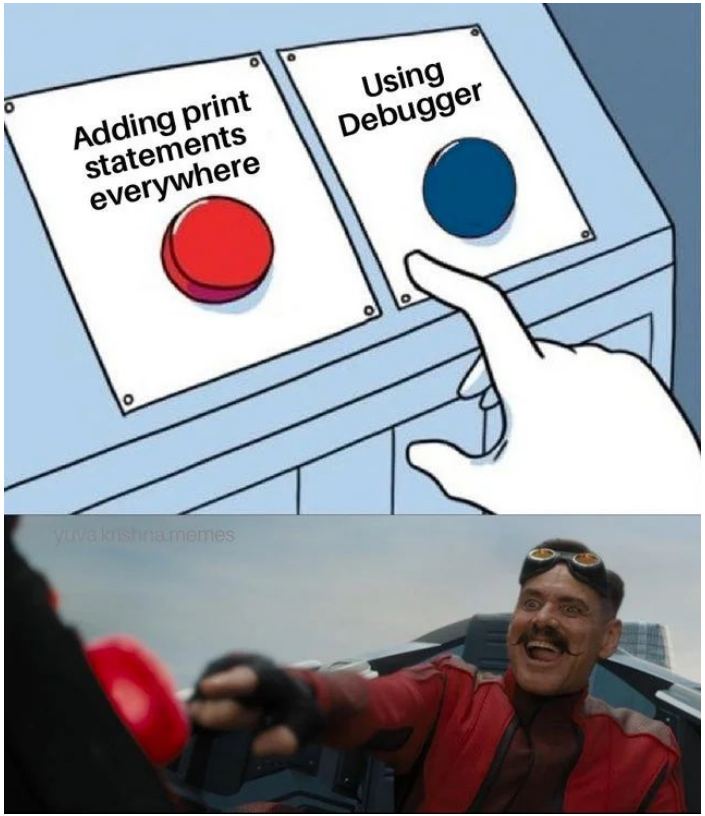
Debugging

Kameswari Chebrolu



Background

- What is the most used language in programming?
  - “Profanity”
- “To err is human, but to really foul things up you need a computer” – Paul R. Ehrlich

Debugging

- "Debugging is like being the detective in a crime movie where you are also the murderer." - Filipe Fortes
- Process of finding compile and run time errors
  - Compile time errors
    - missing semicolons
    - variable type mismatches
    - incorrect passing of arguments
    - undefined or out of scope variables etc

- Run time errors are the hardest, often crash the program
  - dereferencing memory that has not been allocated
  - freeing memory that was already freed
  - opening a file that does not exists
  - reading incorrect or non-existent data (array index out of bound)
  - Processing command line arguments that have not been provided
  - Illegal use of pointers

- Compiling can result in large numbers of error messages
- How to view them?
  - g++ myprogram.cpp 2>&1 | more
  - g++ myprogram.cpp 2>&1 | tee compilation.log

Process

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it” – Brian Kernighan

- Debugging is hard work, and an art
  - Don’t do random trial and error
  - Sit back, think, form a plan and then execute
- Form a hypotheses and see if on right track
  - Seek additional information about the execution that is not evident from the original program or output
  - how often the loop is executed? When did we exit the loop? What is the value of variable i at this point? etc
  - Auto-debuggers can help in all this!
- Focus: gdb for C++/C

Caution:

- Automatic debuggers can also be a tremendous waste of time if not careful
  - Can single-step through the code aimlessly
- Debuggers should always be used to augment a reasoning process, not as a substitute for it!!!

Alternatives

- Add output statements to the code
  - Good to send debugging information to standard error
    - Standard output may be redirected to files or other programs (through pipes etc)
    - Comment out when not needed

```
for (int i = 0; i < numElements; ++i)
{
  // cerr << "i: " << i << endl;
   cout << myArray[i] << endl;
}
// cerr << "Exited loop" << endl;
myArray[0] = myArray[1];
```

Too many comments?

- Set a compile-time symbol DEBUG
  - “#define DEBUG 1” in a .h file (or)
  - g++ -g -c -DDEBUG myProgram.cpp

```
for (int i = 0; i < numElements; ++i)
{
#ifdef DEBUG
   cerr << "i: " << i << endl;
#endif
   cout << myArray[i] << endl;
}
#ifdef DEBUG
cerr << "Exited loop" << endl;
#endif
```

Macros

```
#ifdef DEBUG
#define dbgprint(x) cerr << #x << ": " << x << endl
#define dbgmsg(message) cerr << message << endl
#else
#define dbgprint(x)
#define dbgmsg(message)
#endif
```

```
#include "debug.h"
  ⋮
for (int i = 0; i < numElements; ++i)
{
  dbgprint(i);
  cout << myArray[i] << endl;
}
dbgmsg("Exited loop");
```

```
g++ -g -c -DDEBUG myProgram.cpp

#include "debug.h"
  ⋮
for (int i = 0; i < numElements; ++i)
{
  cerr << "i" << ": " << i << endl;
   cout << myArray[i] << endl;
}
cerr << "Exited loop" << endl;
```

```
g++ -g -c myProgram.cpp

#include "debug.h"
  ⋮
for (int i = 0; i < numElements; ++i)
{
  ;
   cout << myArray[i] << endl;
}
;
```

