

# CIS 213: Foundations of Computer Science—Lab 5

## Debugging with DDD

In this lab you will be introduced to the a debugger, and some of the most popular ways to use it. Copy the program from the lab directory, `~hardnett/pub/cis213/lab5`. The files are:

1. ListP.cpp
2. ListP.h
3. Makefile
4. problem1.cc
5. problem2.cc

## 1 DDD(The debugger)

The debugger is a very useful tool for programmers. We will be learning how to use the debugger. The debugger got its name from the idea that was designed to help you locate bugs in your program logic, and eliminate them. However, the debugger can also be used for the following reasons:

1. If you have to learn how a new program works, then you can run the program and examine it while it executes.
2. If you are learning a programming language, you can use the debugger to learn about the semantics/behavior of the statements in the language.

### 1.1 What does a debugger do?

A debugger gives the programmer the ability to control and monitor the execution of their program while it is running on the computer. The debugger allows the programmer to do the following kinds of tasks:

- You can pause the program like you do a video tape, and then look at what values are in all your variables.
- You can change values of variables without having to recompile or run the program again.
- You can execute your program, one step at a time.
- You can examine functions as they are called, and as they complete.
- You can fast forward the execution to some later point in the program, and have it pause at that point.(using breakpoints)

There are many other features of the debugger, and as time goes on you will learn many of them. This tool is to be added to your collection of program development tools with your editor and compiler.

### 1.2 Learning about DDD

In this lab, you will learn some of the basics about DDD. DDD stands for the Data Display Debugger. This software is free software, and is available on other systems. Although it is free, it is heavily used by programmers all over the world, and it is very similar to debuggers found in Borland C++ and Visual C++ environments.

In order to take advantage of the debugger, your program must be compiled with the `-g` option:

```
e.g. g++ -g program.cc -o program
```

The `-g` option makes your program ready to be used by the debugger. If you do not compile with `-g`, then you will not be able to make use of the debugger. The `-g` should be added to your `g++` command in the Makefile for all of your programs from now on. Look at the makefile in this directory, and notice the `-g` options:

```
host% more Makefile
```

The Makefile is structured in the following manner:

```
<target>: <dependences>
      <command>
```

The semantics for Makefiles:

```
if <target> is specified at the command line and
  if <dependences> exist and they have changed then
    execute <command>
```

To compile the programs as specified in the **Makefile**, you can type the following:

```
host% make all
```

The “all” target depends on all of the the other targets in the file, and so it will compile all of the programs.  
Now lets start the debugger:

```
host% ddd &
```

### 1.3 Configuring DDD

The first thing you will see is a info window and at Tips window. The Tips are usually very good ones, and so read your tips each time you use ddd. You can easily see the next tip, but its usually not worth it to try to read a bunch of them.

At this point, the DDD window has three parts:

- The menu/toolbar area at the top of the window
- The area where your code resides (if you leave out the -g option during compilation, then this window will be blank)
- The command prompt area with the (gdb) prompt

GDB(Gnu Debugger) is really the debugger. DDD is just an interface that makes using the debugger easier(you can run gdb by itself and just get a prompt). The prompt is where debugger commands can be typed in. However, we are using DDD so we will use the menus and icons to issue commands to the debugger instead of typing them in. The other purpose for the window at the bottom is that it acts as the window where your program output will appear. In other words, anything that would normally happen to the shelltool window will happen in this bottom window here.

Now lets configure the debugger:

1. from the menu choose Edit -->Preferences
2. Press the Source Button
3. Across from ”tool buttons location” select source window
4. At the bottom, move the ”source indentation” scroll bar to the right until it reaches 5.
5. select the “Display source line numbers”
6. Press the Data Button
7. select ”detect aliases” & ”use compact layout”. This means that ONLY the last item is not selected.
8. Press Startup
9. across from ”Toolbar Appearance” select ”color” so that everything is selected.
10. Press OK DDD will now ask you about restarting, answer yes and wait
11. Now choose Edit-->Save Options
12. Now choose Edit-->Debugger Settings
13. select the following items

Confirm Potentially dangerous operations

verbosity

Dynamic symbol table reloading....

Printing of objects derived....

Printing of C++ virtual....

Prettyprinting of arrays

Prettyprinting of structures

Printing of char arrays.....

Current source Language (use C++..)

14. deselect Autoloading of shared library symbols....

15. Press OK

16. choose Edit-->Save Options

Now your debugger is configured. You will never have to change these again unless your/we find something we don't like.

