Lab #2 - Introduction to C++

2.S998 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Contents

1 Overview and Objectives

This lab will introduce C and C++ to new users. It assumes nothing regarding C++ background, but does assume some familiarity with basic programming concepts in other programming languages such as for-loops, variable scope, conditional execution (if-then-else) for example.

1.1 More C++ Resources

We will only just touch the basics today. A couple further resources are worth mentioning for following up this lab with your own self-guided C++ exploration.
Text book:

- The text book I recommend is *Practical C++ Programming*, Steve Oualline, O'Reilly Publishers, 2003.

Web sites:

- The www.cplusplus.com web site has a lot of resources. In particular there are a number of good tutorials: `http://www.cplusplus.com/doc/tutorial`
- The standard template library has a number of useful tools ranging from basic tools that we'll use right out of the gate, like strings. Intermediate tools like vectors, maps, and sets. The following is great resource: `http://www.cplusplus.com/reference/stl`

1.2 What you will need to get started

You will need a machine that has:

- A text editor such as Emacs, vi or some ability to edit text files.
- The C++ compiler installed. In this lab, we assume the compiler is the executable `g++`, typically found at `/usr/bin/g++`. You may want to check that this is in your shell's path.

2 Structure of a Program

In this exercise we will follow the example in Listing 1 below, which can also be found at the URL listed at the top of the block. The web site is very good at explaining each component. I recommend reading through the entire page explaining this example.

Your goals in this part are:

1. Open an editor and enter the code block in Listing 1.
   Save the code in a file named `hello.cpp`.
2. Read through the explanation on the web site. (If you are new to C/C++, please do take the time to read this through.)
3. Build the program into an executable.
   On the command line do: `g++ hello.cpp`
   This should generate a file called `a.out`
4. Verify that it runs by running it.
   On the command line do: `./a.out`
5. NOTE: For those still getting familiar with the command-line and shell paths, the `"./"` in the above invocation is important. It tells your shell where to look for the executable `a.out`. Normally the shell looks only in the set of directories in its *path*. By default, the present working directory is not in the shell's path. The notation `"./"` is shorthand for the present working directory. The notation `"../"` refers to the directory above the present directory. Try running `"ls -a"` and you will see that both `"./"` and `"../"` are listed.
6. Rather than building the cryptic `a.out` executable, try building it to have a more meaningful name: `g++ -o hello hello.cpp`. Then you should be able to run the program with `./hello`

Example Code 1.

```
// Code example from: http://www.cplusplus.com/doc/tutorial/program_structure/
// my first program in C++
#include <iostream>

using namespace std;

int main ()
{
  cout << "Hello World!" << endl;
  return 0;
}
```

3 Command Line Arguments

In this exercise we will expand on the first example by adding the ability to handle command line arguments in our program. (This section does not correspond to a section on the cplusplus.com tutorial site.) A *command line argument* is an argument passed to the program on the command line. For example, it would be nice if our Hello World! program had the ability to accept an alternative message on the command line. For example:

```
$ ./hello Greetings!
```

In the `main()` function, there is the option of having two arguments as shown in Listing 2. The first argument is an integer, `argc`, indicating the number of command line arguments. The program itself is always the first argument, so the smallest this number could be is one. The second argument `argv` holds the actual values of all the command line arguments. (We'll punt for now on the notation `char**`, but in short it can be thought of as a pointer to an array of strings.) The first argument is `argv[0]`, the second argument, if it exists, is `argv[1]` and so on.

Your goals in this part are:

1. Open an editor and enter the code block in Listing 2.
   Save the code in a file named `hello_arg.cpp`.
2. Build the program into an executable.
   On the command line do: `g++ -o hello_arg hello_arg.cpp`
   This should generate a executable file called `hello_arg` in the same directory.
3. TIP: when you do an `"ls"` in your terminal to see if the `hello_arg` executable is there, try doing `"ls -F"` instead. This will append an asterisk to all executables making it easier to visually find what you're looking for. In OS X, try `"ls -FG"` to add some color. If you like this idea, consider making it the default for your shell, by putting `alias ls 'ls -FG'` in your `.cshrc` file or `alias ls='ls -FG'` in your `.bashrc` file if you're a bash user.
4. Verify that it runs by running it.
   On the command line do: `./hello_arg`.
5. Modify the program such that instead of the original output, it instead outputs a custom greeting. Your program should respond as follows

