School of Computing and Informatics

BCS 362 – Generic Programming with C++

Class Templates

1 Introduction

Class templates enable you to conveniently specify a variety of related classes – referred to as **class template specializations**. Programming with templates is also known as **generic programming**.

1.1 Class Templates

Class templates encourage software re-usability by enabling a variety of type-specific class template specilaizations to to instantiated from a single class template.

All class templates begin with a keyword **template** followed by a list of **template parameters** enclosed in angle brackets (< >). Each template parameter that represent a type must be preceded by either **typename** or **class**, though **typename** is preferred to avoid ambiguity. Just like other parameters, the names of a parameter must be unique inside template definition.

Type parameter becomes associated with a specific type when you create an object using the class template– at this point, the compiler generates a copy of the class template, in which all the occurrences of the type parameter are replaced with specified type.

1.2 Stack/Queue Class Template-How to represent data

A stack can use various containers to store its elements. A stack requires insertions and deletions only at its top. Therefore, a vector or a deque could be used to store a stack elements. A vector supports faster insertions and deletions at the back. A deque supports fast insertions and deletions and its front and its back. A deque is the default representation for the Standard Library's stack adapter because a deque grows more deficiently than a vector.

A vector is maintained as a contiguous block of memory – when that block is full and a new element is added, the vector allocates a larger contiguous block of memory and copies the old elements into the new block. A deque on other hand, is typically implemented as a list of fixed size built-in arrays – new fixed-size built-in arrays are added as necessary and none of the existing elements are copied when new items are added to the front or back. For this reason, a deque is most appropriate underlying container to implement a stack or a queue.

1.3 deque Sequence Container

Class **deque** provides many of the **vector** and **list** benefits in a single container. The term **deque** is short form of *double-ended queue*. The class **deque** is implemented to provide efficient indexed access (using subscripting) for reading and modifying its elements, much like a vector. This class is also implemented for efficient *insertion and deletion operations at front and back*, much like a **list** (although a list also provides efficient insertions and deletions in the middle of the list). Class deque also provide random-access iterators, so deque can be used with all Standard Library algorithms. One of the most common use of a deque is to maintain a *first-in, first out* queue. Actually, deque is the default underlying implementation for the queue adaptor.

Additional storage for a **deque** can be allocated at either end of the deque in blocks of memory that are typically maintained as a built-in array of pointers to those blocks.

1.4 Stack Class Template

We can create a stack as

```
#include <iostream>
#include <deque>
using namespace std;
template <typename T>
 class Stack
{
private:
  deque<T> stack; //internal representation of a stack
public:
  //return top element of a stack
  const T& top(){ return stack.front(); }
  //push an element to the stack
  void push(const T& val){ stack.push_front(val); }
  //pop an element from the stack
  void pop(){ stack.pop_front(); }
  //check if stack is empty
  bool isEmpty(){ return stack.empty(); }
  //return size of a stack
  int size(){ return stack.size(); }
  const T& bottom(); //this function is defined outside the class
};
```

The member function definitions of a class template are *function templates*, but are not preceded with the **template** keyword and template parameters in angle brackets when they are defined within the class templates body. They do use the class template parameter (for this case **T**) to represent the element type. The constructor (default) constructor of the class template invokes the deque constructor.

1.5 Defining member functions outside the class template definition

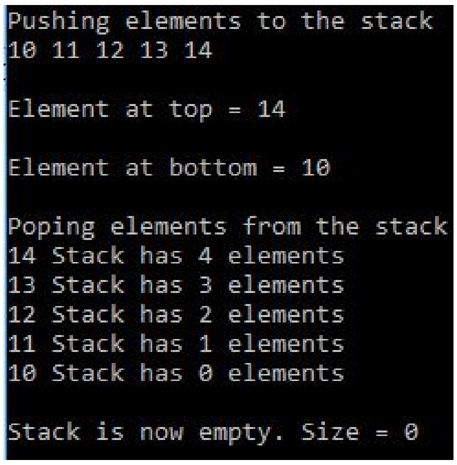
Member function can be defined outside class template definition. If you do this, each function must begin with the template keyword followed by the same set of of template parameters as the class template. The member function must be qualified with the class name and scope resolution operator. For example, function **const T& bottom()** can be defined as

```
template <typename T>
 inline const T& Stack<T>::bottom(){
  return stack.back();
}
```

The main function to test the stack class template is

```
int main()
{
  Stack<int> s;
  cout << "Pushing elements to the stack\n";
  for (int i = 10; i < 15; i++)
  {
    s.push(i);
    cout << s.top() << " ";
  }
  cout << "\n\nElement at top = " << s.top();
  cout << "\n\nElement at bottom = " << s.bottom();
  cout << "\n\nPoping elements from the stack\n";
  while (!s.isEmpty())
  {
    cout << s.top() << " ";
    s.pop();
    cout << "Stack has " << s.size() << " elements \n";
  }
  cout << "\nStack is now empty. Size = " << s.size() << endl;
  return 0;
}
```

which gives the output



```
Pushing elements to the stack
10 11 12 13 14

Element at top = 14

Element at bottom = 10

Poping elements from the stack
14 Stack has 4 elements
13 Stack has 3 elements
12 Stack has 2 elements
11 Stack has 1 elements
10 Stack has 0 elements

Stack is now empty. Size = 0
```

From the output, you can notice that the first element to be pushed into the stack is indeed at the bottom of the stack and the last element is at the top of the stack.

1.6 list Sequence Container

The **list** sequence container (from header **list**) allows insertion and deletion operations at any location in the container. If many insertions and deletions occur at the ends of the container, the **deque** data structure provides a more efficient implementation.

Class template **list** is implemented as a *doubly linked list* – every node in the list contains a pointer to the previous node in the list and to the next node in the list. This enables class list to support bidirectional iterators that allow the container to be traversed both forward and backward.