CS100 Lecture 26

Templates II

Contents

Template specialization

Templates are for **generic programming**, but some things need special treatments.

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

```
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:

```
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?

```
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?

```
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> { /* ... */ };
```

Variadic templates: examples

Curiously Recurring Template Pattern (CRTP)

Example 1: Uncopyable

We have seen this `Uncopyable` in Homework 6:

```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 ...

```cpp
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 `Uncopyable*` can point to two things that are **totally unrelated to each other**!

Example 1: Uncopyable

```cpp
template <typename Derived>
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` inherit from **different bases**: `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 repeating ourselves?

Example 2: Incrementable

```
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 to perform the downcasting.
    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>.
};
```

Curiously Recurring Template Pattern

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

- we can avoid repeating ourselves.
- 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 derived class is, so a safe downcast can be performed.

CRTP idiom adopted in the standard library: `std::enable_shared_from_this`.

Introduction to template metaprogramming

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**.

Compute $n!$ in compile-time?

```cpp
template <unsigned N>
struct Factorial {
  static const unsigned long long value = N * Factorial<N - 1>::value;
};
template <>
struct Factorial<0u> {
  static const unsigned long long value = 1;
};

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

Check whether an integer is a prime in compile-time?

```
template <unsigned N, unsigned Div> struct PrimeTest {
  static const bool result = (N % Div != 0) && PrimeTest<N, Div + 1>::result;
};
template <unsigned N> struct PrimeTest<N, N> { // end
  static const bool result = true;
};
template <unsigned N> struct IsPrime {
  static const bool result = PrimeTest<N, 2>::result;
};
template <> struct IsPrime<1u> {
  static const bool result = false;
};

static_assert(IsPrime<197>::result); // 197 is a prime
static_assert(!IsPrime<42>::result); // 42 is not
```

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!
```

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

Seven basic quantities in physics

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 { /* ... */ };

using mass = quantity<1, 0, 0, 0, 0, 0, 0>;
using force = quantity<1, 1, -2, 0, 0, 0, 0>;
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 = getAccc();
force f = m + a; // Error! No match operator+ for 'mass' and 'acceleration'!
force f = m * a; // Correct.
```

If the arithmetic operations of different `quantity`s are defined correctly, we can avoid dimension mistakes in ***compile-time***!

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`, ...