CS100 Lecture 25

Templates I

Contents

- Function templates

- Class templates

- Alias templates, variable templates and non-type template parameters

Function templates

Motivation: function template

```cpp
int compare(int a, int b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
int compare(double a, double b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
int compare(const std::string &a, const std::string &b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

Motivation: function template

```cpp
template <typename T>
int compare(const T &a, const T &b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

- Type parameter: `T`
- A template is a **guideline** for the compiler:
  - When `compare(42, 40)` is called, `T` is deduced to be `int`, and the compiler generates a version of the function for `int`.
  - When `compare(s1, s2)` is called, `T` is deduced to be `std::string`, ….
  - The generation of a function based on a function template is called the **instantiation** (实例化) of that function template.

Type parameter

```cpp
template <typename T>
int compare(const T &a, const T &b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

- The keyword `typename` indicates that `T` is a **type**.

- Here `typename` can be replaced with `class`. **They are totally equivalent here.** Using `class` doesn't mean that `T` should be a class type.

Type parameter

```cpp
template <typename T>
int compare(const T &a, const T &b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```

- Why do we use `const T &`? Why do we use `b < a` instead of `a > b`?

Type parameter

```cpp
template <typename T>
int compare(const T &a, const T &b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

- Why do we use `const T &` ? Why do we use `b < a` instead of `a > b` ?
  - Because we are dealing with an unknown type!
  - `T` may not be copyable, or copying `T` may be costly.
  - `T` may only support `operator<` but does not support `operator>` .

* Template programs should try to minimize the number of requirements placed on the argument types.

Template argument deduction

To instantiate a function template, every template argument must be known, but not every template argument has to be specified.

When possible, the compiler will deduce the missing template arguments from the function arguments.

```cpp
template <typename To, typename From>
To my_special_cast(From f);

double d = 3.14;
int i = my_special_cast<int>(d);    // calls my_special_cast<int, double>(double)
char c = my_special_cast<char>(d); // calls my_special_cast<char, double>(double)
```

Template argument deduction

Suppose we have the following function template

```
template <typename P>
void fun(P p);
```

and a call `fun(a)`, where the type of `a` (ignoring references) is `A`.

- `A` is never considered to be a reference. For example,

```
const int &cr = 42;
fun(cr); // A is considered to be const int, not const int &
```

Template argument deduction

Suppose we have the following function template

```cpp
template <typename P>
void fun(P p);
```

and a call `fun(a)`, where the type of `a` (ignoring references) is `A`.

- Passing-by-value ignores references and top-level `const` qualifications. For example,

```cpp
const int ci = 42; const int &cr = ci;
fun(ci); // A = const int, P = int
fun(cr); // A = const int, P = int
```

Template argument deduction

Suppose we have the following function template

```cpp
template <typename P>
void fun(P p);
```

and a call `fun(a)`, where the type of `a` (ignoring references) is `A`.

- If `A` is an array or a function type, the **decay** to the corresponding pointer types takes place:

```cpp
int a[10]; int foo(double, double);
fun(a); // A = int[10], P = int *
fun(foo); // A = int(double, double), P = int(*)(double, double)
```

Template argument deduction

Suppose we have the following function template

```
template <typename P>
void fun(P p);
```

and a call `fun(a)`, where the type of `a` (ignoring references) is `A`.

- If `A` is `const`-qualified, the top-level `const` is ignored;
- otherwise `A` is an array or a function type, the **decay** to the corresponding pointer types takes place.

This is exactly the same thing as in `auto p = a;`!

Template argument deduction

Let `T` be the type parameter of the template.

- The deduction for the parameter `T x` with argument `a` is the same as that in `auto x = a;` .

- The deduction for the parameter `T &x` with argument `a` is the same as that in `auto &x = a;` . `a` must be an lvalue expression.

- The deduction for the parameter `const T x` with the argument `a` is the same as that in `const auto x = a;` .

The deduction rule that `auto` uses is **totally the same** as the rule used here!

Template argument deduction

Deduction forms can be nested:

```cpp
template <typename T>
void fun(const std::vector<T> &vec);

std::vector<int> vi;
std::vector<std::string> vs;
std::vector<std::vector<int>> vvi;

fun(vi); // T = int
fun(vs); // T = std::string
fun(vvi); // T = std::vector<int>
```

Exercise: Write a function `map(func, vec)`, where `func` is any unary function and `vec` is any vector. Returns a vector obtained from `vec` by replacing every element `x` with `func(x)`.

Template argument deduction

Exercise: Write a function `map(func, vec)`, where `func` is any unary function and `vec` is any vector. Returns a vector obtained from `vec` by replacing every element `x` with `func(x)`.

