Introduction to Debugging

1 Introduction

Software virtually never works correctly the first time it is run, and the process of finding and fixing bugs often takes much longer than actually writing the program. In fact, the debugging process can be so lengthy and time-consuming that most processor manufacturers include significant hardware features to help programmers by allowing them to pause the program and transfer control to another program when certain conditions are met. Those features are typically used by a program called a *debugger*, rather than by application programmers directly. Most bugs arise because the programmer has reasoned incorrectly about the program's *state*: the value of variables, registers, and memory locations are not what the programmer thought they would be. Hence, the most important needs of the programmer in trying to find bugs are to inspect and modify the program state with as much detail as possible.

A debugger is a program with special privileges allowing it to start, stop, pause, inspect, and even modify the state of other programs while they are running for the purpose of finding software bugs. It is a fundamental tool for software development. One very popular example is the *GNU Debugger* (*GDB*), which is free, open-source, and available for an extremely large variety of computer platforms and OSes. While GDB itself is a command-line tool which takes some effort to learn, NetBeans provides a graphical interface to it that is integrated within the IDE itself, making the process much more intuitive. This introduction to debugging illustrates the most common uses of the debugger within NetBeans.

For those wishing to know more about GDB and its command-line use there are many resources available, for instance: type `man gdb` in most Linux/UNIX distributions; the GDB online manual[1]; numerous online tutorials (just search with Google); or the online help interface within GDB (run by typing `gdb` at the command prompt, then type `help`).

2 Preparation - Compile Flags

When the computer runs your program, it loads the machine code (sequence of bits and bytes) into memory then starts reading and executing the instructions. All variable and function names have at this point been translated by the compiler and linker into memory addresses storing the variables and functions, which are simply numbers usually printed in hexadecimal - not easily understandable. Running the debugger will allow you to view the values, but they will not hold very much meaning for the programmer.

For this information to be useful, we would like to know what function and variable names correspond to the binary values in memory and registers. For that, we need the compiler and linker to produce a mapping called *debug symbols*. With debug symbols enabled, the debugger will be able to specify where it is in terms of functions, source files, and line numbers which is much more convenient. With the GCC compiler (as used by default in NetBeans), that option is `-g`. It will be enabled by default when you build a Debug configuration (see the NetBeans tutorial for more info). It is also enabled by default in the Makefiles provided with the labs.

[1] http://www.gnu.org/software/gdb/documentation/

3 Using the Debugger

3.1 Controlling Program Execution

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 simplest way to do this is through setting a breakpoint.

3.1.1 Breakpoints

A breakpoint is a designated source line where the program will pause each time it is reached. When being debugged, the program will run up to (but not including) the line at the breakpoint. When paused, you can inspect the program's state, change the state (memory, registers), set other breakpoints, etc. Virtually the only thing you cannot do is change the source code or compilation options - those require you to end your debug session and recompile before they take effect. Once done, you may continue. The software will run until it hits another breakpoint (or the same one again).

Breakpoints can be set by clicking on the line numbers at the left of the source editor. You *do not* need to recompile after changing breakpoints. When set, a red square appears. You can clear it by clicking again. Any number of breakpoints can be set. When the program pauses at a breakpoint, the current line about to be executed will be highlighted in green in the editor window.

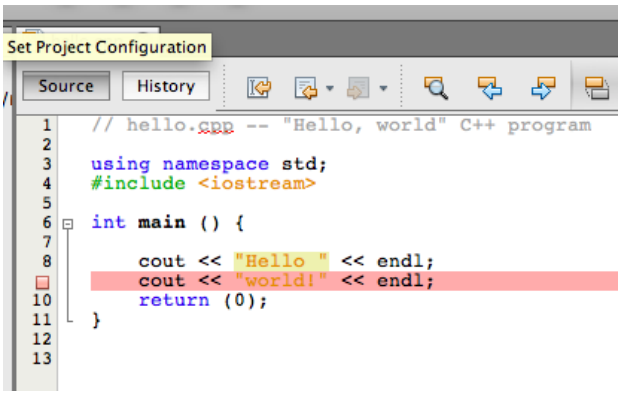

Figure 1: Setting a breakpoint

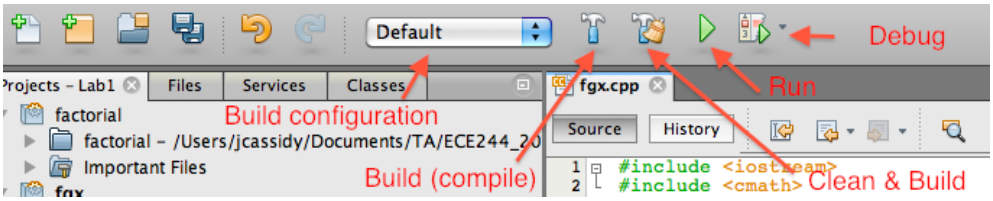

Figure 2: The run-build toolbar, showing the buttons to start a debug session

1. Compile `hello.cpp`, ensuring that debugger symbols are enabled
2. Set a breakpoint on the line where "world" is printed (Fig 1).

