CS100 Lecture 26

Templates II

Contents

Template specialization

Template specialization

Templates are for **generic programming**.

- Write code once and use it for different data types.
- Define generic funcions or classes, parameterized by data types, so that they can work for different data types.

What if a particular data type needs some special treatment?

Specialization for a function template

```
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?

```
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 template with the template parameters taking certain values.

The type `T const &` 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! They constitute **overloading**.

For specialization of a function template, **only full specialization is allowed**.

- The specialized version has no template parameters, and is no longer a template.

Specialization for a class template

It is allowed to write a specialization for a class template.

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

Partial specialization is 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> { /* ... */ };
```

- The specialized version still has template parameters, and is a template.

Variadic templates: an example

A `print` function

```cpp
template <typename First, typename... Rest>
void print(std::ostream &os, const First &first, const Rest &...rest) {
  os << first << std::endl;
  if (/* `rest` is not empty */) // How to test this?
    print(os, rest...); // It becomes `print(os, r0, r1)`,
                        // if `rest` = {`r0`, `r1`}.
}

std::string s = "hello"; double d = 3.14; int i = 42;
print(std::cout, s, d, i);
```

First, understand the different meanings of `...`.

- `typename... Rest` indicates that `Rest` is a template parameter pack.
- `const Rest &...rest` indicates that `rest` is a function parameter pack.
- `rest...` in `print(os, rest...)` is **pack expansion**.

Compile-time instantiation

```cpp
template <typename First, typename... Rest>
void print(std::ostream &os, const First &first, const Rest &...rest) {
  os << first << std::endl;
  if (/* `rest` is not empty */) // How to test this?
    print(os, rest...);
}

std::string s = "hello"; double d = 3.14; int i = 42;
print(std::cout, s, d, i);
```

`print(std::cout, s, d, i)` leads to the generation of the following functions:

```cpp
void print(std::ostream &os, const std::string &first,
           const double &rest0, const int &rest1); // `rest` = {`rest0`, `rest1`}
void print(std::ostream &os, const double &first,
           const int &rest0); // `rest` = {`rest0`}
void print(std::ostream &os, const int &first); // `rest` = {}
```

`sizeof...(pack)`

How many arguments are there in a pack? Use the `sizeof...` operator, which is evaluated at compile-time.

```cpp
template <typename First, typename... Rest>
void print(std::ostream &os, const First &first, const Rest &...rest) {
  os << first << std::endl;
  if (sizeof...(Rest) > 0)
    print(os, rest...);
}

std::string s = "hello"; double d = 3.14; int i = 42;
print(std::cout, s, d, i);
```

Looks good ... But a compile-error?

`sizeof...(pack)`

Looks good ... But a compile-error?

```
b.cpp: In instantiation of ‘void print(std::ostream&, const First&, const Rest& ...)
[with First = int; Rest = {}; std::ostream = std::basic_ostream<char>]’:
b.cpp:11:8:   required from here
b.cpp:7:10: error: no matching function for call to ‘print(std::ostream&)’
    7 |     print(os, rest...);
```

It says that when `First = int, Rest = {}`, we are trying to call `print(os)` (with nothing to print).

Note: `first` is not a parameter pack, so `print` must have at least ***two*** arguments.

Compile-time `if`

Let's see what the function looks like when `First = int, Rest = {}`:

```cpp
void print(std::ostream &os, const int &first) {
  os << first << std::endl;
  if (false)   // `sizeof... (Rest)` = 0
    print(os); // Ooops! `print` needs at least two arguments!
}
```

The problem is that `if` is a **run-time** control flow statement! The statements must ***compile*** even if the condition is 100% `false`!

We need a **compile-time** `if`.

Compile-time `if`: `if constexpr`

```
if constexpr (condition)
  statement1
```

```
if constexpr (condition)
  statement1
else
  statement2
```

`condition` must be a compile-time constant.

Only when `condition` is `true` will `statement1` be compiled.

Only when `condition` is `false` will `statement2` be compiled.

Use `if constexpr`

```cpp
template <typename First, typename... Rest>
void print(std::ostream &os, const First &first, const Rest &...rest) {
  os << first << std::endl;
  if constexpr (sizeof...(Rest) > 0)
    print(os, rest...);
}
```

Solution without `if constexpr`: overloading.

```cpp
template <typename T>
void print(std::ostream &os, const T &x) { os << x << std::endl; }
template <typename First, typename... Rest>
void print(std::ostream &os, const First &first, const Rest &...rest) {
  print(os, first); // Use the first template.
  print(os, rest...); // Use the first template when `sizeof... (Rest)` = 1.
}
```

Curiously Recurring Template Pattern (CRTP)

Example 1: Uncopyable

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

public:
  Uncopyable() = default;
};

class ComplexDevice : public Uncopyable { /* ... */ };
```

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

Example 1: Uncopyable

But if two classes inherit from `Uncopyable` publicly, odd things may happen ...

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

public:
  Uncopyable() = default;
};

class Airplane : public Uncopyable {}; // Copying an airplane is too costly.
class MonaLisa : public Uncopyable {}; // An artwork is not copyable.

