CS100 Recitation 15

GKxx

June 6, 2022

Contents

Contents

Variadic Templates

```cpp
template <typename T>
inline void read(T &x) {
  int c = getchar(), f = 0; x = 0;
  while (!isdigit(c)) {
    f |= c == '-';
    c = getchar();
  }
  while (isdigit(c)) {
    x = x * 10 + c - '0';
    c = getchar();
  }
  if (f) x = -x;
}
template <typename T, typename... Args>
inline void read(T &x, Args &...args) {
  read(x); read(args...);
}
```

Variadic Templates

```
int a; long b; unsigned c; short d;
read(a, b, c, d);
```

Variadic Templates

```
int a; long b; unsigned c; short d;
read(a, b, c, d);
```

The compiler will generate:

```
void read(int &, long &, unsigned &, short &);
void read(long &, unsigned &, short &);
void read(unsigned &, short &);
void read(short &);
void read(int &);
void read(long &);
void read(unsigned &);
```

Pack Expansion

When `Args &...args` becomes

```
long &a, unsigned &b, short &c
```

the expression `read(args...)` expands to

```
read(a, b, c);
```

- e.g. `&args...` expands to `&a, &b, &c`.
- e.g. `const_cast<const Args *>(&args)...` expands to

  ```
  const_cast<const T1 *>(&x1), const_cast<const T2 *>(&
      x2), const_cast<const T3 *>(&x3)
  ```

- `sizeof...(args)` is the number of elements in the parameter pack. (NOT '`sizeof(args)...`'!)

Contents

Check whether Two Types are Same

```
template <typename X, typename Y>
struct AreSame {
  static constexpr bool value = false;
};
template <typename X>
struct AreSame<X, X> {
  static constexpr bool value = true;
};

AreSame<int, double>::value // false
AreSame<int, signed>::value // true
```

Check whether a Type is a Pointer

```
template <typename T>
struct IsPointer {
  static constexpr bool value = false;
};
template <typename T>
struct IsPointer<T *> {
  static constexpr bool value = true;
};
```

<type_traits>

Primary type categories

`is_void` (C++11)	checks if a type is `void` (class template)
`is_null_pointer` (C++14)	checks if a type is `std::nullptr_t` (class template)
`is_integral` (C++11)	checks if a type is an integral type (class template)
`is_floating_point` (C++11)	checks if a type is a floating-point type (class template)
`is_array` (C++11)	checks if a type is an array type (class template)
`is_enum` (C++11)	checks if a type is an enumeration type (class template)
`is_union` (C++11)	checks if a type is an union type (class template)
`is_class` (C++11)	checks if a type is a non-union class type (class template)
`is_function` (C++11)	checks if a type is a function type (class template)
`is_pointer` (C++11)	checks if a type is a pointer type (class template)
`is_lvalue_reference` (C++11)	checks if a type is a *lvalue reference* (class template)
`is_rvalue_reference` (C++11)	checks if a type is a *rvalue reference* (class template)
`is_member_object_pointer` (C++11)	checks if a type is a pointer to a non-static member object (class template)
`is_member_function_pointer` (C++11)	checks if a type is a pointer to a non-static member function (class template)

<type_traits>

Composite type categories

`is_fundamental` (C++11)	checks if a type is a fundamental type (class template)
`is_arithmetic` (C++11)	checks if a type is an arithmetic type (class template)
`is_scalar` (C++11)	checks if a type is a scalar type (class template)
`is_object` (C++11)	checks if a type is an object type (class template)
`is_compound` (C++11)	checks if a type is a compound type (class template)
`is_reference` (C++11)	checks if a type is either a *lvalue reference* or *rvalue reference* (class template)
`is_member_pointer` (C++11)	checks if a type is a pointer to an non-static member function or object (class template)

Type properties

`is_const` (C++11)	checks if a type is const-qualified (class template)
`is_volatile` (C++11)	checks if a type is volatile-qualified (class template)
`is_trivial` (C++11)	checks if a type is trivial (class template)
`is_trivially_copyable` (C++11)	checks if a type is trivially copyable (class template)
`is_standard_layout` (C++11)	checks if a type is a standard-layout type (class template)
`is_pod` (C++11)(deprecated in C++20)	checks if a type is a plain-old data (POD) type (class template)
`is_literal_type` (C++11) (deprecated in C++17) (removed in C++20)	checks if a type is a literal type (class template)

