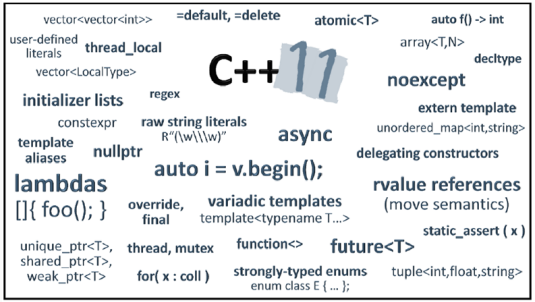
Programmation objet avancée en C++ moderne

Programmation générique - bibliothèque standard.

Mathias Paulin
IRIT-CNRS Université Paul Sabatier

Année universitaire 2016-2017



Classes templates

Introduction

Support à la programmation générique

Programmation utilisant les types comme paramètres.

- Types ou valeurs comme paramètres de définition de
  - Classes
  - Fonctions
  - Alias
- Représentation d'une large gamme de concepts
  - Efficacité en temps et en espace du code résultat
- Pas de contraintes sur les paramètres
  - Pas de restriction sur les types utilisables,
- Résultat fortement typé
  - Utilisation impossible d'un objet sans respecter sa classe.

Classes templates

Les templates par l'exemple

Chaîne de caractères

- Comment gérer différents types de "caractères"

Classe paramétrée par le type de caractère :

```
template <typename C>
class String {
public :
  String();
  explicit String (const C*);
  String(const String &);
  // ...
  C& operator[](int i);
  String& operator+=(C c);
  // ...
private :
  // ...
};
```

`template <typename C>` : définition d'une classe template.

Nom de type `C` :

- Existant jusqu'à la fin de la déclaration
- Impossible de spécifier des propriétés sur `C`
  - expérimental en C++17 : concept

Usage :

```
String<char> cs;

class JChar { ... }
String<JChar> js;
```

Utilisation des classes templates

- Exactement comme les autres.
- Aucun surcoût à l'exécution.
- Code généré plus petit :
  - Seules les méthodes utilisées sont effectivement compilées.

Fonctions templates

Principe des templates applicable aussi aux fonctions

- Fonctions paramétrées génériques
- Principe d'écriture et d'utilisation similaire aux classes.
- Très utilisées dans la bibliothèque standard.

Définition d'une classe template

- Définition des membres d'une classe template de la même façon qu'un membre de classe non template.
  - En ligne dans la déclaration de classe : de façon classique.
  - En dehors de la déclaration : doit être déclaré template :

```
template <typename C>
String& String<C>::operator+=(C c) {
  // add c at the end of this string
  return *this;
}
```

- Qualification avec <C> redondante dans le scope de String<C>
  - String<C> ::String est le constructeur.
  - template <typename C> String& String<C> ::operator+=(...).

Définition d'une classe template

- Surcharge des fonctions templates par rapport à leurs paramètres possible.
- Surcharge du paramètre template impossible
  - Notion de spécialisation.
- Surcharge du nom de classe impossible

```
template <typename C>
class String { /* ... */};

cass String { /* ... */};  // error : double definition of class String
```

- Un type utilisé comme paramètre template doit fournir tous les services nécessaire à la classe template.
  - Pour le moment, vérification automatique impossible
  - Notion de *concept* dans la norme C++17 (expérimental avec gcc)

Instanciation d'une classe ou fonction template

- Instanciation : génération à partir d'une définition template et de paramètres.
- Spécialisation : version pour une valeur particulière de paramètres.
  - A la charge du compilateur.
  - Peut être forcée par le programmeur.
- Instanciation paresseuse :

```
String<char> cs;

void f() {
  String<JChar> js;
  cs = "Hello world!";
}
```

Génération (Instanciation) de :

- déclarations String<char> et String<JChar>.
- constructeurs par défaut et destructeurs.
- String<char> ::operator=(const char*).

- Composition par instanciation : permet d'éliminer de nombreux appels direct ou indirect à des fonctions : code plus efficace.