```cpp
template <typename Func, typename T>
auto map(Func func, const std::vector<T> &vec) {
  std::vector<T> result;
  for (const auto &x : vec)
    result.push_back(func(x));
  return result;
}
```

Usage:

```cpp
std::vector v{1, 2, 3, 4};
auto v2 = map([](int x) { return x * 2; }, v); // v2 == {2, 4, 6, 8}
```

Forwarding reference

Also known as "universal reference" (万能引用).

Suppose we have the following function template

```
template <typename T>
void fun(T &&x);
```

A call `fun(a)` happens for an expression `a`.

- If `a` is an rvalue expression of type `E`, `T` is deduced to be `E` so that the type of `x` is `E &&`, an rvalue reference.

- If `a` is an lvalue expression of type `E`, `T` **will be deduced to** `E &`.
  - The type of `x` is `T & &&` ??? (We know that there are no "references to references" in C++.)

Reference collapsing

If a "reference to reference" is formed through type aliases or templates, the **reference collapsing** (引用折叠) rule applies:

- `& &`, `& &&` and `&& &` collapse to `&`;
- `&& &&` collapses to `&&`.

```cpp
using lref = int &;
using rref = int &&;
int n;

lref&  r1 = n; // type of r1 is int&
lref&& r2 = n; // type of r2 is int&
rref&  r3 = n; // type of r3 is int&
rref&& r4 = 1; // type of r4 is int&&
```

Forwarding reference

Suppose we have the following function template

```cpp
template <typename T>
void fun(T &&x);
```

A call `fun(a)` happens for an expression `a`.

- If `a` is an rvalue expression of type `E`, `T` is deduced to be `E` so that the type of `x` is `E &&`, an rvalue reference.

- If `a` is an lvalue expression of type `E`, `T` **will be deduced to** `E &` and the reference collapsing rule applies, so that the type of `x` is `E &`.

Forwarding reference

Suppose we have the following function template

```
template <typename T>
void fun(T &&x);
```

A call `fun(a)` happens for an expression `a`.

- If `a` is an rvalue expression of type `E`, `T` is `E` and `x` is of type `E &&`.
- If `a` is an lvalue expression of type `E`, `T` is `E &` and `x` is of type `E &`.

As a result, `x` is always a reference depending on the value category of `a`! Such reference is called a **universal reference** or **forwarding reference**.

The same thing happens for `auto &&x = a;`!

Perfect forwarding

A forwarding reference can be used for **perfect forwarding**:

- The value category is not changed.
- If there is a `const` qualification, it is not lost.

Example: Suppose we design a `std::make_unique` for one argument:

- `make_unique<Type>(arg)` forwards the argument `arg` to the constructor of `Type`.
- The value category must not be changed: `make_unique<std::string>(s1 + s2)` must move `s1 + s2` instead of copying it.
- The `const`-qualification must not be changed: No `const` is added if `arg` is non-`const`, and `const` should be preserved if `arg` is `const`.

Perfect forwarding

- The value category must not be changed: `make_unique<std::string>(s1 + s2)` must move `s1 + s2` instead of copying it.

- The `const`-qualification must not be changed: No `const` is added if `arg` is non-`const`, and `const` should be preserved if `arg` is `const`.

The following does not meet our requirements: An rvalue will become an lvalue, and `const` is added.

```
template <typename T, typename U>
auto make_unique(const U &arg) {
  return std::unique_ptr<T>(new T(arg));
}
```

Perfect forwarding

- The value category must not be changed: `make_unique<std::string>(s1 + s2)` must move `s1 + s2` instead of copying it.

- The `const` -qualification must not be changed: No `const` is added if `arg` is non-`const` , and `const` should be preserved if `arg` is `const` .

We need to do this:

```cpp
template <typename T, typename U>
auto make_unique(U &&arg) {
  // For an lvalue argument, U=E& and arg is E&.
  // For an rvalue argument, U=E  and arg is E&&.
  if (/* U is an lvalue reference type */)
    return std::unique_ptr<T>(new T(arg));
  else
    return std::unique_ptr<T>(new T(std::move(arg)));
}
```

Perfect forwarding

`std::forward<T>(arg)` : defined in `<utility>` .

- If `T` is an lvalue reference type, returns an lvalue reference to `arg` .

- Otherwise, returns an rvalue reference to `std::move(arg)` .

