

CSE 351

GDB Introduction

Lab 2

- Reading and understanding x86_64 assembly
- Debugging and disassembling programs
- Today:
 - General debugging for C with GDB

scanf and sscanf ()

- Lab 2 uses `sscanf` (string scan format), which parses a string into values

Example:

```
char *mystring = "123, 456"  
int a, b;  
sscanf(mystring, "%d, %d", &a, &b);
```

- The first argument is parsed according to the format string.
- After this code is run, `a = 123` and `b = 456`.

```
printf("Variable a=%d and b=%d\n", a, b);
```

- This will print to the console "Variable a=123 and b=456"
- Notice the similarities to `printf()`!

Format Specifier

- Notice the string formatter `"%d"`
 - `"%d"`, signed int
 - `"%u"`, unsigned int
 - `"%c"`, char
 - `"%f"`, float
 - `"%s"`, string (match until it finds white-space)
 - `"%x"`, hexadecimal int
 - `"%p"`, pointer
- Subtle differences between `printf` and `scanf`
 - <http://www.cplusplus.com/reference/cstdio/printf/>
 - <http://www.cplusplus.com/reference/cstdio/scanf/>

GDB Background

- GNU Debugger
- GDB can help you debug your program in four ways:
 - It can run your program
 - It can stop your program on specified conditions
 - It allows you to examine what has happened once the program has stopped
 - It allows you to modify your program's execution at runtime
- Today we will be going over many of the features that will make GDB a great resource for you this quarter
- Useful in future classes! CSE 333, CSE 451, CSE 484 etc.

Interactive Demonstration

- I encourage you to either follow along on your own machine or find someone next to you who is doing so.
- Download **calculator.c** from the class calendar page.
- We want to compile this file with debugging symbols included. To do this, we must use the **-g** flag in GCC:

```
gcc -Wall -std=gnu99 -g calculator.c -o calculator
```
- Without debugging symbols, GDB is not nearly as useful.

Loading the Program

- In order to load a binary into GDB, you simply pass the name of the executable to the **gdb** program.
- Try this on your machine:
 - **gdb calculator**
- You should see a bunch of version and license information.
- The last line before the (**gdb**) prompt is always the symbol loading status.
 - If you ever see (**no debugging symbols found**) you may have a problem.
 - In this case, you should see no such message.

Exiting GDB

- Before we go any further, it might be helpful to understand how to exit GDB.
- There are multiple ways to exit:
 - **Ctrl-D**
 - Typing **quit**
 - Typing **q**
- Many GDB commands can simply be abbreviated to their first letter.
- If you ever want to stop the current GDB command, just use **Ctrl-C**.

Running the Executable

- There are multiple ways you can begin execution of a program in GDB.
- The **run** command will start your program and keep running until it hits a critical error or the program finishes.
 - Try entering **run**, or just **r**.
- The **start** command will load your program into memory and break at the beginning of **main()**
 - You will see that most times **run** is all you need, but there are cases when you want to just start stepping through **main()**.
- If you want to specify command-line arguments, you just pass those to **run** or **start**.
 - To run **calculator**, we need to pass three arguments.
 - Try entering: **run 2 3 +**

Viewing Program Source Code

- To examine source code while debugging use the **list** (**l** for short) command.
 - Useful when trying to find line numbers.
- For example, let's look at the code for **main()**.
- Type **list main**.
 - This will display 10 lines of code around the beginning of the **main()** function.
- If you want to display 10 lines around line 45, enter **list 45**
- If you want to display a range of line numbers, such as lines 30-70, enter **list 30,70**

Setting Breakpoints

- In order to step through code, we need to be able to pause execution.
- GDB allows you to set breakpoints, just like when you debugged Java programs in Eclipse or jGRASP.
- The **break** (**b** for short) command creates breakpoints.
- Let's set a breakpoint at the entry to **main()**.
 - Enter: **break main**
- Now enter **run** and see the program break at **main()**.
- Each breakpoint is given a number.
 - Our breakpoint is given the number 1.
 - To disable our breakpoint temporarily, enter: **disable 1**
 - To enable our breakpoint again, enter: **enable 1**
 - To delete our breakpoint, enter: **delete 1**
- To see a summary of all your breakpoints, enter: **info break**

Stepping Through Code

- The **next** (**n** for short) command allows you to step through one line of C code at a time, stepping over function calls.
- The **step** (**s** for short) command is the same, except it steps into function calls.
- The **finish** (**fin** for short) command, steps out of the function.
- It works exactly like you would hope, most of the time...
 - Caveat: if you loaded some external library that was not compiled with debugging symbols, then calls to that library will look confusing when you step into them.
- Break your program at the beginning of **main()**, enter **next** until you arrive at a call to **printf()**, and then enter **step** to step into the call to **printf()**.
 - Note that it doesn't step into that function call, because it wasn't compiled with debugging symbols
- If you have halted execution and wish to continue running the program, use the **continue** (**c** for short) command.
 - Use that now to run the program to completion.

Printing Variables

- GDB has its own print function that is extremely useful.
- Let's print out our command-line arguments in various formats.
- Set a breakpoint on line 47 by entering: **b 47**
- Restart running the calculator program with some custom command line arguments.
- Continue until the breakpoint on line 47 is hit.
- Once there, print out the values of the three variables holding your arguments (**a, b, operator**) by typing the following:
 - **print a**
 - **print b**
 - **print operator**

Printing Variables (cont.)

- Now let's try printing out the values of the variables in different formats.
- Try the following:
 - `print /x operator`
 - `print /t a`
 - `print print_operation`
 - `print *argv`
 - `print *argv[1]`
 - `print *argv[3]`
- What do each of these do?

Debugging

- Let's look at how GDB enables us to easily identify runtime errors.
- Try making the program divide by zero:
 - `run 1 0 /`
- If you keep continuing, eventually the program will throw an arithmetic exception, and GDB will tell you exactly that.
- If you want to see a backtrace, just type **backtrace** (**bt** for short) and it will show you the chain of function calls that led to the error.
 - Viewing a backtrace can be very helpful in debugging.

Future Topics

- Next week we will be going over some more advanced topics to get you through Lab 2
- These include, but are not limited to:
 - Disassembling programs
 - Stepping through assembly code
 - Printing register values
 - Examining memory
- If time permits, we can start getting into some of those now, but if not feel free to start messing with those on your own.