

Debugging Reference Guide

By Maia Burton

Managed by Professor Matthew Butner

University of California - Davis

2023

Table of Contents

Contents

1	Debugging Prerequisites	4
1.1	Use the Smallest Input Possible	4
1.2	Knowing the Correct Results	4
1.3	NOTE: What is an Expression?	4
2	Debugging Set Up	6
2.1	Reading this Guide	6
2.2	Using the Debugger Overview	6
2.3	Terminal Compiling and Debugging	6
2.3.1	Compiling Your Code	6
2.3.2	Practical Programmer: Running Commands on the Shell	7
2.3.3	Start the Debugging Program	7
2.3.4	Example: Attempting to Debug the Source Code - Incorrect Way	8
2.3.5	Example: Debugging with Command Lines	9
2.3.6	Additional GDB Info	9
2.4	IntelliJ Debugging	10
2.4.1	IntelliJ Makes Your Project	10
2.4.2	IntelliJ Doesn't Make Your Project	10
2.4.3	Additional Notes	10
3	Execution Commands	11
3.1	Run Command	11
3.1.1	NOTE: IntelliJ's Play	12
3.2	Continue Command	12
3.3	Finish Command	12
3.4	Step Commands	13
3.4.1	What Does Count Mean?	13
4	Breakpoint Commands	14
4.1	Setting Breakpoints	14
4.1.1	→ GDB Breakpoint Location	14
4.1.2	Halfway Point of Execution	16
4.1.3	GDB Breakpoint Example	17
4.2	Conditional Breakpoints	17
4.2.1	Conditional Breakpoint Example	18
4.3	Watchpoint Command	19
4.4	Breakpoint <i>Info</i> Command	20
4.5	Breakpoint Delete, Disable, Enable Commands	21

5	Examining Commands	22
5.1	IntelliJ Viewing Methods	22
5.2	Displaying and Printing Commands	22
5.3	Variable <i>Info</i> Command	22
5.4	Backtrace Command	23
5.5	What does Backtrace Do?	23
5.6	When to Use Backtrace	24
5.7	Up and Down Commands	24
5.7.1	Moving Up Command	24
5.7.2	Moving Down Command	24
6	Assembly Language - UC Davis ECS 50	25
6.1	What are Labels?	25
6.2	Labels in C and C++	26
6.3	Tips for Writing Assembly:	26
7	Glossary of Debugging Terms	27

1 Debugging Prerequisites

Before you start the debugger, you need to do these two things to get the most out of it.

- Use the smallest input possible
- Know the correct results

1.1 Use the Smallest Input Possible

Sometimes you only need a specific input to cause the error in your program, and debugging with extra input would be unnecessary. Smaller inputs means less output, and you will have less code to double check.

Less output, and less code to check

Example:

Assume you are writing a program to sort a list of numbers. If you want to test how it handles duplicate numbers, it would be much easier and faster to debug an input such as `[3, 3, 2]` instead of `[5, 6, 7, 4, 5]`.

1.2 Knowing the Correct Results

Before you begin using the debugger, you **MUST** know what the correct results should be at **every moment** your code is running. If you don't know what the results should be, you can't know if your program is doing what it should or not. It's a good idea to solve the problem by hand to identify the steps your program should take and to determine what the values of your variables should be at any given step.

Example:

If you were writing a calculator program and your program told you $2 + 2 = 8$, you would have to know that the correct answer is 4 to notice there's a problem. Just because a program doesn't crash, does not mean it works.

1.3 NOTE: What is an Expression?

Most, if not all, of the commands covered in this reference guide apply to variables, arithmetic expressions, and function calls. These can all be referred to as expressions.

An expression is any piece of code that would result in a value. Essentially, anything that isn't an if/else statement, loop, or declaration is an expression.

All of the following are expressions

- `x`
- `12 + 5 * 9`
- `x > y`
- `sqrt(z)`
- `pow(a, strlen(my_string))`

All of the following are NOT expressions

- `for (int i = 0; i < len; i++)`
- `if (x > y)`
- `while (i < 10)`

2 Debugging Set Up

2.1 Reading this Guide

Different debugging platforms refer to commands in different ways. This guide will be using the GDB and DDD command names with the equivalent command in IntelliJ. Example images from the GDB and IntelliJ's Clion. The LLDB, the default debugger in Xcode on macOS, will not be covered in this reference guide. Below is a link that will take you to a GDB to LLDB layout.

- [GDB to LLDB Command Map](http://lldb.llvm.org/use/map.html) : lldb.llvm.org/use/map.html
- [Full GDB Documentation](http://sourceware.org/gdb/onlinedocs/gdb/index.html#SEC_Contents) : sourceware.org/gdb/onlinedocs/gdb/index.html#SEC_Contents

2.2 Using the Debugger Overview

1. Choose your debugging platform
2. Compile your code
3. Set breakpoints
4. Start the debugger

Debuggers run on your program's **executable** (.out or .exe file). This means that you need to compile your program first and fix any syntax errors. A common mistake is to try to run the debugger on your source code (.c, .cpp, .h files), which will not work.

2.3 Terminal Compiling and Debugging

1. Choose GDB or DDD
2. Compile your code
3. Open the executable on the debugger
4. Set breakpoints
5. Start debugging

2.3.1 Compiling Your Code

If using the GDB or DDD debugger, remember to include the `-g` option when compiling your program.

- C++: `g++ -Wall -Werror -g [myCode].cpp -o [nameOfExecutable]`

- C: `gcc -Wall -Werror -g [myCode].c -o [nameOfExecutable]`

Practical Programmer: Don't forget to **save** and **recompile** your program after edits!! Sometimes if your changes don't seem to be working, you might not be working with the most current executable.