## 1.4 Learning About the DDD layout

As mentioned before, DDD has the following screen layout:

- The menu/toolbar area at the top of the window
- The area where your code resides (if you leave out the -g option during compilation, then this window will be blank)
- The command prompt area with the (gdb) prompt

The menu/toolbar area has a list of drop-down menus like File, Edit, View, etc. The toolbar has 3 parts: The first part is the "()" textbox that just displays the currently selected object to have a command applied to it. The second part is the list of icons that are some of the most frequently used commands to apply to selected objects. The last part is a list of commands starting with Run, Interrupt, Step, etc. These are commands that affect the debugger as a whole, and are not applied to single objects to be debugged.

A debuggable object can be a variable, function, class, etc.

The large sub-window in the middle is where the program you are debugging is. It should have line numbers next to each line. You can use the scroll bar to the right to move up and down within the window (You can also use the Page Up and Page Down buttons). The size of this sub-window can be resized by dragging the little square in its lower right corner. You can drag it up and down.

The last sub-window at the bottom is where all debugger commands (including some that are not in the menus) can be entered. This is also the window where you will see your program running (i.e. this is like "shelltool" for the debugger).

To learn more about the debugger you will need to use it to help you on your programs, and you can use the Help menu option. There is also some information on the class web page.

## 2 Setting BreakPoints at a given line

Setting breakpoints is the primary use of a debugging tool. A breakpoint is a place in your program where you would like the program to pause during execution. When the program pauses you can view the values of variables and even change their values to see what would happen. Think of a breakpoint as being placed where you would normally place a debugging cout statement. The advantage is that you don't have to edit and recompile your program to put breakpoints in, while this is the case with debugging cout statements.

## 2.1 Using the Debugger on a program

We will be using the debugger this semester for 2 purposes. First, we would like to use it to learn more about how C++ programs work. Second, you should learn to use the debugger to help you fix the bugs in your programs. The more you use the debugger, the better you will be with it.

first you need to try to trace the program by hand. Use your editor, xcoral or xemacs to look at the file problem1.cc.

```
host% xemacs problem1.cc &
```

### **DO NOT CHANGE THIS FILE**

Trace this file using the input: 3

Be careful, it is a rather tricky program to trace. Try to determine what the output will be.

NOTE: Use your tables to keep track of variables.

After you have some output on your paper, then just run the program and give it input 3:

```
host% prob1
```

Were you right? If not, that is why we are going to use the debugger to understand how it gets that output. If you were right, then you can confirm what you have done on paper by using the debugger.

Run it a few times, try different inputs to see what it does. Now we will use the debugger to learn about the code and to learn how to use the debugger.

Open the program file `prob1` in the debugger using the `File` menu.

## 2.2 STEP 1: SETTING A BREAKPOINT

Here is how to set a breakpoint at any given line. Look in the program in the window and find the line near line 15 that has `"cout << "`, this is how to set the breakpoint:

1. move the mouse to the left margin where the line numbers are, and put the cursor there. The file name followed by the line number should appear in the `"():"` debuggable object box at the top.
2. Now click the stop sign icon, and stop sign should appear on that line.

Now you have a break point set.

Now lets set a BreakPoint for if statement on line 19, and the for loop on line 23. Follow the steps above and set the breakpoints.

Now you should have 3 stop signs/breakpoints created. You can verify this in two ways:

1. You can look up down the files for the stop signs

OR

2. You can select the menu `Source\verb+-->+Edit Breakpoints` and see a list of them.

Once the breakpoints are set you are ready to run the program.

## 2.3 STEP 2: STARTING THE PROGRAM

Click the "Run" button in the row of buttons. The "Run" button starts the program from the beginning. Notice that the first breakpoint that is hit is the stop at the `cout`. You will see a little green arrow pointing to that line. This arrow points to the line that WILL BE executed NEXT..

## 2.4 VIEWING LOCAL VARS

Notice we have finished executing the declarations of the integers. We can look at these local variables:

Select the menu item `Data->Display Local Variables`.

this will make the debugger create a window with a box of all of the local variables. This box will stay and change everytime the values of the local variables change.(if we call a function, those local variables will be displayed instead)

You can see what the initial values of the vars are.

## 2.5 CONTROLLING THE EXECUTION

One of the strengths of the debugger is that we can control how fast the program executes by dictating which instructions it executes next. There are two widely used commands for this:

Continue(Cont) : This will make the program continue execution till the next breakpoint.