Classes templates

Vérification de type

Vérification de type

- Très nombreuses informations disponibles lors de l'instanciation
  - Grande flexibilité de génération.
  - Code généré très efficace.
- Vérification statique et indication d'erreur difficile.
  - Vérification de type sur code généré.
  - Report d'erreur avec des informations inconnues du programmeur.
    Très difficile de comprendre une erreur pour un programmeur non expert ...
- Origine de la plupart des erreurs d'instanciation : impossible d'écrire :

```
template <Container Cont, typename Elem>
  requires Equal_comparable<Cont::value_type, Elem<()
int find_index(Cont &c, Elem e);
```

Notion de concept

- Impossible d'écrire

```
template <Container Cont, typename Elem>
  requires Equal_comparable<Cont::value_type, Elem<()
int find_index(Cont &c, Elem e);
```

  - *Cont* est un type de container
  - On doit pouvoir comparer une valeur de type *Elem* avec un élément de *Cont*

- Besoin de définir un cadre et un vocabulaire pour exprimer ces obligations
  - Une obligation est un prédicat.
  - `Container<T>` est vrai si `T` est un type de container, faux sinon.
- Un tel prédicat est appelé un *concept*.
  - pas (encore) un composant du langage C++ mais doit être intégré dans la démarche de programmation.

Notion de concept

- Spécification d'un concept comme un ensemble de commentaires.
- Exemple : concept Container<T>
  - Expression des propriétés devant vérifier T pour être un type de container
  - T doit avoir un opérateur d'indexation []
  - T doit avoir un méthode membre size()
  - T doit avoir un type membre value_type indiquant le type de ses éléments ...

```
template <typemane T> class MyContainer {
public :
  using value_type = T;
  MyContainer(...) { /* ... */ }
  size_t size() const {return ...;}
  T& operator[](unsigned int i) { return ...; }
private :
  // ...
};
```

Équivalence de type

- Génération de type à partir d'une déclaration de template et de ses arguments
  - Génération du *même* type pour de *mêmes* d'argument
  - Signification de *même* ?

```
using uchar = unsigned char;
using UChar = unsigned char;

String<char> s1;
String<unsigned char> s2;
String<uchar> s3;
String<UChar> s4;
```

- s1 et s2 : type différent
- s2, s3 et s4 : même type

```
template<typename T, int N>
class Buffer;
using CString = String<char>;

Buffer<CString, 10> b1;
Buffer<char, 10> b2;
Buffer<char, 20-10> b3;
```

- b1 et b2 : type différent
- b2 et b3 : même type

Équivalence de type

- Génération de type à partir d'une déclaration de template et de ses arguments
  - Génération du *même* type pour de *mêmes* d'argument
  - Génération de type différents pour des arguments en relation.

```
class Shape { /* ... */ };
class Circle : public Shape {
/* ... */
};

Shape *p{new Circle()}; //polymorphisme, conversion de Circle* vers Shape*
vector<Shape>* q{new vector<Circle>{}}; // error
vector<Shape> vs{vector<Circle>{}};     // error
vector<Shape*> vs{vector<Circle*>{}};   // error
```

- Conversions possibles mais à la charge du programmeur.

Classes templates

Données et méthodes membres

Membres de classe template

Comme une classe standard, une classe template peut avoir :

- des données et méthodes membres (constantes ou non),
- des alias et des types membres,
- des membre template.

Données membres

- Données d'instance (non static) initialisables en ligne ou par constructeur

```
template<typename T> struct X {
  int m1 = 7;
  T m2;
  X(const T& x) : m2{x}{}
};
```

```
X<int> xi{7};
X<string> xs{"wjwf mf d—"};
```

- données d'instance peuvent être ***const*** mais pas ***constexpr***

Méthodes membres

- Méthodes d'instance (non static) pouvant être définies dans ou hors de la classe

```
template<typename T>
struct X {
  void mf1() { /* ... */ } // Définition dans la classe
  void mf2();
};

template<typename T>
void X<T>::mf2() {  // Définition hors classe
  /* ... */
}
```