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

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

5. Note that the program prints "Hello," and stops. The source line is also highlighted in red. The program execution has paused, waiting for your action. In this case, simply click continue (green circle with inset arrow), or go to Debug → Continue. The program will print the rest and finish.

You can also view a list of all breakpoints in all source files (useful for larger programs) in the breakpoints pane (bottom of window, or menu Window → Debugging → Breakpoints).

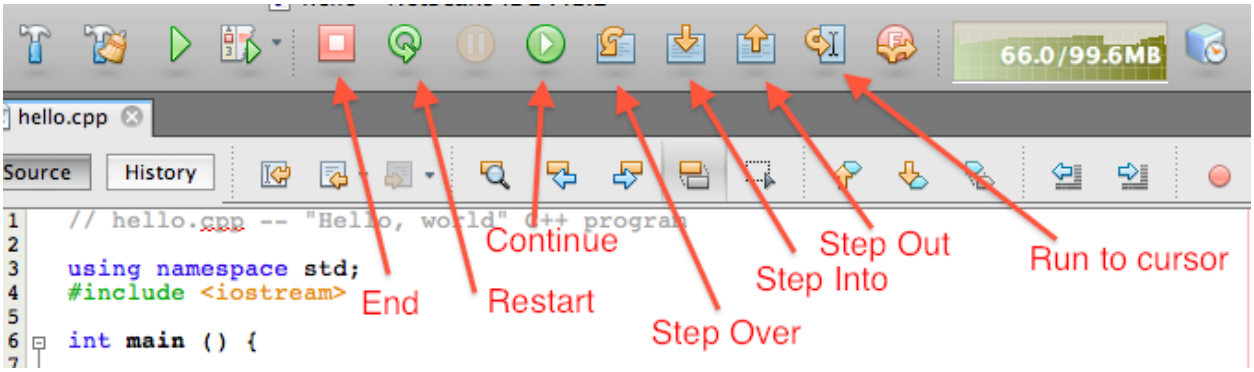

Figure 3: The debugging toolbar

3.1.2 Stepping

Once paused, there are several ways you may step through your program, with the basic ones listed below:

Continue	Executes until the next breakpoint (including arriving at the current breakpoint again) or watch is triggered
Step Into	Executes only the first line in the called function then stops
Step Over	Executes the next function call in its entirety, without stepping you through the intermediate instructions
Step Out	Runs to the end of the current function, then stops at the next line in the calling function
Run to Cursor	You may select a source line using the cursor in the editor, and the program will run up to that point before pausing again

3.2 Inspecting Program State

3.3 The call stack

A *stack* is one of the most basic data structures in computing. As the name suggests, it can be imagined as a stack of heavy objects where you can only lift one at a time - as a result, items must be added or removed from the top. Placing an item on the stack is known as *pushing*, while removal

is called *popping*. For instance, if items A, B, C were pushed onto the stack in order, then A would be at the bottom and C at the top. If this stack were to be popped three times, the resulting items would be C, B, A since it goes from the top down. This ordering is also called *LIFO* for *Last In First Out*.

The *call stack* is a stack used by computers to keep track of what functions have been called, their arguments, and where the current function should return to. Each time a function call is made, the current instruction pointer (location within the code) is pushed onto the stack along with the function arguments. The use of a stack is natural for this purpose, because the processor needs to know where to go when it reaches the end of the function. With a stack, the most recent value (the one we need to return to) is at the top. The stack is also used to create local storage, but that is not important for this discussion.

When debugging a program, the call stack can be inspected by clicking the "Call Stack" pane below the editor, or by using the menu entry Window → Debugging → Call Stack. Entries on the call stack are displayed with the innermost (current) function at the top. The function that called the current function is next, then the function that called it, on down to `main()`. You may double-click on one of the entries to open the source file which defines the function. When doing so, you can also see the values of variables in the Variables pane, or by hovering over any reference to the variable in the source code.

```
1  #include <iostream>
2  #include <fstream>
3  #include <cmath>
4  using namespace std;
5
6  float f(float);
7  float g(float);
8
9  void readLinesFromFile(string filename,int nLines);
10
11 float f(float x)
12 {
13     return g(x-1) + 2*x;
14 }
15
16 float g(float y)
17 {
18     return y*y + 3;
19 }
20
21 int main(int argc,char **argv)
22 {
23     ifstream is;
24     string str;
25
26     coot << f(2) << endl;
27     cout << "f(2) = " << f(2) << endl;
28
29     return 0;
30 }
```