Next: This will make the program execute the next instruction i.e. it will execute the instruction the arrow is pointing at.

Lets execute the next instruction, the cout(NOTE: The program has been suspended all of this time). When you see commands in the <...> then that means click the button.

<NEXT>

<NEXT>

Now you see the output asking for you to input a number. Input the number 3 and press return.

Note: The local vars now have  $n = 3$ . The number inputted.

EVALUATING INSTRUCTIONS: Suppose now you want to evaluate the instruction in the if before actually executing. For example, you want to know if  $n\%2$  is 0 before you do the if statement. You can use the print command. This has to be entered in the bottom part where the

(gdb) prompt is:

(gdb) print n%2

now try:

(gdb) print n%2 == 0

Will n be incremented or not? Now lets use the other execution control button, the continuation. If we do that, then the program will go to the next breakpoint and we can look a n to determine if the if stmt executes the n++ or not:

<CONT>

Now we are at the beginning of the for loop. Notice that has now been added to the locals box, since i is a local variable declared in the for loop.

Now, just use next and print to continue executing the program. Keep your eyes on the values of the variables, and try to predict what each statement will do as you click <NEXT> each time. You can use print to help you with your predictions. Do this for a few minutes until you feel comfortable with these commands.

## 3 The Program With Functions

Now we are going to debug a program that has functions in it. Inherently functions are sometimes hard for beginning students to understand. The debugger can help in reinforcing the correct way that the functions work.

You need to load the other program:

1. Choose the menu option File\verb+->+Open Program
2. Select the prob2 file, and click <OK>

Now you should see some new source code that has some vectors in it. Now, you can see that there are function calls in main to functions: func1, func2, and print.

At this time, we just want to see how we can follow the code from function to function. In other words, we know that the functions will be called in a particular order, and we want to examine the functions each time they are called.

Now, there are many ways to set a breakpoint on a function. This is when the breakpoint is at the beginning of the function.

FIRST WAY:

1. click(just one click) on the func1 call on line 52.
2. notice the name is in the () box at the top
3. click the stop sign(it will not appear here)

Now notice that the stop sign is not where you selected. You will find it in the definition of the func1 function. Scroll up to the top of the file. The stop sign is at the first instruction of the func1 function. So, everytime this function is called from anywhere, it will always pause.

SECOND WAY:

1. find the function you want to set the breakpoint in  
i.e. find the definition(body) of function func2.
2. select the name of the function with a single-click.
3. click the stop sign

Notice that now there is stop sign in the func2 function; just like the func1 function. Now you have a breakpoint in the function.

Now set a breakpoint on the print function using one of the methods from above.

Now, since we are going to look at function calls, we will need to look at the arguments or parameters of each function in addition to the local vars. At this time, the display at the top should have a box that says "locals: no frame selected". That just means that the program has not started, and so there are no locals to show. To add a box for arguments:

- Select Data->Display Arguments
- move the boxes around so they are side by side.

Now we are going to run this program in the following way:

1. Click <RUN> to get things started.
2. Enter 5 for the number.
3. When you stop in a function use <NEXT> to execute a few instructions. notice that the local vars and args will change to match each function being called.
4. When you think you have examined the function enough then use the <CONT> to go to the next breakpoint.
5. Each time you go to a new function, you can determine where it was called from by going to the menu Status->BackTrace.

In the backtrace the current function is on the bottom of the list, and the function that called it is above, and so on.

NOTE: at this time, it is not possible to see into vectors. We hope this is corrected very soon.

You have been introduced to a lot of the debugger, but there are plenty of other features. You can read the tip of the day each time you use DDD for more ideas. Also there is a website [www.ddd.org](http://www.ddd.org) and man pages with documentation.

## 4 QUESTIONS

Create an answer file called `lab5.ans` and put answers the following questions in the file:

1. Explain the commands `Cont`, `Next`, and `print`.
2. Explain how you can use breakpoints to understand the sequence of function calls in program.
3. Suppose you have a loop that is suppose to print all of the data for a vector. Suppose it does not print one or two of those items. Explain how you could use breakpoints and the above commands to figure out which iteration is at fault.
4. Give a situation that you may use this tool to help you with in your programs. What feature do you like the most.

Now submit your file using:

```
host% turnin cis213 lab5
```

Be sure that you see that your `lab5.ans` file was submitted.