . .
delete [] ptab;
```

Mauvaise allocation

- 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 indénie !
t1 [ 10 ] = 5;
```

Solutions aux fuites mémoire

- smart pointers pour remplacer les pointeurs.
- conteneurs standards pour masquer l'allocation dynamique.
- ajoutent un aspect sémantique aux pointeurs

Smart pointers en C++ >= 11

Les smart pointers (pointeurs intelligents) s'occupent de la gestion de certains types de pointeurs et de la libération de la mémoire les concernant, tout en conservant la syntaxe habituelle des pointeurs.

Les principaux Smart Pointers

- `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 peut être transmise par la fonction std::move qui ”déplace” le pointeur.
- l’instance originelle n’est alors plus utilisable

exemple unique_ptr

```
unique_ptr< Student > ps0( new Student( ... ) );
// ou : auto ps0 = std::make_unique<Student>();
std :: cout << p s0 << std :: endl ;
//auto ps1 = ps0; // Pas de constructeur de recopie.
// ps1 prend la propriété, ps0 devient invalide.
unique_ptr< Student > p s1( std ::move( p s0 ) );
// std::cout << *ps0 << std::endl; // Erreur segmentation.
ps1->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.

exemple shrared_ptr

```
shared_ptr< Student > ps0( new Student( ... ) );
// ou : auto ps0 = std::make_shared<Student>( ... );
std :: cout << ps0 << std :: endl ;
auto ps1 = ps0;
std :: cout << ps1.use_count () << std :: endl ;
ps1->setFirstName( "Hector" );
std :: cout << ps0 << std :: endl ;
std :: cout << ps1 << std :: endl ;
```

Généricité

L'usage des templates apporte une grande puissance au C++, au prix d'une certaine technicité syntaxique …

principes

La généricité consiste à développer un code ou une structure de données avec des types non explicitement définis i.e. génériques.

Applications de la généricité

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

On parle 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.

Templates et 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 );
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 ];
} }
```

Variadic templates

- Fonctions ou classes avec nombre de paramètres variables.
- Syntaxe : `template < typename... TS >`
- Utilisation de la récursion dans de nombreux cas avec spécialisation pour arrêt.
- Classes variadiques cf. `std :: tuple` par exemple.

Fonctions variadiques

```
// Fin de la récursion.
template < typename T >
void printer( T const & t ) {
std :: cout << t << std :: endl ;
}
// Cas général.
template < typename T, typename... TS >
void printer( T const & t , TS const &... ts ) {
std :: cout << t << ", ";
printer( ts ... );
}
// Utilisation.
printer( 1.0f , 3, 'a ' , "Hello !" );
```

Bibliothèque standard STL

La Standard Template Library apporte un framework dédié au maniement des collections basé sur l'usage des templates. La STL est un framework essentiel du C++ qui a beaucoup évolué avec les versions récentes de C++

Standard Template Library

Concepts :

- Conteneurs
- Itérateurs
- Algorithmes

Principaux conteneurs STL

- `std :: array`
- `std :: vector`
- `std :: deque`
- `std :: forward list`
- `std :: list`
- `std :: set`
- `std :: map`

Itérateurs et générateurs

- Généralisation des pointeurs pour le parcours/manipulation des conteneurs
- Algorithmes
- Manipulation des conteneurs
- Traitement, recherche, dénombrement, tri, ...

Exemples d'itérateurs et générateurs

```
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 de foncteur

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

Autre exemple : classe 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

- Fonctions anonymes.
- Utiles pour passer en paramètre d'un algorithme.
- forme générale : `[capture list ] (params) code;`.

exemples de lambdas

```
std :: vector< int > v0( 10 );
std :: vector< int > v1;
std :: generate( begin( v0 ), end( v0 ),
[ ] ( ) { return std :: rand()%10; } );
std :: copy if ( begin( v0 ) , end( v0 ),
std :: back_inserter ( v1 ) ,
[ ] ( int i ) { return i%2; } );
```

Lambdas : Capture de l'environnement

- `[]` : pas de capture.
- `[&]` : capture des variables externes à la lambda par référence.
- `[=]` : capture par valeur.
- `[a]` : capture de la variable a par valeur.
- `[&a]` : capture de la variable a par référence.

Exemple de capture par référence dans un lambda

```
std :: vector< int > v( 10 );
int i = 0;
std :: generate( begin( v ), end( v ), [&i ]() { return i++; } );
```

Threads

`std::thread`

- Constructeur variadique.
- Passage de la fonction à appeler en paramètre + arguments.
- Possibilité de passer une lambda.
- Attention : passage par référence avec `std::ref()`.
- Compilation avec l'option -pthread.
- Synchronisation avec `std::mutex`.

Exemple : thread

