ECE244 Lab 1

1 Objectives

The objectives of this assignment are multifold: to learn basic UNIX commands, to learn how to submit your solutions to the `autotester` program, and to provide you with practice on the edit-compile-run-debug cycle of C++ programs in a UNIX environment. The C++ programming language is, in many ways, a superset of the C programming language you are already familiar with, so doing this Lab Assignment in C++ should be straightforward. This lab will also provide an introduction to using the NetBeans IDE (*Integrated Development Environment*), which provides extensive functionality to make source code entry, browsing, building, and debugging more convenient.

2 Problem Statement

In this lab you will first create a directory for ECE244 assignments in the file system under your UNIX account on ECF. You will then copy a simple C++ program, compile it, check it for correct output, edit the program, recompile and submit to the autotester. Next, you will use the GNU Debugger (GDB) from within NetBeans to debug several programs. The first few programs give in-depth guidance on how to accomplish the goals, while the latter programs leave you with just hints. To follow the hints, consult the debugging guide which will have more details.

3 Additional Background Material

- Online NetBeans debugger intro here[1]
- Video debugger tutorial here[2]

4 Procedure

4.1 Setting up your directory

Log in to the ECF system using your own ID and password. Create a new directory for this course called `ece244`, and protect it by setting the permissions so only you can read or write it. See the "ECF Unix Environment" document reading for details (hint: `chmod` $-R$ `go-rwx ece244`).

It is your responsibility to ensure that others cannot copy your work. If another student is able to copy your work, and we determine that two submitted solutions stem from the same source, then we have no option but to take action against both parties.

[1] http://www.cs.uga.edu/ shoulami/sp2009/cs1301/tutorial/NetBeansDebuggerTutorial/NetBeansDebuggerTutorial.htm
[2] http://www.youtube.com/watch?v=joWldbcp1So

Download the lab1 files from the course website. They will be in an archive called `lab1.zip`. Extract the file into your `ece244` folder using the command `unzip lab1.zip`[3] This will create two folders: `lab1` and `tutorials`.

4.2 Preparation: tutorials and background documents

Change your directory to the tutorials directory (hint: `cd tutorials`). Please read the three background documents listed below and do the exercises contained within.

- ECF_Unix_guide.pdf: The ECF UNIX Environment, commands and filesystem
- NetBeans_intro.pdf: "Getting Started with NetBeans" guide
- Debugging_intro.pdf: "Intro to Debugging" guide

It is critical that you read the entirety of these three documents – taking shortcuts now will lead to you having a poor understanding of the tools we will use in this course, and make later labs very difficult to complete. Make sure in particular that you understand the process to check and submit your work. Many students receive a grade of 0 on a lab, not for failing to do the work, but because they did not submit the files correctly.

4.3 Importing and building the "Hello, World" program

The classic "Hello, World!" program is shown below. It can be found in the `hello` folder under `lab1` which was extracted from the archive. Use the `cd` command to change your current directory to the textttthello directory.

```
1  // hello.cpp -- "Hello, world" C++ program
2
3  using namespace std;
4  #include <iostream>
5
6  int main () {
7
8      cout << "Hello " << endl;
9      cout << "world!" << endl;
10     return (0);
11 }
12
```

1. Open NetBeans (from the command line, you can type `netbeans` to start it).
2. Create a new project (File → New Project) from existing sources
   (a) Choose C/C++ from existing sources since the files are already there, then click next
   (b) Click the Browse button and select the folder you just created (`<...>/ece244/lab1/hello`)

[3] How to use the zip and unzip commands is covered in the ECF Unix Environment document.

(c) Leave all other settings at default

3. In the project navigator at left, open `hello.cpp` by double-clicking. Note the C++ keywords are highlighted.

4. NetBeans will by default have built (compiled and linked) the program after importing. If you want to build again after making source changes, you can do so by clicking the hammer, or through the menu Run → Build Main Project. The hammer-and-broom icon is for clean and build (ie. delete all non-source files first)

5. Run the program by clicking the green right arrow button in the toolbar or from the menu Run → Run Main Project. You should see the expected output in the window pane below the source editor.

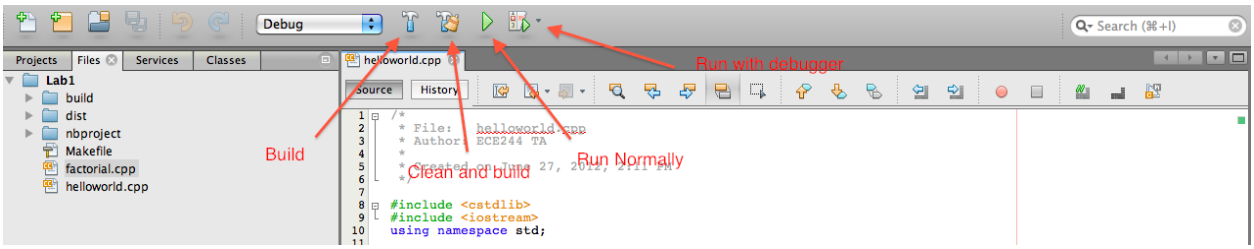

Figure 1: NetBeans basic toolbar for building and running projects

4.4 Starting with the debugger: using breakpoints

Before going on, be sure you have read the "Intro to Debugging" document, and worked through the examples within it.

