CS11 Advanced C++

Spring 2020 – C++ Standard Library Templates

The C++ Standard Library

- The C++ Standard Library makes heavy use of both class and function templates
- Example:
  - std::string is an instantiation of std::basic_string<char>
  - Also std::wstring, std::u32string, etc.
- The collection templates are a very well known part of the C++ Standard Library
  - Very sophisticated class templates to provide collection functionality for programs
- These collections, associated algorithms, and iterators, used to be known as the Standard Template Library
  - Now they are simply the “C++ Standard Library collections”

Standard Template Library (STL)

- The STL is a very influential library and set of approaches to providing generic containers
- Primary architect: Alexander Stepanov
  - AT&T Bell Labs, then later Hewlett Packard
- Andrew Koenig motivated proposal to ANSI/ISO Committee in 1994
- Proposal accepted/standardized in 1994
- Continuous refinements, increased support

C++ Standard Library / STL

- The STL aimed to provide a set of generic containers, algorithms, and iterators that provide many of the basic algorithms and data structures of computer science
- Generic
  - Heavily parameterized; lots of templates
- Containers
  - Collections of other objects, with various characteristics
- Algorithms
  - For manipulating the data stored in containers
- Iterators
  - “A generalization of pointers”
  - Cleanly decouple algorithms from containers

A Simple STL Example

- You want an array of numbers

```
std::vector<int> v{3};  // Vector of 3 elems
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];
```

- Now you want to reverse their order

```
std::reverse(v.begin(), v.end());
```

- **`vector<int>`** is the generic container
- **`reverse()`** is a generic algorithm
- **`reverse()`** uses iterators associated with **`v`**

C++ Standard Library Algorithms

- Algorithms are generic function templates
  - *(well, mostly…)*
  - Parameterized on iterator type – *not* container
- Example: the **`find()`** algorithm

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator first,
                   InputIterator last,
                   const T& value) {
  while (first != last && *first != value) ++first;
  return first;
}
```

  - Searches for **`value`** in range **`[first, last)`**.

Algorithms and Iterators

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator first,
                   InputIterator last,
                   const T& value) {
```

- **`InputIterator`** isn't a specific type

```
while (first != last && *first != value) ++first;
```

  - Just needs to support **`*`** (dereference), **`++`** (increment), and equality operators

- Pointers also satisfy these constraints

```
float a[5] = { 1.1, 2.3, -4.7, 3.6, 5.2 };
float *pVal;
pVal = find(a, a + 5, 3.6);  // float* as iterators
```

The Big Picture

- This set of required functionality for the iterator-type is called a concept
  - In this case, the concept is named “InputIterator”
- A type that satisfies these requirements is said to “model the concept”
  - Or, it “conforms to the concept”
- Example:
  - **`int*`** is a model of Input Iterator because **`int*`** provides all of the operations that are specified by the Input Iterator requirements

What about **`reverse()`**?

- The **`reverse()`** algorithm needs more!
  - Specifically, its iterators also need the `--` operator.
- **`reverse()`**'s arguments must model the BidirectionalIterator concept.
  - Like InputIterator, but with more requirements.
- BidirectionalIterator refines the InputIterator concept.
  - This is *exactly like* class-inheritance
  - Different terms because these *aren't* classes

Iterator Concept Hierarchy

- Trivial Iterator – supports dereference
  - That's it. Yep, it's trivial.
- Input Iterator – supports increment
  - Only read support is guaranteed.
  - Only single-pass support guaranteed.
- Forward Iterator – like Input Iterator
  - Supports multi-pass algorithms.
- Bidirectional Iterator – supports decrement
- Random Access Iterator
  - Supports arbitrary-size steps forward and backward

Output Iterators

- Output Iterators don't appear in the iterator concept hierarchy
- Different, very limited set of requirements
  - Support assignment
  - Support increment
  - Support postincrement-and-assign
    - `*iter++ = value;`
- "It's like a tape."
  - You can write to the current location
  - You can advance to the next location