2.3.2 Practical Programmer: Running Commands on the Shell

You don't need to quit the debugger to run commands on the shell. Simply type `shell` before your command to have it run on the shell instead of the debugger. This can be incredibly useful when you make edits to your program. Without closing the debugger, you can recompile your code and keep all of your breakpoints where they are.

Examples:

- `shell ls`: run the `ls` command
- `shell cd some_dir`: change directories to `some_dir`

NOTE: When debugging with DDD, after you update your program, the DDD needs the specified file and to be reloaded.

2.3.3 Start the Debugging Program

GDB vs. DDD

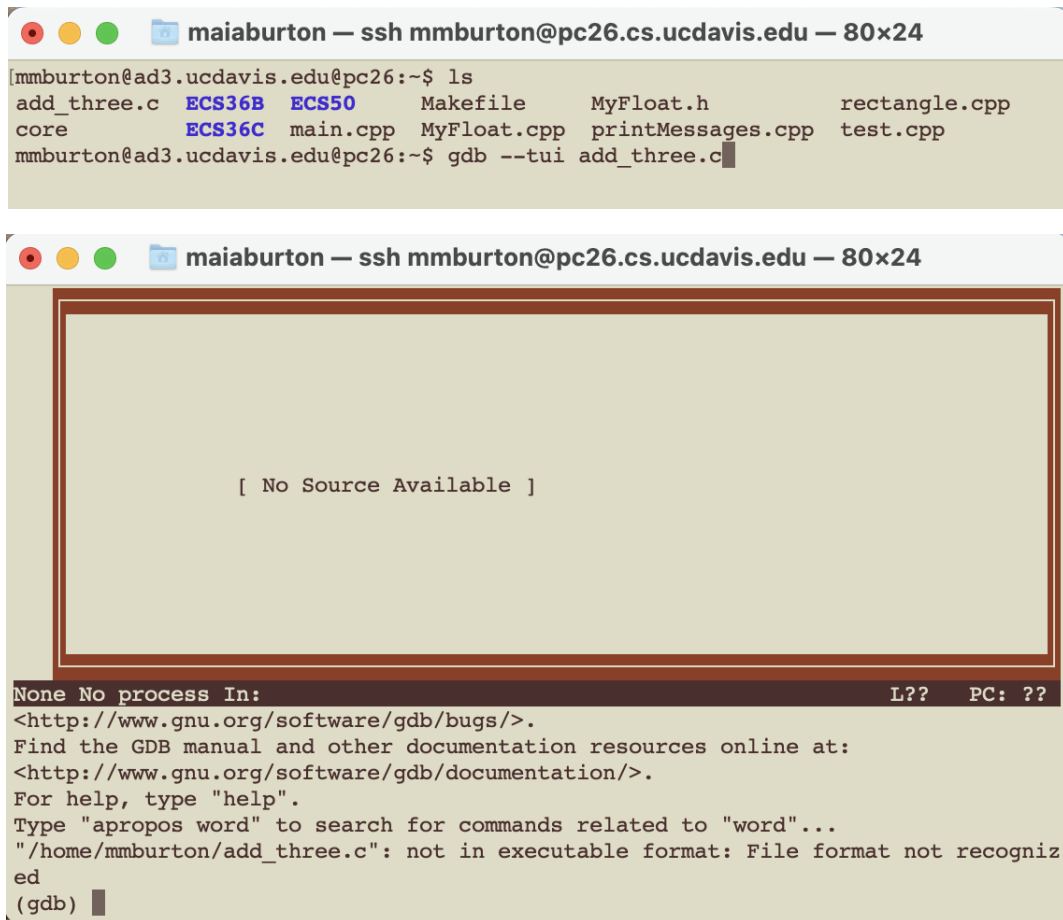
For C and C++ programs, the GDB and DDD are terminal based debuggers that you can use. DDD is more like a GUI interface that lets you display a little more in terms of where breakpoints are and in terms of variables. You can also do more clicking actions, but typically commands are typed in both options. Whichever option you choose, the commands should be the same. Below is how to open each one on your terminal window.

- GDB: `gdb --tui [nameOfExecutable]`
- DDD: `ddd [nameOfExecutable]`

Once you've selected and opened a terminal debugger program, the interface will look blank. Hit ENTER/RETURN on your keyboard to display your code.

2.3.4 Example: Attempting to Debug the Source Code - Incorrect Way

In the first image, the programmer displays all the files in their current directory including `add_three.c`. The programmer then attempts to open the GDB with the C language source file. While the second image is normal when opening the GDB, the code does not appear after the programmer hits ENTER/RETURN. Instead, they receive the message: “`add_three.c: not in executable format`”.