One of the most fundamental functions of the debugger is to "pause" a program either at a specified source line, or when a certain condition is true to let you inspect what is going on. The most basic way to do this is through setting a breakpoint, which will be illustrated below:

1. Set a breakpoint on the line where "world" is printed

2. Run the program normally, noting that it runs just like before: the breakpoint does nothing unless we start the debugger.

3. Now run with the debugger. Either click the button that shows a breakpoint with a green triangle at lower right (see Fig 2), or menu Debug → Debug Main Project[4].

4. Note that the program prints "Hello," and stops. The source line is also highlighted in green. The program execution has paused, waiting for your action. In this case, simply continue. The program should print the rest and finish.

[4]If you have multiple projects open, be sure that `hello` is the main before you do this. See the NetBeans intro for more detail

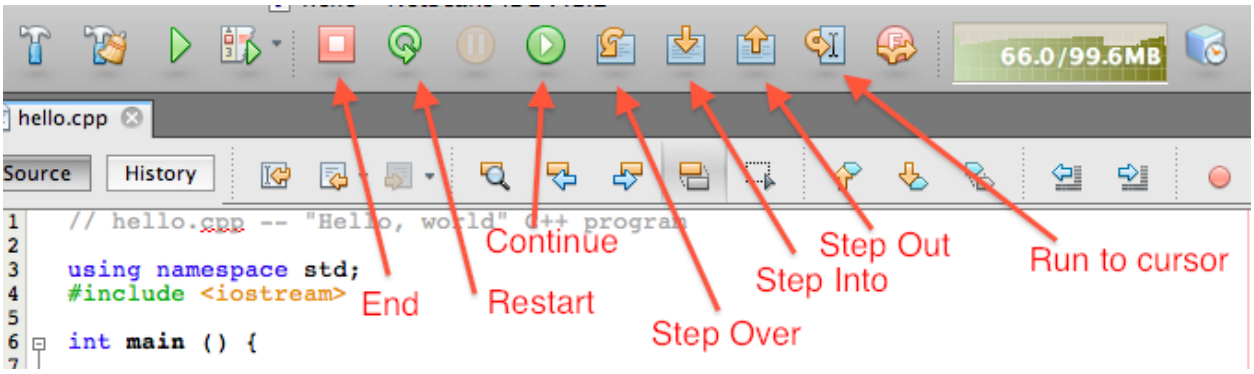

Figure 2: The debugging toolbar

4.5 Using exercise to test your program

One utility provided to help you is the `exercise` script[5]. It will run your program and provide it with test input, then compare the output expected against what is actually provided. You will be notified whether or not the two differ. When you run the `hello` program, you should see:

```
Hello
world!
```

Figure 3: Output produced

Now let's test the program before submitting it to see if that works:

1. Run the program and verify that you get the output shown in Fig 3
2. Use the `exercise` script to test your `hello` program
3. You should find that it fails - it was expecting something more like Fig 4
4. Edit the code so that it produces the expected output
5. Test again, hopefully it works

You should run through this process for every lab that you submit to ensure you are producing correct output. Remember that output produced must be exactly what is specified in the handout - no difference is permitted in spaces, capitalization, spelling, etc. The functionality marks are produced by a test script which checks for exact equivalence. Note also that (where user input controls the program) the exercise script will only run one or a few test cases. In the future, you will be evaluated on more cases than are run in `exercise` to ensure your program can handle a range of input.

[5]For details, read the document on the ECF Unix Environment

```
Hello, world!
```

Figure 4: Expected output

4.6 Finding a run-time bug with `harmonic.cpp`

The next program, found in the `harmonic` folder, is supposed to ask the user for a number and then print the corresponding harmonic number. The N-th harmonic number is defined as

$$\sum_{1}^{N} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} \tag{1}$$

In this program, it has been implemented as a `for` loop with a variable used to accumulate the sum as it goes. However, you will find that although the program compiles correctly and runs it does not produce the expected output.

1. Create a project and compile the file
2. Run for a few test cases to see if correct output is produced
3. Find the bug. Suggested procedure:
   (a) Set a breakpoint within the loop body
   (b) Run the program with a reasonable (small) input number, say 3
   (c) Check the value of the variables (Variables window) when the breakpoint is encountered
   (d) Hit continue to go through the loop step-by-step and watch the values
4. Change the code and recompile
5. Re-test to ensure correct functioning

When you believe you have fixed the bug, you should run
`~ece244i/public/exercise 1 harmonic`
to confirm that the program generates the expected output.

4.7 Another run-time bug: temperature conversion with `convert.cpp`

Our third program takes a temperature in degrees Fahrenheit, Celsius, or Kelvin, and prints out its equivalent in the other scales. Where T_F, T_C, and T_K are the temperatures in each of the scales, the conversion between them is defined by the equations below:

$$T_F = \frac{9}{5} T_C + 32.0 \tag{2}$$

$$T_K = T_C + 273.0 \tag{3}$$

For your convenience in testing, several equivalent values with physical significance are given below.

