Debugging

1. Why debug
2. Avoiding Problems
3. Recognizing Errors
4. Using GDB
6. Debugging thought process
6. Using a GUI/IDE (CLion etc.)
7. Quick note on Valgrind

Why Debug

- Because your program won’t “compile”
- Because your program “crashes”
- Because your program “doesn’t work”
- Because your program “works”
  - Seriously

Avoiding Problems

- Think → plan → paper → type → test → fix → integrate
- Don’t write your code in one big file
- Keep versions of working code (git hub is great)
- Break your program/problem/solution/code into pieces (e.g. functions)
  - Don’t Repeat Yourself (DRY)
    - Copy-pasting code is easier to screw up than you think
- Take your time. It’s easy to cut corners, take shortcuts, or write a ton of code all at once to “save time”
  - But then when problems come they are hard to find and take longer to solve
  - Spend a bit more time up front to do things carefully and test everything along the way
    - Prevents problems
    - Problems show up in controlled environment
    - Solve problems before they become a bigger problem (when integrated into your whole program)
    - Slow and steady wins the race
- “Unit testing”:
  - Write a piece of code, e.g. a function, and then write a small program to test it.
    - Tedious to compile a new program for every function right?
      - I’ll give some makefiles that make the process easier.
  - Also let’s you isolate an issue while debugging

“Compile” Errors

- Build system errors
  - When your build system is setup incorrectly you can’t even try to compile
- Compiler errors
  - When there are problems in your code
- Linker errors
  - When your code is (possibly) fine but it can’t be put together with the rest of your code into a whole program

Build System Errors

- Typically for C/C++ we use *make* which runs *recipes* defined in a *makefile*
- It’s old and finicky and annoying and powerful and great and terrible too
- CMake, Automake, and some other build systems are tools make using *make* easier
  - CLion uses Cmake to generate a Makefile

Build System Errors

- When *make* has errors, you’ll see something like:
  - make: *** [something] Error 1
  - make: *** No rule to make ‘something’. Stop
- But often times when compiling or linking fails, *make* will report that.
- Try to recognize when the error is with *make* or with your code. For example, your *recipe* might not be including all the source (object) files it needs for the *target*
- If might complain about “separators”:
  - In the good old days tabs were tabs and spaces were spaces
  - Nowadays people like replacing tabs with spaces
    - Your editor might do this: “\space\space\space\space” when you press “tab”
  - *Make* wants tabs to be a tab (\t character)

Compiler Errors

- When you “compile” a “program” two things happen:
  - Source code gets turned into *object* files
    - main.cpp → main.o
  - Object files (and libraries) get *linked* into a new file
    - main.o + “C++ Standard Library” → a.out
      - Things like std::cout aren’t built into the language, they are part of a library. That’s why you #include <iostream>. This can cause issues, where you think “oh I’m using this standard thing why is it complaining?” when forgetting the #include
  - Depending on how you set things up the object (.o) part might be hidden from view (gcc / g++ can do it in one step).

Compiler Errors

- Can get really long and complicated *looking*
  - Errors cascade. Especially true for “parse” errors.
  - A small mistake, like forgetting or having an extra ; }) , > or] makes what comes later look broken, even if it’s fine
    - The compiler is trying to match your code with grammar rules. If you mistype something, it might keep going past where it should when applying a rule, and so subsequent rules are getting applied strangely, leading to more errors, and so on.

Compiler Errors

- So your terminal window has filled up with hundreds of errors, words and filenames you don’t recognize
    - Don’t despair, just scroll up.
    - Typically the main cause of your error is at the top of the list
    - Typically it will tell you exactly where the problems started: the file, line number, sometimes even a little arrow pointing at the issue, sometimes even a suggestion on how to fix it
    - You might have to read a few lines
        - The error may say that it’s coming from a function, in a file, from another file, etc.
            - It’s trying to help you *trace* a path to the problem
        - Just stop reading when you see complicated code and filenames, function names you don’t recognize.
            - For example, using C++ templates wrong can lead to some really excessively long error messages telling you all about how what you did didn’t work with the “behind the scenes” code
- Many languages are similar, but you might be looking at the bottom of the errors and reading up

Linker Errors

- Look for the words “ld” or “collect”: that’s the linker
- Most common linker error is “undefined reference to <something>”
  - There’s a big difference between *declaring* and *defining* something.
  - void function_name(int a, int b); // declaration
  - void function_name(int a, int b) { ‘’ definition

        return a + b;