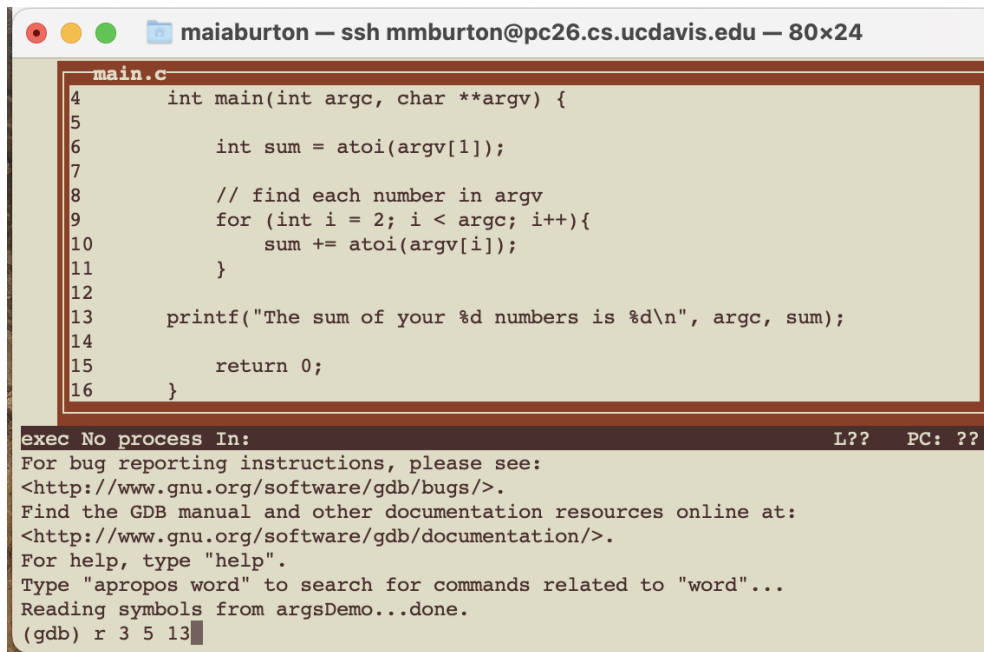


The first screenshot shows a terminal window titled "maiaburton — ssh mmburton@pc26.cs.ucdavis.edu — 80x24". The user runs `ls` and lists files: `add_three.c`, `ECS36B`, `ECS50`, `Makefile`, `MyFloat.h`, `rectangle.cpp`, `core`, `ECS36C`, `main.cpp`, `MyFloat.cpp`, `printMessages.cpp`, and `test.cpp`. Then, the user runs `gdb --tui add_three.c`. The second screenshot shows the same terminal window. A large rectangular box with a red border is centered on the screen, containing the text "[No Source Available]". Below this box, the terminal shows the GDB prompt `(gdb)` and a message: `None No process In: L?? PC: ??`. Below this, there is a link to the GDB manual: `<http://www.gnu.org/software/gdb/bugs/>`. Then, it says: `Find the GDB manual and other documentation resources online at: <http://www.gnu.org/software/gdb/documentation/>`. For help, type "help". Type "apropos word" to search for commands related to "word"... `"/home/mmburton/add_three.c": not in executable format: File format not recognized` (gdb) █

2.3.5 Example: Debugging with Command Lines

A program that requires command line arguments will compile and open on the GDB exactly the same as a program without them. To debug with command line arguments, simply append them after the `run` command. In this example, the executable is named `argsDemo`.

`argsDemo` takes in any amount of integer command line arguments and outputs their sum. In the image below, the debugger is already opened with the program's executable. To run the debugger, the programmer types `run` followed by the numbers 3, 5, and 13.



```
maiaburton — ssh mmburton@pc26.cs.ucdavis.edu — 80x24
main.c
4   int main(int argc, char **argv) {
5
6       int sum = atoi(argv[1]);
7
8       // find each number in argv
9       for (int i = 2; i < argc; i++){
10          sum += atoi(argv[i]);
11      }
12
13      printf("The sum of your %d numbers is %d\n", argc, sum);
14
15      return 0;
16  }

exec No process in:                               L??  PC: ??
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from argsDemo...done.
(gdb) r 3 5 13
```

NOTE: `argc` is the number of command line arguments and `argv` is the name of the array containing the command line arguments. Also remember that `argc` equals one more than the number of command line arguments you enter because the program name counts as a command line argument. `argc = {argsDemo, 3, 5, 13}`.

2.3.6 Additional GDB Info

1. Shortcuts: The GDB and DDD only need a unique prefix form of their commands. For instance, the command `continue` can be run with `c`, `co`, `con`, `cont`, etc. In this guide, the smallest unique prefix will be red within the command word - `continue`.

2. Optional Parts: Some GDB and DDD commands will be shown with square brackets. These are **optional** parts of the command such as `run [command line arguments]`. If your program doesn't require command line arguments, you can just type `run`.

3. Repetitive Commands: Hitting **Enter** on your keyboard without typing a command will prompt the GDB to run the last line that was run. This is useful if you want to run the program one line at a time and don't want to type the same command over and over again. This is commonly done with **step**, **next**, or **continue**.

2.4 IntelliJ Debugging

If you are using IntelliJ, you most likely will not need any help getting the debugger feature working. It depends on how your project was created.

2.4.1 IntelliJ Makes Your Project

If you have created a project in any IntelliJ platform, your CMake file is automatically created for you and no work is needed to compile and debug your program. You can skip on to the next section and simply click on the debugger icon to start debugging.

2.4.2 IntelliJ Doesn't Make Your Project

If you did NOT create a project on the IntelliJ platform, you will need to configure your program yourself which is taught in the following section.

2.4.3 Additional Notes

There are some GDB commands, such as **watch**, which have no equivalent in IntelliJ. If you want to use a debugging feature that IntelliJ doesn't have, you can write GDB commands into the GDB tab under the debugging window of IntelliJ.

Debugging Command Categories

- **Execution Commands:** involve actions that affect how the program executes.
 - Does the program execute one line at a time or multiple lines at a time?
 - Does a function call count as one line of execution or does each line within the function count as well?

Breakpoint Commands: are commands that pause the program at specific locations that you choose.

- If your project doesn't complete, you may want to pause the program halfway through and see if anything is wrong.
 - Why does your program break when a variable changes, but only sometimes?
- **Examining Commands:** allow you to verify the values of your expressions.
 - Are the values of your array how they should be?
 - Is my boolean variable true?

3 Execution Commands


Execution commands refer to actions that affect the execution of the program. Your examining commands will then let you view these values.

3.1 Run Command

→ **What Does It Do?**

The run command starts the program on the debugger with any given command-line arguments (if provided). If you use the `run` command when the program is already running, it will restart the program from the beginning.

