CSU NUPAC Tutorials
2019

Gabriel Palacino

As usual I’m borrowing a lot of the material from Sam Meehan (University of Washington)
Introduction

- Debugging is something you have already seen a bit of
  - You have therefore dealt with it in some regard
- There are a few helpful guidelines to follow/know when debugging
  - Compile-time / run-time difference
  - Compile-time debugging
  - Run-time debugging: the hard way
  - Run-time debugging: the fancy way
Compile Time

- The compiler is your friend
  - Even if at times it may be frustrating, following its directions ensures we write good code

- Errors: You must fix these
  - No compilation → no object file → no linking → no program to run
Compile Time

- The compiler is your friend
  - Even if at times it may be frustrating, following its directions ensures we write good code
- Warnings: You should fix these (programmer dependent)
  - Compilation happens but there may be unexpected behavior
  - Will your code be used by others? Will it be “relied” upon? If yes, fix these!
- Different compilation options (e.g. `-Wall`) to suppress/show - Link Here
Compile Time

- The compiler is your friend
  - Even if at times it may be frustrating, following its directions ensures we write good code

- Warnings: You should fix these (programmer dependent)
  - Compilation happens but there may be unexpected behavior
  - Will your code be used by others? Will it be “relied” upon? ➔ If yes, fix these!

- Different compilation options (e.g. `-Wall`) to suppress/show - [Link Here](#)

```
#include <iostream>

int main()
{
    std::cout << "Hello" << std::endl;
    int x = 4;
    return 0;
}
```

Need additional compilation flag
Compile Time

- The compiler is your friend
  - Even if at times it may be frustrating, following its directions ensures we write good code
- Warnings: You should fix these (programmer dependent)
  - In ATLAS, we even have flags that promote warnings to errors in some cases (i.e. -Wall -Werror)
  - Code written in this way will be incredibly solid and robust

Need additional compilation flags
Compile Time

- How to approach more than one error?
- Rogers & Hammerstein: “Let’s start at the beginning, a very good place to start.”

Area = 0.5*height*(base+top)
Compile Time

- How to approach more than one error?
- Rogers & Hammerstein: “Let’s start at the beginning, a very good place to start.”

Don’t start here!

11 bugs exist in this code, can you find them?
Compile Time [1]

- How to approach more than one error?
  - Rogers & Hammerstein: “Let’s start at the beginning, a very good place to start.”
- Go to the top of your compilation call and fix the first error
  - Fixing the first one may have a cascade effect
How to approach more than one error?

- Rogers & Hammerstein: “Let’s start at the beginning, a very good place to start.”
- Go to the top of your compilation call and fix the first error.
  - Fixing the first one may have a cascade effect.
Compile Time [3]

- Continue process until all errors have been resolved ...
- One of the most common errors is forgetting the “;” at the end of a line
  - “I spent 1 hour searching for a missing semi-colon last night …”
  - Don’t do this ➔ use the line number information!

10 errors → 4 errors
Compile Time [4]

- Continue process until all errors have been resolved …
- Can’t use a variable until you declare it (i.e. `int a;`)
  - This is probably the result of a typo or copy-pasting code
Compile Time [5+6]

- Continue process until all errors have been resolved ...
- As you progress, it will be more efficient to fix multiple unrelated errors at once
  - Particularly useful when compilation time becomes long
Compile Time

- Success! Are we finished?
  - It compiles without any warnings and without any hidden errors
Run Time [1]

- Even if your program compiles, it may not run (properly) : WHY?
- Disallowed behavior : “throw an exception” : [More info here](#)
  - Internal checks for behavior that is considered an exceptional circumstance and therefore not catchable at compile time but would otherwise cause errors or undefined behavior at runtime
  - It is possible to define “exception points” in your own code

THIS IS STILL BUGGY!
Run Time [1]

- Even if your program compiles, it may not run (properly) : WHY?
- Disallowed behavior : “throw an exception” : More info here
  - Internal checks for behavior that is considered an exceptional circumstance and therefore not catchable at compile time but would otherwise cause errors or undefined behavior at runtime
  - It is possible to define “exception points” in your own code
#define CERR as a Debugger

- The most useful rule of thumb: **"Print stuff to the terminal via std::cout"**
  - The only way you know what your program is doing is via the terminal
- Need to find the issue before you can fix it
  - Use the `#define CERR` preprocessor from last time

