1

Function and Class Templates

CSC 340

March 30, 2016

2

Overview

- **Generic Programming**
- Function Templates
- Class Templates
- Standard Template Library

3

Generic Programming (Java pre 5.0)

- Use inheritance to implement generic programming - for example, all of the Collections API classes are written in terms of the Object class
- Limitations:
  - Primitive types cannot be directly added into a standard Java collection. Wrapper classes must be used
  - Excessive use of downcasting - time consuming and delaying typing errors until runtime

Using a Collection in Java, one would have to explicitly cast an object stored in the collection back to the appropriate type

4

Generic Programming (C++, Java)

- Java now uses Generics
- C++ uses templates: a blueprint or pattern for creating functions or classes
- Two types:
  - Function templates
  - Class templates (parameterized classes)

4

5

Purpose

- Relieve the programmer of the burden of having to write multiple versions of the same function just to carry out the same operation on different types

5

6

Example

- Consider a function, swap

```
void swap(int& X, int& Y) {
  int temp = X;
  X = Y;
  Y = temp;
}
```

- Drawbacks?

6

We might have implemented swap for one of our sort methods

It only works on one type - integers. What happens if we want this to work for an array of doubles? floats? Envelopes?

7

Overview

- Generic Programming
- **Function Templates**
- Class Templates
- Standard Template Library

8

Function Templates

- A function template has one or more template parameters

```
template<class T>
return_type function_name(T param)
```

- T is a template parameter - it refers to a data type that will be supplied when the function is called

9

Swap, revisited

- Multiple swaps for multiple types:

```
void swap(int& X, int& Y) {
  int temp = X;
  X = Y;
  Y = temp;
}

void swap(string& X, string& Y) {
  string temp = X;
  X = Y;
  Y = temp;
}
```

10

Swap Function Template

- Works with any data type that supports the assignment operator

```
template <class TParam>
void swap(TParam & X, TParam & Y) {
  TParam temp = X;
  X = Y;
  Y = temp;
}
```


11

Specific Swap Function

- This version can co-exist with the template function

```
void swap(string * X, string * Y) {
  string temp = *X;
  *X = *Y;
  *Y = temp;
}
```


12

Calling swap

- When calling swap, the only requirement is that both parameters are the same data type, and that they are modifiable via the = operator

```
int A = 5;
int B = 6;
swap(A, B);

string P(“string one”);
string Q(“other string”);
swap(P, Q);

string * R = new string(“John”);
string * S = new string(“Blarg”);
swap(R, S);
```

13

Illegal Template Calls

- Some combinations won't work - you can't pass constants, and you can't pass incompatible types

```
int B = 6;
// illegal
swap(10, B);

// illegal
swap("Harry", "Sally");

bool E = true;
// illegal
swap(B, E);
```


14

Another example

- The display function can display any data type that overloads the stream insertion operator

```
template <class T>
void display(const T & val) {
  cout << val;
}
```


15

Additional Parameters

- A function template can have additional parameters which are not template parameters

```
template <class T>
void display(const T & val, ostream & os) {
  os << val;
}

display("Hello", cout);
display(22, outfile);
```

16

Additional Parameters

- There can be additional template parameters, as long as each is used at least once in the function's parameter list
- T1 is any container that support the begin() operation
  T2 is the data type stored in the container

```
template <class T1, class T2>
T2 & getFirst( const T1 & container, T2 & value )
{
  value = *container.begin();
  return value;
}
```


17

Calling getFirst

```
vector<int> vec;
vec.push_back( 32 );
int n;
cout << getFirst( vec, n );

list<double> testList;
testList.push_back( 42.24 );
double x;
getFirst( testList, x );
```


This will make more sense after we talk about STL

18

Tips for Defining Templates

- Start with an ordinary function that accomplishes the task with one type - it's often easier to deal with a concrete case rather than the general case
- Then, debug the ordinary function
- Next, convert the function to a template by replacing type names with a type parameter