   ```
   $ ./hello_arg Greetings!
   $ Greetings!
   ```

Example Code 2.

```
// Handling command line arguments

#include <iostream>
using namespace std;

int main (int argc, char **argv)
{
  cout << "Total number of arguments: " << argc << endl;
  cout << "The first argument is: " << argv[0] << endl;
  return 0;
}
```

4 Variables and Data Types

In this exercise we will follow the example in Listing 3 below, which can also be found at the URL listed at the top of the block. The web site discusses C++ variable names, fundamental variable types, and variable declarations. Later on we will be creating our own ”types” in the form of C++ classes, but a good understanding of the C/C++ fundamental types is important. *Take the time to read through this web page.*

Your goals in this part are:

1. Open an editor and enter the code block in Listing 3.
   Save the code in a file named `var_types.cpp`.

2. Build the program into an executable.
   On the command line do: `g++ -o var_types var_types.cpp`
   This should generate a executable file called `var_types` in the same directory.

3. Verify that it runs by running it.
   On the command line do: `./var_types`

4. Modify the program such that instead of assigning and processing the results as done in lines 12-16, we instead grab two variables from the command line, add them, and output the result.

   Save your modified program in a file with a different name such as: `var_types_cmd.cpp`

   To do this you will need to do two things. First you will need to include a library, for converting ASCII strings to numbers, near the top of the program, `#include <cstdlib>`. Next you will need to invoke the `atoi()` function to convert the string to an integer. This snippet shows the general idea:

   ```
   #include <cstdlib>
   ....
   int a;
   a = atoi(argv[1]);
   ```

5. Verify that your program works by running on the command line:

   ```
   $ ./var_types_cmd  12 44
   $ 56
   ```

6. Repeat the above two steps, but in this case the goal is to handle floating point numbers. You will need to declare your variables as the type `float`, and invoke the `atof` function instead.

7. TIP: You can read more about the `atof` and `atoi` functions by typing `"man atoi"` on the command line.

Example Code 3.

```
// Code example from: http://www.cplusplus.com/doc/tutorial/variables/
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result << endl;

  // terminate the program:
  return 0;
}
```

5 Control Structures

In this exercise we will begin using the three most common control structures found in C++, the if-else, while-loop, and for-loop constructs. The idea, if not the C++ syntax, should be familiar to those with experience in other programming languages. In this section we will be following the topic described at the cplusplus.com tutorial page:

http://www.cplusplus.com/doc/tutorial/control/

This lab exercise will have three sub-parts, for each of the control structures, if-else, while-loop, and for-loop.

5.1 The if and else Structures

In the first part, your goals are:

1. Read through the first part of the web tutorial on the *conditional structure*, namely the if and else structure.
2. We're going to make a modified copy of the program you created in Section 4 in Step 4, the program `var_types_cmd.cpp`. Start by making a copy of this file and renaming it to `add_nums.cpp`

   ```
   $ cp var_types_cmd.cpp add_nums.cpp
   ```

3. In the original version of this program the two numbers being added were taken from the command line, `argv[1]`, and `argv[2]`. If these arguments were not provided, this could result in a segmentation fault (program crash) since `argv[2]` doesn't exist unless provided. In this step, use the condition structure to check for the number of arguments. If the wrong number of arguments is provided, output a warning message, otherwise handle the addition and output as normal.
4. Verify that your program works by running on the command line. It should look something like:

   ```
   $ ./add_nums 3 8
   11
   $ ./add_nums 38
   please provide exactly two arguments.
   ```

5.2 The while-loop Structure

In the second part, your goals are:

1. Read through the second part of the web tutorial on the *Iteration structures (loops)*, namely the while loop structure.
2. Open an editor and enter the code block in Listing 4. (cut-paste from the web site)
   Save the code in a file named `countdown.cpp`.

