UNIVERSITY OF CALIFORNIA, MERCED

BERKELEY · DAVIS · IRVINE · LOS ANGELES · MERCED · RIVERSIDE · SAN DIEGO · SAN FRANCISCO SANTA BARBARA · SANTA CRUZ

CSE30: Data Structures
Lab1

1 Introduction

In this lab you will become acquainted with the CodeSync development environment, you will explore the structure of a proper C++ project (a real project is never just one file), you will write some basic C++ programs, and you will write some unit tests for your code.

2 CodeSync

If you are reading this, you are most probably reading it in CodeSync, which means you have found your way here. CodeSync is divided into four major areas, most of which are self-explanatory. On the left sidebar there should be a listing of the files and folders in the project, which is similar to any other file explorer utility. The left sidebar, shows a list of participants who are currently logged into the project, at the top, and a chat message interface at the bottom, which participants can use to send messages to the group. Along the bottom area of the interface is a terminal window, which will be used to compile and run our programs, and finally, the majority of the area in the middle of the screen is where the files are displayed.

Depending on your access level in a given project, you may or may not be able to edit the files or use the terminal. In class demos, your access level is read-only, so you can look but you can not make any modifications. For your lab projects, you have a higher access level so you are allowed to do anything in the project, until the deadline, at which point the system automatically demotes your access level and you will no longer be able to make any modifications. That is why it is imperative that you complete your labs on time.

3 The C++ Project Structure

If you look at the sidebar, you will see that our C++ project is made up of a number of folders. These are `bin`, `dep`, `inc`, `obj`, `scratchpad`, `src`, and `test`. The majority of our time will be spent in the `src`, `inc`, and `test` folders. There are also two files, namely `.gitignore`, and `Makefile`. You should never need to edit those two files, but their purpose and function will be explained as later in the course.

The `src` folder contains the source code for the project (most of it anyway). There will typically be a file called `app.cpp` located in the `src` folder, and that will be where we have defined our main function, which is the function that is called first when the program is executed.

The `inc` folder contains what are known as header files, which are just files that contain functions that we can use in our program, that is to say, any function in a header file from the `inc` folder, can be used in our app.cpp file. You may be wondering why it's necessary to separate these into different files, but trust me, there is a very good reason, which we will talk about later. What we are doing now is just starting off on the right note. At present there are two files in the inc folder, namely RandomSupport.h, and TimeSupport.h. There are useful functions in there for things like generating (good) random numbers, and for measuring the running time of pieces of our code (we have done both of these in our first class demo). In addition to these two files, you will sometimes be asked to create your own ones, as it is the case in this lab.

The test folder is where we write our unit tests, which is basically how we ensure that the code we have written is correct. For unit tests we use a framework called igloo, the code of which can be found in the dep folder. You never need to edit any files in the dep folder since they are dependencies, meaning it's other people's code, so we should not modify it at all.

For completeness, the bin folder is where the compiler produces the executables we need. The obj folder is where the compiler produces the object code, which is an intermediary step in the compilation process that we do not need to worry about in this course. There will sometimes be files in the obj folder, so just leave them alone. Finally, the scratchpad folder is something that we will sometimes use. As the name implies, it is just another C++ source code file that we can use to try out things and just mess around, without contaminating our main source code file. Of course, any header files in the inc folder are also available in the scratchpad. That's actually part of the reason why putting functions in header files is useful. Now these functions can be used in multiple source code files, all we have to do is import them. Much better than copy pasting function code between files.

4 Running the code in CodeSync

Once again, if your access level allows this, you will be able to run the code you write in CodeSync. One thing to note about CodeSync is that the concept of saving files basically does not exist. That is because CodeSync saves your file after every keystroke, so you never need to. You will typically be writing code in the app.cpp file in the src folder (but sometimes you will also me making your own header files). When you are happy with the code you have written and you want to try it out, simply go to the terminal and type make. This will compile your code and produce an executable, provided that there were no errors. If there are errors, they will be displayed on the terminal. Once you have compiled correctly, you can run your code by typing make run on the terminal. Any input or output from your program will be through the terminal.

