CS100 Recitation 14

GKxx

May 30, 2022

Contents

Contents

Define a Function Template

```
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;
}
```

Define a Function Template

```
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, 53)` 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 on `std::strings`, `T` is deduced to be `std::string`.
  - **Instantiation** of a template.

Type Parameter

```
template <typename T, class U>
void fun(const T &, const U &);
```

- `typename` and `class` here are **equivalent**: `U` doesn't have to be a class type.

Type Parameter

```
template <typename T, class U>
void fun(const T &, const U &);
```

- `typename` and `class` here are **equivalent**: `U` doesn't have to be a class type.

```
fun(42, 3.14);  // T=int, U=double
std::string s = "Hello";
fun(s, 42);     // T=std::string, U=int
fun<int, int>(42, 3.14); // T=U=int, 3.14 converts to 3
```

Requirements on Types

Is this good?

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

Requirements on Types

Is this good?

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

- What if `T` is uncopyable, or the copying is too costly?

Requirements on Types

Is this good?

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

- What if `T` is uncopyable, or the copying is too costly?
- To make a class fit here, you need to define both `operator<` and `operator>`!

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

Deduction Fails...

```
template <typename T, typename U>
void fun(const U &x) {}
fun(42); // Error: What is T?
```

Deduction Fails...

```
template <typename T, typename U>
void fun(const U &x) {}
fun(42); // Error: What is T?
```

You need to write it explicitly:

```
fun<double>(42); // Correct. T=double, U=int.
```

Contents

Define a Class Template

- HW5-1: `IntArray` $\Longrightarrow$ `Array<T>`.
- Define member functions inside the class, and outside the class.
- Template arguments could be left out inside the class.

```
// inside the class
Array<T> &operator=
    (const Array<T> &other) {
  // ...
}
```

```
// leave out <T>
Array &operator=
    (const Array &other) {
  // ...
}
```

Class Templates

Template is a guideline for the compiler:

- When we use `Array<double>`, `T` becomes `double` and the compiler generates the code for `class Array<double>`.
- Classes instantiated from the same class template with different template arguments **have nothing to do with each other**. They are independnet classes.

```
template <typename T>
class Array {
  void fun(Array<double> &ad) {
    // Error whenever T is not double!
    auto x = ad.m_size; // m_size is private
  }
};
```

Templated Member Functions

```
template <typename T>
class Array {
 public:
  Array(std::size_t n) : m_size(n), m_data(new T[n]{}) {}
  template <typename Iterator>
  Array(Iterator begin, Iterator end)
      : Array(std::distance(begin, end)) {
    std::copy(begin, end, m_data);
  }
};
```

Templated Member Functions

Define it outside the class:

```
template <typename T>
template <typename Iterator>
Array<T>::Array(Iterator begin, Iterator end)
    : Array(std::distance(begin, end)) {
  std::copy(begin, end, m_data);
}
```

- It should **NOT** be declared as

```
template <typename T, typename Iterator>
```

Templated Member Functions of Non-template Classes

```
class Widget {
 public:
  template <typename Container>
  void add_this_to(Container &c) const {
    c.emplace_back(this);
  }
};
```

Templated Member Functions of Non-template Classes

Define it outside the class:

```
class Widget {
 public:
  template <typename Container>
  void add_this_to(Container &) const;
};
template <typename Container>
void Widget::add_this_to(Container &c) const {
  c.emplace_back(this);
}
```

Instantiation

- For a class template, only when the class is used will the code for the class be generated.
- For a function template, only when the function is called will the code for the function be generated.

Instantiation 实例化

- For a class template, only when the class is used will the code for the class be generated.
- For a function template, only when the function is called will the code for the function be generated.
- For a class template, a member function is compiled only when the function is called.

Instantiation

```
template <typename T>
class Array {
 public:
  void add_five() {
    for (std::size_t i = 0; i != m_size; ++i)
      m_data[i] += 5;
  }
};
Array<int> ai(10);
ai.add_five();                  // OK
Array<std::string> as(10);
// Still OK. add_five is not compiled here.
```

Instantiation

```
template <typename T>
class Array {
 public:
  void add_five() {
    for (std::size_t i = 0; i != m_size; ++i)
      m_data[i] += 5;
  }
};
Array<int> ai(10);
ai.add_five();                  // OK
Array<std::string> as(10);
// Still OK. add_five is not compiled here.