- Méthodes membres pouvant être virtuelles

Membres static

- Un membre static, non défini dans la classe, doit avoir une définition unique dans un programme.

```
template<typename T>
struct X {
  static constexpr Point p{100, 250}; // Point doit être un type littéral
  static const int m1 = 7;
  static int m2 = 8;   // erreur m2 n'est pas constant
  static int m3;
  static voif f1() { /* ... */ }
  static void f2();
};

template<typename T> int X<T>::m1 = 88; // erreur : double définition
template<typename T> int X<T>::m3 = 99;

template<typename T> int X<T>::f2() { /* ... */ }
```

Alias de type membres

- Très utiles pour unifier les notations et pour la meta-programmation.

```
template<typename T>
class X {
  using value_type = T;
  using iterator = Vector_iter<T>; // Vector_iter défini ailleurs
  // ...
};
```

Type membres

- Défoinition d'énumération ou de sous-types.

```
template<typename T>
class X {
  enum E1 {a, b};
  enum class E2;
  struct C2;
}
```

Membres template

Une classe ou une classe template peut avoir des membres qui sont aussi template.

- Utile pour représenter des types liés de façon flexible mais robuste.

```
template<typename Scalar>
class complex {
  Scalar re, im;
public :
  complex() : re{}, im{} {}
  template<typename T>
    complex(T rr, T ii=0) : re{rr}, im{ii} {}

  complex(const complex&) = default;
  template<typename T>
    complex(const complex<T>& c) : re{c.real()}, im{c.imag} {};

  Scalar real() {return re;}
  Scalar imag() {return im;}
};
```

Membres template

- Permet des conversions contrôlées entre les complexes, sans erreur d'arrondi

```
complex<float> cf;  / valeur par défaut
complex<double> cd{cf}; // OK, conversion double vers float
complex<float> cf2{cd}; // ERROR,  pas de conversion float vers double

class Quad {
  // no conversion to int
};
complex<Quad> cq;
complex<int> ci{cq}; // ERROR, pas de conversion Quad vers int
```

- Construction possible d'un complex<T1> à partir d'un complex<T2> si construction possible d'un T1 à partir d'un T2.

Membres template

- Un membre template ne peux pas être virtuel.
  - Incompatible avec la gestion du polymorphisme par table de fonctions virtuelle.
  - Editeur de lien trop complexe à réalisé pour supporter ceci.

Membres amis

- Une classe template peut désigner des classes amies.

```
class C;

template<typename T>
class My_class {
  friend C;
  friend class C1;
}
```

```
template<typename T>
class My_other_class {
  friend T;
  friend My_clas<T>;
}
```

- La relation d'amitié n'est ni transitive, ni héritée

Membres amis

- Une classe template peut désigner des fonctions amies.

```
template<typename T> class Matrix;
template<typename T> class Vector {
  T v[4];
public :
  friend Vector operator*<>(const Matrix<T>&, const Vector&);
}
template<typename T> class Matrix {
  Vector<T> v[4];
public :
  friend Vector<T> operator*<>(const Matrix&, const Vector<T>&);
}

template<typename T>
Vector<T> operator*(const Matrix<T>& m, const Vector<T>& v) {
  //... accès direct à m.v[i] et v.v[i]
}
```

Fonctions template

Introduction

Rôle et usage simple

- Classe template : définition de container.
- Fonction template : manipulation des container.

Tri d'un vecteur

```
template<typename T> void sort(vector<T>&); // déclaration

void f(vector<int>& vi, vector<string>& vs) {
  sort(vi);    // sort(vector<int>&;
  sort(vs);// sort(vector<string>&;
}
```

- Paramètres template déduits des paramètres de la fonction.

Rôle et usage simple

Tri d'un vecteur

```
template<typename T>
void sort(vector<T> &v) {
  // Shell sort (Knuth, Vol3. p. 84)
  const size_t n = v.size();

  for (int gap=n/2; 0<gap; gap/=2)
    for (int i=gap; i<n; ++i)
      for (int j=i-gap; 0<=j; j-=gap)
        if (v[j+gap]<v[j])
          swap(v[j], v[j+gap];
}
```

- Fonction générique sur le type des éléments à trier
- Nécessite un opérateur < défini sur le type T.