        }
- The compiler is ok with you calling any function that’s been *declared*
- The *linker* gets upset when the function hasn’t been *defined*
- These issues can also affect variables, but rarely (usually a variable declaration is also a definition:
  - Not true for **extern** *“global variables”*

        extern std::string dog_name; // variable declaration in HEADER file

        std::string dog_name = “Ford”; // variable definition in SOURCE file … technically it’s also a declaration
- Sometimes the compiler and linker are both ok with you having an undeclared function, but it will give you a warning “implicit declaration”. You don’t want to implicitly declare functions. Treat this warning as an error

Warnings vs. Errors

- Treat warnings as errors. Look them up, assume you made a mistake and wrote code that doesn’t do what you want it to.
- Some warnings are unavoidable, especially when working with really old code in the C standard library
  - e.g. a lot of stuff returns integral types equivalent to what you want but not what you want
  - Or you may have a “narrowing”, e.g. :

    ```
    long int banana_man();
    int monkey = banana_man();
    ```
- A common warning, especially when dealing with pointers, is “something ***makes*** something ***from*** something ***without a cast***”
- If you see those words stop and investigate thoroughly

  for example:

  ```
  int bad_idea() { return INT_MAX; }
  int *x;
  x = bad_idea(); // this would cause a “makes pointer from integer without a cast” error
  ```
    - You can “fix” it like this:

      ```
      x =(int *) bad_idea(); // the (int *) is a cast.  Casts convert one data type to another.
      ```
  - Don’t use casts.
  - If you are 100% sure that you need a cast for what you’re doing, then do something else instead
    - Solve all casting issues without casts.
    - Sometimes you gotta cast though
      - Nope.

Typos

- Typos are a massive source of errors
  - Leaving out a ; ,)] }
  - Typing a name wrong
  - Typing the wrong name
    - Give your variables recognizable names!

      if (k – i > j) a[i] = b[j-k]; // what?????
      - Happens a lot when grabbing code from the internet, books, professors. Make your code *self-documenting*.

Consistency

- A ton of different errors can happen because you:
  - Didn’t include a header
  - Didn’t compile with the needed source files
  - Call a function with the wrong arguments
  - Made a name typo
  - Don’t have a definition
  - Don’t have a declaration
  - And so on…

Consistency

- When reading errors if you see any of these:

  “implicit declaration” // often a typo, or forgot to include a header (or declare in the header)

  “error: … not declared in this scope” // often a typo, or forgot to include a header (or declare in the header)

  “undefined reference” // you’ve got the declaration but not the definition. Typo? Compiling (linking) the needed .cpp (.o) file?

  “multiple definition of” // exactly what it sounds like. You defined something, either a variable, or a function with the same signature, multiple times

  “redefinition of” // are you using include guards?

  “two few arguments” // you called the function with fewer arguments than you declared it with

  “error: cannot convert” // you used the wrong types in a function call or assignment

  - “no known conversion”

  “no matching...” // did you call your function with wrong types or number of arguments?

  “conflicting types” // do the *function signature* in your declaration and definition match?

  - It’s probably a problem with consistency.
    - Check for typos
    - Check that you’re including the right header
    - Check that your header has include guards

      ```
      #ifndef HEADER_NAME_H
      #define HEADER_NAME_H
      /* header code goes here */
      #endif // HEADER_NAME_H
      ```

    - Check that you declaration and definition match
    - Check that you are calling the function with the right number and types of parameters
    - Check that the types on both side of an assignment match
    - Check that you are linking the needed source files together

Debugging Thought Process

- Why is this happening now?????????
  - The deadline is in two hours gosh darn it
  - I just want to sleep.
  - please kill me now
- What happened?
  - “it” isn’t working
- Well what is “it”?
  - What was supposed to happen?
  - Why did that happen instead?
  - Where did it happen?

Debugging Thought Process

- WHAT went wrong

  versus

- WHY did it go wrong

  versus

- WHERE did it go wrong

  versus

- What, where, and why WOULD it go wrong

  ^^^^^ this ^^^^^

How WOULD it break?