```
// Fonctions à appeler.
void fct0 () { ... }
void fct1( int n ) { ... }
void fct2( int & n ) { ... }
std :: thread th0( fct );
std :: thread th1( fct , 5 );
int a = 5;
std :: thread th2( fct , std :: ref ( a ) );
std :: thread th3( []() { std :: cout << "Lambda!\n"; });
th0 . join ();
th1 . join ();
th2 . join ();
th3 . join ();
```

Futures

- Exécution de tâches asychrones.
- Plus simple que les threads.
- Exemple : tâche asynchrone

exemple de future

```
std :: future< int > f0 = std :: async( std :: launch :: async , fact ,
. . .
int res = f0 . get (); // Récup. du résultat.
```

Héritage

```
class B : public/protected/private A
public : /* les membres publics de A restent publics :
accessibles par l'héritier i.e. dans la classe B,
accessibles à partir des instances de B.*/
protected : /* les membres publics de A deviennent protected :
accessibles dans la classe B,
inaccessibles à partir des instances de B.*/
private : /* Les membres publics et protected deviennent privés :
inaccessibles dans la classe B,
inaccessibles à partir des instances de B.*/
```

Héritage multiple

- Il est possible d'hériter de plusieurs classes contrairement à Java.
- Risques d'ambiguïtés si plusieurs ancêtres possèdent des attributs ou méthodes de même prototype.

Exemple d'héritage multiple

```
struct A {
int a;
};
struct B {
int a;
};
struct C : public A, B {
void display () {
std :: cout << a << std :: endl ; // ???
std :: cout << A:: a << std :: endl ;
std :: cout << B:: a << std :: endl ;
}; };
```

Polymorphisme : principe

- Définir une hiérarchie de types
- Pouvoir manipuler ces types à partir du type ancêtre.
- Garder des noms de méthodes homogènes

Exemple polymorphisme

```
class Tool { ... };
class Screwdriver : public Tool { ... };
class Hammer: public Tool { ... };
Tool   screwdriver = new Screwdriver ();
```

Problèmes d'héritage

```
struct Tool {
 void display () {
 std :: cout << "This is a tool .\n";
};
struct Screwdriver : public Tool {
void display () {
  std :: cout << "This is a screwdriver .\n";
  }
};
```

Problème d'héritage (suite)

```
int main() {
std :: unique_ptr< Tool > screwdriver
= std :: make_unique< Screwdriver >();
screwdriver->display (); // a chage ?
}
```

-> Résultat : `This is a tool`

Explications problème héritage

- Par défaut le type est déterminé statiquement (à la compilation) et ne dépend pas du type utilisé pour l'instanciation.
- Dans ce cas le type de screwdriver est `std:: unique_ptr< Tool >`.
- Donc la méthode display de Tool est appelée.

Mots-clefs du Polymorphisme

- `virtual` : définition d'une méthode virtuelle qui peut être surchargée par les héritiers.
- `virtual ... =0`: méthode virtuelle pure i.e. la classe contenant cette méthode ne peut être instanciée.
- `override` : indique que la méthode doit surcharger la même méthode (même prototype) de l'ancêtre.
- `nal` : indique que la méthode ne peut plus être surchargée par les héritiers.

Exemple basé sur le polymorphisme

```
struct Tool {
virtual void display () {
 std :: cout << "This is a tool .\n";
};
struct Screwdriver : public Tool {
void display () {
 std :: cout << "This is a screwdriver .\n";
};
int main() {
std :: unique_ptr< Tool > screwdriver = std :: make_unique< Screwdriver > ()
screwdriver->display (); // This is a screwdriver.
}
```

Mécanisme polymorphisme 1/2

- `virtual` indique que la méthode n'est plus appelée directement.
- Une table virtuelle est créée à l'instanciation pour la classe de base et pour ses héritiers qui contient des pointeurs vers les méthodes virtuelles uniquement.
- Pour la classe de base, ici `Tool`, la table contient un pointeur vers la méthode `Tool::display`.
- Pour la classe héritière, ici `Screwdriver`, la table contient un pointeur vers la méthode `Screwdriver :: display`.

Mécanisme polymorphisme 2/2

- Lors de la compilation le type `std:: unique_ptr< Tool >` est bien attribué à l'instance.
- Lors de l'instanciation la table virtuelle est par contre remplie lors de la construction et contient donc le pointeur vers la méthode `Screwdriver :: display`.
- Lors de l'appel, la méthode `display` est appelée sur le type `Tool` mais c'est le pointeur de la table virtuelle qui est utilisé.

Conséquences du polymorphisme

- L'appel de méthodes ne dépend plus du type de l'instance.
- Uniquement valable pour les méthodes virtuelles.
- Les méthodes standards ne sont pas affectées.

Le polymorphisme engendre un surcoût :

1. mémoire : La table virtuelle augmente la taille de l'instance avec un pointeur par méthode virtuelle.
2. temps d'exécution : L'appel d'une méthode virtuelle est plus coûteux que pour une méthode standard : indirection, il faut récupérer le pointeur dans la table avant de faire l'appel.

compléments sur les destructeurs

