1 Lab 4

Date: Feb 20, 2020

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a `#` character to end-of-line.

1.1 Aims

The aim of this lab is to introduce you to object-oriented programming in C++. After completing this lab, you should have some familiarity with the following topics:

- The use of `virtual` functions in C++ and the overhead associated with using them.
- Implementing abstract classes in C++.
- The use of prefix notation for representing arithmetic expressions without needing to use parentheses.

1.2 Exercises

This lab has three exercises. The first shows that C++ achieves its aims of being a better C by making the association of related data and code explicit, without introducing any overhead. The second exercise introduces object-oriented programming in C++. The final exercise asks you to essentially repeat the previous exercise but with less help.

The provided Makefile will build an executable called `main` in the current directory (which is usually a directory for the individual exercises). It assumes that there is a `main.cc` and will build the `main` executable from all the `*.cc` files in the directory. It automatically tracks dependencies. It also generates a `.gitignore` file so that useless files do not get committed to github.

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab4` branch and a new `submit/lab4` directory. Start a `script` session to log your interaction into a `lab4.LOG` file.

Copy all the lab4 exercises into your `submit/lab4` directory by copying the contents of the `~/cs240/labs/lab4/exercises`:

```
$ cd ~/i240?/submit/lab4
$ cp -r ~/cs240/labs/lab4/exercises .
```

1.2.2 Exercise 1: Encapsulating Code With Data

Change over to the `./exercises/1-point` directory.

```
$ cd exercises/1-point
$ ls -l
```

You should see that the directory contains a `point.cc` file which implements 2D points as per the specification in the header file `point.hh`. It also contains a `main.cc` driver program. Look at these source files. Read the comments and understand the code.

Simply type `make -f ../Makefile` in that directory to build the program. The `-f` option tells it to use a `Makefile` other than that in the current directory; in this case, we are using a `Makefile` from the parent directory.

It should compile a `main` executable without any errors or warnings. It should run successfully. The output has two sections:

The first section gives the pair-wise distance between the different points in the `points[]` global in `main.cc`. You should quickly validate some of the distances.

The second section is more interesting. It prints out the size of a `double` as well as the size of a `Point`. Since a `Point` contains two doubles, the output shows that there is **zero** memory overhead for a `Point`. Basically, the `toString()` and `distance()` functions are just like regular C functions except:

- They have "funny" names `Point::toString` and `Point::distance` In fact, these are the names used when they are defined in `point.cc`.
- They are invoked with the same syntax as accessing a field of a `Point`. If `p1` is a `Point`, we access its data member `x` using `p1.x`; similarly, we run its member function `toString()` using `p1.toString()` and find its distance from `Point p2` using `p1.distance(p2)`.

Once you are sure you understand the code, make changes to add a `Line` class which contains two `Point`'s. It should support a `length()` member function which returns the length of the line as well as a `toString()` method which returns a string representation of a line consisting of the string representation of the first end-point concatenated with the string `" -- "` concatenated with the string representation of the second end-point.

Hints:

- The length of a line is simply the distance between its end-points.
- `std::string` objects can be concatenated using `+`.

Change the driver code in `main.cc` to output the lengths of the lines defined by pairs of points in the `points[]` array: i.e. `points[0] -- points[1]`, `points[2] -- points[3]`, ... Your code should work irrespective of the length of the `points[]` array. In particularly, you should ensure that your program behaves "reasonably" when there are an odd number of `Point`'s in `points[]`. You can comment out the calls to `outDistances()` and `outSizes()` in `main()`.

With the provided `points[]` array, your output should be something like:

```
(0, 0) -- (3, 4) length = 5
(1, 1) -- (4, 5) length = 5
```

1.2.3 Exercise 2: Object-Oriented Programming in C++

The `exercises/2-shape` directory contains `tostring`, `point` and `shape` modules along with a `main.cc` driver program. Look at the corresponding files in that order; be sure to read the comments to understand the code.

Compile the program using the `Makefile` in the parent directory. When you run the program, look at the output closely to verify the following:

- The `main()` function only contains a single call to `shapeP->perimeter()` with `shapeP` being a pointer to a `Shape`. We see from the output that the correct function gets called automatically: `Circle::perimeter()` if `shapeP` points to a `Circle`, `Rectangle::perimeter()` if `shapeP` points to a `Rectangle`. This is referred to as **runtime dispatch** or **runtime polymorphism**.
- Recollect that in the previous exercise, `Point` had a size of 16 bytes but its size has now increased to 24 bytes. The main difference between this exercise's `Point` and that in the previous exercise is the inheriting of the **virtual** `toString()` function. Hence the memory increase represents the overhead resulting from the implementation of **virtual** functions.