<type_traits>

Supported operations

is_constructible (C++11) **is_trivially_constructible** (C++11) **is_nothrow_constructible** (C++11)	checks if a type has a constructor for specific arguments (class template)
is_default_constructible (C++11) **is_trivially_default_constructible** (C++11) **is_nothrow_default_constructible** (C++11)	checks if a type has a default constructor (class template)
is_copy_constructible (C++11) **is_trivially_copy_constructible** (C++11) **is_nothrow_copy_constructible** (C++11)	checks if a type has a copy constructor (class template)
is_move_constructible (C++11) **is_trivially_move_constructible** (C++11) **is_nothrow_move_constructible** (C++11)	checks if a type can be constructed from an rvalue reference (class template)
is_assignable (C++11) **is_trivially_assignable** (C++11) **is_nothrow_assignable** (C++11)	checks if a type has a assignment operator for a specific argument (class template)
is_copy_assignable (C++11) **is_trivially_copy_assignable** (C++11) **is_nothrow_copy_assignable** (C++11)	checks if a type has a copy assignment operator (class template)
is_move_assignable (C++11) **is_trivially_move_assignable** (C++11) **is_nothrow_move_assignable** (C++11)	checks if a type has a move assignment operator (class template)
is_destructible (C++11) **is_trivially_destructible** (C++11) **is_nothrow_destructible** (C++11)	checks if a type has a non-deleted destructor (class template)
has_virtual_destructor (C++11)	checks if a type has a virtual destructor (class template)
is_swappable_with (C++17) **is_swappable** (C++17) **is_nothrow_swappable_with** (C++17) **is_nothrow_swappable** (C++17)	checks if objects of a type can be swapped with objects of same or different type (class template)

Examples

Raise a compile-error with customized information text when `Shape` is not an abstract class:

```
static_assert(std::is_abstract<Shape>::value,
              "'Shape' must be an abstract class!");
```

Examples

Raise a compile-error with customized information text when Shape is not an abstract class:

```
static_assert(std::is_abstract<Shape>::value,
              "'Shape' must be an abstract class!");
```

`std::vector` doesn't allow value-type to be cv-qualified:

```
template <typename Tp, typename Alloc>
class vector {
  static_assert(std::is_same<Tp, typename std::remove_cv<
      Tp>::type>::value, "std::vector must have a non-
      const, non-volatile value type!");
  // ...
};
```

Examples

```
template <typename Tp, typename Alloc>
class vector {
  static_assert(std::is_same<Tp, typename std::remove_cv<
      Tp>::type>::value, "std::vector must have a non-
      const, non-volatile value type!");
  // ...
};
```

Why is `typename` here?

Examples

```
template <typename Tp, typename Alloc>
class vector {
  static_assert(std::is_same<Tp, typename std::remove_cv<
      Tp>::type>::value, "std::vector must have a non-
      const, non-volatile value type!");
  // ...
};
```

Why is `typename` here?

- When `Tp` is unknown, the compiler does not know what `std::remove_cv<Tp>` is.
- Is `std::remove_cv<Tp>::type` a type or a `static` data member? You tell the compiler.

Examples

```
template <typename T, bool is_const>
class Slist_iterator {
 public:
  using reference = typename std::conditional<is_const,
      const T &, T &>::type;
  using pointer = typename std::conditional<is_const,
      const T *, T *>::type;
};
```

Examples

```
template <typename T, bool is_const>
class Slist_iterator {
 public:
  using reference = typename std::conditional<is_const,
        const T &, T &>::type;
  using pointer = typename std::conditional<is_const,
        const T *, T *>::type;
};
```

How is `std::conditional` implemented?

Examples

```
template <bool Condition, typename TrueT, typename FalseT>
struct Conditional {
  using type = TrueT;
};
template <typename TrueT, typename FalseT>
struct Conditional<false, TrueT, FalseT> {
  using type = FalseT;
};
```

std::distance

```
template <typename Iterator>
?? distance(Iterator begin, Iterator end);
```

What's the return-type?

std::distance

```
template <typename Iterator>
?? distance(Iterator begin, Iterator end);
```

What's the return-type?

```
typename Iterator::difference_type
```

std::distance

Is this correct?

```
template <typename Iterator>
  typename Iterator::difference_type
    distance(Iterator begin, Iterator end) {
  return end - begin;
}
```

std::distance

Is this correct?

```
template <typename Iterator>
  typename Iterator::difference_type
    distance(Iterator begin, Iterator end) {
  return end - begin;
}
```

Yes, but only for random-access-iterators.