```cpp
template <typename T, typename U>
auto make_unique(U &&arg) {
  return std::unique_ptr<T>(new T(std::forward<U>(arg))).
}
```

Variadic templates

To support an unknown number of arguments of unknown types:

```cpp
template <typename... Types>
void foo(Types... params);
```

It can be called with any number of arguments, of any types:

```cpp
foo();          // OK: `params` contains no arguments
foo(42);        // OK: `params` contains one argument: int
foo(42, 3.14);  // OK: `params` contains two arguments: int and double
```

`Types` is a **template parameter pack**, and `params` is a **function parameter pack**.

Parameter pack

The types of the function parameters can also contain `const` or references:

```cpp
// All arguments are passed by reference-to-const
template <typename... Types>
void foo(const Types &...params);
// All arguments are passed by forwarding reference
template <typename... Types>
void foo(Types &&...params);
```

Pack expansion

A **pattern** followed by `...` , in which the name of at least one parameter pack appears at least once, is **expanded** into zero or more comma-separated instantiations of the pattern.

- The name of the parameter pack is replaced by each of the elements from the pack, in order.

```cpp
template <typename... Types>
void foo(Types &&...params) {
  // Suppose Types is T1, T2, T3 and params is p1, p2, p3.
  // &params... is expanded to &p1, &p2, &p3.
  // func(params)... is expanded to func(p1), func(p2), func(p3).
  // func(params...) is expanded to func(p1, p2, p3)
  // std::forward<Types>(params)... is expanded to
  //   std::forward<T1>(p1), std::forward<T2>(p2), std::forward<T3>(p3)
}
```

Perfect forwarding: final version

Perfect forwarding for any number of arguments of any types:

```cpp
template <typename T, typename... Ts>
auto make_unique(Ts &&...params) {
    return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

This is how `std::make_unique`, `std::make_shared` and `std::vector<T>::emplace_back`
forward arguments.

`auto` and template argument deduction

We have seen that the deduction rule that `auto` uses is exactly the template argument deduction rule.

If we declare the parameter types in an **lambda expression** with `auto`, it becomes a **generic lambda**:

```cpp
auto less = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
std::string s1, s2;
bool b1 = less(10, 15); // 10 < 15
bool b2 = less(s1, s2); // s1 < s2
```

`auto` and template argument deduction

We have seen that the deduction rule that `auto` uses is exactly the template argument deduction rule.

Since C++20: Function parameter types can be declared with `auto` .

```cpp
auto add(const auto &a, const auto &b) {
  return a + b;
}
// Equialent way: template
template <typename T, typename U>
auto add(const T &a, const U &b) {
  return a + b;
}
```

Class templates

Define a class template

Let's make our `Dynarray` a template `Dynarray<T>` to support any element type `T`.

```cpp
template <typename T>
class Dynarray {
  std::size_t m_length;
  T *m_storage;

public:
  Dynarray();
  Dynarray(const Dynarray<T> &);
  Dynarray(Dynarray<T> &&) noexcept;
  // other members ...
};
```

Define a class template

A class template is a **guideline** for the compiler: When a type `Dynarray<T>` is used for a certain type `T`, the compiler will **instantiate** that class type according to the class template.

For different types `T` and `U`, `Dynarray<T>` and `Dynarray<U>` **are different types**.

```cpp
template <typename T>
class X {
  int x = 42; // private

public:
  void foo(X<double> xd) {
    x = xd.x; // access a private member of X<double>
    // This is valid only when T is double.
  }
};
```

Define a class template

Inside the class template, the template parameters can be omitted when we are referring to the self type:

```cpp
template <typename T>
class Dynarray {
  std::size_t m_length;
  T *m_storage;

public:
  Dynarray();
  Dynarray(const Dynarray &); // Parameter type is const Dynarray<T> &
  Dynarray(Dynarray &&) noexcept; // Parameter type is Dynarray<T> &&
  // other members ...
};
```