5 C++ Basics

This section provides some pointers to online resources that you can use to brush up on your C++, in case you are not familiar with the syntax. Fortunately, C++ is very similar in syntax to Java so you should be able to read it easily. The best place to learn C++ online is http://cplusplus.com/doc/tutorial/, which covers all the aspects of the language we will be using. For this week, we will be going up to functions, so please go over the following sections and make sure you are comfortable with the basics. Here are links to the specific sections:

Structure of a program	`http://cplusplus.com/doc/tutorial/program_structure/`
Variables and types	`http://cplusplus.com/doc/tutorial/variables/`
Constants	`http://cplusplus.com/doc/tutorial/constants/`
Operators	`http://cplusplus.com/doc/tutorial/operators/`
Basic Input/Output	`http://cplusplus.com/doc/tutorial/basic_io/`
Control Structures	`http://cplusplus.com/doc/tutorial/control/`
Functions	`http://cplusplus.com/doc/tutorial/functions/`

Feel free to copy and paste any code you want from the tutorials linked above into your `scratchpad.cpp` file in the `scratchpad` folder, Please note that whatever is in the scratchpad has to be a complete C++ program. To compile your scratchpad, simply type `make scratch` in your terminal, and to run it, type `./bin/scratch`.

6 Creating Header Files

As you are aware, any real program makes use of code that was written before. Even a "Hello World" program in C++ requires you to include a header called `iostream`. Now `iostream` comes with the compiler that is installed, but sometimes we need to write our own headers. To do so, go to the `inc` folder in your CodeSync project and create a new file with the Create File button on located right under the project title, and above the files and folders. Call your file `Functions.h` and create it. One you have done that, it will open in a text editor in the main area of the CodeSync interface. When creating your own header files, always do this first: At the top of the file, before you write anything else, write the following 2 lines:

```
#ifndef Functions_h
#define Functions_h
```

Then at the end of the file, write the line

```
#endif
```

All the code you write in the file, should be between the `#define` statement and the `#endif` statement. It is extremely important that you do this, otherwise things start breaking, and it is difficult to diagnose the problem.

Now that we have created the file, it is time to define some functions in it. We will create a simple function that finds the maximum of two given integers. Start by declaring the function called `max`, which takes in two integers and returns an integer. The specification of the function's return type, and parameter types is called the *signature* of the function. Here is the `max` function:

```
int max(int x, int y){
    return x;
}
```

The more observant amongst you will point out that the function implemented above is not correct, and to you I say: "Don't worry about that right now". There is a good reason that I have left the function incorrect on purpose. In the case of this function it is very easy to get it right the first time, but in general other functions you write will not be so easy. So it is often desirable to start with an incorrect solution, and through the process described in the next section, correct the function to the degree that we are 100% sure that it is correct. I am talking of course about unit testing. We will always unit test our code, otherwise we can't trust that it works correctly. Donald Knuth, who is a pioneer in Computer Science was corresponding with a colleague, and he had sent them a program to try. Knuth included the following note "Beware of bugs in the above code; I have only proved it correct, not tried it." Basically, testing is important. It is very important. It is probably more important that writing the code itself.

7 Unit Testing

Recall from the previous section that we should have a file called `Functions.h` and it looks something like this:

```
#ifndef Functions_h
#define Functions_h

int max(int x, int y){
    return x;
}

#endif
```

We basically have a function `max` which we need to test for correctness. To do so, navigate to your `test` folder and open the file `test.cpp`. At present, it should look as follows:

```
#include <igloo/igloo.h>

using namespace igloo;

int main() {
// Run all the tests defined above
return TestRunner::RunAllTests();
}
```

It basically runs an empty suite of test cases, which is not too useful for anyone. Since we are planning on testing the `max` function, which is defined in `Functions.h`, we have to include the `Functions.h` file here, which we can do right under the line `#include<igloo/igloo.h>`.

Next, we need to define a suite of tests for our `max` function. We do so in the space right under the line `using namespace igloo`. We define a context, which is basically giving a name to the suite of tests we are performing. We can do something like this:

```
Context(MaxFunctionTests){

};
```

Don't forget the semi-colon after the closing brace of the context definition. Inside the context, we define specs, where each spec is basically an individual test case. We now need to think about what test cases would our function need to pass, in order to convince us that it works correctly. That is to say, we need to come up with input/output pairs that the function will be tested with.

One such pair is to provide 1, and 0, as inputs and expect the result to be 1, because indeed 1 is greater than 0. This is a good test case but if it was the only test we performed, we could easily have an incorrect function that passes it. In fact our `max` function, the way it is written now, would pass the test, even though we know it is incorrect. The current implementation of the `max` function always tells us that whatever number was given first is the maximum, because it always return `x` without doing any checking.