```
Tool   screwdriver = new Screwdriver ;
. . .
delete screwdriver ;
```

`delete` est appelé sur le type déterminé statiquement, ici `Tool`. Donc le destructeur de la classe `Screwdriver` n'est pas appelé : fuite mémoire. Solution : rendre le destructeur virtuel.

Destructeur virtuel

```
struct Tool {
. . .
virtual ~Tool() = 0;
};
Tool ::~ Tool() {}
struct Screwdriver {
. . .
};
~Screwdriver() { ... }
```

Interfaces

Une interface est une classe contenant uniquement des méthodes virtuelles pures. Il n'y a pas de mot-clé spécifique comme interface en Java. Exemple : `c++ class UneInterface { public : virtual method( ... ) = 0; };`

Classes abstraites

Une classe est abstraite si au moins une de ses méthodes est virtuelle pure :

```
class UneClasseAbstraite {
  int value ;
  public :
     UneClasseAbstraite() { ... }
     virtual method( ... ) = 0; // au moins une méthode VP.
};
```

Exemple : Design pattern Factory 1/3

```
struct Tool { // Abstract class.
  std :: string name;
  virtual void use() = 0;
  virtual void display () {
  std :: cout << "This is a " << name;
};}
struct Screwdriver : public Tool {
  std :: string type ;
  Screwdriver( std :: string type ) type( type ) { ... }
  void use() override { ... }
  void display () {
  Tool :: display ();
  std :: cout << " of type " << type ;
  } };
```

Exemple : Design pattern Factory 2/3

```
enum class ToolType { screwdriver , hammer, ... };
struct ToolFactory {
  template < typename T >
  static std :: unique_ptr< Tool > create(ToolType tooltype ,
  T &... args )
  {
   switch( tooltype) {
     case ToolType :: screwdriver :
        return std :: make_unique<Screwdriver >( args );
        break;
     case ...
} }
};
```

Exemple : Design pattern Factory 3/3

```
using ToolBox = std :: vector< std :: unique_ptr< Tool > >;
ToolBox toolbox ;
toolbox . push back( ToolFactory :: create(hammer,
. . . ) );
toolbox.push back( ToolFactory::create(screwdriver ," Phillips" ) );
for( auto & tool : tool box ) {
  tool->use ();
  tool->display ();
}
```

Métaprogrammation

- Idée : calculer/transformer le code à la compilation pour améliorer les performances à l'exécution du programme.
- Effet de bord du mécanisme des templates.
- Techniques : utilisation de `templates` et de `constexpr`.

Exemple : Fonction meta-factorielle

```
template < unsigned int N >
struct fact {
static unsigned int const value
= N   fact< N - 1 >::value ;
};
template <>
struct fact< 0 > {
  static unsigned int const value = 1;
};
std :: cout << fact< 10 >::value << std :: endl ;
```

Expression templates 1/7

- Objectifs : suppression des temporaires et des recopies.
- Evaluer le code le plus tard possible (évaluation retardée)

Exemple expression template : addition de vecteurs 2/7

```
class vec3f {
vec3f & operator=( vec3f const & v ) {...}
}; vec3f operator+( vec3f const & v ) {...}
. . .
vec3f v0 = v1 + v2 + v3 + v4;
```

Complexités de ces opérations 3/7

1. addition de v3 et v4 (for) dans un vec3f temporaire,
2. addition de v2 et du temporaire (for) dans un nouveau temporaire,
3. addition de v1 et du nouveau temporaire (for) dans un autre temporaire,
4. recopie du dernier temporaire dans v0 (for).

Total : 3 vec3f temporaires + 4 boucles for.

Version sans classe (C)

```
for( ... ) {
  v0[ i ] = v1[ i ] + v2[ i ] + v3[ i ] + v4[ i ];
}
```

Total : pas de vec3f temporaire, 1 boucle for.

Expression templates 4/7

- Allier la programmation haut-niveau/objet avec l'efficacité des langages bas-niveau.
- Objectif : suppression des temporaires, des parcours inutiles.
- Idée : surcharger les opérateurs et retarder l'évaluation.

Exemple 5/8