Assertions

- A way of defensive programming
- A boolean test that should be true if things are working as expected
  - Assert macro is defined in <assert.h> for C and <cassert> for C++
  - Program stops with an informational message whenever the asserted condition fails

```
unsigned long gcd( unsigned long m, unsigned long n ) {
   assert( m >= n );
   while ( n != 0 ) {
      unsigned long r{m % n};
      m = n;
      n = r;
   }
   return m;
}
```

- Controlled by compile-time symbol NDEBUG (“not debugging”)
  - If NDEBUG is defined, then each assertion is compiled as a comment
  - If NDEBUG is not defined, then assertion works

gdb

How to use gdb?

- Compile your program with the -g compiler flag (-g helps in debugging)
- g++ -g -o broken broken.cpp
- gdb broken
  - starts the debugger for this program
  - But does not run it yet!

Commands

- Typing “help” at the prompt prints out a list of possible topics you can ask help for:
  - `(gdb) help`
- “run” will run the program
  - `(gdb) run`

- To stop the program at different points in its execution, can use breakpoints
  - To specify a breakpoint upon entry to a function
    - (gdb) break function
    - (gdb) break bug.c:function
    - (gdb) b function (shortcut b for break)
  - Set a breakpoint by specifying a file and line number
    - (gdb) break 26 (line no)
    - (gdb) break bug.c:26

- To list where the current breakpoints are set
  - (gdb) info breakpoints
- To delete the breakpoint numbered 2 (as specified via info)
  - (gdb) delete 2
- Can also set breakpoints to only trigger when certain conditions are true.
  - (gdb) break 23 if i == count - 1
    (e.g. for (int i = 0; i < count; i++))

- If we wish to continue from a breakpoint, we can use “continue”
  - (gdb) continue
  - different from “run” which starts the program from the beginning
  - Continue will take to next breakpoint

- Can check the values of certain key variables at breakpoint
  - (gdb) print n
  - (gdb) p n (shortcut print is p)
- Can also use print to evaluate expressions, make function calls, reassign variables, and more
  - (gdb) print buffer[0] = 'Z'
  - (gdb) print strlen(buffer)
- Can print out a group of variables in a particular function
  - info args prints out the arguments (parameters) to the current function you're in:
    - (gdb) info args
  - info locals prints out the local variables of the current function:
    - (gdb) info locals

- At a breakpoint, we can ask the debugger to print out the contents of the file via list
  - (gdb) list
  - We can also list the contents of files by name and line number
    - (gdb) list bug.C:1
- There are two commands to step through: “step” and “next”
  - Next will step over function calls
    - (gdb) next
  - Step will step “into” function calls
    - (gdb) step

- To see the value of a specific variable whenever we reach a breakpoint or are stepping through our program, we can use the “display” command
  - (gdb) display result
  - (gdb) n
  - (gdb) n

- Can examine the contents of the stack via backtrace which prints the current functions on the stack
  - (gdb) backtrace
  - (gdb) bt (bt shortcut for backtrace)
  - Also prints the arguments to the functions on the call stack

- You can use the up and down commands for moving up and down the stack frames
  - up moves you up one stack frame (e.g. from a function to its caller)
    - (gdb) up
  - down moves you down one stack frame (e.g. from the function to its callee)
    - (gdb) down
  - Helpful if you're stuck at a segfault and want to know the arguments and local vars of the faulting function's caller or callee

Profiling

- Debuggers enable us to search for/localize bugs and observe program behaviour
- Profilers allow us to examine the program's overall use of system resources
  - focused primarily on memory use and cpu consumption
  - Can determine parts of code that are time consuming and need to be re-written for faster execution

Workflow

- Have profiling enabled while compiling the code
  - g++ -pg -o profile profile.cpp
- Execute the program code to produce the profiling data
  - ./profile
  - Generates gmon.out (a binary file)
    - If gmon.out already exists, it will be overwritten.
    - Note program must exit normally (don;t use control-c)
- Run the gprof tool on the profiling data file
  - gprof profile gmon.out > analysis.txt

Analyzing the output

- Flat profile: shows the total amount of time your program spent executing each function
- Call graph: shows how much time was spent in each function and its children
  - Can find functions that, they themselves may not have used much time, but called other functions that did!

References

http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf (gdb reference card)

https://www.thegeekstuff.com/2012/08/gprof-tutorial/ (gprof)

http://www.cs.toronto.edu/~krueger/csc209h/tut/gdb_tutorial.html (debugging example)