Rôle et usage simple

- Version générique sur l'opérateur de comparaison

Tri d'un vecteur

```
template<typename T, typename Compare=std::less<T>>
void sort(vector<T> &v) {
  // Shell sort (Knuth, Vol3. p. 84)
  const size_t n = v.size();
  Compare cmp; // make a default compare object

  for (int gap=n/2; 0<gap; gap/=2)
    for (int i=gap; i<n; ++i)
      for (int j=i-gap; 0<=j; j-=gap)
        if (cmp(v[j+gap],v[j]))
          swap(v[j], v[j+gap];
}
```

Rôle et usage simple

- Version générique sur l'opérateur de comparaison.
  - Plusieurs ordres possibles

Tri d'un vecteur

```
struct NoCase {
  // case insensitive compare
  bool operator()(const string& a, const string& b) const;
}

void f(vector<int>& vi, vector<string>& vs) {
  sort(vi); // sort using <
  sort<int, std::greater<int>>(vi); // sort using >

  sort(vs); //case sensitive sort
  sort<string, No_Case>(vs); //case insensitive sort
}
```

Fonctions template

Arguments des fonctions template

Déduction des paramètres template

- Fonction template : essentielles à l'écriture de programmes génériques.
- Déduction de paramètres template cruciale
  - Déduction de type et de valeurs à partir d'un appel de fonction template.
  - Possible si la liste d'arguments définit de manière unique les paramètres template.

```
template<typename T, int max>
struct Buffer {
  T buf[max];
};

template<typename T, int max>
T& lookup(Buffer<T, max>& b, const char* p);

string& f(Buffer<string, 128>& buf, const char*p) {
  return lookup(buf, p); // use th lookup() where T is string and max is 128
}
```

Déduction des paramètres template

- Classe template : déduction de paramètres impossible
  - Conséquence de la flexibilité des constructeurs.
- Utilisation d'une fonction pour réaliser la déduction et la création.

```
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 a, T2b) {
  return {a, b}
}

auto x = make_pair(1, 2);                    // x is a pair<int, int>
auto y = make_pair(string("New York"), 7.7); // y is a pair<string, double>
```

- Possibilité de préciser de façon explicite les arguments template.
  - ex : static_cast, dynamic_cast, ...

Surcharge et dérivation de templates

Le mécanisme de déduction de type est compatible avec la surcharge et la dérivation :

```
template<typename T>
  class B{ /* ... */ }

template<typename T>
  class D : public B<T> { /* ... */ }

template<typename T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd) {
  f(pb);     // appel de f<int>(pb)
  f(pd);     // appel de f<int>(satic_cast<B<int>*>(pd))
}
```

Surcharge et paramètres non déduits

Un paramètre non déduit est traité comme un paramètre de fonction non template

```
template<typename T, typename C>
T get_nth(C& p, int n); // get the nth element of C

struct Index {
  operator int(); // user defined conversion from Index to int
};

void f(vector<int>& v, short s, Index i) {
  int i1 = get_nth<int>(v, 2); // exact match
  int i2 = get_nth<int>(v, s);  // standard conversiont : short to int
  int i3 = get_nth<int>(v, i); // user-defined conversion : Index to int
}
```

Alias template

- Définition d'alias de template avec spécialisation
  - totale,
  - partielle.

```
template<typename T, typename Allocator = allocator<T>> class vector;

using Cvec = vector<char>; // alias with both arguments bounded
Cvec vc {'a', 'b', 'c'};

template<typename T>
using Vec = vector<T, My_allocator<T>>; // alias with 2nd argument bounded
// Vec is still an alias
Vec<int> fib = {1, 1, 2, 3, 5, 8, 13};
```

Alias template

- Alias compatible avec spécialisation

```
template<int>
struct int_exact_traits {
  // idea : int_exact_traits<N>::type is an int type with exactly N bits
  using type = int;
}

template<>
struct int_exact_traits<8> {
  using type = char;
}

template<>
struct int_exact_traits<16> {
  using type = short;
}
```

Alias template

- Alias compatible avec spécialisation

```
template<int N>
using int_exact = typename int_exact_traits<N>::type

int_exact<8> a = 7; // int_exact<8> is an int with 8 bits

// simpler to write than

int_exact_traits<8>::type b = 7;
```

- Un alias ne peut pas être spécialisé
  - Ambiguïté de spécialisation.