19

Overview

- Generic Programming
- Function Templates
- **Class Templates**
- Standard Template Library

20

Class Templates

- Let you create new classes at runtime
- Use a class template when you would otherwise be forced to create multiple classes with the same attributes and operations
- STL (Standard Template Library) Container classes - vector, list, set - are good examples

21

Defining a Class Template

- General format for declaring a class template. The parameter T represents a data type:

```
template <typename T>
class className
{};

// OR
template <class T>
class className
{};
```

22

Defining a Class Template

- Class templates are defined ONLY IN THE .h FILE
- Read that again - class templates are not defined in separate compilation units (they're not really classes, they are patterns that the compiler uses to define classes when needed)

22

23

Array Example

- Typed arrays: https://gist.github.com/jrob8577/30a29237e05f77bcdf6c
- Array class template: https://gist.github.com/jrob8577/dca3b3149d4cb78f0a7b
- A client can create any type of array using our template, as long as it permits the use of the assignment operator =: https://gist.github.com/jrob8577/d31a99928f57bbad16d2

23

24

Next Assignment

- Make Queue and Node template classes
- One gotcha - we can no longer return -1 in the event that the queue is empty. We will return the default type, as created using the default constructor for the specified type

```
template <typename T>
T Queue<T>::front() {
  // example of default constructor for type T
  return T();
}
```

24

This leads to some edge cases in code we've written - we can no longer test for -1, and should instead ensure the queue is not empty…

25

Overview

- Generic Programming
- Function Templates
- Class Templates
- **Standard Template Library**

26

STL

- Standard Template Library
  - Reference: http://www.sgi.com/tech/stl/
- Created by Alexander Stepanov and Meng Lee at Hewlett Packard
- Helps programmers avoid having to reinvent standard container implementation code
- Not part of the C++ core, but part of the C++ standard
- Key components: containers, iterators, algorithms

27

Container Types

- Sequence
  - vector, deque, list
- Container Adapters
  - stack, queue, priority_queue
- Associative Containers
  - set, multiset, map, multi map

28

Including the STL

- The class name is the same as the header to require `<vector>, <list>, <deque>, <stack>, <bitset>`
- Also
```
<queue> // queue and priority_queue
<set> // set and multiset
<map> // map and multimap
```

28

29

Common Member Functions

- default constructor, copy constructor, destructor
- empty
- max_size (max number of elements for a container)
- size (number of elements currently in the container)
- operator = (assign one container to another)
- overloaded <, <=, >, >=, ==, !=
- swap

29

30

Common Member Functions

- `begin()` returns an iterator pointing at the first element
- `end()` returns an iterator pointing after the last element
- `erase()` erases one or more elements
- `clear()` erases all elements
- `rbegin()` returns reverse_iterator
- `rend()` return reverse_iterator

30

reverse iterators iterate backwards

31

`list` Sequence Container

- Efficient insertion and deletion at any location
  - `deque` is more efficient if insertions and deletions are at the ends of the container
- Implemented as a doubly linked list
  - supports bidirectional iterators
- Does not support `at()` or `operator []`

32

`deque` Sequence Container

- Provides benefits of `vector` and `list` in the same container
  - Rapid indexed access (like `vector`)
  - Efficient insertion and deletion at its end (like `list`)
  - Supports random access iterators
  - Often used for FIFO queue
  - Noncontiguous memory layout

33

Common `deque` Operations

- `push_back()`
- `push_front()`
- `pop_back()`
- `pop_front()`
- `size()`
- `operator[]`
- `at()`

34

Common `stack` Operations

- Constructor can take a reference to a `Container` (or create an empty stack)
- `empty()`
- `size()`
- `top()`
- `pop()`
- `push()`

35

Common `queue` Operations

- Constructor can take a reference to a `Container` (or create an empty queue)
- `empty()`
- `size()`
- `front()`
- `back()`
- `pop()`
- `push()`

36

Associative Containers

- Keys are always kept in sorted order
  - iterator always traverses in sort order
- `Set` - Single key values
- `Multiset` - Duplicate keys allowed
- `Map` - Contains pairs of (key, value)
  - key is unique
- `Multimap`
  - keys do not need to be unique

37

Error Handling

- Exceptions are never thrown by STL classes
- Very little error checking exists in the STL classes
  - Read the docs!

