6.096 Problem Set 4

1 Additional Material

1.1 Templates and Header Files

The compiler does not compile a templated function until it encounters a use of it – until that function is used with a particular type parameter, at which point it is compiled for that type parameter. Because of this, if you define a templated class in a header file and implement its (templated) functions in a .cpp file, the code in the .cpp file will never get compiled unless you use the templated functions within that .cpp file. To solve this problem, templated classes and functions are generally just implemented in the header file, with no .cpp file.

1.2 Templates and `friend` Functions

There are some syntax oddities when using friend functions with templated classes. In order to declare the function as a friend of the class, you need a function prototype (or the full function definition) to appear before the class definition. This is a problem, because you'll often need to have the class already defined in order for the function prototype to make sense. To get around this issue, when writing friend functions with templated classes, you should include declarations in the following order:

1. A templated *forward declaration* of the class. (A forward declaration is a statement like `class SomeClass;`, with no class body, that just alerts the compiler that the class definition is coming.)

2. A templated function prototype for the friend function (or the entire function definition).

3. The full templated class definition, including the `friend` statement. When you write the name of the function in the `friend` statement, you need to include an extra `<>` after it to indicate that it is a templated function (e.g., `friend MyClass operator+<>(const MyClass &c1, const MyClass &c2;)`.

4. The operator function definition, if you didn't include it above.

2 Multi-Type `min`

2.1

Using templates, implement a `min` function which returns the minimum of two elements of any comparable type (i.e., it takes two arguments of some type `T`, and works as long as values of type `T` can be compared with the `<` operator).

(To test this function, you may need to omit your usual `using namespace std;` line, since there is already an `std::min` function.)

2.2

Implement the `min` functionality from part 1 using only preprocessor macros. (*Hint:* You will probably need the ternary operator – the `?:` syntax.)

3 Casting

Assume you implemented Problem 5 from Problem Set 3 correctly. This would mean you would have a working `Polygon` class, and inheriting from that a `Triangle` class and a `Rectangle` class. Now imagine you have a pointer declared as `Rectangle *rect;` that has been properly initialized.

3.1

Write a line of code showing how you would cast `rect` to a `Triangle *` without checking for type correctness (i.e., without checking whether it actually points to a `Triangle`). Do not use C-style casts.

3.2

Now write a line of code that does the same thing, but checks for type correctness and throws an exception or returns a null pointer if `rect` does not actually point to a `Triangle`.

4 Templated Stack

A *stack* data structure stores a set of items, and allows accessing them via the following operations:

- *Push* – add a new item to the stack
- *Pop* – remove the most recently added item that is still in the stack (i.e. that has not yet been popped)

- *Top* – Retrieve

For more explanation, see http://en.wikipedia.org/wiki/Stack_(data_structure).

4.1

Using templates, implement a `Stack` class that can be used to store items of any type. You do not need to implement any constructors or destructors; the default constructor should be sufficient, and you will not need to use `new`. Your class should support the following 3 public functions (assuming `T` is the parameterized type which the `Stack` is storing):

- `bool Stack::empty()` – returns whether the stack is empty
- `void Stack::push(const T &item)` – adds `item` to the stack
- `T &Stack::top()` – returns a reference to the most-recently-added item
- `void Stack::pop()` – removes the most-recently-added item from the stack

Use an STL `vector`, `deque`, or `list` to implement your `Stack`. You may not use the STL `stack` class. Make the member functions `const` where appropriate, and add `const`/non-`const` versions of each function for which it is appropriate. In the case where the `Stack` is empty, `pop` should do nothing, and `top` should behave exactly as normal (this will of course cause an error).

When working with templated classes, you cannot separate the function implementations into a separate .cpp file; put all your code in the class definition.

(*Hint:* You can represent pushing by adding to the end of your `vector`, and popping by removing the last element of the `vector`. This ensures that the popped item is always the most recently inserted one that has not yet been popped.)

4.2 `friend` Functions and Operator Overloading (Optional)

Make a `friend` function that implements a `+` operator for `Stacks`. The behavior of the `+` operator should be such that when you write `a + b`, you get a new stack containing `a`'s items followed by `b`'s items (assuming `a` and `b` are both `Stacks`), in their original order. Thus, in the following example, the contents of `c` would be the same as if 1, 2, 3, and 4 had been pushed onto it in that order. (`c.top()` would therefore return the value 4.)

```
1 Stack<int> a, b;
2 a.push(1);
3 a.push(2);
4 b.push(3);
5 b.push(4);
6 Stack<int> c = a + b;
```

Put your code for this section in the same file as for the previous one.

5 Graph Representation (Optional)

A *graph* is a mathematical data structure consisting of *nodes* and *edges* connecting them. To help you visualize it, you can think of a graph as a map of "cities" (nodes) and "roads" (edges) connecting them. In an *directed graph*, the direction of the edge matters – that is, an edge from A to B is not also an edge from B to A. You can read more at Wikipedia: http://en.wikipedia.org/wiki/Graph_(mathematics).

One way to represent a graph is by assigning each node a unique ID number. Then, for each node ID n, you can store a list of node ID's to which n has an outgoing edge. This list is called an *adjacency list*.

Write a `Graph` class that uses STL containers (`vector`s, `map`s, etc.) to represent a directed graph. Each node should be represented by a unique integer (an `int`). Provide the following member functions:

- `Graph::Graph(const vector &starts, const vector &ends)`
  Constructs a `Graph` with the given set of edges, where `start`s and `end`s represent the ordered list of edges' start and endpoints. For instance, consider the following graph:

  The `vector`s used to initialize a `Graph` object representing this graph would be:

`start:`	1	1	1	5	5	4
`end:`	2	3	4	4	2	2

- `int Graph::numOutgoing(const int nodeID) const`
  Returns the number of outgoing edges from `nodeID` – that is, edges with `nodeID` as the start point

- `const vector<int> &Graph::adjacent(const int nodeID) const`
  Returns a reference to the list of nodes to which `nodeID` has outgoing edges

(*Hint:* Use the following data type to associate adjacency lists with node ID's: `map<int, vector<int> >`. This will also allow for easy lookup of the adjacency list for a given node ID. Note that the `[]` operator for `map`s inserts the item if it isn't already there.)

The constructor should throw an `invalid_argument` exception of the two `vector`s are not the same length. (This standard exception class is defined in header file `stdexcept`.) The other member functions should do likewise if the nodeID provided does not actually exist in the graph. (*Hint:* Use the `map::find`, documented at http://www.cplusplus.com/reference/stl/map/find/.

6.096 Introduction to C++

January (IAP) 2011