Definition	° C	° K	° F
Boiling point (liquid nitrogen)	-196.0	77.0	-320.8
Freezing point (pure water)	0.0	273.0	32.0
Boiling point (pure water)	100.0	373.0	212.0
Normal body temperature	37.0	310.0	98.6

Table 1: Selected temperature values

1. Create a NetBeans project
2. Compile the program, fixing errors as needed[6]. NetBeans will help you navigate the error messages, and flag syntax errors. You can run

   `~ece244i/public/exercise 1 convert`

   to automatically run some test cases, but you should also try some test cases of your own.
3. If you find that it does not, try to find and fix the error(s). As often happens with software, be aware there may be more than one bug to be fixed.

HINT Sometimes when debugging a process involving many steps, the *single-step* function in the debugger can be very useful to follow along. This can help you confirm that program control is following the path you think (or not!). Often, this is a far more efficient process than inserting random print statements or staring at code.

Important Make sure you do not change the output formatting. The exact same spacing and number of decimal places must be given to receive credit.

4.8 Debugging a large program

Part of the code distribution is early version of VPR (Versatile Place and Route) for FPGA, the software part of Vaughn Betz's PhD thesis. It is a fairly large piece of software, consisting of over 20k lines of source. To illustrate the use of the debugger, an error has been deliberately inserted into VPR to cause a run-time error. Trying to locate a fault by inserting print statements would be next to impossible in a program of this scope. Instead, you will use the debugger available in NetBeans to locate the fault quickly and precisely.

The steps below will walk you through compiling and running the program, which should cause it to die with a segmentation fault. Recall from the debugging tutorial that a segmentation fault occurs when the program tries to read memory that it's not allowed to. This may be because of an invalid pointer, or exceeding the bounds of a valid pointer. When programming, it is good practice to initialize all pointers to `NULL` (value 0x0) because that value is never a valid memory address and so is easily recognized as an uninitialized pointer. Otherwise it is hard to tell by inspection what values are valid or not.

1. Create a NetBeans project for the `vpr` folder using the existing source and Makefile

[6] Read the "Getting Started with NetBeans" document which discusses some common errors

2. Compile the project (there should be no compile-time errors)
3. Run the program - it should just print a usage message explaining the (complicated) command-line options
4. Run the program again with the command-line options below (see "Getting Started With NetBeans" to see how you can set command-line options) - it should get partway through and then die reporting "Segmentation fault"

```
vpr e64-4lut.net 4lut_sanitized.arch placed.out routed.out -nodisp
```

5. Run again using the debugger to localize the exception
   (a) When the segfault happens, the Operating System will send the program a "signal" (i.e. a message) which is intercepted by GDB/NetBeans - it will pop up a dialog box asking you what to do.
   (b) Choose to "discard and pause", i.e. do not pass the signal to the program, but pause it and let you work with the debugger.
   (c) Look at the sequence of functions the got you to this point starting at `main()` (inspect the *call stack*[7]) to see where you are and how you got there. The function names won't have significance to you, but that's OK; just poke around.
   (d) Inspect the local variables to see their values.
   (e) The fault should happen on an illegal memory access. The debugger will show you the line on which the bad memory access occurs. Look at the variables on that line, especially pointers (and remember that an array is really a pointer to a block of memory). Which pointer/array variable has an invalid value?
6. Scroll around within the function a bit to see where the pointer's value is (or should be) defined - in this case, it should be clear and easy to fix.
7. Demonstrate how you located the fault to your lab TA (details below in Sec 5.1).

.

The command line to run the example file is also saved in `test.sh`[8] for your convenience.
A few helpful hints as you try debugging:

- You can inspect the value of a variable while debugging either by using the Variables window, or by hovering the mouse cursor over the variable name.
- One hidden "feature": to inspect the base pointer in an array expression such as `p[i][j]` you have to highlight the base pointer name (`p`) first and then hover the cursor over it.

[7] See debugger tutorial for details - look in the Call Stack pane at the bottom
[8] To run from the command line, simply type `test.sh`

5 Deliverables

5.1 Demonstration

Demonstrate the following to one of your lab TAs:

- Find the source file and line number where the `vpr` program is crashing.
- Figure out how the program arrived at this point (inspect the call stack).
- Identify the type of crash and why it is occurring.
- Find the (deliberately commented out) line causing the error and uncomment it.
- Recompile and demonstrate correct function of the program.

5.2 Autotester submission

Please submit the following files to the autotester, after you have fixed the code in each so they produce correct output and pass `exercise`.

- `hello.cpp`
- `harmonic.cpp`
- `convert.cpp`

Remember to use `exercise` to test each program. When you are satisfied both programs are correct and perfectly match the expected output, copy all three files into one directory and use

```
~ece244i/public/submit 1
```

to submit the assignment for marking. Carefully examine the output from submit to ensure your program was submitted correctly and passed basic checks, or you may receive 0 for this part of the lab. You can submit your work as many times as you like; only the last submission will be graded. Also check that you have set your file permissions correctly to prevent others from copying your work.