Organisation et structuration du code

Trois approches possibles

1. Inclusion des définitions de template avant leur utilisation dans une unité de compilation.
   - 1 seul fichier : template.h (définiton)
2. Inclure les déclarations de template (uniquement) avant leur utilisation dans une unité de compilation. Inclure les définitions plus tard dans l'unité de compilation.
   - 2 fichiers : template.h (déclaration) et template.inl (définition)
3. Inclure les déclarations de template (uniquement) avant leur utilisation dans une unité de compilation. Définir les templates dans une autre unité de compilation.
   - 2 fichiers : template.h (déclaration) et template.cpp (définition)

- La compilation séparée des templates (cas 3 ci-dessus) n'est pas offerte par les compilateurs.

Organisation et structuration du code

- Solution courante : inclure les définitions templates dans unités de compilation.
- Une définition template est décrite dans un fichier .h

```
#include <iostream>
template<typename T>
void out(const T& t) {
  std::cerr << t << std::endl;
}
```

- Cette définition est incluse quand on en a besoin.

```
// file user1.cpp
#include "out.h"
// use out
```

```
// file user2.cpp
#include "out.h"
// use out
```

Organisation et structuration du code

- Solution courante : inclure les définitions templates dans unités de compilation.
- Le compilateur à la charge de la génération et de la gestion du code
  - Génération lorsque nécessaire.
  - Optimisation du code redondant
- Approche similaire à la gestion des fonctions `inline`
- Inconvenient
  - Risque d'augmentation des dépendances.
  - Risque d'interférence avec code utilisateur.
- Solutions
  - Éviter les macros.
  - Utiliser des `namespace`
  - Minimiser les dépendances entre un template et son environnement.
  - Utiliser la solution 2

Organisation et structuration du code

- Solution alternative : inclure les déclarations puis les définitions templates.
- Une définition template est décrite dans deux fichier .h (par exemple)

Déclaration

```
// file out.h
template<typename T>
void out(const T& t);
```

Définition

```
// file out_impl.h
#include <iostream>
template<typename T>
void out(const T& t) {
  std::cerr << t << std::endl;
}
```

- Cette définition est incluse quand on en a besoin.

```
// file user3.cpp
#include "out.h"
// use out
#include "out_impl.h"
```

Organisation et structuration du code

- Solution alternative : inclure les déclarations puis les définitions templates.
- Minimize les risque de collisions avec le code utilisateur
- Inconvenient
  - Problèmes avec les fonctions non-template et les membres static
  - Duplication de code dans plusieurs unités de compilation.
- Solutions
  - Utiliser la solution 1
  - Séparer le code template du code non template.
  - Encapsuler les templates lorsque possible.

Lambda expression

Introduction

Définition

Une Lambda expression (ou fonction lambda) est une notation simplifiée pour définir et utiliser un objet fonction anonyme.

- Par abus de langage, on appelle un tel objet une lambda.

Une Lamda expression est définie par une séquence de

- Une liste de capture, éventuellement vide, délimitée par `[]`.
- Une liste, optionnelle, de paramètres, délimitée par `()`.
- Un spécifieur optionnel `mutable` indicant si la lambda peut modifier son état.
- Un spécifieur optionnel `noexcept`
- Une déclaration optionnelle du type de retour sous la forme `->`
- Un corps, délimité par `{}` décrivant le code de la fonction.

Définition

- Une liste de capture, éventuellement vide, délimitée par `[]`.
  - Capture de noms depuis l'environnement de définition de la lambda pouvant être utilisés dans son corps.
  - Type de capture : référence ou copie
- Une liste, optionnelle, de paramètres, délimitée par `()`.
  - Arguments nécessaires à l'exécution de la lambda.
- Un spécifieur optionnel `mutable` indicant si la lambda peut modifier son état.
  - Modification des données capturées par valeur.

Usage

- Argument de type fonction pour un algorithme.
- Évaluation partielle de fonction.
- Programmation fonctionnelle d'ordre supérieur.

