CS100 Lecture 25

Templates I

Contents

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

Function templates

Motivation: function templates

```
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;
}
```

We need to write and maintain *multiple copies* of the same code in the functions.

Can we write the code *one time* to define `compare` functions for different data types?

Motivation: function templates

```
template <typename T> // Use the data type T as a parameter
int compare(const T &a, const T &b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

- Specify a set of functions that use the same code but handle different data types.
- A template is a **guideline** for the compiler to generate a specific function:
  - `compare(42, 40)`: `T` is deduced to be `int`, and the compiler generates a function for `int`.
  - `compare(s1, s2)`: `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

```
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

```
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

```
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>`.

Templates should try to minimize the number of requirements placed on the type arguments.

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.

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

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

Template argument deduction

Suppose we have the following function template

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

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

- `A` is never considered to be a reference type. 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

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

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

- Passing-by-value ignores top-level `const` qualification. For example,

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

Template argument deduction

Suppose we have the following function template

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

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

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

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

Template argument deduction

Suppose we have the following function template

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

and a call `fun(a)`, where the type of the argument `a` 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 x = a;`!

Template argument deduction

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

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

`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**:

```
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`.

```
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;
}
```

Template argument deduction

Define a function template to deal with sequences for different element types.

```
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>
```

Template argument deduction

Write a function `map(func, vec)`, where `func` is any unary function and `vec` is any vector. It returns a vector by replacing every element `x` in `vec` with `func(x)`.

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

Usage:

```
int f(int x) { return x * 2; }
std::vector v{1, 2, 3, 4};
auto v2 = map(f, 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 &&`.
- If `a` is an lvalue expression of type `E`, **`T` will be deduced to `E &`**.
  - The type of `x` is `E & &&`??? (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 `&&`.

```
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:

```
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 &&`.
- 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** that passes the arguments of a function to another function, so that:

- The value category of every argument is not changed.
- The `const` qualification on every argument is not changed.

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

- `make_unique<Type>(arg)` forwards the argument `arg` to the constructor of `Type`.
- `arg`'s value category must not be changed: `make_unique<std::string>(s1 + s2)` must move `s1 + s2` (call `std::string`'s move constructor), instead of copying it.
- `arg`'s `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

A forwarding reference can be used for **perfect forwarding** that passes the arguments of a function to another function, so that:

- The value category of every argument is not changed.
- The `const` qualification on every argument is not changed.

The following is not perfect forwarding:

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

- If the argument of `make_unique` is an rvalue without `const` qualification, what `T`'s constructor receives is an lvalue (since it has a name), with `const` added.

Perfect forwarding

A forwarding reference can be used for **perfect forwarding** that passes the arguments of a function to another function, so that:

- The value category of every argument is not changed.
- The `const` qualification on every argument is not changed.

We need to do this:

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

```
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:

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

`Types` : a **template parameter pack** representing zero or more template parameters.

`params` : a **function parameter pack** representing zero or more function parameters.

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

```
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
```

Parameter pack

Parameter pack declaration: `...` is *on the left of* pack names in parameter declarations.

The types of the function parameters in the pack can contain `const`, `&` and `&&`:

```
// 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

Parameter pack expansion: `...` is *on the right of* pack names in expressions.

An expression that contains one or more parameter pack names followed by `...` is **expanded** into a sequence of comma-separated expressions:

- In the expressions, the name of the parameter pack in the original expression is replaced by each of the elements from the pack, in order.

```
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:

```
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.

Class templates

Define a class template

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

```
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

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

```
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 ...
};
```

Define a class template

When a type `Dynarray<T>` is used for a certain type `T`, the compiler will **instantiate** that class type based on the class template.

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

```
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.
  }
};
```

Member functions of a class template

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

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

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

```
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!

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

```
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:

```
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:

```
// 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

`using` can also be used to declare **alias templates**:

```
template <typename T>
using pii = std::pair<T, T>;

pii<int> p1(2, 3); // std::pair<int, int>
```

Variable templates (since C++14)

An example:

```
template <typename T>
T pi = T(3.141592653589793);

auto pi_d = pi<double>; // The `double` version of π
auto pi_f = pi<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, ...

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

array<int, 10> a;
```

Summary

Idea of templates: **Generic programming**.

- Write code that can deal with different data types.
- The compiler generates (***instantiates***) the actual code based on the template code.

Function templates:

- Template argument deduction:
  - Pass-by-value: Top-level `const` qualification is 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 an lvalue reference if the argument is an lvalue, and an rvalue reference if the argument is an rvalue.
  - `const` qualification is not changed.
- `std::forward<T>(x)`.

Variadic templates: Accept an unknown number of arguments of unknown types.

Summary

Class templates:

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

Other things:

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