3. Build the program into an executable.
   On the command line do: `g++ -o countdown countdown.cpp`
   This should generate a executable file called `countdown` in the same directory.

4. Verify that it runs by running it.
   On the command line do: `./countdown`.

5. Modify the program such that instead of accepting a single number from the user, it repeatedly accepts a number, checks whether it is even or odd, outputs the result, and quits when the first even number is provided.

   Save your modified program in a file with a different name such as: `even_odd.cpp`

   HINT: You may want to use the C++ modulus operator, e.g., (7%2) = 1, (17%3) = 2.

6. Verify that your program works by running on the command line. It should look something like:

```
$ ./even_odd
Enter a number: 3
The number 3 is odd
Enter a number: 19
The number 19 is odd
Enter a number: 2
The number 2 is even. Goodbye!
```

Example Code 4.

```
// Code example from: http://www.cplusplus.com/doc/tutorial/control/
// custom countdown using while

#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

5.3 The for-loop Structure

In the third part, your goals are:

1. Read through the second part of the web tutorial on the *Iteration structures (loops)*, namely the for-loop structure.

2. Open an editor and enter the code block in Listing 5. (cut-paste from the web site)
   Save the code in a file named `for_countdown.cpp`.

3. Build the program into an executable.
   On the command line do: `g++ -o for_countdown for_countdown.cpp`
   This should generate a executable file called `for_countdown` in the same directory.

4. Verify that it runs by running it.
   On the command line do: `./for_countdown`.

5. Modify the program such that it accepts two integers from the command line from the user, and outputs all multiplication combinations between the two numbers ranging from $[0, N]$ for each number.

   Save your modified program in a file with a different name such as: `forloop_mult.cpp`

6. Verify that your program works by running on the command line. It should look something like:

```
$ ./forloop_mult 10 10
0 0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9 10
0 2 4 6 8 10 12 14 16 18 20
0 3 6 9 12 15 18 21 24 27 30
0 4 8 12 16 20 24 28 32 36 40
0 5 10 15 20 25 30 35 40 45 50
0 6 12 18 24 30 36 42 48 54 60
0 7 14 21 28 35 42 49 56 63 70
0 8 16 24 32 40 48 56 64 72 80
0 9 18 27 36 45 54 63 72 81 90
0 10 20 30 40 50 60 70 80 90 100
```

Example Code 5.

```
// Code example from: http://www.cplusplus.com/doc/tutorial/control/
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

6 Simple Classes

In this exercise we will begin using C++ classes. Everything prior to this section in today's lab is really in the domain of the C programming language. The use of *classes*, or more precisely the *objects* which are *instances* of classes, is what makes C++ an "object-oriented programming language". In this section we will be following the topic described at the cplusplus.com tutorial page:

`http://www.cplusplus.com/doc/tutorial/classes/`

This lab exercise will have two sub-parts, (a) building and modifying the example class on the web site, (b) implementing your class and program in separate files.

6.1 A simple class example (from the web site)

In the first part, your goals are:

1. Read through the first part of the web tutorial on Classes(I) at `http://www.cplusplus.com/doc/tutorial/classes/`
2. Open an editor and enter the code block in Listing 6. Save the code in a file named `simple_rect.cpp`.
3. Build the program into an executable. On the command line do: `g++ -o simple_rect simple_rect.cpp` This should generate a executable file called `simple_rect` in the same directory.
4. Verify that it runs by running it. On the command line do: `./simple_rect`.
5. Prepare to make a modified version of this program by copying the file `simple_rect.cpp` to `my_rect.cpp`
6. Modify the program in following three ways: (1) In addition the `x`, and `y` member variables, add a member variable of type `string` indicating the rectangle color. You will need to include the string library, `#include <string>`. (2) In addition to the `set_values` member function, add another member function for setting the rectangle color. It will take single (string) argument. (3) Add yet another member function called `print()` taking no arguments, but writes the area of the rectangle and color to the terminal using `cout`.
7. Modify your `main()` function to give the rectangle a color, and use the `print()` function instead of the original call to `cout` to convey the rectangle properties.
8. Build your program: `g++ -o my_rect my_rect.cpp`. Verify that your program works by running on the command line. It should look something like:

```
$ ./my_rect
area: 12x
color: yellow
```

Example Code 6.

```
// Code example from: http://www.cplusplus.com/doc/tutorial/classes/
// classes example
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area() << endl;
  return 0;
}
```

6.2 Building your implementation over distinct files

One of the nice things about C++ is that objects (classes), once defined and implemented, may be used in many programs. In the example implemented above, instances of the `CRectangle` class could only be used in that particular program. By splitting out the class definition and class implementation into separate files, objects of that class may be used multiple programs.