as.add_five(); // Error.
```

Templated Type Aliases

```
template <typename T>
using pvec = std::shared_ptr<std::vector<T>>;

pvec<int> pvi; // pvi is a shared_ptr<vector<int>>
pvec<double> pvd; // pvd is a shared_ptr<vector<double>>
```

This is what `typedef` unable to do.

Contents

Example

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

Example

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

const char *cs1 = "hello", *cs2 = "world";
auto result = compare(cs1, cs2); // Oops!
```

- T becomes `const char *`.
- Comparing two pointers instead of two strings!

Specialization of Function Templates

```
template <typename T>
int compare(const T &a, const T &b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
template <>
int compare(const char *const &p1,
            const char *const &p2) {
  return std::strcmp(p1, p2);
}
```

- Parameter types should match the corresponding types in the previously declared template.
  - This is a specialization for `T = const char *`.

Specialization and Function Matching

- Non-template > Template-specialization > Template.

```
template <>
int compare(const char *const &p1,
            const char *const &p2) {
  std::cout << "template specialization" << std::endl;
  return std::strcmp(p1, p2);
}
int compare(const char *const &p1,
            const char *const &p2) {
  std::cout << "non-template" << std::endl;
  return std::strcmp(p1, p2);
}
const char *cs1 = "hello", *cs2 = "world";
auto result = compare(cs1, cs2); // "non-template"
```

Specialization of Class Templates

If we want to make a specialized version for `Array<bool>`, just like `std::vector`...

```
template <typename T>
class Array { /* ... */ };

template <>
class Array<bool> {
  // Show your talented design for bools here.
};
```

Partial Specialization

Recall that std::vector has another template parameter: the allocator.

```
template <typename T, typename Alloc>
class vector { /* ... */ };

template <typename Alloc>
class vector<bool, Alloc> { /* ... */ };
```

Partial Specialization

Recall that `std::vector` has another template parameter: the allocator.

```
template <typename T, typename Alloc>
class vector { /* ... */ };

template <typename Alloc>
class vector<bool, Alloc> { /* ... */ };
```

- `vector<bool, Alloc>` is a specialization for `T=bool`, but it is still a template.
  - `Alloc` is the remaining template parameter.

Partial Specialization

`std::unique_ptr` has a partial specialization for arrays:

```
template <typename T>
class unique_ptr { /* ... */ };

template <typename T>
class unique_ptr<T[]> { /* ... */ };
```

- `std::unique_ptr<T[]>` does not provide `operator->`, but it provides `operator[]`.
- Used for managing dynamic arrays of unique ownership.

Partial Specialization

std::unique_ptr has a partial specialization for arrays:

```
template <typename T>
class unique_ptr { /* ... */ };

template <typename T>
class unique_ptr<T[]> { /* ... */ };
```

- std::unique_ptr<T[]> does not provide operator->, but it provides operator[].
- Used for managing dynamic arrays of unique ownership.
- **Use STL containers** instead of std::unique_ptr<T[]>, unless for special purposes!

Partial Specialization

Partial specialization is **NOT allowed** for function templates:

```
template <typename T>
void swap(T &a, T &b) {
  T tmp(a);
  a = b;
  b = tmp;
}
template <typename T>
void swap<Array<T>>(Array<T> &a, Array<T> &b) { // Error!
  a.swap(b);
}
```

Contents

Integer Parameters

A new way of passing arrays:

```
template <unsigned N>
void print_array(const int (&arr)[N]) {
  for (unsigned i = 0; i != N; ++i)
    std::cout << arr[i] << " ";
}
```

Integer Parameters

A new way of passing arrays:

```
template <unsigned N>
void print_array(const int (&arr)[N]) {
  for (unsigned i = 0; i != N; ++i)
    std::cout << arr[i] << " ";
}
```

You may also use range-for:

```
template <unsigned N>
void print_array(const int (&arr)[N]) {
  for (auto x : arr)
    std::cout << x << " ";
}
```

`arr` is recognized as a **reference to array** here.