- Two ways to find source of a problem:
  - Go step by step and figure out what happened
    - Bewildering. Doesn’t make sense unless you know why you wrote broken code, but if you knew why it was broken, then you wouldn’t have written it that way….
  - Ask yourself why it would go wrong and work backwards
    - “my problem is: <this> which could only happen if <that> , so why did <that>”
- Or ask a rubber duck
    - Talking through a problem helps. Having a rubber duck sitting next to you makes you look a little less crazy when talking to yourself
      - Also tricks your brain into helping remember stuff
- Or ask a person
  - Very helpful
  - But now your talking about Rick and Morty and making plans for lunch and your code is still broken

Debugging Tools/Methods

- Print statements
  - Honestly helpful, especially when writing code for the first time. But when debugging...
  - Everyone does it and knows they shouldn’t

    ```
    cout << “I got here” >> endl;
    cout << “Variable x is:” << y
    ```

  - Problems:
    - Adding more code to broken code, shouldn’t we fix the broken part before adding more?
    - What if the print statement is broken?
    - Do we even know what we need to print?
      - lot’s of guess work, changing things, recompiling and running over and over and over again
    - Messy
    - Have to remove them later

Debugging Tools/Methods

- Print statements
  - Anything you can do with a print statement you can do with a debugger
  - If you are going to be running a lot of code and want to track its progress, they do help. Consider using ***debugging macros***

```
#define DEBUG 0
/* some code … */
#if DEBUG
    std::cout << “Debugging is on!” << std::endl;
#endif
/* more code … */
```

- Let’s you setup different debugging cases

```
#define DEBUG 3
#if DEBUG > 0
    std::cout << “Debugging is on! << std::endl;
#endif
#if DEBUG == 1
    std::cout << “Debugging level one << std::endl;
#elif DEBUG == 2
    std::cout << “Debugging level two << std::endl;
#endif
```

Downsides:

- Macros are seen as archaic
- Still adding code

Pluses:

- Can turn tons of debugging statements on and off easily
- Can have cases, run code conditionally, and more….

But why use print statements?

- Because we want to know if we get to a certain point in our program
  - Just set a *break* point there
- Because we want to know the value of a variable
  - Just set a *watch* on the variable
- Because we want to evaluate something we otherwise wouldn’t
  - Can watch that too!

Using GDB

- GDB is like a shell. You can tell it to run your program, but then you can control how your program is run, and inspect what’s happening.
- It’s on the command line: but it has a command line GUI! (x-curses)
- Using it is actually fairly easy
  - (as easy as remembering command line commands ever is)
  - But gets easier when it’s integrated into your IDE
    - CLion
    - KDevelop
    - NetBeans
    - CodeLite
    - Eclipse CDT
    - Visual Studio

Using GDB

- First we need to compile our code with debugging symbols.
  - gcc -g ... // e.g. gcc -g main.c
    - technically -g generates symbols for your system default debugger
    - Probably gdb, but if it's not, -ggdb forces the symbols to be for gdb
- Next we can do either:

```
> gdb
```

or

```
> gdb <program_name> // e.g. gdb a.out
```

- Either way gdb starts and you get a prompt: (gdb)
  - If you just did

    ```
    > gdb
    ```

  - Then you will do:

    ```
    (gdb) file <program_name>
    ```

    To load the file

Using GDB

(gdb) Ctrl + x, a   // that’s two separate commands.

- We get a “gui” showing us our source code! Yay!
  - Or we didn’t because program isn’t running yet

(gdb) break main // break point on main

(gdb) run [arguments] // if your program takes arguments

Using GDB

```
(gdb) break #                      // break on line number in the current source file
(gdb) break file.cpp:#             // break on line number in file.cpp
(gdb) break <function_name>        // break everytime <function_name> is called
(gdb) print <expression>           // the name of a variable or any expression
(gdb) info locals                  // print values of local variables
(gdb) info args                  // print arguments to current function
```

NOTE: function/frame/context are all kind of the same thing, it’s about what’s on your ***program stack***

NOTE: the highlighted line is **about** to run

- If you have a big conditional:

  ```
  if ( condition_a || condition_b || condition_b )
  ```

  - It can be tricky to know why it evaluated true or false, especially when it’s complicated

```
(gdb) print condition_a
```

- Ex: `(gdb) print (x > y)`

Using GDB

```
(gdb) continue            // continue running
(gdb) finish              // run until the current function completes
(gdb) until #           // run until line number
(gdb) return <value>      // return the current function with <value>
(gdb) jump #              // jump to line number
(gdb) where            // show a stack trace, also (gdb) backtrace
(gdb) step             // run the current line, shorthand: s, (step into)
(gdb) next             // run the current line including a function (step over)
```

Using GDB

(gdb) watch <expression>

Can "watch" a variable or any arbitrary expression and automatically break when it changes

- Have to worry about scope. Can't really watch variables that don't exist. Need to have it available in the current *stack frame*.
- Very powerful and useful but also kind of limited. Once a function returns it's frame is popped off the stack and you don't have the watch anymore :(
- Combining watch points and break points helps overcome this.
- Or use...

