CSCI 262 Data Structures

Spring 2019

Lab 12: Inheritance

(Back to Labs)

Goals

- Understand the basics of inheritance

Overview

This lab will walk you through some exercises concerning inheritance. As usual, we ask that you work with a partner or a team. If you end up solo, please raise your hand and we'll pair you with someone else (or make a three-person team if necessary). You are welcome to change partners throughout the semester!

Instructions

Step 0: go ahead and create a new project for this lab. Download lab12a.zip and unpack to the directory of your choice. Import the files from lab12a.zip into your project. Note there is just one .cpp file, which contains a `main()` function, and two header files. As provided, the code should compile and run.

Step 1: simple inheritance

Spend some time looking through the code provided. There are two classes provided, `int_counter` and `mod_counter`. Start with `int_counter`. The `int_counter` class models a very simple object which keeps track of how many times the object has received an `increment()` command. There is a default constructor, which simply initializes the object to start at a count of zero. There is also a `reset()` method, which can be used to reset the counter to zero. Finally, there is a `to_string()` method to obtain a printable representation of the counter's current state.

Take a look now at `mod_counter`. This class models a counter which rolls over at a given value, returning to zero - "mod" is short for "modulus" in this case - the counter represents the number of times the counter has received the `increment()` command, mod some value. You can see that `mod_counter` inherits from `int_counter`. The `mod_counter` class provides a constructor which requires a parameter (the modulus at which to roll over), and overrides the `increment()` method. The other two methods are simply inherited, as the inherited behavior is appropriate for those methods.

Taking another look at `int_counter`, you may notice that we've used a visibility that we've previously not discussed: `protected`. The reason we haven't discussed it until now is that the protected visibility only has meaning in the context of inheritance. The difference between protected and private is that private members are **not** visible to the inheriting (child) classes, while protected members remain protected in the child classes. Because `mod_counter` needs to examine and update the `_value` instance variable, it was necessary to make it protected; alternately, of course, we could have created setters and getters and used those from the child class. (Or made everything public, but that would make it possible for a user to modify the `_value` to some invalid state for the `mod_counter` - e.g., some value greater than the modulus.)

Finally, note that `mod_counter` has its own private variable to hold the modulus at which the counter will roll over.

If you fire up the program and run it, you can see the two kinds of counter in action; the `main()` function creates pointers to one of each of the types of counter; the `mod_counter` object is initialized with a modulus of 7.

Take some time to play around with the code, keeping in mind the lecture we recently had on inheritance and polymorphism. In particular, try the following (don't worry about messing up the code - we're going to throw it away in a moment anyway):

- What happens if you modify `int_counter` to eliminate the use of the `virtual` keyword? Does it matter for all the methods, or just some?
- What happens if you use an array of `int_counter`, rather than an array of pointers to `int_counter`?

Step 2: abstract base class

Go ahead and discard all of the code used in step 1, or else create a new project. Then download lab12b.zip and add all of its files to your project.

Exploring this code base, you can see we've added a couple of classes. One is simply called `counter`, and is now the base of our class hierarchy; we've modified `int_counter` to inherit from `counter`. The other new class is called `bit_counter`, and it also inherits from `counter`.

The `bit_counter` class is just a fun way of showing binary counting, for various widths of binary numbers. Essentially you construct it with a particular bit width, and when it is converted to a string, it represents its count as a binary number with that many digits. The count is actually stored as a string of zeros and ones, rather than as an integer: theoretically, it can represent **much** larger values than an `int` - just use a very large width. It isn't terribly important that you examine this code in any depth, but feel free!

Pay special attention to the `counter` class. Notice that all the methods in this class are pure virtual, making this class an *abstract* class. You can verify for yourself that you cannot create any `counter` objects. However, we are going to use the fact that all of the counter objects ultimately inherit from `counter` to demonstrate polymorphic behavior in `main()`.

The main() function declares an array of 5 `counter` pointers; the first two are bit counters of different widths, next is an integer counter, followed by a mod counter and another integer counter. Note that we are simply looping over the array of counter objects, not worrying or caring what type is actually stored on the other end of the pointer; we simply execute `to_string()` and `increment()` on each counter. We are guaranteed that these methods exist, if we can instantiate the objects at all, because the methods were declared in the common base class `counter`.

Step 3: your contribution

For the final step, we ask that you create another counter type: a "bounded" integer counter. This counter will count up to a fixed bound, and then go no further. That is, if its bound is 100, the counter will start at zero, count up to 100, and then stay on 100 forever (unless `reset()` is called).

Call your new class `bounded_counter`, and have it inherit from `int_counter`. (Do everything inline, so you only need a bounded_counter.h file and no additional .cpp file.) Decide which methods need to be overridden, and what additional members you need. It should only be possible to construct a `bounded_counter` with a specified bound - there should be no default constructor.

You can test your new class by modifying main.cpp; replace the last `int_counter` object in the array with a `bounded_counter` object constructed with a bound of 10. Then run the code; you should get the following output:

```
00000000  0000  0  0  0  Enter to continue, q to quit
00000001  0001  1  1  1  Enter to continue, q to quit
00000010  0010  2  2  2  Enter to continue, q to quit
00000011  0011  3  3  3  Enter to continue, q to quit
00000100  0100  4  4  4  Enter to continue, q to quit
00000101  0101  5  5  5  Enter to continue, q to quit
00000110  0110  6  6  6  Enter to continue, q to quit
00000111  0111  7  0  7  Enter to continue, q to quit
00001000  1000  8  1  8  Enter to continue, q to quit
00001001  1001  9  2  9  Enter to continue, q to quit
00001010  1010  10  3  10  Enter to continue, q to quit
00001011  1011  11  4  10  Enter to continue, q to quit
00001100  1100  12  5  10  Enter to continue, q to quit
00001101  1101  13  6  10  Enter to continue, q to quit
00001110  1110  14  0  10  Enter to continue, q to quit
00001111  1111  15  1  10  Enter to continue, q to quit
00010000  0000  16  2  10  Enter to continue, q to quit
00010001  0001  17  3  10  Enter to continue, q to quit
...
```

Submission instructions: create a .zip file containing your bounded_counter.h file, a README containing your name and your partner's name (and anything else you want to tell us), and submit the .zip file to Canvas.

Grading:

Correctly functioning `bounded_counter` class	9 points
README	1 point
Total:	**10 points**