Function Objects

- Anything that can be called like a function
  - A generalization of functions
  - Can be a true function pointer
  - Can be an instance of a class that overloads **()**
- Allows customization of algorithm operations
  - Can pass these things to C++ Standard Library algorithms
- Also known as “functors”

Function Pointers

- C/C++ functions can be referred to by name
  - **`sin(x)`**, **`cos(x)`**, **`sqrt(x)`**, etc.
- Can also refer to functions via function pointers
  - Like a normal pointer, but function can be called through it
  - Function's signature is part of the pointer's type
    - Number and types of arguments, return type
- Above funcs take a **`double`** and return a **`double`**
  - A function pointer for them could be like this:

```
double (*fp)(double);
```

  - Variable name is **`fp`**
  - Points to a function that takes a **`double`** and returns a **`double`**

Using Function Pointers

- Normally refer to functions to invoke them

```
double rot = coord * sin(angle);
```

  - Invokes **`sin`**, using **`angle`** as argument
- Can also get a function's address via its name

```
double (*fp)(double);
...
fp = sin;  // No arguments to sin here!
...
double res = fp(input);
```

  - Use **`fp`** like a normal function
  - Can set **`fp`** to *any* function with the same signature
    - **`sin`**, **`cos`**, **`tan`**, **`sqrt`**, **`log`**, **`exp`**, your own functions, etc.

Functor Concepts

- Generator `f()`
  - No arguments.
- Unary Function `f(x)`
  - One argument.
- Binary Function `f(x, y)`
  - Two arguments.
- Special concepts for `bool` return-types
  - Predicate `bool p(x)`
  - Binary Predicate `bool p(x, y)`
- Others, too…

Simple Functor Example

- You want a collection of 100 random values

```
vector<int> values{100};
generate(values.begin(), values.end(), rand);
```

- Can create and use your own generator functions

```
int randomColorValue() {
  return rand() & 0x00FFFFFF;
}
...
vector<int> randColors{10};
generate(randColors.begin(), randColors.end(),
  randomColorValue);
```

Functors with State

- You want the sum of a vector of integer values
  - Create a functor with state
  - A class with overloaded **()** is perfect for this

```
struct adder {
  int sum;
  adder() : sum{0} { }
  void operator()(int x) { sum += x; }
};
```

- Apply functor with **for_each** algorithm

```
adder result =
  for_each(values.begin(), values.end(), adder{});
cout << "Sum is " << result.sum << "\n";
```

The **`for_each()`** Algorithm

- Example implementation of **`for_each()`**:

```
template <typename InputIterator, typename Function>
Function for_each(InputIterator first,
                  InputIterator last, Function f) {
  while (first != last) {
    f(*first);
    ++first;
  }
  return f;
}
```

- Our example:

```
adder result =
  for_each(values.begin(), values.end(), adder{});
```

  - An **`adder`** object is initialized; a copy is passed to **`for_each()`**
  - Function-template uses object **`f`** as a function on each element
  - Function returns the object **`f`**, which is then copied into **`result`**

Printing The Numbers

- Now you want to print the numbers, separated with commas.
- Use **`copy()`** algorithm and Output Iterators

```
copy(values.begin(), values.end(),
  ostream_iterator<int>(cout, ", "));
```

  - Note that **`ostream_iterator`** template-param must match element-type of collection.

C++ Standard Library

- C++ Standard Library function objects are much more sophisticated than the examples in this lecture
- Reason: C++ also supports function-object composition, partial binding of arguments, etc.
  - Implementation details are pretty baroque, and not nearly as elegant as that provided by functional languages
  - Details are also changing in C++17, C++20!
  - (Much of the current complexity is to support backward compatibility, and will disappear in C++17/C++20)
- Nonetheless, simple implementations like this can be used to customize collection and algorithm behavior