```
class vec3f {
. . .
vec3f operator+( vec3f const & v ) {
vec3f tmp( size () );
for( std :: size_t i = 0 ; i < size() ; ++i ) {
tmp[ i ] = this [ i ] + v[ i ];
}
return tmp;
}
. . .
};
```

Expression templates 6/8

1. Une expression générique contient 2 opérandes et 1 opérateur. Dans notre exemple, une expression peut contenir 2 vec3f, ou être composée d'autres expressions.
2. L'opérateur [] applique l'opérateur Op sur les 2 opérandes.
3. Si les opérandes sont des expressions, l'application s'applique récursivement.

Code correspondant

```
template < typename L, typename Op, typename R >
struct expr {
   L const & l ;
   R const & r;
Expr( L const & l , R const & r ): l(l), r(r) {}
auto operator [](  std :: size_t i ) {
   return Op :: apply( l[ i ] , r[ i ] );
   };
}
```

Expression templates 7/10

Surcharge de l'opérateur + pour les expressions : plus de calcul, construction d'une expression.

```
template < typename L, typename R >
expr< L, Plus , R > operator+( L const & l , R const & r ) {
   return expr< L, Plus , R >( l , r );
}
```

Expression templates 10/10

Définition de l'opération à effectuer.

```
struct Plus {
static unsigned int apply( unsigned int a,
return a + b; unsigned int b ) {
};
}
```

Composition des expressions :

```
vec3f v0, v1;
. . . v0 + v1; // expr< vec3f, Plus, vec3f >
. . . v0 + v1 + v2; // expr< vec3f, Plus, expr< vec3f, Plus, vec3f > >
. . .
```

Evaluation d'expression par opérateurs [] ou = (1/2)

```
template < typename L, typename Op, typename R >
struct expr {
. . .
auto operator []( std :: size_t i ) {
return Op:: apply( l [ i ] , r [ i ] );
};}
```

Evaluation d'expression par opérateurs [] ou = (2/2)

Exemple

```
class vec3f {
. . .
template < typename Expr >
array & operator=( Expr const & e )
{ v. resize ( e. size () );
for( std :: size_t i = 0 ; i < v.size() ; ++i )
{ v[ i ] = e[ i ];
}
return this ;
}
. . .
};
```

Boost

- bibliothèque C++ open-source libre (licence Boost)
- contient beaucoup de fonctionnalités non intégrées au C++
- ”bac à sable” avant éventuel introduction dans le standard

Contenu de Boost

- Asio : réseau
- Algorithm
- DateTime
- Filesystem
- GIL : Generic Image Library
- Interprocess
- Log
- Serialization . . .

Lecture json avec boost 1/2

Utilisation de `boost::property tree`.

Lecture d'un fichier json : `json { "login": "jubertie", "server_name": "192.168.0.254", "array": [ "1", "2", "7" ] }`

Lecture json avec boost 2/2

```
boost::property tree
boost :: property tree :: ptree p;
boost :: property tree :: read json ( "config . json" , p );
auto login = p.get( "login", "none" );
auto server name = p.get( "server name", "localhost" );
auto missing = p.get( "missing", "missing" );
std :: cout << "login=" << login << std :: endl ;
std :: cout << "server name=" << server name << std :: endl ;
std :: cout << "missing=" << missing << std :: endl ;
for( auto x: p. get child ( "array" ) ) {
std :: cout << x. second . get value< int >() << std :: endl ;
}
```

C++17 : langage

- Mot clé typename comme alternative à class pour les paramètres template template
- Nouvelles règles de déduction pour `auto`
- Imbrication simplifiée des namespaces : `namespace X::Y { ... }` au lieu de `namespace X { namespace Y { ... } }`
- if statique à la compilation : `if constexpr(expr)`
- if avec initialisation : `if( init ; expr )`
- Déduction de type pour les constructeurs :

```
std :: pair( 1.0f, 'a' ) au lieu de std :: pair< oat, char>( 1.0f, 'a')
```

C++17 : bibliothèque

- Suppression de types et fonctions dépréciées : std:: auto ptr, std :: random shuffle, ...
- Ajout de la bibliothèque `filesystem`
- Versions parallèles des algorithmes

```
std :: byte : type octet
std :: string view
std :: optional
std :: any
```

C++2x

- Modules : remplacement du mécanisme `#include` par `import` :
- amélioration du temps de compilation, plus de `#ifdef ... HPP #de ne` pour les fichiers . hpp.
- Concepts
- Ranges
- Coroutines …