

{DEBUGGING CHEAT SHEET}

MAIA BURTON - ORGANIZED BY PROFESSOR BUTNER
UNIVERSITY OF CALIFORNIA - DAVIS

GETTING STARTED

Set Up

GDB: To run the debugger, type
`gdb --tui name_of_executable`

NOTE: Make sure to use the name of the *executable*, NOT the .cpp or .c file

IntelliJ: Most, if not all, IntelliJ commands are in the toolbar at the top of the screen.

Before You Begin

- **Know the Correct Answer:** Debugging lets you view the state of your expressions and locate where your program's behavior deviates from your expectations. If you don't know what your program **should** be doing, you can't know if it's doing something wrong.

- **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. If the input `cat` and `alphabet` will both cause an error, which would you rather examine step by step?

Additional Information

GDB Prefixes

GDB only needs a unique prefix form of a command to get started. The shortest possible prefix is bolded. For example, `continue`

Non-existing IntelliJ Commands

Some GDB commands have no equivalent in IntelliJ. To use a nonexistent IntelliJ command, you can write it in GDB into the GDB tab under the debugging window of IntelliJ.

Debugging Reference Guide



For examples and additional information for each command, see the reference guide.

EXECUTION COMMANDS

Run Command

Use the run command to start the debugger on your program. In GDB, type

`r`un [command line arguments]

On IntelliJ platforms, the 'play' symbol [] will run the program, **not** the debugger. To start the debugger, click on []

Continue - GDB; Pause/Resume - IntelliJ


Continue/Resume will run until your program ends, your program crashes, or the debugger encounters a breakpoint.

- **Continue - GDB** - Type `c`ontinue


- **Resume - IntelliJ** - Click []

When to Use it?: You should resume the debugger after verifying your variables match your expectations and you decide to continue to the next breakpoint.

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. In GDB: step into is denoted as `s`tep count . In IntelliJ, click [].

When to Use it?: When you want to see inside the function you're calling. If you have not verified if the function that's called returns the correct result.

- **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. In GDB: step over is denoted as `n`ext . In IntelliJ click the [].

When to Use it?: If you have already verified that the function that's called produces the correct result.

BREAKPOINT COMMANDS

What Is It?: Breakpoints are special markers that pause the program at a specific point. This lets you examine the program's state and behavior at those points.

When Can I Create One?: Breakpoints need to 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.

Setting Breakpoints

In GDB - Type `b`reak [location]
Location can be a line number, function name, or label.

On 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.

When to Use Breakpoints?: Place these in the areas you suspect are the most likely places to cause an error. When the program pauses verify the results are correct.

If you can't make an estimated guess where the error is, see the reference guide for help.

Conditional Breakpoints

Conditional breakpoints only stop a program if a statement is true.

- **In GDB** - type
`b`reak [loc.] if [cond. statement]

- **In IntelliJ** - Set a breakpoint and right-click it. This opens a side panel for you to set your conditional.

When to Use it?: If a function is called multiple times throughout a program, but only fails during specific calls or if there is an error in a loop but only after iteration x.

BREAKPOINT CONT.

Watchpoint Command

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

- **In GDB** - type `w`atch [expression]

- **In IntelliJ** - Right-click an expression and select Add to watch.

When to Use it?: If you want to monitor the behavior of an expression throughout the program without having to predict a particular place where this may happen.

Breakpoint Info Command

The `i`nfo `b`reak command is used in GDB, to show you all the breakpoints you have created. Each breakpoint will be displayed next to a number which is used as its ID.

You will want to use this command and the ID's to delete, disable, and enable breakpoints.

Break Point Delete, Disable, and Enable

- **Delete** - `d`elete [ID] will remove the breakpoint.

- **Disable** - `d`isable [ID] will keep the breakpoint where it was set, but will not pause the program when the line is reached.

- **Enable** - `e`nable [ID] undoes the disable feature on the breakpoint. The program will pause at the previously disabled breakpoint.

In IntelliJ - The same function applies to IntelliJ. To delete, click on the breakpoint. To disable and enable, right-click the breakpoint and follow the dropdown panel.