In this next part, your goals are:

1. Open an editor and enter the code blocks in Listing 7 into three distinct files named as indicated on the first line in each block.
2. Build the program into an executable.
   On the command line do: `g++ -o distributed_rect CRectangle.cpp distributed_rect_main.cpp`
   This should generate a executable file called `distributed_rect` in the same directory.
3. Verify that it runs by running it.
   On the command line do: `./distributed_rect`.

Example Code 7.

```
// distributed_rect_main.cpp

#include <iostream>
#include "CRectangle.h"
using namespace std;

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area() << endl;
  return 0;
}
```

```
// CRectangle.h

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);};
};
```

```
// CRectangle.cpp

#include "CRectangle.h"
using namespace std;

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}
```

6.3 Constructors and Destructors

We won't have time to work through examples on different ways to implement your class' constructor(s) or destructor(s). Minimally you need to understand that they are functions that are called automatically when an instance of a class is created (constructor) and when it is destroyed or goes out of scope (destructor).

Be sure to at least read the explanation on this topic in the latter part of the web page:

http://www.cplusplus.com/doc/tutorial/classes/

7 Derived Classes

An important feature of C++ over C is the ability to define classes in a hierarchical structure. Subclasses of a class may be defined that inherit (or override) properties of its parent class. This has enormous utility with respect to the goal of "code re-use". Class inheritance is what enables MOOS applications and IvPHelm behaviors to be written from a general MOOSApp and IvPBehavior superclass.

In this exercise we will explore the use of class inheritance with a simple example. Readers should regard this as a launching point to explore further the details of C++. If you've gotten this far and through this exercise in today's lab, you have a fighting chance to understand the autonomy specific code used in this course. However, we recommend you keep going with your C++ exploration in this course. It is an essential, and very empowering language in marine robotics.

In this section we will be following the topic described at the cplusplus.com tutorial page:

http://www.cplusplus.com/doc/tutorial/inheritance/

In the exercise, your goals are:

1. Read through the first part of the web tutorial on Friendship and Inheritance at http://www.cplusplus.com/doc/tutorial/classes/
   While the notion of class friendship is useful, it is not our focus here. I recommend skimming this material for now.

2. Open an editor and enter the code block in Listing 8.
   Save the code in a file named `derived.cpp`.

3. Build the program into an executable.
   On the command line do: `g++ -o derived derived.cpp`
   This should generate a executable file called `derived` in the same directory.

4. Verify that it runs by running it.
   On the command line do: `./derived`.

5. Prepare to make a modified version of this program by copying the file `derived.cpp` to `diamond.cpp`

6. Modify the program by adding a new derived class called `CDiamond`, and implement its `area()` function accordingly. Add the use of the CDiamond class to the `main()` function.

7. Build your program: `g++ -o diamond diamond.cpp`. Verify that your program works by running on the command line. It should look something like:

```
$ ./diamond
20
10
5
```

Example Code 8.

```
// Code example from: http://www.cplusplus.com/doc/tutorial/inheritance/
// derived classes
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

7.1 Polymorphism

If you have time still in this lab, explore the issue of polymorphism discussed at:

http://www.cplusplus.com/doc/tutorial/polymorphism/

In particular, focus on the issue of virtual members and abstract base classes. Try to re-work the Polygon example above by defining `area()` as a virtual function as discussed on the web site. See if you can learn the difference between a `"virtual function"` and a `"pure virtual function"`. Should the `area()` function for the `CPolygon` class above be `"virtual function"` or a `"pure virtual function"`? Why? If if the former, what would be a reasonable implementation?