1 Lab 5

Date: Feb 27, 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 give you more exposure 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++.
- The use of stack machines for evaluating arithmetic expressions.

1.2 Exercises

This lab has two exercises. The first is a very simple illustration of object-oriented programming in C++. The second exercise continues the final exercise from your *previous lab*.

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 `lab5` branch and a new `submit/lab5` directory. Start a `script` session to log your interaction into a `lab5.LOG` file.

Copy all the lab5 exercises into your `submit/lab5` directory by copying the contents of the `~/cs240/labs/lab5/exercises`:

```
$ cd ~/i240?/submit/lab5
$ cp -r ~/cs240/labs/lab5/exercises .
```

1.2.2 Exercise 1: Using Virtual Functions

Change over to the *<./exercises/1-animal>* directory.

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

You should see that the directory contains a `animal.hh` file which implements simple classes giving a taxonomy of animals. The `Animal` class is abstract because it contains a **pure virtual function** `says()`. Hence it cannot be instantiated.

All of our `Animal`'s have `name`'s and that is reflected in the fact that an `Animal` is always created with a `name` field. We have `Dog`, `Cat` and `Cow` as sub-classes of `Animal`. Their constructors call the `Animal` constructor in their initialization lists.

Each `Animal` sub-class provides an implementation for the `says()` method. Since no other methods remain unimplemented, the `Animal` sub-classes are concrete and can be instantiated.

Simply type `make -f ../Makefile` in the `1-animal` 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.

The compilation should fail with an error. Figure out why and fix the problem.

Once you have fixed the problem, the compilation should succeed with building a `main` executable. It should run successfully and have each animal make its trademark sounds.

Dogs eat meat, cats eat fish and cows eat grass. Add an `eats()` method to all the animal classes which returns the kind of food each animal eats. Then modify `main()` so as to have it output what each animal eats, rather than what each animal says.

1.2.3 Exercise 2: Extending Expressions

The *<./exercises/2-expr>* directory contains a solution to the last exercise in the previous lab.

Compile the program using the `Makefile` in the parent directory. When you run the program, it should produce the output required for the previous lab.

A stack machine is one where all memory is accessed as a stack using instructions which operate only on the stack. For example, the *Java Virtual Machine* used for running Java is organized as a stack machine.

The dc(1) program is a basic stack machine. A sample session with `dc` is shown below:

```
$ dc
1 p c
1
f
1 2 3 * + p c
7
4 5 * 8 4 / + p c
22
$
```

The above input to `dc` contains the following commands:

integers Pushes the value of the integer onto the top of the `dc` stack.

operators `+`, `-`, `*`, `/` Pop the top two elements from the stack and push onto the stack the result of applying the operator to the two operands popped off the stack.

`p` Print the value on top of the stack followed by a newline.

`c` Clear the stack.

Make additions to the expression classes to output code for `dc`. Specifically, add a method `std::string dcCode()` to each expression class to produce `dc` code for that expression. This is fairly straightforward:

- The `dc` code for an `IntExpr` is simply the value of the contained integer.
- The `dc` code for an `AddExpr`, `SubExpr`, `MulExpr`, `DivExpr` is the concatenation of the `dc` code for the left operand, the `dc` code for the right operand followed by the appropriate operator `+`, `-`, `*` or `/` respectively.
- The `dc` code for the top-level expression should be followed by `p` to have `dc` print the value on top of its stack, followed by a `c` to have `dc` clear out its stack.

Modify the `main()` program to have it output **only** lines containing the `dc` code for each expression argument.

So for example, running the modified program on the last example given in the previous lab should result in the following:

```
$ ./main  "+ + + / 22  6 * 5 3 - 2 5 * 3 5"
22 6 / 5 3 * + 2 5 - + 3 5 * + p c
$
```

Of course, we can pipe the `dc` code generated by our program into `dc`:

```
./main "- 3 2" "* + 1 2 5" | dc
1
```

```
15
$ ./main  "+ + + / 22  6 * 5 3 - 2 5 * 3 5" | dc
30
$
```

1.2.4 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab5.LOG` in your `lab5` directory. Add all your files to git and commit. Then merge your `lab5` 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.