Exemple : paramètre d'un algorithme

Version objet fonction anonyme (lambda) :

```
void print_modulo(const vector<int> &v, ostream &os, int m) {
  // output v[i] if v[i]%m==0
  for_each(begin(v), end(v),
    [&os,m](int x){if (x%m == 0) os << x << '\n'; }
  );
}
```

Exemple : paramètre d'un algorithme

Version objet fonction non anonyme :

```
class Modulo_print {
  ostream & os; // members hold capture list
  int m;
public:
  Modulo_print(ostream &s, int mm) : os{s}, m{mm} {} // capture
  void operator()(int x) const {
    if (x%m == 0) os << x << '\n';
  }
};

void print_modulo(const vector<int> &v, ostream &os, int m) {
  // output v[i] if v[i]%m==0
  for_each(begin(v), end(v),
    Modulo_print{os, m}   / Object creation, use and destruction
  );
}
```

Exemple : paramètre d'un algorithme

Une lambda peut être nommée ...

```
void print_modulo(const vector<int> &v, ostream &os, int m) {
  // output v[i] if v[i]%m==0
  auto modulo_print = [&os,m](int x){ if (x%m == 0) os << x << '\n'; };

  for_each(begin(v), end(v), modulo_print);
}
```

Le nommage permet de

- Être plus rigoureux dans la conception d'une lambda.
- Construire des lambda récursives.

Lambda expression

Capture de l'environnement

Capture de l'environnement ([])

- Indique si la lambda peut accéder à des données de son environnement de définition
- Capture vide : []

```
void algo(vector<int> &v) {
  sort(begin(v), end(v)); // sort values of v
  sort(begin(v), end(v),
       [](int x, int y){return abs(x)<abs(y); }); // sort absolute values of v
}
```

  - Impossible d'accéder à des variables de l'environnement de définition.
  - Accès uniquement aux paramètres de la lambda ou à des variables globales.

Capture de l'environnement (`[]`)

- Capture implicite par référence : `[&]`
  - Toutes les variables locales sont capturées par références. Elle peuvent être modifiées par la lambda.
- Capture implicite par valeur : `[=]`
  - Toutes les variables locales sont capturées. Les noms désignent des copies des variables au moment de l'appel à la lambda.
- Capture explicite : `[capture-list]`
  - La liste `capture-list` contient les noms des variables locales auxquelles la lambda peut accéder par référence ou par copie.
  - Les variables précédées par `&` sont capturées par référence.
  - Les variables précédées par `=` sont capturées par copies.
  - La liste de capture peut contenir le nom `this`.

Capture de l'environnement ([])

- Capture implicite par référence, avec exceptions : `[&, capture-list]`
  - Toutes les variables locales sont capturées par références, sauf celles apparaissant dans la liste `capture-list` qui sont capturées par copie.
- Capture implicite par valeur, avec exceptions : `[=, capture-list]`
  - Toutes les variables locales sont capturées par valeur, sauf celles apparaissant dans la liste `capture-list` qui sont capturées par références. Les noms dans la liste doivent être précédés par &.
- Un nom précédé par & est toujours capturé par référence, un nom qui n'est pas précédé par & est toujours capturé par valeur.

Capture de l'environnement ([])

- Capture de l'objet `this`
  - Utile si la lambda doit accéder à des membres de l'objet de création.
  - Les membres sont toujours capturés par références.
  - `[this]` signifie que les membres sont accédés à travers l'objet et non copiés dans la lambda.
- Lambda `mutable`
  - Permet de modifier localement les données capturées par valeur.

```
void algo(vector<int>& v {
  int count = v.size();
  std::generate(begin(v), end(v), [count]()mutable{return --count;});
}
```

  - Seule la copie est modifiable. La capture reste par valeur.

Lambda expression

Lambda et durée de vie

Durée de vie d'une lambda

Une lambda peut survivre à son environnement de définition.

- Utilisation d'une lambda dans un thread différent de sa définition.
- Stockage d'une lambda pour utilisation future.
  - Exemple : ajout d'un couple (nom, action dans un menu)

```
void setup(Menu &m) {
  Point p1, p2, p3;
  // compute positions of p1, p2 and p3
  m.add("draw triangle", [&]{m.draw(p1, p2, p3)} ); // Disaster to come ...
}
```

  - Qu'est-ce qui sera dessiné lors de l'utilisation du menu ?
  - Que se passe-t-il si draw modifie ses paramètres ?
- Si une lambda doit survivre à son créateur, la capture DOIT être faite par copie.