  Look at the memory sizes of all the other objects. In each case, you see 8 extra bytes due to the overhead necessary to support virtual functions. For example, `ToString` and `Shape` do not contain any data, but have a size of 8 bytes. `Circle` contains 32 bytes of data (24 bytes for the virtualized `origin Point` and 8 bytes for the `radius`), but has size 40. Finally, `Rectangle` contains 48 data bytes (two virtualized `Point`'s), but has size 56.

  In all cases, these 8 extra bytes result from each instance of a class containing a pointer to a shared table of pointers to the **virtual** functions for that class. This table is referred to as the **virtual table** or **vtable** and the pointer in the object instance is referred to as the **vtable pointer**.

After understanding the code, add a suitable `area()` method to all the `Shape` classes. Modify the driver program to have each output line also show the area

of the shape.

1.2.4 Exercise 3: Evaluating Prefix Arithmetic Expressions

The normal notation used for arithmetic expressions is **infix** notation, where the operator appears in-between the operands. This is easy to read, but necessitates the use of precedence and associativity rules as well as the use of parentheses to override the default. For example, `1 + (2 * 3)` can be written simply as `1 + 2 * 3` because `*` has higher-precedence than `+`. OTOH, the parentheses in `(1 + 2) * 3` cannot be removed without changing the meaning.

An alternate is to use a **prefix** notation where the operator precedes its operands. Examples of prefix expressions followed by the equivalent infix expression:

- Prefix `+ 1 * 2 3` is infix `1 + 2*3`.
- Prefix `* + 1 2 3` is infix `(1 + 2)*3`.
- Prefix `+ / 40 5 - 3 2` is infix `40/5 + 3 - 2`.

Prefix notation works because each of the operators takes exactly 2 operands (we cannot use unary minus with prefix notation).

In Exercise 3, you will write code for evaluating prefix arithmetic expressions involving integers and the binary arithmetic operators `+`, `-`, `*` and `/`. Actually, all the code needed to extract the structure of an expression from a string has been provided to you. This structure has been represented using C++ classes which use virtual functions. You are required to add code to evaluate the value of the arithmetic expression represented by this internal structure.

Compile the code in the Exercise 3 directory `exercises/3-expr`. You can run the `main` executable with command-line arguments representing prefix arithmetic expressions. For each command-line argument, the program will print out its equivalent fully-parenthesized infix expression:

```
$ ./main '+ 1 * 2 3'  '* + 1 2 3'  '+ / 40 5 - 3 2'
(1) + ((2) * (3))
((1) + (2)) * (3)
((40) / (5)) + ((3) - (2))
$
```

Look at the source files. The `tostring` module is unchanged from the previous exercise. `main.cc` is a trivial driver program which simply calls a parser (from the `parse` module) to extract the internal structure of a prefix arithmetic expression contained in a command-line argument. The internal structure is represented as an instance of an `Expr` class with code in the `expr` module. The

main program simply prints out the string representation of the internal structure. This string representation is built by `Expr::toString()` which produces the fully-parenthesized infix representation.

Note the use of smart pointers for representing `ExprPtr`'s. This makes memory management trivial.

What you are required to do in this exercise is to add an `Expr::eval()` method which evaluates the value of the expression represented by the instance on which it is called. Modify the `main.cc` driver to print out the evaluated expression so that the output will look like the following:

```
$ ./main "+ 1 2" "- 3 4" "* 5 6" "/ 77 8"
(1) + (2) = 3
(3) - (4) = -1
(5) * (6) = 30
(77) / (8) = 9
$ ./main "* + 1 2 4" \
         "- * 3 4 5" \
         "/ * 5 6 10" \
         "/ 77 + 8 5"
((1) + (2)) * (4) = 12
((3) * (4)) - (5) = 7
((5) * (6)) / (10) = 3
(77) / ((8) + (5)) = 5
$ ./main "+ + + / 22  6 * 5 3 - 2 5 * 3 5"
((((22) / (6)) + ((5) * (3))) +
   ((2) - (5))) + ((3) * (5)) = 30
$
```

[Newlines have been added in the output above for formatting reasons].

Hints:

- The code for `Expr::eval()` is similar but simpler than the code for `Expr¬ ::toString()`.
- The code for `Expr::eval()` cannot easily be factored out into a common function like the function like `binaryExprToString()` used for implementing `Expr::toString()`.
- The evaluation of an `IntExpr` is simply the contained `value`.
- The evaluation of a binary expression is the result of applying the expression's operator to the evaluation of its operand expressions.

1.2.5 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab4.LOG` in your `lab4` directory. Add all your files to git and commit. Then merge your `lab4` branch into the `master` branch and commit your changes.

1.3 References

The following are links to reference material. Look around for tutorial material with which you may be more comfortable.

Online *<https://en.cppreference.com/w/>*.

Wikipedia article on *Virtual Functions*.

Virtual functions.