Integer Parameters

Recall the old way:

```
void print_array(const int *arr, unsigned N) {
  for (unsigned i = 0; i != N; ++i)
    std::cout << arr[i] << " ";
}
```

`arr` is simply recognized as a pointer here.

Integer Parameters

Generalization for value type:

```
template <typename T, unsigned N>
void print_array(const T (&arr)[N]) {
  for (const auto &x : arr)
    std::cout << x << " ";
}
```

- Be careful with unknown types! (use `const auto &` to avoid copying)

Integer Parameters

```
template <typename T, bool is_const>
class Slist_iterator {
  // When is_const is true, it is a const_iterator.
  // Otherwise, it is an iterator.
};

template <typename T>
class Slist {
 public:
  using iterator = Slist_iterator<T, false>;
  using const_iterator = Slist_iterator<T, true>;
  // ...
};
```

Constant Expression

- Anything used as a template argument must be **known at compile-time**. (Why?)

Constant Expression

- Anything used as a template argument must be **known at compile-time**. (Why?)

```
int n;
std::cin >> n;
std::array<int, n> arr; // Error! n is runtime-determined.
```

Constant Expression

- Anything used as a template argument must be **known at compile-time**. (Why?)

```
int n;
std::cin >> n;
std::array<int, n> arr; // Error! n is runtime-determined.

int maxn = 1000;
std::array<int, maxn> arr; // Error! maxn is not a
    constant expression
```

Contents

Constant Expression

```
// const variable initialized with constant expression is
    constant expression.
const int maxn = 1000;
std::array<int, maxn> arr; // OK.
```

Constant Expression

```
// const variable initialized with constant expression is
    constant expression.
const int maxn = 1000;
std::array<int, maxn> arr; // OK.
```

The `constexpr` keyword:

```
constexpr int maxn = 1000; // Undoubtedly constant
    expression.
std::array<int, maxn> arr;
```

Constant Expression

- `constexpr` explicitly declares that a variable must have a compile-time known value.
- `constexpr` tells the compiler: You can work out its value!

Constant Expression

- constexpr explicitly declares that a variable must have a compile-time known value.
- constexpr tells the compiler: You can work out its value!
- But if the compiler cannot, it reports an error.

```
int n; std::cin >> n;
constexpr int m = n; // Error.
```

Contents

constexpr Functions

Since C++11, functions can be defined as `constexpr`, but with some constraints.

```
constexpr int add(int a, int b) {
  return a + b; // OK.
}
int x = add(42, 35); // run the function at compile-time!
int a, b; std::cin >> a >> b;
int y = add(a, b); // run the function at runtime.
```

constexpr Functions

Parameter types and return-type of constexpr functions must be a literal type.

```
constexpr std::string cat(const std::string &a, // Error!
                          const std::string &b) {
  return a + b;
}
```

constexpr Functions

C++11 constexpr functions must contain only a return statement, but C++14 allows more control-flow statements.

```
// Allowed since C++14
constexpr int power(int x, int n) {
  int result = 1;
  while (n--)
    result *= x;
  return result;
}
// The version for C++11
constexpr int power(int x, int n) {
  return n == 0 ? 1 : power(x, n - 1);
}
```

More at Compile-time...

C++ has been trying to do more work at compile-time...

- Constructors may be `constexpr` .
- STL containers like `std::vector`, `std::string` have `constexpr` constructors and member functions since C++20.

More at Compile-time...

C++ has been trying to do more work at compile-time...

- Constructors may be `constexpr` .
- STL containers like `std::vector`, `std::string` have `constexpr` constructors and member functions since C++20.
- Non-type template parameters must be integral type (integers, `bool`, `char`) before C++20...
  - Floating-point types or class types are not allowed.

More at Compile-time...

C++ has been trying to do more work at compile-time...

- Constructors may be `constexpr` .
- STL containers like `std::vector`, `std::string` have `constexpr` constructors and member functions since C++20.
- Non-type template parameters must be integral type (integers, `bool`, `char`) before C++20...
  - Floating-point types or class types are not allowed.
- Since C++20, anything can be used as template parameters.

An Interesting Example

```
template <unsigned long N>
struct binary {
  static constexpr unsigned long value
      = (binary<N / 10>::value << 1) + (N & 1);
};

template <>
struct binary<0ul> {
  static constexpr unsigned long value = 0;
};
```