Lambda expression

Appel et retour d'une lambda

Appel et retour

- L'appel d'une lambda suit les mêmes règles que l'appel d'une fonction.
- Le type de retour d'une lambda peut être déduit de son corps si
  - Il n'y a pas de `return` : le type de retour est `void`
  - Il n'y a qu'un `return` : le type de retour est le type de l'expression retournée.
- Si le type ne peut être déduit, il DOIT être défini.

```
int y;
auto addx = [=](int x){return x+y;}; // return type is int
auto z1 = addx(y);  // z1 is int

auto z2 = [=]{if (y) return 1; else return 2;}; // error : return type unknown
auto z3 = [=]()->int {if (y) return 1; else return 2;}; // OK : return type is int
```

Lambda expression

Type d’une lambda

Type d'une lambda

- Type appelé "*closure type*", unique pour une lambda
  - Fonction de l'environnement et du corps
  - Deux lambdas ne peuvent avoir le même type.
- Utilisation de la déduction de type
  - Une lambda peut initialiser une variable de type `auto`
- Type template `std::function<R(AL)>`
  - `R` : type de retour de la lambda, `AL` : Liste de type des paramètres de la lambda
- Une lambda sans capture peut-être affectée à un pointeur de fonction

```
double (*p1)(double) = [](double a) {return sqrt(a);}; // OK
double (*p2)(double) = [&](double a) {return sqrt(a);}; // error : capture
double (*p3)(double) = [](int a) {return sqrt(a);}; // error : argument type mismatch
```

Type d'une lambda

- Utilité de `std::function<R(AL)>` : lambda récursives
  - Lambda pour inverser une chaine de caractère de type "C"
  - Nommage et capture du nom pour écriture récursive

```
auto rev = [&rev](char* b, char* e) {
  if (1 < e-b) {swap(*b, *--e); rev(++b, e);}
};
```

  - Erreur de compilation ! Utilisation d'une variable `auto` impossible avant que le type réel ne soit déduit (ici, `void`)
  - Ecriture correcte

```
std::function<void(char* b, char* e)> rev = [&](char* b, char* e) {
  if (1 < e-b) {swap(*b, *--e); rev(++b, e);}
};
```

Bibliothèque standard

Introduction

Bibliothèque standard

- Ensemble des composants spécifiés par le standard ISO C++, dont le comportement est le même quelle que soit l'implantation du standard.
  - gcc, apple clang, icc, visual c++, ...
  - Quelque variations en terme de performances toutefois.
- DOIT être utilisée autant que possible (et en priorité) pour assurer
  - La portabilité du logiciel.
  - La maintenance du logiciel.
  - La durée de vie du logiciel.
  - La performance du logiciel.

Cours sur la bibliothèque standard ?

Impossible à faire ! Présentation des grands concepts.

à découvrir et manipuler en TD et en TP !

Bibliothèque standard

Services offerts

Services offerts par la bibliothèque standard

- Éléments du langage : range-for, gestion mémoire, RTTI.
- Spécificités liées à l'implantation : `numeric_limits`, ...
- Fonctions primitives ne pouvant être implantée facilement : `is_polymorphic`, ...
- Éléments de programmation parallèle de bas niveau (lock free)
- Support pour la programmation multi-thread.
- Support pour la programmation concurrente.
- Fonctions de bas niveau : `memmove`, ...
- Éléments non primitif génériques : `list`, `map`, `sort`, I/O streams, ...
- Cadre pour l'extension de la bibliothèque et la programmation générique.
- Composants utiles et généraux : random generators, complex, regexp, ...

Contraintes de conception

La bibliothèque standard a été conçue et développée pour être

- Utilisable par tout le monde, du débutant à l'expert
- Suffisamment performante pour servir de base au développement d'autres bibliothèques
- Primitive au sens mathématique : composants prévus pour un seul rôle
- Pratique, efficace et sûre pour les usages communs.
- Complète dans les services offerts.
- Extensible pour exploiter les types et opérations utilisateurs de la même manière que les types et opérations de la bibliothèque standard.