With that in mind, it would be good to write some tests, that would reveal this bug. The first thing we can do is to reverse the order of parameters passed into the function. So we pass in 0, and 1, and we expect the result to be 1. Do you think that a function that passes both of these tests is 100% correct?

Consider the following function:

```
int max (int x, int y){
    return 1;
}
```

Indeed, the function above would pass both of the tests above, but it is still incorrect. It always says the result is 1, no matter what. To address this issue, we need to add more tests. For example, we could add a third one where the inputs were 5, and 2, and we expect 5, and a fourth one where the inputs are 7, and 9, and the result is 9.

I hope you agree if our function passes all four of the tests above, we can be very confident that it is correct, which we could not have been without running these tests.

So then to implement the test suite, we go in the context we defined earlier and we write four specs. This is how they should look:

```
#include <igloo/igloo.h>
#include <Functions.h>

using namespace igloo;

Context(MaxFunctionTests){
    Spec(OneGreaterThanZero){
        Assert::That(max(1,0), Equals(1));
    }

    Spec(ZeroLessThanOne){
        Assert::That(max(0,1), Equals(1));
    }

    Spec(FiveGreaterThanTwo){
        Assert::That(max(5,2), Equals(5));
    }
    Spec(SevenLessThanNine){
        Assert::That(max(7,9), Equals(9));
    }
};

int main() {
// Run all the tests defined above
return TestRunner::RunAllTests();
}
```

To run the tests, you can type `make test` in the terminal, followed by `./bin/test`. Running the tests produces the following output:

```
..FF
MaxFunctionTests::SevenLessThanNine failed:
Expected: equal to 9
Actual: 7

MaxFunctionTests::ZeroLessThanOne failed:
Expected: equal to 1
Actual: 0

Test run complete. 4 tests run, 2 succeeded, 2 failed.
```

This means that it passed 2 tests, the ones where the larger number was given first, but it failed the others. Namely for the `SevenLessThanNine` test it was expecting 9, but it actually got 7. Similarly for the other test.

With this information, we have a better idea of how to implement the function, so we now go back to the `Functions.h` file and we fix our `max` function. We can replace the code with the following:

```
int max(int x, int y){
    if (x > y){
        return x;
    }
    else{
        return y;
    }
}
```

Re-running the tests again, with `make test` and `./bin/test`, we get the following output:

```
....
Test run complete. 4 tests run, 4 succeeded, 0 failed.
```

Getting the above, with the tests we have written is the only way we can have some confidence that the function works correctly.

8 Tasks

Complete the following tasks before the deadline:

1. The `even` function:
   - Create a function called `even` in your `Functions.h` header file. The `even` function should take in an integer and should return `true` if the integer is even, otherwise it should return `false`. The function does not have to be correct at this point. It only needs to take in an integer and return a boolean.
   - Create a `Context` in your testfile called `EvenFunctionTests`, and in it write as many specs as you need to convince yourself that if a function passes all of them, then it should be correct.
   - Correct your `even` function if necessary.
2. The `sum` function:
   - Create a function called `sum` in your `Functions.h` header file. The `sum` function should take in an integer and should return return the sum of all positive integers up to and including that integer. The function does not have to be correct at this point. It only needs to take in an integer and return a boolean.
   - Create a `Context` in your testfile called `SumFunctionTests`, and in it write as many specs as you need to convince yourself that if a function passes all of them, then it should be correct.
   - Correct your `sum` function if necessary.
3. The main program in `app.cpp`
   - Include your `Functions.h` file after `iostream`
   - In your `main` function, make a call to your even function to test whether 0 is an even number.
   - In your `main` function, use your `sum` function to add up all the integers up to and including 100. Print out the result.

9 Assessment

This lab is out of 100 points. They will be awarded as follows:

- 10 points for the correct implementation of the `even` function.
- 35 points for the unit tests of the `even` function.
- 10 points for the correct implementation of the `sum` function.
- 35 points for the unit tests of the `sum` function.
- 10 points for the function calls in the `app.cpp` file.

With your unit tests you need to demonstrate that you understand what it is that makes the function correct vs incorrect. If we can easily find an incorrect function that passes all your unit tests, you will be lose a lot of points.