→ **How Does it Work on the Debugger?**



- **GDB** - Type `run [commandLine args]`
- **IntelliJ** - Click  * Edit run configurations with command lines *

→ **Coding Clue**

Set breakpoints before running your program, otherwise the program will run through without giving you any information about your error. **You need at least ONE breakpoint before starting the GDB or DDD debugger.**

3.1.1 NOTE: IntelliJ's Play

IntelliJ has both a play and debugging button. Even if you have breakpoints set, the run button will not stop the program when it reaches one.


- To run your program **without testing**, click 
- To investigate your code and test it, click 

3.2 Continue Command

→ What Does It Do?

Continue/Resume will resume running your program on the debugger until your program ends, your program crashes, or the debugger encounters another breakpoint. Your program must have started to run in order for it to resume.

→ How Does it Work on the Debugger?

- GDB - Type `continue`
- IntelliJ - Click []

→ When to Use it?

You should resume the debugger after verifying your variables match your expectations and decide to continue to the next breakpoint. You should be confident that the code between the the current breakpoint and the next works and won't need to be inspected further. If not, you should examine each line closely using either `next` or `step`.

3.3 Finish Command

→ What Does It Do?

Finish will continues the execution of the current function until it returns to its caller, ignoring all breakpoints. Suppose function A calls function B and execution has stopped in B. The `finish` command would return you to function A, at the point where function A called function B. If you are in the `main` function, using finish will work the same way as the continue command.

→ How Does it Work on the Debugger?


- GDB - Type `finish`
- IntelliJ - Use the `step out` button in the debugger toolbar. This will continue execution until the current function returns to its caller, siimilar to GDB's `finish`.

→ **When to Use it?**

This command is helpful when you accidentally step into a function or you are done inspecting its code.

3.4 Step Commands


Step Into - Runs the next line of code, if there's a function call to go into, it **WILL** go to the start of the function and stop there.

- **GDB**: Type `step [count]`
- **IntelliJ** - Click [

→ **When to Use Next?**

Use this command when you want to see inside the function you're calling. If you have not verified whether or not the function being called returns the correct result, use `step`.

Step Over - Runs the current line of code, but **does NOT** go into function calls. The entire function is run, and the debugger will stop at the next line of the caller method.

- **GDB** - Type `next [count]`
- **IntelliJ** - Click [

→ **When to Use Step?**

If you have already verified that the function being called produces the correct result, you should use `next`.

3.4.1 What Does Count Mean?

In GDB, `step` and `next` will run one line of code by default, but you can specify a different number if you want.

REMINDER: If you hit **Enter** without a command in GDB, the debugger will run the last line you entered. If you specified `step 5` once and hit enter again, you will move another 5 steps. The same applies for `next`.

4 Breakpoint Commands

→ What are Breakpoints?

Breakpoints are special markers that pause the program at a specific location or condition. This lets you examine the program's state and behavior at those locations.

→ When to Create Breakpoints

Breakpoints can be set either **before** you use the `run` command or while the program is paused. This means you need to set at least one before starting your program.

4.1 Setting Breakpoints

- **GDB** - Type `break [location]`
- **IntelliJ** - Click in the gutter of your user interface. This is on the left side of your screen between the line number and your code

4.1.1 → GDB Breakpoint Location

The location of a breakpoint can be specified by a file name and a line number. If a file name is not given, the specified line number will be placed in the current file. Additionally, if you are only debugging one file, a file name is not needed.

The location section of a breakpoint can be a line number of the current file you are in, a specified line in a specified file, or a function name.

- `break 13`
- `break [file_name]:[line_number]`
- `break funcA`

```

1  #include <stdio.h>
2  int main() {
3      //Defined integer variables
4      int num1 = 0;
5      int num2 = 0;
6      int num3 = 0;
7
8      //Prompt user for three integers
9      printf("Enter first integer: ");
10     scanf("%d", &num1);
11     printf("Enter second integer: ");
12     scanf("%d", &num2);
13     printf("Enter third integer: ");
14     scanf("%d", &num3);
15
16     //Print the sum
17     printf("The sum is: %d\n", num1 + num2 + num3);
18     //return 0 is the end of the program
19     return 0;
20 }

```

The red dot in the IntelliJ screenshot is where you need to click with your cursor to set a breakpoint. The lighter gray section is called the **gutter**.

→ How to Use Breakpoints

Place breakpoints in the areas you suspect are the most likely places to cause an error. When the program pauses at a breakpoint, verify your program's results are correct. Once you've verified the most likely places an error would be but still haven't found it, you need to widen your search and place them in new spots.

If you don't have any guess at all where an error might be, place a breakpoint at the halfway point of execution.

4.1.2 Halfway Point of Execution

If you can't make an estimated guess where the error is: Place a breakpoint at the halfway point of **execution**. The halfway point of execution is the halfway point from the moment your program begins running, **NOT** the middle of your program file. Scan over your program and try to identify the high level steps.