Bibliothèque standard

Organisation de la bibliothèque

Organisation de la bibliothèque

Bibliothèque placée dans le `namespace std`.

La bibliothèque standard est organisée par famille de services.

- Containers : classes template pour le stockage et l'organisation des données.
- General utilities : gestion mémoire, timers, gestion des types, ...
- Algorithms : ensemble d'algorithmes génériques (sort, search, scan, ...)
- Diagnostic : gestion des erreurs et des exceptions.
- String : gestion de chaînes de caractères.
- Input/Output : gestion de flux d'entrée sortie
- Langage support : éléments de base du langage C++
- Numerics : outils numériques, `complex`, `random`, ...
- Concurrency : programmation concurrente, `mutex`, `condition_variable`, ...

Containers

`<vector>`	Tableau 1D redimensionnable
`<deque>`	File à double entrée
`<forward_list>`	Liste simplement chaînée
`<list>`	Liste doublement chaînée
`<map>`	Tableau associatif
`<set>`	Ensemble ordonné
`<unordered_map>`	Table de hashage
`<unordered_set>`	Ensemble non ordonné
`<queue>`	File
`<stack>`	Pile
`<array>`	Tableau 1D de taille fixe
`<bitset>`	Tableau de bool

General utilities

`<utility>`	Operateurs, paires
`<tuple>`	Tuples
`<type_traits>`	Traits de type (propriétés)
`<typeindex>`	Transforme un `type_info` en un index ou hash code
`<functional>`	Objets fonction
`<memory>`	Outils de gestion de la mémoire
`<scoped_allocator>`	Allocateurs de scope (bloc de visibilité)
`<ratio>`	Arithmétique rationnelle à la compilation
`<chrono>`	Utilitaires pour la gestion du temps et des timers
`<ctime>`	Gestion de temps à la mode "C"
`<iterator>`	Itérateurs et support pour les itérateurs

Algorithms

`<algorithm>`	Algorithmes généraux - Tri, recherche, séquence, ...
`<cstdlib>`	Interface à la bibliothèque standard C - bsearch, qsort, ...

Diagnostic

`<exception>`	Classe de base pour la gestion d'exception
`<stdexcept>`	Exception standard - `runtime_error`, `logic_error`, ...
`<cassert>`	Macro assert (identique C)
`<cerrno>`	Gestion d'erreur à la "C"
`<system_error>`	Gestion des erreurs système

Strings and characters

`<string>`	String of `T`
`<cctype>`	Classification des caractères
`<cwctype>`	Classification des caractères - Wide)
`<regex>`	Gestion des expressions régulières

Concurrency

`<atomic>`	Types et opérations atomiques
`<condition_variable>`	Synchronisation par variable conditionnelle
`<future>`	Gestion des tâches asynchrones)
`<mutex>`	Classes pour l'exclusion mutuelle
`<thread>`	Gestion de la programmation multi-threads

Input/Output

`<iostream>`	Flux standard d'entrée sortie
`<ios>`	Classes de bases pour les flux standard)
`<streambuf>`	Buffer de flux
`<istream>`	Template des flux d'entrée
`<ostream>`	Template des flux de sortie
`<iomanip>`	Manipulateurs d'entrée/sortie)
`<sstream>`	Flux d'entrée sortie vers chaînes
`<fstream>`	Flux d'entrée sortie vers fichiers
`<iosfwd>`	Pré-déclaration des fonctionalités d'entrée sortie
`<cstdio>`	printf and co.

Langage support

`<limits>`	Limites numériques
`<new>`	Gestion dynamique de la mémoire
`<typeinfo>`	RTTI - Identification dynamique du type
`<initializer_list>`	Gestion des listes d'initialisation

Numerics

`<complex>`	Gestion des nombres complexes
`<valarray>`	Vecteurs et opérations sur les vecteurs
`<numeric>`	Opérations numériques générales
`<random>`	Générateurs de nombre aléatoires

Questions ?