Using GDB: An Introduction
Written by Andrew M. Morgan

To use the freely available GNU debugger (gdb) first, compile your program with the –g command line option. Example:

```
g++ -g –Wall myProjectSource.cpp –o myProjectExec
```

Next, run gdb by providing the name of your executable (note: do not include any command line arguments to your program at this time!). Example:

```
gdb myProjectExec
```

This will start the debugger, which will provide a very simple gdb prompt – there is no graphical interface for gdb-proper (note: there are "front ends" that have been developed that wrap gdb in a graphical interface).

The following is a brief list of some of the most useful commands in gdb – this is not an exhaustive list, but should get you started and provide the commands you utilize most often.

run <arguments>: This will begin running your program. If your program accepts command line arguments, include them after the run command.

`b <sourceFile:lineNumber>`: This will set a break point at the specified line number of the specified line number. For example, "b myProjectectSource.cpp:38" will set a breakpoint at line 38 of the file myProjectSource.cpp.

`b <functionName>`: This will set a break point at the very beginning of the specified function. Example: "b main" will set a break point at the first executable statement of the main function.

`b`: This will set a breakpoint at your current location. If you have used step or next to run some statements, just issuing the "b" command will set a breakpoint at the line you are currently at.

`p <variableName>`: This will print the current value of a variable within the current scope.

`n`: This command will execute the entire next statement. If the statement is a function call, it will execute the entire function and advance to the next executable statement after the function call.

`s`: This command will execute the next statement, but if the statement is a function, "s" will step into the function and stop at the first executable statement within the function.

`c`: This command will continue execution of your program from the current statement until the program ends or a break point is reached.

`l`: This command will list 5 lines of code prior to the current line the debugger is stopped at and 5 lines after the current line. If you issue this command multiple times in a row, it will continue to show the 10 lines that follow the previously listed code.

`i b`: This command will print useful information about all breakpoints currently set. It will show the breakpoint number, which is useful for setting conditional breakpoints or deleting breakpoints. It will also show the source file and line number that each breakpoint is set at.

`d <breakpointNumber>`: This will delete the breakpoint number specified. Get the breakpoint number for a specific breakpoint with the "i b" command described above.

`d`: This will delete all breakpoints (you will need to confirm when prompted).

`cond <breakpointNumber> <condition>`: This will turn a breakpoint into a conditional break point. For example, "cond 2 iVal > 10" will update breakpoint number 2 to be conditional on the condition "iVal > 10". This way, the break point will only cause the debugger to stop at the location specified by break point 2 if iVal is greater than 10. If iVal is less than or equal to 10, the breakpoint will be ignored.

`until <lineNumber>`: This will continue execution of your program from the current location and stop at the specified lineNumber. It is similar to setting a breakpoint then continuing, except that no breakpoint is set, so it is a "one time operation".

`disable <breakPointNumber>`: This will disable the specified breakpoint number so that it no longer causes the debugger to stop at the breakpoint's location. It is different from deleting the breakpoint because the breakpoint remains in the breakpoint list and can be re-enabled without having to fully specify the breakpoint's location.

`enable <breakPointNumber>`: This will re-enable a breakpoint that had been deactivated using the deactivate command described above. Once re-enabled, the debugger will stop when it reaches the breakpoint's location.

`where`: This command will print out the entire call stack to your current location. This is useful for knowing how you got to the location the debugger has stopped at. For example, it will show you that main called foo (on line #43), and foo called bar (on line 86) and that you are currently in bar.cpp at line 173.

`up`: This will cause the debugger to move "up" the call stack so that you can inquire about variables within that scope. For example, if main calls foo, and foo calls bar, and you are at a breakpoint down in bar, the "up" command will switch to the location of the call to foo within the bar function. Note: The "current statement" the debugger is waiting at is still down in bar, but this command simply lets you go up the call stack to see what variables a level up are. Issuing "up" multiple times continues to go up the call stack until you get to main.

`down`: Goes back down the call stack (see "up" above)