→ Halfway Point Example

```
1. int addTwo(int num1, int num2) {
2.     int c;
3.     c = num1 + num2;
4.     return c
5. }
6.
7. int main() {
8.     int sum;
9.     int a, b;
10.
11.     printf("Enter the first and 12. second number \n");
13.     scanf("%d %d, &a, &b);
14.     sum = addTwo(a, b); // call addTwo function
15.     printf("The sum of the two numbers is %d", sum);
16.
17. }
```

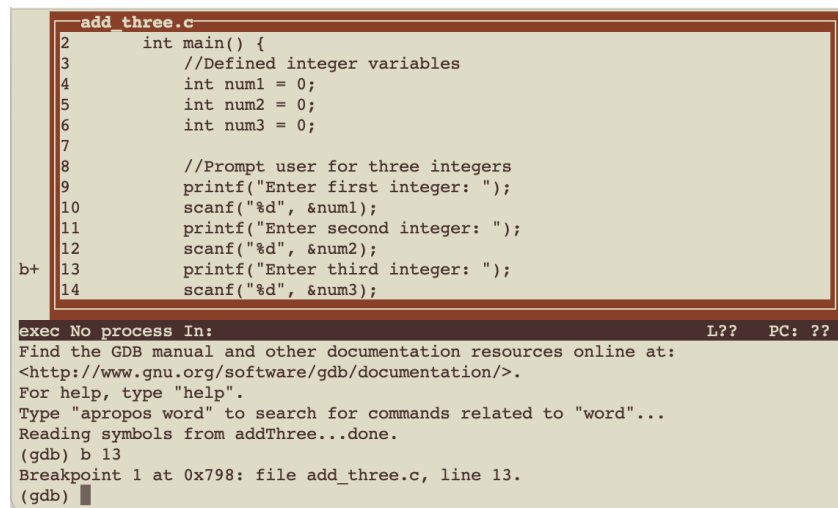
//addTwo.c

In the example program above, the halfway point of the file would be on line 8 or 9. However, following the path of execution, the program starts at `main()` and steps into `addTwo()` on line 14. This means that the *halfway point of execution* is on line 2.¹

If the breakpoint was placed on line 8, which is the middle of the file, nothing would happen. The debugger would only execute one line of code - `int main()`. This would give you no information about how your program is doing.

¹**NOTE:** The header of the function doesn't truly have an impact on the program and is more for the computer to understand what you want from it. You placing a breakpoint on the function header line or the first line of the function does not make a difference. In the example, a breakpoint on line 1 and 2 is essentially the same.

4.1.3 GDB Breakpoint Example



```
add_three.c
2   int main() {
3       //Defined integer variables
4       int num1 = 0;
5       int num2 = 0;
6       int num3 = 0;
7
8       //Prompt user for three integers
9       printf("Enter first integer: ");
10      scanf("%d", &num1);
11      printf("Enter second integer: ");
12      scanf("%d", &num2);
13      printf("Enter third integer: ");
14      scanf("%d", &num3);
15  }
```

```
exec No process in: L?? PC: ??
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from addThree...done.
(gdb) b 13
Breakpoint 1 at 0x798: file add_three.c, line 13.
(gdb)
```

Because the example program only has one file, the following two examples will both place a breakpoint on line 13.

- `break 13`
- `break addTwo.c:13`

4.2 Conditional Breakpoints

→ What Do Conditional Breakpoints Do?

Conditional breakpoints will only pause a program if the specified condition is true.

→ How Does it Work on the Debugger?

- **GDB** - Type `break [location] if [conditional statement]`
- **IntelliJ** - Set a regular breakpoint, then right click the red breakpoint that appears. Define your conditional in the panel that opens.

→ When to Use Conditional Breakpoints?

There are two main reasons to use a conditional breakpoint.

1. If a function is called multiple times throughout a program but is only failing during specific calls, it is most likely that it is being passed invalid arguments. A conditional statement will help you examine only the function call that is failing instead of pausing at every single call.
2. If there is an error in a loop, but only at a specific iteration, a conditional statement can allow you to skip past all the iterations that operate correctly.

4.2.1 Conditional Breakpoint Example

What the Program Does

The C code below shows a function called `convert()` that takes in a character in the English alphabet and outputs the letter's numerical position. A = 1, B = 2, ..., Z = 26. The library function `toupper()` turns any input letter to its uppercase equivalent and only one array is needed instead of one lowercase and one upper.

```
1. \\ alphabetNumbers.c
2. unsigned convert(char letter){
3.
4.     char nums[26] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
5.     'L', 'M', 'N', 'O', 'P', 'Q', 'R', 's', 'T', 'U', 'V', 'W', 'X', 'Y',
6.     char capLetter = toupper(letter);
7.
8.     // search for the letter in the array
9.     for (unsigned letterNum = 0; letterNum < sizeof(nums); letterNum++){
10.         if (capLetter == nums[letterNum]){
11.             return letterNum + 1;
12.         }
13.     }
14.     return 0;
15. }
```

The Problem:

All the letters of the alphabet work the way they should except for the letter S, lower or uppercase. This is because the `char` array that the for loop is searching has all capital letters except for S. If this function were in a bigger program and called multiple times throughout runtime, a conditional breakpoint would be very helpful.

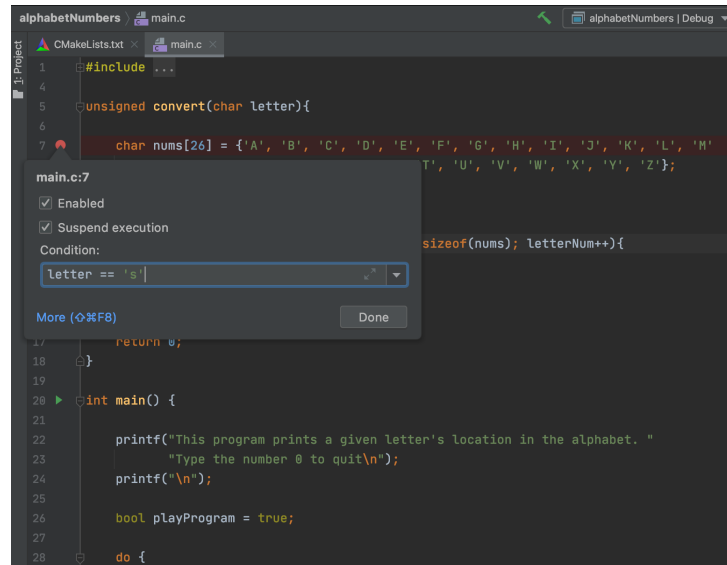
Why Use a Conditional Breakpoint?