An example of inspecting the call stack for the program above is given in Fig 4, with the program starting (as always) in `main`. To compute the value to be printed, `main()` has to call `f()`, which in turn calls `g()`. So, initially the call stack will contain a return address from `main()` and the values of the command-line arguments `argc` and `argv`. When it reaches the call to `f()`, the

return address (next instruction within `main()`) will be pushed, along with the argument `x` (in this case with value 2).

As you may be able to recognize from the figure, a breakpoint was set at the `cout<<` statement in `main()`. The program was run and paused at the breakpoint. From there, single-stepping was used to get into `f()` and then into `g()`. You can try this process yourself using the source code in the folder `fgx`.

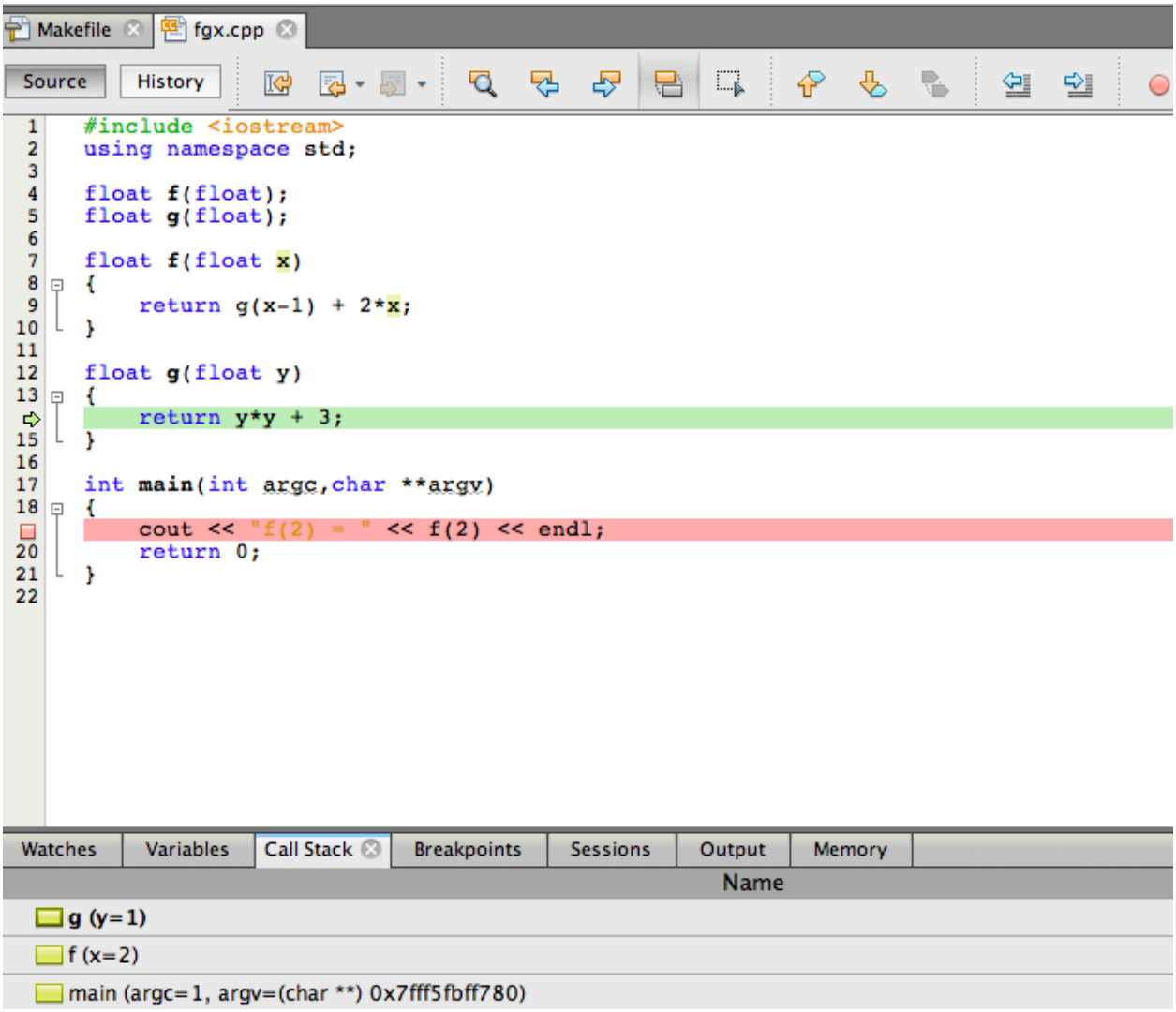

Figure 4: Inspecting the call stack

3.3.1 Exercises

1. Load `fgx.cpp` and set a breakpoint in `main()` on the line where `f(2)` is called. Run the program and inspect the call stack when the program pauses at the breakpoint. Now run **Step Into** several times to see how the call stack changes as the function calls are carried out.