{DEBUGGING CHEAT SHEET}

MAIA BURTON - ORGANIZED BY PROFESSOR BUTNER
UNIVERSITY OF CALIFORNIA - DAVIS

EXAMINING COMMANDS

These commands are used to view and verify the expressions in your program.

In IntelliJ: You can view your expressions in a few ways: rolling over them with your mouse, viewing the variable window, or the watch window.

Display/Print Command

- `display` - Shows you the condition of an expression until you disable the command
- `print` - Shows you the condition of an expression once

Which Should I Use?: Choose depending on how long you want to examine an object. It's a good idea to display continuously changing expressions or ones difficult to visualize such as data structures.

What Should I Look At?: Variables you think are causing problems or might be contributing to the error. In general, more is better.

Variable *Info* Command

- `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 info about the registers

Viewing Labels

(*Assembly Language - ECS 50 Only*)

To print labels, they need to be type casted. If the label is the beginning of the array, you need the address of the label to get the array.

FUNCTION CALL STACK

Sometimes when you are debugging a function, you need to see what is going on in the function that called the current one.

What is a function stack?

When a function is called, a stack frame is created at the top of the stack with function details like parameters, local variables, and the return address. As functions call other functions, new frames are stacked on top of each other. When a function completes execution, its stack is removed and control is passed back to the calling function.

If `func_a` called `func_b` called `func_c`, and you set a break point inside of `func_c`, the call stack would look like:

- `func_a`
- `func_b`
- `func_c`

Backtrace

A backtrace is a summary of how your program got where it is. It shows one line per frame, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

When to Use it? Sometimes the error isn't your function, it's been called inappropriately. When you want to see the series of function calls that led to a specific function call use backtrace.

COMMON PROBLEMS: If you see your program is passing incorrect arguments to functions or infinite recursion.

- **GDB:** Type `bt` or `backtrace`
- **IntelliJ:** backtrace is not directly used with IntelliJ. You can use other facilities such as the Stack Trace Analyzer within IntelliJ, but this is not covered in this course

MORE EXAMINING

Up and Down Commands

Up and Down are part of the backtrace command. Up will move you up the call stack and down will take you down.

- **GDB:** up moves you one level up the function call stack. In other words, go back to the function that called the one you are in.

- **GDB:** down moves you one level down the function call stack. In other words, go to the function the one you are in is calling.

Arrays

To view an array, specify the array, the index to start, and the number of elements you want to view:

```
[command] theArrayName[startIndex]@[numOfElementsToView]
```

Example: `display nums[0]@len` assuming `len` has the length of the array, display all elements in `nums`

Example: `display nums[len - 5]@5` assuming `len` has the length of the `nums`, display the last 5 elements of `nums`

Example: `display nums[len - 5]@5` displays 3 elements before and after element at index `i` in `some_array`

NOTE: If you have a matrix that is **dynamically allocated**, you can only display one row at a time.

If you have a one big chunk array, the entire matrix can be printed by setting it to start at 0 and giving the length as `numRows * numCols`

OTHER INFORMATION

Common Mistakes

Command Line Arguments: Don't forget that `argv[0]` is the name of your program. The beginning of passed arguments to a problem from the terminal is `argv[1]`. If you are writing command line argument validation, don't forget to count the program name as an element.

Debugging Tips

Never assume that anything is correct. If you haven't double-checked it, it could be wrong. If you don't find the error where you think it will be, expand your search.

Your error might not be in the function where the issue becomes apparent. The error might be in the arguments that it was passed. Double check your function calls.

Four Rules of Thumb

1. **Always have an exit button:** Use try/catch functions, make sure you don't have infinite loops, have a literal exit button if applicable
2. **Leave comments**
3. **Be conscious of your global variables**
4. **Never assume your user will do what you want them to:** input validation, conditional statements

Finding a completely unknown error

If you don't even know where to start looking for an error:

1. Set a break point at the halfway execution point of your program
2. Check if the state of your program is correct. if it isn't, the error must be in the first half.
3. Set a new break point halfway between your new area of unknown
4. Repeat steps 2 and 3 until you locate your bug