(gdb) display <expression>

Can have an expression evaluated every time program breaks

- Doesn't automatically break when expression changes. Does display everytime program breaks
- Can use on out of scope variables, e.g.

```
(gdb) display bubbleSort::i  //works
```

Using GDB

- And so much more!
- It’s super powerful, you can even write scripts for it, setup conditional stuff, all kinds of stuff.
  - In a working environment there might be some really complicated development stuff setup that gdb can leverage
- One of those things that you could specialize in
- But there’s so many things to do
  - Writing code
  - Version Control
  - Debugging
  - Building
  - Linting (fixing up spaces and formatting etc.)
  - And so much more!
- Can’t be an expert in all of these
- Don’t want to have to learn a tool when it’s time to use the tool
  - Learn it first: debug working code!
- Command line is awesome and powerful but that’s a lot of stuff to memorize
  - Especially if you’re only going to use one or two commands.
    - For example, git is super powerful, especially on the command line, but most people just:

      git clone
      git add
      git commit
      git push
      git pull
- **That’s what an IDE is for: Integrated Development Environment**
  - **Turns arcane command line magic into buttons!**

CLion

- Free for students: jetbrains.com
- Can do other languages too (most IDE’s have plugins for other languages)
- They also make IDE’s for other languages:
  - IntelliJ – Java
  - PyCharm – Python
  - RubyMine – Ruby
  - Android Studio – Kotlin, ADB, etc. (official Google collaboration)
  - They all look very similar, so the buttons I show you in CLion will be similar there
- I swear I’m not a corporate shill!
- What about Eclipse, Visual Studio, etc.
  - BIG DIFFERENCE
  - Jetbrains products WORK when you install them and tell you what’s wrong and how to fix it when they don’t
  - FOSS is GOOD, Eclipse is BAD
    - Just my opinion
      - Also I’m right

CLion

- Code editor with fancy tools (completion, refactoring, jump to definition/declaration, etc.)
- Build system (Cmake)
- Debugging – uses GDB! (or LLDB, your choice)
- Version Control (git, vcs, svn)
  - Github integration : saves your API token (or username/pass)
- Linting
  - Set code formatting styles and it will reformat your code according to that style
- Constantly testing your code as you write it (most IDE’s do this)
  - **FINDS ERRORS AND TELLS YOU ABOUT THEM AND POTENTIAL FIXES AS YOU TYPE THEM**
  - Stop debugging ERRORS, start debugging PROBLEMS
- Workflow for students:

```
while (project != finished) {
    while (feature != implemented {
        Write code;
        test code;
        if (code works) {
            integrate feature into project;
            implemented = true;
        }
    }
    Push code to github;
    Log onto delmar (or hoare);
    Pull code from github;
    Test code on delmar (or hoare);        // IMPORTANT: if working with system calls you’ll want to spend more time on delmar/hoare.  Also get your delmar/hoare makefile working at beginning
    if (project works on delmar (or hoare) finished = true;
}
```

Debugging with CLion

- Just like debugging with GDB
- But there’s buttons!
- A few important differences:
  - display is called “watches” (watch is called watch point)
  - “step” is called “step into”   // typical of IDE’s
  - “next” is called “step over” // typical of IDE’s
  - “finish” is called “step out” // typical of IDE’s
  - Run to cursor …. like “until”
  - More helpful bits that apply to working in a GUI…
  - Not all GDB functionality is there as a button, but the GDB console is available
- Variable values show up in your code area too
- Sometimes you can’t expand out complex pointer structures and might need to use the GDB Console to explicitly dereference what you’re looking at.
  - *Sometimes*

GDB with multiple processes

- (gdb) set detach-on-fork [on/off]
- (gdb) set follow-on-fork-mode [parent/child]
- (gdb) info inferiors
- (gdb) inferior #
- (gdb) attach PID   //get from ps -e | grep <program_name>
- (gdb) detach

Valgrind

- It exists
- It’s used to debug memory
- Check for leaks etc.
- Sudo apt-get install valgrind