`std::distance`

Recognize and choose different implementations:

- For input-iterators, increment `begin` until it reaches `end` and count the steps.
- For random-access-iterators, return `end - begin`.

std::distance

Recognize and choose different implementations:

- For input-iterators, increment begin until it reaches end and count the steps.
- For random-access-iterators, return end - begin.

Require every iterator to have a type alias member: `iterator_category`.

- `std::vector<T>::iterator_category` is `std::random_access_iterator_tag`
- `std::list<T>::iterator_category` is `std::bidirectional_iterator_tag`
- `std::forward_list<T>::iterator_category` is `std::forward_iterator_tag`

std::distance

```
template <typename Iterator>
auto distance_impl(Iterator begin, Iterator end,
                   std::input_iterator_tag) {
  // ...
}
template <typename Iterator>
auto distance_impl(Iterator begin, Iterator end,
                   std::random_access_iterator_tag) {
  return end - begin;
}
template <typename Iterator>
inline auto distance(Iterator begin, Iterator end) {
  using category = typename Iterator::iterator_category;
  return distance_impl(begin, end, category{});
}
```

- “Compile-time polymorphism”.

Traits

Wait... What about pointers?

- Pointers are also iterators,
- but they do not have a '`iterator_category`' member!

Traits

Wait... What about pointers?

- Pointers are also iterators,
- but they do not have a '`iterator_category`' member!

```
template <typename Iterator>
struct Category {
  using type = typename Iterator::iterator_category;
};
template <typename T>
struct Category<T *> {
  using type = std::random_access_iterator_tag;
};
```

Traits

```
template <typename Iterator>
struct Category {
  using type = typename Iterator::iterator_category;
};
template <typename T>
struct Category<T *> {
  using type = std::random_access_iterator_tag;
};
template <typename Iterator>
inline auto distance(Iterator begin, Iterator end) {
  return distance_impl(begin, end,
                       typename Category<Iterator>::type{});
}
```

std::iterator_traits

The standard library has defined `std::iterator_traits`:

```
template <typename Iterator>
struct iterator_traits {
  using value_type = typename Iterator::value_type;
  using difference_type
      = typename Iterator::difference_type;
  using pointer = typename Iterator::pointer;
  using reference = typename Iterator::reference;
  using iterator_category
      = typename Iterator::iterator_category;
};
```

std::iterator_traits

Specialization of `std::iterator_traits` for pointers:

```
template <typename T>
struct iterator_traits<T *> {
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using pointer = T *;
  using reference = T &;
  using iterator_category
      = std::random_access_iterator_tag;
};
```

Contents

Test whether a Function Exists

SFINAE: Substitution Failure Is Not An Error

- When substitution failure happens, the compiler tries some other solutions instead of reporting an error.

Test whether a Function Exists

SFINAE: Substitution Failure Is Not An Error

- When substitution failure happens, the compiler tries some other solutions instead of reporting an error.
- If we can transform an error into a substitution failure...

Test whether a Function Exists

SFINAE: Substitution Failure Is Not An Error

- When substitution failure happens, the compiler tries some other solutions instead of reporting an error.
- If we can transform an error into a substitution failure...

```
namespace detail {
  template <typename T = Expr,
            typename = decltype(make_sin((T *)nullptr))>
  std::true_type helper(int);
  std::false_type helper(double);
}
constexpr bool make_sin_defined
    = decltype(detail::helper(0))::value;
```

- That's how things like `std::is_copy_constructible` are implemented.

Test whether a Member Function is const

```
namespace detail {
  template <typename T = Point_handle,
            typename = decltype(((const T *)nullptr)->
                get_x())>
  std::true_type helper(int);
  std::false_type helper(double);
}
constexpr bool get_x_is_const
    = decltype(detail::helper(0))::value;
static_assert(get_x_is_const, "Why don't you define get_x
    as a const member function?");
```

std::enable_if

Enable a function to be called according to a given condition:

```
template <typename T>
inline auto read(T &x) // Only allow T to be integral type
    -> typename std::enable_if<std::is_integral<T>::value,
                               void>::type {
  // ...
}
```

std::enable_if

Allow conversion from iterator to const_iterator:

```
template <typename T, bool is_const>
class Slist_iterator {
 public:
  template <typename Other,
            typename = typename std::enable_if<
                std::is_same<Slist_iterator<T, false>,
                             Other>::value
                && is_const>::type>
  Slist_iterator(const Other &oi);
};
```

Contents

Compile-time Factorial

Compile-time factorial:

```
template <unsigned N>
struct Factorial {
  static constexpr unsigned value
      = N * Factorial<N - 1>::value;
};
template <>
struct Factorial<0> {
  static constexpr unsigned value = 1;
};
```

You may use it like this:

```
std::cout << Factorial<5>::value;  // prints 120
std::cout << Factorial<10>::value; // prints 3628800
```

The results are calculated during compile-time!