Suppose you call this function for each letter in the alphabet and discover S doesn't work. Rather than place a regular breakpoint at the beginning of the function, stopping 17 times before reaching S, you can use a conditional statement that only pauses your program when the letter S is input.

Using the Conditional Breakpoint

The GDB command - `break 8 if letter == 's'` places a breakpoint at the beginning of the for loop and pauses the debugger only if the input is `s`. Nothing will happen when other letters are entered.

You can also write longer conditional breakpoints such as with an `or` statement. To be case insensitive, the conditional statement can be written as `break 8 if letter == 's' || letter == 'S'`.



For IntelliJ, set a regular breakpoint - by clicking in the gutter. Then right click the breakpoint to open a panel shown above. Simply type in the conditional you want to set, then click Done.

4.3 Watchpoint Command

What are Watchpoints?

A watchpoint will pause the program every time a specified expression changes.

→ **How Does it Work on the Debugger?**

- **GDB** - Type `watch [expression]`
- **IntelliJ** - IntelliJ does **NOT** have a **watch**. What IntelliJ calls watch, is essentially a GDB *display* ². Watch does not exist for IntelliJ platforms.

→ **When to Use it?**

Use watchpoints when you know something is going wrong with a variable, but you don't know when/where. This is also helpful if you want to monitor the behavior of an expression throughout the program without having to predict a particular place where this may happen.

Like conditional breakpoints, you may end up pausing your program more than you would like. It can be helpful to use more complex expressions to be more efficient.

²See the *Examining Commands* section

4.4 Breakpoint *Info* Command

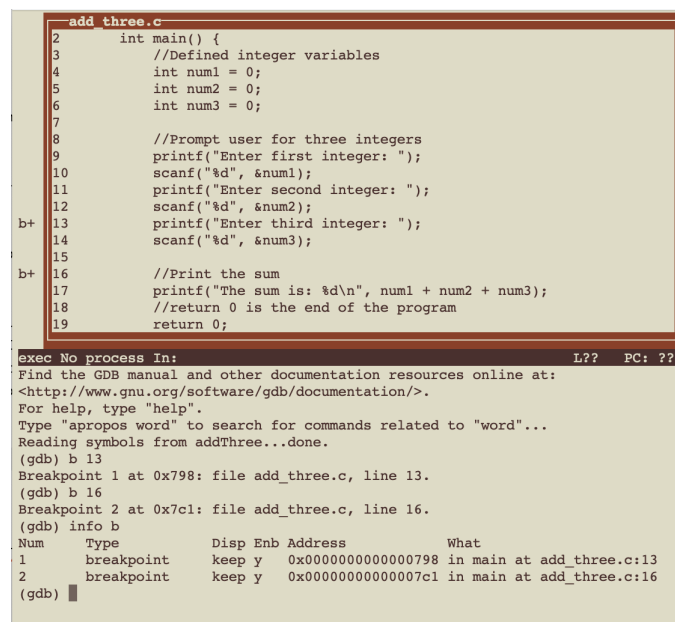
→ What Does it Do?

The `info` command is used in the GDB to show you all the information about a specified field. After typing `info break` into the GDB on a program, the debugger will display information about all different types of breakpoints

- Type of breakpoint
- ID number
- Whether it is active or not
- Its location

The assigned ID's are used to delete, disable, and enable breakpoints. These characteristics are explained in the next section.

- GDB - Type `info break`



```
add_three.c
2      int main() {
3          //Defined integer variables
4          int num1 = 0;
5          int num2 = 0;
6          int num3 = 0;
7
8          //Prompt user for three integers
9          printf("Enter first integer: ");
10         scanf("%d", &num1);
11         printf("Enter second integer: ");
12         scanf("%d", &num2);
b+ 13         printf("Enter third integer: ");
14         scanf("%d", &num3);
15
b+ 16         //Print the sum
17         printf("The sum is: %d\n", num1 + num2 + num3);
18         //return 0 is the end of the program
19         return 0;
}

exec No process in: L?? PC: ??
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from addThree...done.
(gdb) b 13
Breakpoint 1 at 0x798: file add_three.c, line 13.
(gdb) b 16
Breakpoint 2 at 0x7c1: file add_three.c, line 16.
(gdb) info b
Num      Type             Disp Enb Address            What
1        breakpoint       keep y  0x0000000000000798 in main at add_three.c:13
2        breakpoint       keep y  0x00000000000007c1 in main at add_three.c:16
(gdb)
```

The example image above shows two existing breakpoints.

- **Num:** The breakpoint ID
- **Type:** Regular, conditional, watch, etc
- **Disp:** Specifies what happens to the breakpoint when it gets hit.
 - **Keep** means that the breakpoint will not be removed.
 - **Del** means that the breakpoint will be deleted after the first hit.
- **Enb:** Specifies whether the debugger ignores or listen to the breakpoint
- **What:** The location of the breakpoint

4.5 Breakpoint Delete, Disable, Enable Commands

→ What Do They Do?

- **Delete:** Removes a breakpoint
- **Disable/Enable:** “Comments” and “Uncomments” breakpoints
The breakpoints will stay where they are placed, but the debugger will ignore them if the breakpoint is disabled

→ How Does it Work on the Debugger?