Uncopyable *foo1 = new Airplane();
Uncopyable *foo2 = new MonaLisa();
```

Ooops ... A pointer of type `Uncopyable *` can point to two things that are **totally unrelated to each other**!

Example 1: Uncopyable

```cpp
template <typename Sub> // Use the subclass as the template parameter
class Uncopyable {
  Uncopyable(const Uncopyable &) = delete;
  Uncopyable &operator=(const Uncopyable &) = delete;

public:
  Uncopyable() = default;
};

class Airplane : public Uncopyable<Airplane> {};
class MonaLisa : public Uncopyable<MonaLisa> {};
```

Now `Airplane` and `MonaLisa` are uncopyable, but inherit from **different base classes**: `Uncopyable<Airplane>` and `Uncopyable<MonaLisa>` are different types.

Example 2: Incrementable

```cpp
template <typename T>
class Iterator {
  T *cur;

public:
  auto &operator++() {
    ++cur;
    return *this;
  }
  auto operator++(int) {
    auto tmp = *this;
    ++*this;
    return tmp;
  }
};
```

```cpp
class Rational {
  int num;
  unsigned denom;

public:
  auto &operator++() {
    num += denom;
    return *this;
  }
  auto operator++(int) {
    auto tmp = *this;
    ++*this;
    return tmp;
  }
};
```

```cpp
class AtomicCounter {
  int cnt;
  std::mutex m;

public:
  auto &operator++() {
    std::lock_guard l(m);
    ++cnt;
    return *this;
  }
  auto operator++(int) {
    auto tmp = *this;
    ++*this;
    return tmp;
  }
};
```

Example 2: Incrementable

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 the repetition?

Example 2: Incrementable

```cpp
template <typename Sub> // Use the subclass as the template parameter
class Incrementable {
public:
  auto operator++(int) {
    // Use `static_cast` here to perform the downcasting.
    auto real_this = static_cast<Sub *>(this); // `real_this` points to a `Sub`.
    auto tmp = *real_this;
    ++*real_this; // Use the `operator++()` of `Sub`.
    return tmp;
  }
};

class A : public Incrementable<A> {
public:
  A &operator++() { /* ... */ }
  // The `operator++(int)` is inherited from `Incrementable<A>`,
  // which should use the `operator++()` of `A`.
};
```

Curiously Recurring Template Pattern

Idea: Use the subclass as the template argument of the base class that is generated from a class template.

By writing the common parts of classes `X`, `Y`, `Z` in a base class `Base`,

- we can avoid repetition.
- However, `X`, `Y` and `Z` have a common base (which may lead to weird things), and `Base` does not know ***who*** is inheriting from it.

By letting `X`, `Y`, `Z`, ... inherit from `Base<X>`, `Base<Y>`, `Base<Z>`, respectively,

- each class inherits from a unique base class, and
- the base class knows what the subclass is, so a safe downcasting can be performed.

Template metaprogramming

Template metaprogramming

Metaprogramming (元编程)： Write code that generates code.

Template metaprogramming: Write templates that are used by the compiler to generate code.

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.

Know whether a type is a pointer?

```
template <typename T>
struct is_pointer {
  static const bool result = false;
};

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

- `is_pointer<int *>::result` is true.
- `is_pointer<int>::result` is false.

`<type_traits>`

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

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

Compute $n!$ in compile-time?

```cpp
template <unsigned N>
struct Factorial {
                                  // N! = N x (N-1)!
  static const unsigned long long value = N * Factorial<N - 1>::value;
                                  // `Factorial<N - 1>`: Recursive generation of
                                  // a class for `N - 1` until `N - 1` = 0.
};

template <>
struct Factorial<0u> { // Specialization for N = 0
  static const unsigned long long value = 1; // 0! = 1
};

int main() {
  int a[Factorial<5>::value]; // 120, which is a compile-time constant.
}
```

Seven basic quantities in physics

When performing computations in physics, the correctness in **dimensions** is important.

```
double mass = getMass();
double acceleration = getAcc();
double force = mass + acceleration; // Ooops! A mistake here, but it compiles.
                                    // Dimension of `mass`: M
                                    // Dimension of `acceleration`: L / (T x T)
```

Can we avoid such mistakes at **compile-time**? That is, to make mistakes in dimensions a **compile error**.

Seven basic quantities in physics

```cpp
// Each of the seven basic quantities corresponds to a template parameter.
template <int mass, int length, int time, int charge,
          int temperature, int intensity, int amount_of_substance>
struct quantity { /* ... */ };

// All other physical quantities can be constructed from the multiplication of
// positive and negative powers of the basic quantities.
using mass = quantity<1, 0, 0, 0, 0, 0, 0>;
using force = quantity<1, 1, -2, 0, 0, 0, 0>; // (mass x length) / (time x time)
using pressure = quantity<1, -1, -2, 0, 0, 0, 0>;
using acceleration = quantity<0, 1, -2, 0, 0, 0, 0, 0>;

mass m = getMass();
acceleration a = getAcc();
force f = m + a; // Error! No match `operator+` for 'mass' and 'acceleration'!
force f = m * a; // Correct.
```

If the arithmetic operators for `quantity`s are defined correctly, we can avoid dimension mistakes at ***compile-time***!

Template metaprogramming

Template metaprogramming is a very special and powerful technique that makes use of the compile-time computation of C++ compilers.

- Some computations are performed at compile-time to save time at run-time.

Learn a little bit more in recitations.

Summary

Template specialization

- Specify template arguments to define special behaviors for them.
- Full specialization: The specialization has no template parameters.
- Partial specialization: The specialization still has template parameters.
- Function templates cannot have partial specializations.

Summary

Variadic template example: A `print` function.

- `pack...`: pack expansion.
- `sizeof...(pack)` returns the number of arguments in a parameter pack. It is compile-time evaluated.
- `if constexpr`: compile-time `if`, which ***compiles*** statements based on a conditional expression that can be evaluated at compile-time.

Summary

Curiously Recurring Template Pattern (CRTP)

- Let `X` inherit from `Base<X>`.
- Each class inherits from a unique base class.
- The base class knows what the subclass is, so a safe downcasting can be performed.

Template metaprogramming

- Shift work from run-time to compile-time, thus enabling higher run-time performance and earlier error detection.