What can it Accomplish?

Examples:

- Ensuring dimensional unit correctness
- Optimizing matrix operations
- Generating custom design pattern implementations
- Designing “domain-specific embedded languages” (DSEL).

A Simple Example: Dimensions

From *C++ Template Metaprogramming* Chapter 3:

```
#include "dimensions.hpp"
#include <iostream>
int main() {
    quantity<double, mass> m(3.0);
    quantity<double, force> f(6.0);
    quantity<double, acceleration> a(2.0);
    quantity<double, force> f2 = m * a;
    std::cout << f2.value() << std::endl;
    std::cout << (m * a == f) << std::endl;
    f += -m * a + f;
    std::cout << f.value() << std::endl;

    // This should cause a compile-error.
    std::cout << (m + f).value() << std::endl;
    return 0;
}
```

What Have We Learned?

1. Getting Started	
2. Variables and Basic Types	
3. Strings, Vectors, and Arrays	
4. Expressions	
5. Statements	Exception handling (try-catch, throw)
6. Functions	
7. Classes	
8. The IO Library	
9. Sequential Containers	
10. Generic Algorithms	Lambdas
11. Associative Containers	
12. Dynamic Memory	Smart pointers
13. Copy Control	Rvalue references and move semantics
14. Overloaded Operations and Conversions	
15. Object-Oriented Programming	
16. Templates and Generic Programming	Type deduction related to rvalue refs
17. Specialized Library Facilities	
18. Tools for Large Programs	
19. Specialized Tools and Techniques	

- Chapter 4. Smart Pointers
  - Item 18: Use std::unique_ptr for exclusive-ownership resource management.
  - Item 19: Use std::shared_ptr for shared-ownership resource management.
  - Item 20: Use std::weak_ptr for std::shared_ptr-like pointers that can dangle.
  - Item 21: Prefer std::make_unique and std::make_shared to direct use of new.
  - Item 22: When using the Pimpl Idiom, define special member functions in the implementation file.
- Chapter 5. Rvalue References, Move Semantics, and Perfect Forwarding
  - Item 23: Understand std::move and std::forward.
  - Item 24: Distinguish universal references from rvalue references.
  - Item 25: Use std::move on rvalue references, std::forward on universal references.
  - Item 26: Avoid overloading on universal references.
  - Item 27: Familiarize yourself with alternatives to overloading on universal references.
  - Item 28: Understand reference collapsing.
  - Item 29: Assume that move operations are not present, not cheap, and not used.
  - Item 30: Familiarize yourself with perfect forwarding failure cases.
- Chapter 6. Lambda Expressions
  - Item 31: Avoid default capture modes.
  - Item 32: Use init capture to move objects into closures.
  - Item 33: Use decltype on auto&& parameters to std::forward them.
  - Item 34: Prefer lambdas to std::bind.
- Chapter 7. The Concurrency API
  - Item 35: Prefer task-based programming to thread-based.
  - Item 36: Specify std::launch::async if asynchronicity is essential.
  - Item 37: Make std::threads unjoinable on all paths.
  - Item 38: Be aware of varying thread handle destructor behavior.
  - Item 39: Consider void futures for one-shot event communication.
  - Item 40: Use std::atomic for concurrency, volatile for special memory.

Exception Handling

- `try-catch`, `throw` and the exception classes defined in the standard library
- *Effective C++* Item 8, 25, 29.

Concurrency

Multi-threading in C++: `<thread>`, `<future>`, ... (since C++11)
Play with them in CS110!

The Boost Library

A collection of high quality, open source, platform- and compiler-independent libraries. http://boost.org

- Lambda: (Unlike C++11 lambda)

```
std::for_each(v.begin(), v.end(),
              std::cout << _1 * 2 + 10 << '\n');
```

- Template metaprogramming library (mpl)
- Math and numerics
- Inter-language support (e.g. boost python)
- Memory management
- Graph library
- ......