2. Load `factorial.cpp`, which calculates the factorial function recursively. Set a break at the

call to `factorial` and step in to watch how the call stack changes. You can also inspect the return value in `f_n_minus_one`. Run a few times and try using the step out and step over functions as well to see how they differ.

3.4 Local variables

The debugger also allows you to inspect the value of local variables. If you click on the "variables" tab below the source editor (if it's missing, go to the menu Window → Debugging → Variables), you will be able to see the values of all local variables when the program is stopped.

3.5 Watches

Watches are the NetBeans name for the what GDB does when given the `display` command. It is a set of variables or expressions that are displayed in a window when execution is stopped. For instance, if the values of variables `x` and `y` are of interest, we can view them in the Variables pane. If, however, it is some function of them (for instance `x+2*y`) that we want to view, a Watch can be established to print out that expression every time the program is stopped. You can also use pointer arithmetic, such as dereference (`*` operator) or take-address (`&` operator) in the expressions.

3.6 Exception handling

Sometimes due to a bug your program will terminate early with a serious error called an *exception*, often printing an error message to the terminal. By definition, an exception condition is exceptional - a properly-written program should never terminate this way. You should always check for (or prevent) errors, catch them, and handle them properly.

3.6.1 Common Exceptions

Some of the more common exceptions that you should watch out for are:

- Dividing by zero, or other invalid arithmetic operations
- Reading or writing past the end of an array
```
int x[100];
x[100] = 1; // oops!  (remember x[100] runs from x[0] to x[99])
```
- Reading or writing memory that does not belong to the program, for instance with an uninitialized pointer or reading past the end of a block of memory
```
int *p; // uninitialized pointer
p=2; // what does p point to?
```
- Trying to read something into a buffer that is too small
```
char dst[10];
strcpy(dst,very_long_string); // copying a string that is too large
gets(dst,f); // reads a line from file - length unknown (hopefully less than 10!)
```
- Trying to execute an invalid instruction (eg. if the call stack gets overwritten and an invalid return address is used leading to invalid instructions being issued)

- Running out of stack space (often if a program uses too much recursion, or is stuck in an infinite loop)
```
void f()
{
f(); // just keeps calling itself infinitely...
}
```

The debugger can be extremely useful in locating such errors, because when the debugger is running and a program exits due to an exception it pauses at that point (just like a breakpoint) and provides information about the exception. You can then use the debugger to inspect the program state and determine why the exception occurred. First and most trivially, it shows you the source line where the error occurred which is sometimes sufficient. Next, there are other tricks you can try:

- Check the values of local variables - are they what you expect?
- Examine the call stack and arguments - how did we get here? Do the arguments make sense?
- Look in memory, make sure the contents of your data structures are valid

Some of the more common exceptions are listed below, with tips on how to fix them.

3.6.2 Segmentation fault

For security reasons, modern processors and operating systems only allow programs to read and write to their own areas (segments) of memory, as allocated by the OS (your program can request more using `new`, or `malloc` in C). A *segmentation fault* is raised when the program tries to access memory other than what it is permitted to. The most common cause of a segmentation fault is trying to write or read an invalid address - usually through a pointer that is either uninitialized (contains random values), NULL, incorrectly allocated/freed, or by accessing past the end of an array[2]. There is an excellent free and open-source tool called *Valgrind* (info here[3]) that helps diagnose invalid `new`/`delete` and read/write problems. It is installed on ECF machines if you want to try it.

The example file `segfault.cpp` and its associated project illustrate use of the debugger to trace a segmentation fault.

3.6.3 Divide by zero

As the name implies, division by zero is an illegal operation, causing a floating point exception that will halt your program. If a divide-by-zero occurs in your program, you should check whether zero is within your expected range for the divisor (generally it should not be since divide-by-zero is not physically meaningful). If so, it will be necessary to treat it as a special case. Otherwise, you should check why you are getting a value outside the expected range using the techniques outlined above.

[2] Check zero-based vs. one-based indexing: remember that int x[100] is 100 elements and runs from x[0] to x[99]
[3] www.valgrind.org