Lab A

Case Study – Using Template Classes

This set of exercises is a case study that illustrates a number of features of C++ working in combination. You will work with the **IntStack** class, converting it into a template class **Stack** that can handle a variety of data types, including **String**. The proper functioning of the stack class and the test program when run with **String** objects relies on some of the features built into the **String** class for construction and conversion. The trace statements in the **String** class code have been retained to help you see the inner workings.

Suggested Time: 30 minutes

Exercise 1

You are given as a starting point an integer stack class **IntStack** and a test program. The files for the stack class are named **stack.h**, and **stack.cpp**. In the test program **tststack.cpp** the commands are "ipush", "ipop" and "iprint" (for the integer stack). You will convert the class to a template class, and modify the test program to use the new class, instantiating a stack object with **int** as the data type.

1. Review then build and run the test program.

2. Modify the stack class definition so that it is a template class called **Stack** with one class parameter **T**.

3. Modify the stack class implementation to implement the template class.

4. Modify the test program (**tststack.cpp**) to create a stack object from the template class, using **int** as the data type parameter. Build and run your program.

Exercise 2

In this exercise you will modify your test program to create a second stack object which holds **String** items. You will add three new commands ("spop", "spush", and "sprint") so that your program can exercise the string stack as well as the integer stack.

1. Add an include of the **String** class header file **strn.h** to the test program (**tststack.cpp**), and add the **String** implementation file **strn.cpp** to the project.

2. Add code to instantiate a stack object, passing **String** as the data type.

3. Add code for the three commands ("spop", "spush", and "sprint") that exercise the string stack. You will have to handle conversions between **char** pointers used for I/O and **String**.

4. Build and run the program. Make sure you understand all the trace output of constructors, conversions, and destructors.

Exercise 3

Our implementation of the stack class uses a fixed size array, which is not flexible, because all stacks must have the same size. We could provide another implementation using dynamic memory allocation, creating storage for the stack on the heap. Templates offer another possible implementation. Besides having class types as arguments, templates can take ordinary data types as arguments as well. In this exercise you will add a second parameter to the template specification which will be an **int** that is used for the size of the array. Thus dynamic memory allocation is not used, but we can instantiate stack objects of different sizes as well as different data types.

1. In the file **stack.h** defining the stack class, remove the **const** definition of the symbol **STACKSIZE** and instead use it as a second parameter in the template specification. Note that this parameter can have a default value.

2. Add this second parameter to the template specification before each of the functions and in the use of **Stack** for scope resolution.

3. Modify the test program to create a stack of **String** items of size 3.

4. Build and run your program. Note that now only three **String** constructors are invoked when the string stack object is created.

Lab B

Introduction to STL Programming

In this lab you will write a word dictionary using the STL map container.

Suggested Time: 15 minutes

Instructions

1. Open the starter project.
2. Declare an instance of a map container inside the main routine. Since this a word dictionary, the type of both key and the value will be the STL string class. As a dictionary in alphabetic order, the comparison function object should be less.
3. Populate the dictionary with words of your choosing by using associative array property (i.e. the overloaded [] operator).
4. Inside the input loop lookup the definition of any word (use std::map.find()).

Lab C

STL Programming

In this lab you will filter and sort a vector of Rational numbers.

Suggested Time: 20 minutes

Instructions

1. Open the starter project and review the Rational class. There are several operator overloads, including < and > which can be used for sorting. The include for vector, iterator, and algorithm have been added.

2. There exists an empty instance of a vector container **w** inside the main routine followed by a for loop to output the filtered and sorted results.

3. Implement a function **FillterWholeNumbers** which can be used to determine if a Rational is whole.

4. Using std::copy_if filter all values of **v** where the rational value is a whole number, i.e. numerator mod denominator is zero. Tip: In the call to copy_if use std::back_inserter to populate **w** and **FillterWholeNumbers** as the predicate. Test your changes.

5. Use std::sort to sort w in either ascending or descending order. Use std::less<Rational> for ascending or std::greater<Rational> for descending. Test your changes.

Lab D

Smart Pointers

In this lab you will modify a program so that it implements a vector of unique pointers to objects.

Suggested Time: 20 minutes

Instructions

1. The starter project is a copy of the polymorphism sample from chapter 10. Notice the program uses an array of pointers to const Employee. The function PayReport accepts a pointer to a pointer, which matched the actual argument passed in the call. The last important feature is the loop to delete the instances which have been allocated on the heap.

2. Delete the code in main().

3. Include the <vector> header and the <memory> header, which holds the definitions for smart pointers

4. In main() create a vector named vEmp. The vector will hold elements of type unique_ptr<const Employee>.

5. Use push_back on the vector instance to add new instances of Sally, Wally, and Sue. You will need to typecast the arguments to conform to the vector definition.

6. Create a loop that iterates over the vector, output the name and pay of each employee. Run the program and verify results.

7. Comment out the loop you ctreated. Without making any changes to the PayReport function call PayReport with the appropriate actional arguments. Tip: use the vector::size and vector::data methods. Run the program and verify results.