CS100 Lecture 23

Templates

Contents

- Function templates
- Class templates
- Template specialization
- `type_traits`

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

```
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.

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

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

- The type of **the expression** `a` is never a reference: Whenever a reference is used, we are actually using the object it is bound to.
- 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

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

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

- If `A` is `const`-qualified, the top-level `const` is ignored:

  ```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

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

and a call `fun(a)`, where the type of **the expression** **`a`** 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

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

and a call `fun(a)`, where the type of **the expression** `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 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)`.

```
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:

```
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 <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** 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 <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 <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:

- `makeUnique<Type>(arg)` forwards the argument `arg` to the constructor of `Type`.
- The value category must not be changed: `makeUnique<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: `makeUnique<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.

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

Perfect forwarding

- The value category must not be changed: `makeUnique<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:

```
template <typename T, typename U>
auto makeUnique(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 makeUnique(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 makeUnique(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**:

```
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);
}
```

Curiously Recurring Template Pattern (CRTP)

Example 1: We have seen this `Uncopyable` in Homework 7:

```cpp
class Uncopyable {
  Uncopyable(const Uncopyable &) = delete;
  Uncopyable &operator=(const Uncopyable &) = delete;

 public:
  Uncopyable() = default;
};
```

A class can be made uncopyable by inheriting `Uncopyable`.

But if both `A` and `B` should be uncopyable, can we make each of them inherit a unique base class?

Curiously Recurring Template Pattern (CRTP)

```cpp
template <typename Derived>
class Uncopyable {
  Uncopyable(const Uncopyable &) = delete;
  Uncopyable &operator=(const Uncopyable &) = delete;

 public:
  Uncopyable() = default;
};

class A : public Uncopyable<A> {};
class B : public Uncopyable<B> {};
```

Every class `C` inherits a unique base class `Uncopyable<C>`!

Curiously Recurring Template Pattern (CRTP)

Example 2: With the prefix incrementation operator `operator++` defined, the postfix version is always defined as follows:

```
auto operator++(int) {
  auto tmp = *this;
  ++*this;
  return tmp;
}
```

How can we avoid repeating ourselves?

Curiously Recurring Template Pattern (CRTP)

```cpp
template <typename Derived>
class Incrementable {
 public:
  auto operator++(int) {
    // Since we are sure that the dynamic type of `*this` is `Derived`,
    // we can use static_cast here.
    auto real_this = static_cast<Derived *>(this);
    auto tmp = *real_this; ++*real_this; return tmp;
  }
};
class A : public Incrementable<A> {
 public:
  A &operator++() { /* ... */ }
  // The operator++(int) is inherited from Incrementable<A>.
};
```

CppCon2022: How C++23 Changes the Way We Write Code Jump to 23:55 to see how C++23 can simplify CRTP.

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>
```

Template variables (since C++17)

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

```
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.

```
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**:

```
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
```

Template specialization

Specialization for a function template

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

What happens for C-style strings?

```cpp
const char *a = "hello", b[] = "world";
auto x = compare(a, b);
```

This is comparing two pointers, instead of comparing the strings!

Specialization for a function template

```cpp
template <typename T>
int compare(T const &lhs, T const &rhs) {
  if (lhs < rhs) return -1;
  else if (rhs < lhs) return 1;
  else return 0;
}
template <> // specialized version for T = const char *
int compare<const char *>(const char *const &lhs, const char *const &rhs) {
  return std::strcmp(lhs, rhs);
}
```

Write a specialized version of that function with the template parameters taking a certain group of values.

The type `const T &` with `T = const char *` is `const char *const &`: A reference bound to a `const` pointer which points to `const char`.

Specialization for a function template

It is also allowed to omit `<const char *>` following the name:

```cpp
template <typename T>
int compare(T const &lhs, T const &rhs) {
  if (lhs < rhs) return -1;
  else if (rhs < lhs) return 1;
  else return 0;
}
template <>
int compare(const char *const &lhs, const char *const &rhs) {
  return std::strcmp(lhs, rhs);
}
```

Specialization for a function template

Is this a specialization?

```cpp
template <typename T>
int compare(T const &lhs, T const &rhs);
template <typename T>
int compare(const std::vector<T> &lhs, const std::vector<T> &rhs);
```

No! These functions constitute **overloading** (allowed).

Specialization for a function template

Is this a specialization?

```cpp
template <typename T>
int compare(T const &lhs, T const &rhs);
template <typename T>
int compare<std::vector<T>>(const std::vector<T> &lhs,
                            const std::vector<T> &rhs);
```

- Since we write `int compare<std::vector<T>>(...)`, this **is** a specialization.
- However, such specialization is a **partial specialization**: The specialized function is still a function template.
  - **Partial specialization for function templates is not allowed.**

Specialization for a class template

It is allowed to write a specialization for class templates.

```cpp
template <typename T>
struct Dynarray { /* ... */ };
template <> // specialization for T = bool
struct Dynarray<bool> { /* ... */ };
```

Partial specialization is also allowed:

```cpp
template <typename T, typename Alloc>
class vector { /* ... */ };
// specialization for T = bool, while Alloc remains a template parameter.
template <typename Alloc>
class vector<bool, Alloc> { /* ... */ };
```

`type_traits`

Know whether two types are the same?

```cpp
template <typename T, typename U>
struct is_same {
  static const bool result = false;
};
template <typename T> // specialization for U = T
struct is_same<T, T> {
  static const bool result = true;
};
```

- `is_same<int, double>::result` is false.
- `is_same<int, int>::result` is true.
- Are `int` and `signed` the same type? Let `is_same` tell you!

Know whether a type is a pointer?

```
template <typename T>
struct is_pointer {
  static const bool result = false;
};
template <typename T>
struct is_pointer<T *> { // specialization for <T *> for some T.
  static const bool result = true;
};
```

- `is_pointer<int *>::result` is true.
- `is_pointer<int>::result` is false.
- Is `std::vector<int>::iterator` actually a pointer? Is `int[10]` the same thing as `int *`? Consult these "functions"!

`<type_traits>`

`std::is_same`, `std::is_pointer`, as well as a whole bunch of other "functions": Go to this standard library.

This is part of the **metaprogramming library**.

Template metaprogramming

Template metaprogramming is a very special and powerful technique that makes use of the compile-time computation of C++ compilers. (It is Turing-complete and pure functional programming.)

Learn a little bit more in recitations.

In modern C++, there are many more things that facilitate compile-time computations: `constexpr`, `consteval`, `constinit`, `concept`, `requires`, ...