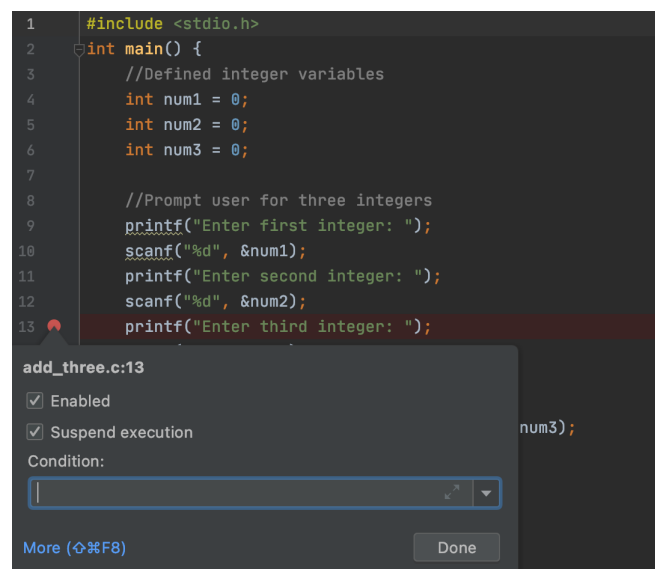
With the GDB:

- **Delete** - `Delete [breakpoint ID]` will remove a breakpoint.
- **Disable** - `Disable [breakpoint ID]` will keep the breakpoint where it is set, but will not pause the program when the line is reached.
- **Enable** - `enable [breakpoint ID]` undoes the disable feature on the breakpoint. The program will pause at the previously disabled breakpoint.

With IntelliJ:

The same function applies to IntelliJ.

- **Delete** - Click on the breakpoint
- **Disable and Enable** - Right-click the breakpoint and click the **enable** check box in the the dropdown panel



5 Examining Commands

The following commands in this section are used to view expressions and verify they produce the correct output.

In IntelliJ: You can view your expressions by rolling over them with your cursor, viewing the `variable` window, or the `watch` window.

5.1 IntelliJ Viewing Methods

5.2 Displaying and Printing Commands

→ What Do They Do? What's the Difference?

The `display` and `print` command will show you the value of an expression. The difference between `display` and `print` is its duration.

- `display [expression]` - Shows you the value of an expression until you disable the command
- `print [expression]` - Shows you the value of an expression once

→ Which Should You Use?

Choose which to use depending on how long you want to examine an expression. It's a good idea to display continuously changing expressions or ones that are difficult to visualize such as data structures.

→ What Should I Look At?

You should be verifying variables that you think are causing problems or might be contributing to the error. In general, the more the better.

5.3 Variable *Info* Command

As mentioned in the breakpoint info command section, the `info` command shows you information about a specified field. In addition to breakpoints, you can also view variables, arguments, and registers.

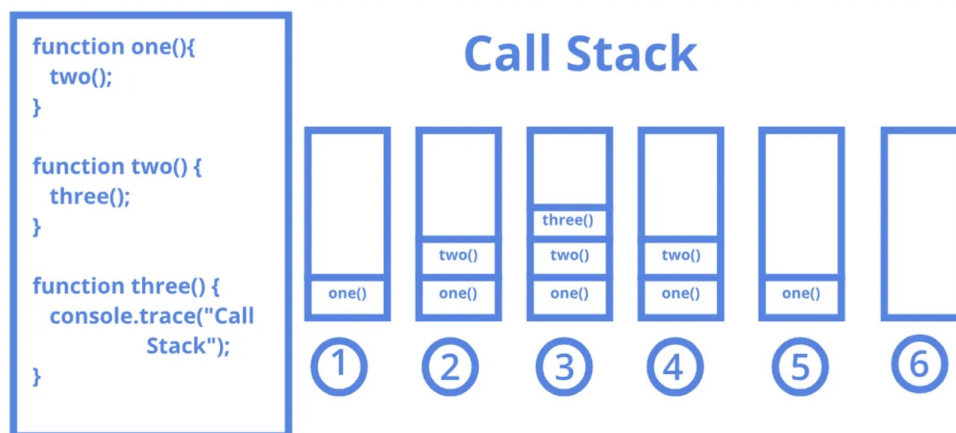
- `info locals` - Prints out the values of all the local variables
- `info args` - Prints out the values of all the arguments to a function
- `info regs` - Prints out the values of all the registers

5.4 Backtrace Command

→ What is a Stack?

A stack is a data structure that contains its elements in a specific way. The first element placed in it, is the last element to come out. Think of making a pile of books. The first book you use will be at the very bottom of the pile. You will have to take off books from the top first to reach the first one you put in.

A **call stack** is a stack that specifically stores information about the active subroutines of a computer program. In the image below, function `one()` is called first, which calls function `two()`, which calls function `three()`. Each numbered image, illustrates how the local variables of each function are stored and removed from the call stack. When execution leaves a function become its complete, all its data is saved. Once execution returns, it's removed from storage.



A stack is a chain of function calls. Sometimes when you are debugging a function, it can be helpful to see what was happening in the function that called it.

→ How Does it Work on the Debugger?

- **GDB** - Type `bt` or `backtrace`
- **IntelliJ** - Although IntelliJ does not have a direct `backtrace` command like GDB, you can view the call stack in the “Debugger” tab. This tab shows the call stack with the current function at the top, and you can navigate through it to inspect the sequence of function calls.

5.5 What does Backtrace Do?

A backtrace is a summary of how your program got where it is. It shows the series of function calls to get to the point where you are currently now and where you are at. It shows one line per call, starting with the currently executing function, followed by its caller, and on up

the chain.

Example:

Imagine function A called function B and then C and crashes on function D. After debugging function D, you know it works fine. A backtrace can show you if some function along the way passed invalid arguments.

5.6 When to Use Backtrace

Sometimes an error isn't because of the code in your function, but actually because it's been called inappropriately. When you want to see the series of function calls that led to a specific function call, use backtrace.