Add “debugger” line from last time
#define CERR as a Debugger

- The most useful rule of thumb: *“Print stuff to the terminal via std::cout”*
  - The only way you know what your program is doing is via the terminal
- Need to find the issue before you can fix it
  - Use the #define CERR preprocessor from last time ... LIBERALLY
Run Time

- Memory issue: segmentation fault “seg fault”
  - You are trying to access memory that is outside of your authority
- Inspect the contents of the program just before the crash via std::cout
- Our program no longer crashes at the same std::stof() call
  - It is an issue with the arguments
Run Time [2]

- The program executes but doesn’t do anything yet ...
Run Time [3]

- Fixed that → now provide it the proper number of arguments
- Still seems to be an issue with std::stof() : “no conversion”
  - It’s as if c++ doesn’t know how to make one of the arguments into a float
Run Time Success ??

- Finally it runs and gives us a number: “Area : 2”
  - Is this correct? (base = 2, top = 1, height = 1) → NOPE
- Validation by hand for a simple example is critical: keep it simple stupid
  - In ATLAS, people spend large amounts of time validating that our code with simple examples
  - In industry: “unit tests”

- Debug by printing out to the screen
  - Start with most basic portion of calculation
Run Time Success ??

- Finally is runs and gives the correct answer: “Area : 1.5”

- Finally is runs and gives the correct answer : “Area : 1.5”
- Let’s validate further : trapezoids can be upside down
  - It gives the correct answer ... but just by chance
Run Time Success

- Finally we have a debugged and successfully executing program
- Can you find a fringe case where the user may break this?
  - Learning/working to identify fringe cases is key to being successful at higher levels
  - Physicists live “on the limits”
“Fancy Debugging”

- Everything before this has been “manual” debugging
  - Simple: you know how to write std::cout statements
  - Effective: I got a PhD debugging like this
  - Tedious: Requires you to recompile every F*#(!@& time

- Auxiliary programs exist to assist with debugging: “debuggers”
  - Example: GNU-debugger = “gdb” (tutorial1, tutorial2, or just Google it)

- What is allows you to do
  - Step through program line by line
  - Inspect the contents of variables and where they exist in memory
  - No recompiling with a bunch of std::cout statements
Exercise on “Simple” Program

- Goal of program to compute

\[ \sum_{i=0}^{n} \frac{x^n}{n!} \]

- Example: \( n=2, x=3 \)
  - Term1 = 1 (remember 0!=1)
  - Term2 = 3
  - Term3 = 4.5
  - Sum = 8.5
Exercise on “Simple” Program

- Goal of program to compute

\[ \sum_{i=0}^{n} \frac{x^i}{i} \]

- Example: n=2, x=3
  - Term1 = 1 (remember 0!=1)
  - Term2 = 3
  - Term3 = 4.5
  - Sum = 8.5

\[ 8.5 \neq \text{Infinity} \]
How can we find the issue?

- **OLD**: Put a bunch of print statements throughout the main() and functions
- **NEW**: Run program via gdb and inspect with “debug symbols”
  - Compile with debug symbols included: “-g” option
  - Run with gdb: “./broken” → “gdb broken”
Working example

- Set a “breakpoint”
  - `$ b <line_number>`
- The code will “pause” while exec.
  - Further execution done deliberately
Working example

- Execute the code within gdb
  - `$ run <executable> <arguments>`
- Execution proceeds like normal hitting a breakpoint
Working example

- Proceed through code line-by-line
  - $ step
- Executes current line
  - Perform operation
  - Assign variables = change memory
Working example

- Move ahead in code
  - $ \text{next}$

- No operations performed
  - Useful because no memory changes
  - Navigate code in new way
Working example

- Shortcuts exist as well
  - $ <\text{return}>$ : last command
    - You don’t have to type “<return>” but just press ENTER
  - $ s $ : same as “step”
  - $ n $ : same as “next”
Working example

- Print a “backtrace”
  - `$bt`: dump that #($%

- What has my code done thus far?
  - Often times you see this when a seg fault and memory dump occurs
  - This time it is controlled
Working example

- Inspect a specific variable
  - `$ print <var_name>`
- Allows you to see the value without a std::cout statement
  - Example: "fact" = 0 !!!!!!
Working example

- Make this one minor change
- Re-executing program and validating the result

\[ \sum_{i=0}^{2} \frac{3^i}{i} = 8.5 \]
Conclusions

- Debugging is hard
- Debugging will take a lot of time
- You will become good at debugging
- Debuggers are powerful
- You will become powerful
  - ... like Chuck Norris