Member functions of a class template

If we want to define a member function outside the template class, a template declaration is also needed.

```cpp
class Dynarray {
public:
  int &at(std::size_t n);
};

int &Dynarray::at(std::size_t n) {
  return m_storage[n];
}
```

```cpp
template <typename T>
class Dynarray {
public:
  int &at(std::size_t n);
};
template <typename T>
int &Dynarray<T>::at(std::size_t n) {
  return m_storage[n];
}
```

Member functions of a class template

A member function will not be instantiated if it is not used!

```cpp
template <typename T>
struct X {
  T x;
  void foo() { x = 42; }
  void bar() { x = "hello"; }
};
```

```cpp
X<int> xi; // OK
xi.foo();  // OK
X<std::string> xs; // OK
xs.bar();          // OK
```

No compile-error occurs: `X<int>::bar()` and `X<std::string>::foo()` are not instantiated because they are not called.

Member functions of a class template

A member function itself can also be a template:

```cpp
template <typename T>
class Dynarray {
public:
  template <typename Iterator>
  Dynarray(Iterator begin, Iterator end)
      : m_length(std::distance(begin, end)), m_storage(new T[m_length]) {
    std::copy(begin, end, m_storage);
  }
};
std::vector<std::string> vs = someValues();
Dynarray<std::string> ds(vs.begin(), vs.end()); // Values are copied from vs.
```

Member functions of a class template

A member function itself can also be a template. To define it outside the class, **two** template declarations are needed:

```cpp
// This cannot be written as `template <typename T, typename Iterator>`
template <typename T>
template <typename Iterator>
Dynarray<T>::Dynarray(Iterator begin, Iterator end)
    : m_length(std::distance(begin, end)), m_storage(new T[m_length]) {
  std::copy(begin, end, m_storage);
}
```

Alias templates, variable templates and non-type template parameters

Alias templates

The `using` declaration can also declare **alias templates**:

```cpp
template <typename T>
using pii = std::pair<T, T>;
pii<int> p1(2, 3); // std::pair<int, int>
```

Variable templates (since C++14)

Variables can also be templates. Typical example: the C++20 `<numbers>` library:

```cpp
namespace std::numbers {
  template <typename T>
  inline constexpr T pi_v = /* unspecified */;
}
auto pi_d = std::numbers::pi_v<double>; // The `double` version of π
auto pi_f = std::numbers::pi_v<float>;  // The `float` version of π
```

Non-type template parameters

Apart from types, a template parameter can also be an integer, an lvalue reference, a pointer, ...

Since C++20, floating-point types and *literal class types* with certain properties are also allowed.

```cpp
template <typename T, std::size_t N>
class array {
  T data[N];
  // ...
};
array<int, 10> a;
```

Non-type template parameters

Define a function that accepts an array of **any length**, **any type**:

```cpp
template <typename T, std::size_t N>
void print_array(T (&a)[N]) {
  for (const auto &x : a)
    std::cout << x << ' ';
  std::cout << std::endl;
}
int a[10]; double b[20];
print_array(a); // T=int, N=10
print_array(b); // T=double, N=20
```

Summary

Idea of templates: **Generic programming**.

- Write code that can deal with different types. e.g. Containers parameterized on element types.

- The compiler generates (*instantiates*) the actual code in terms of the template code.

Function template:

- Template argument deduction:
  - Pass-by-value: Reference and top-level `const` -qualification are ignored. Decay happens.
  - Pass-by-reference: Reference collapsing happens.
  - Same as the deduction rule of `auto` .

Summary

Perfect forwarding:

- Forwarding reference: `auto &&`, or `T &&` where `T` is a template type parameter.
  - Deduced to lvalue-reference if the argument is an lvalue, and rvalue-reference if the argument is an rvalue.
  - `const` qualifications are preserved.
- `std::forward<T>(x)` : Casts `x` to `T &&`.

Variadic template and pack expansion: Used to accept an unknown number of arguments of unknown types, and perform some operations on them.

- "unknown" is for the author of the templates. The compiler generates the required versions of them.

Summary

Class templates:

- The class is parameterized on some template parameter. Different template arguments yield different classes.
- The member functions are instantiated only when they are called.

Other things:

- Alias templates
- Variable templates
- Non-type template parameters