You want to know how the function or where it got called.

COMMON MISTAKES: Passing invalid arguments or infinite recursion are common mistakes and ones that can be detected using backtrace.

5.7 Up and Down Commands

The Up and Down commands are used with the backtrace command. Up will move you up the call stack and down will take you down.

5.7.1 Moving Up Command

- **GDB** - Use the **up** command to move one level up the call stack and **down** to move one level down. This helps in examining the sequence of function calls leading to the current point of execution
 - **up** [count] - moves up the call stack by one frame (or count frames if specified)
 - **down** [count] - moves down the call stack by one frame (or count frames if specified)
- **IntelliJ** - Click [^ ↑]

5.7.2 Moving Down Command

- **GDB** -
- **IntelliJ** - Click [^ ↓]

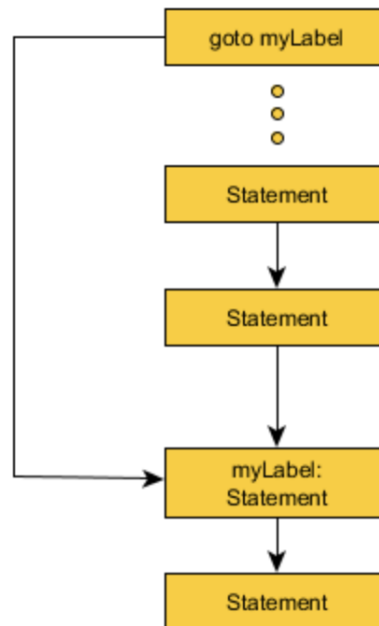
6 Assembly Language - UC Davis ECS 50

6.1 What are Labels?

Labels in assembly language are identifiers which mark a location in the code and are primarily used with jump commands (`jmp`, `jz`, `jnz`, etc), allowing the control flow of the program to move to the labeled line. This can be critical for implementing loops, conditional execution, and for structuring the flow of the program.

Using the `goto` statement, the execution of the program moves to the location of the label. When the code reaches the `goto` statement, the control of the program “jumps” to the label. Then the execution continues normally. If the execution reaches the labeled statement without a jump the program will execute it just like any other.

Labels themselves do not execute; instead, they serve as destinations within the control flow.



Simple Loop Example:

```
mov ecx, 5      ; set loop counter to 5
label_loop:     ; label marking the beginning of the loop
dec ecx         ; decrement loop counter
jnz label_loop  ; jump to 'label_loop' if zero flag is not set (ecx not zero)
```

Conditional Execution Example:

```
cmp eax, ebx    ; compare two registers
je equal        ; jump to label 'equal' if eax is equal to ebx
label_1:
```

```

; code executed if not equal
jmp end
equal:
; code executed if equal
end:

```

6.2 Labels in C and C++

When programming in higher-level languages like C or C++, it's often advised to avoid using `goto` because it can make the code difficult to follow and maintain. In assembly, however, structured programming constructs like `for`, `while`, and `if-else` are not available, making labels and jumps necessary for controlling program flow.

Coding Clue: When writing assembly language programs, it's smart to write it first in C or C++ which is easier to read, then translating it.

Converting C loop to Assembly

```

for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}

```

An Equivalent assembly version might look like this:

```

mov ecx, 0          ; Initialize counter
loop_start:
cmp ecx, 10         ; Compare counter with 10
jge loop_end       ; If counter is greater or equal to 10, exit loop
; Code to print value of ECX, simplified
inc ecx            ; Increment counter
jmp loop_start     ; Jump back to the start of the loop
loop_end:

```

6.3 Tips for Writing Assembly:

- **Label Usage:** Use descriptive labels to make the assembly code easier to understand. Avoid excessive jumps which can make the flow hard to follow.
- **Comments:** Always comment your assembly code extensively. Given its low-level nature and the manual control you have over the CPU, explaining your intent can greatly aid in maintenance and debugging.
- **Simulate Higher-level Constructs:** Try to structure your assembly code to mimic higher-level constructs where possible. This can make it easier to translate high-level logic into assembly and vice versa.

7 Glossary of Debugging Terms

Argument: A value provided to a function or program when it is called or initiated. In debugging, watching how arguments change can be crucial for identifying issues.

Backtrace: A summary of the current call stack, showing a list of all function calls that led to the current point of execution.

Breakpoint: A debugging tool that allows a program to pause execution at a specific point so that variables and program flow can be examined.

Command-Line Arguments: Parameters or flags passed to a program which affect how it runs. Debuggers like GDB allow you to specify these when starting a program with the run command.

Conditional Breakpoint: A breakpoint that triggers only when a specified condition is true, allowing for more targeted debugging.

Debugger: A tool that allows programmers to control the execution of a program, inspect the state of the program, and potentially alter its operation to track down errors.

Expression: Any valid unit of code that resolves to a value. In debugging, expressions can be monitored to see how values change over time.

Function Call Stack: The stack structure that stores information about the active functions/subroutines of a program. Debuggers can navigate this stack to show the sequence of function calls.

GDB (GNU Debugger): A popular debugger for programs written in C, C++, and other languages, providing detailed control over the debugging process.

IntelliJ: An integrated development environment (IDE) that includes tools for debugging Java and other languages, with a graphical user interface for managing breakpoints, watching expressions, and stepping through code.

Label: In assembly language, a marker for a position within source code, often used with jump commands to control flow of execution.

Step Into: A debugging command that executes the next line of code; if the line contains a function call, the debugger enters the called function.

Step Over: A debugging command that executes the next line of code but does not enter into functions (unlike Step Into), which means it treats the function call as a single step.

Watchpoint: A type of breakpoint that pauses the execution of a program when the value

of an expression changes, used to monitor variables or expressions for changes.