37

38

Iterator Basics

- An iterator is a generalization of pointers
- Facilitates cycling through the data in a container
- Although an iterator is not a pointer, you can think of it and use it as if it were
  - Supports the ++, —, ==, !=, and * operators

38

39

Iterator Example

- Declare a vector of integers, and an iterator to iterate over the vector

```
vector<int> container;
vector<int>::iterator p;
for( p = container.begin(); p !=
container.end(); p++ )
  cout << *p;
```

39

40

Types of Iterators

- Input Iterator (InIt)
  - Used to read an element from a container
  - Moves only in the forward direction, one element at a time
  - Cannot pass through a sequence twice
- Output Iterator (OutIt)
  - Used to write an element to a container
  - Moves only in the forward direction, one element at a time
  - Cannot pass through a sequence twice

41

Types of Iterators

- Forward Iterator (FwdIt)
  - Reads and writes in forward direction only
  - Retains its position in the container
- Bidirectional Iterator (BidIt)
  - Reads and writes in both directions
  - Retains its position in the container
- Random-Access Iterator (RadIt)
  - Reads and writes in randomly accessed positions
  - permits pointer arithmetic

Note that each iterator's capabilities are a superset of the previously listed iterators

42

Iterator Support by Container

Containers	Supported Iterators
stack, queue, priority queue	no iterators
list, set, multiset, map, multimap	bidirectional
vector, deque	random

43

Invalidating Iterators

- Some operations on a container may invalidate iterators
  - Results in a stale iterator
- Example:
  - `list.push_back()` does not invalidate iterators
  - `vector.push_back()` invalidates all iterators
  - `list.erase()` invalidates iterators that pointed to the delete elements
- Check the documentation for each method!

44

Generic Algorithms

- Includes over 60 function templates
  - Sorting
  - Searching
  - Copying
  - accumulate
  - Inner_product
  - Partial sum
  - Many More
- http://www.sgi.com/tech/stl/table_of_contents.html

45

Generic Algorithms for Sorting

- Syntax

```
void sort( Iterator begin, Iterator end );
void sort( Iterator begin, Iterator end, Comparator
comp );
```

- Example

```
Vector<int> vector;
sort( v.begin(), v.end(), less<int>() );
```

- less<int>() is called a function object

46

Function Object Example

- Default comparison of STL container objects when sorting uses the < operator:

```
template<typename T>
class less {
  public:
    bool operator() ( const T & x, const T & y) const
    {
       return x < y;
    }
};
```


47

Predefined Function Objects

- Template class that contains a function
- Makes the STL more flexible - permits sorting on your own criteria
- Predefined objects in the STL:

```
divides<T>
equal_to<T>
greater<T>
greater_equal<T>
less<T>
…
```


48

Searching

- Syntax

```
Iterator find( Iterator begin, Iterator end, const Object& x );
Iterator find_if( Iterator begin, Iterator end, Predicate pred );
```

- Example

```
template<int len>
class StringLength {
  public:
    bool operator() (const string & s ) const {
      return s.length() == len;
    }
};

// Returns first string of length 9
find_if( v.begin(), v.end(), StringLength<9>() );
```


Predicate is a boolean function object

Many other searches, see docs!

Copying

- Including - copy, copy_backwards, remove, remove_copy, remove_if, replace, and many others

- Example

```
vector<int> source(10, 37);
vector<int> target( 10 );
copy( source.begin(), source.end(), target.begin() );
```