Introduction to GDB and Debuggers

Marvin Smith

September 7, 2011

What happens when our code has bugs?

What are our solutions?

- Cascades of cout statements?
- Re-write your code?
- Bug your T.A.? (wink, wink)

Debuggers

Use a debugger! Debuggers allow you to step through your code to determine what is actually happening when you run your program.

What is the output of variable c?

```
#define square(x) x*x
#include <iostream>

using namespace std;

int main()
{
    int a = square(3+5);
    int b = a*2;
    int c = b/4;
    cout << c << endl;

    return 0;
}
```

Result

Answer: 11

Correct Solution: 32

Where is the bug?

Approach

This is an unusually difficult problem because everything looks perfectly fine. Rather than use cout statements, lets try GDB.

Using GDB

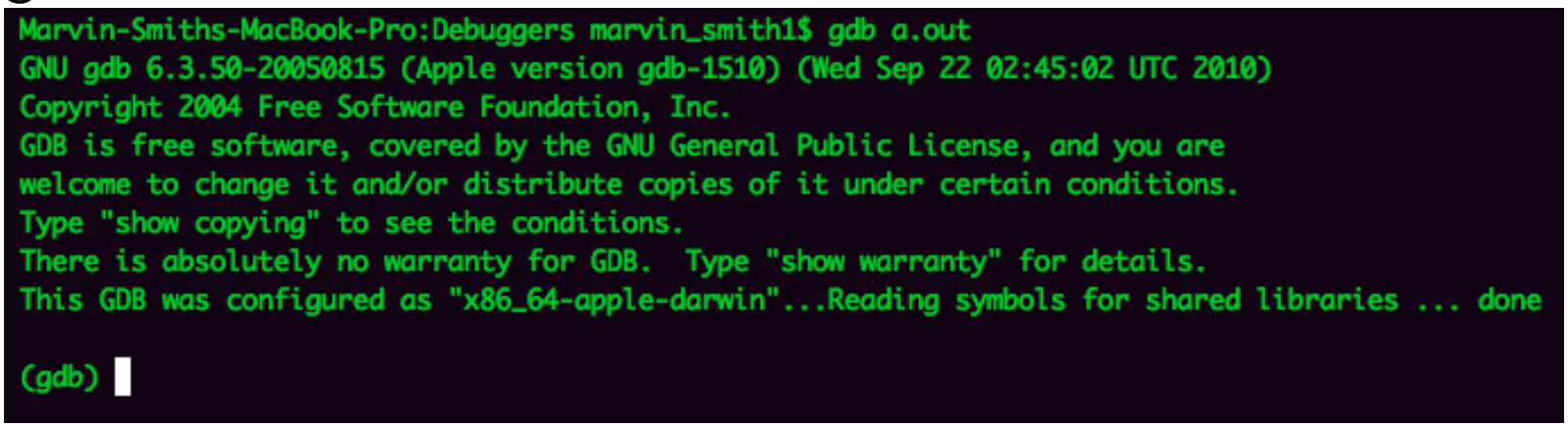
To allow for the use of GDB, compile your code with the following command...

`g++ test.cpp -g`
-g gives the binary files the ability to be accessed by gdb.

Using GDB

To start GDB, run

`gdb a.out`



Starting Your Code

To start running your program, type **run**. Make sure that you ran GDB with a binary file. GDB will run your program until you reach a breakpoint, your program finishes, or it crashes...

```
(gdb) run
Starting program: /Users/marvin_smith1/Documents/Personal_Sandbox/Personal_Files/Slideshows/Debuggers
/a.out
Reading symbols for shared libraries ++. done
3
Remaining elements
6
5
4
a.out(46858) malloc: *** error for object 0x100100080: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug

Program received signal SIGABRT, Aborted.
0x00007fff8230c0b6 in __kill ()
(gdb)
```

Using GDB

There are several commands that are useful.

```
list        - list 10 lines of code.
b x         - establish breakpoint at line x
step        - step into next line
next        - go to next line and execute
continue    - continue program until end/break
print x     - output value of variable x
backtrace   - print the function trace to the point
```

(gdb) list

```
list          - list the next 10 lines of code
list x        - list 10 lines surrounding line x
list x,y      - list lines of code from x to y
```

```
(gdb) list
71
72      /*
73         Main Driver
74         - This demo is going to push a couple values
75           on, pop a value off, then check what is remaining
76           on the stack when the destructor is called
77      */
78      int main(){
79
80         Stack stack;
(gdb)
```

(gdb) break

Breakpoints are stops in code where you can freeze program operation. Breakpoints are useful in order to stop and evaluate your program.

```
break             - create breakpoint at current line
break x           - create breakpoint at line x
break file:x      - create breakpoint at line x in file
break function    - create breakpoint at function
clear x           - remove breakpoint at line x
```

NOTE: Clear works the same as breakpoint

```
(gdb) break 82
Breakpoint 1 at 0x10000095d: file test1.cpp, line 82.
(gdb)
```

```
(gdb) break test1.cpp:86
Breakpoint 1 at 0x100000931: file test1.cpp, line 86.
(gdb)
```

(gdb) print

When running your program, it is useful to print the value of variables stored in your program. Note that GDB can access private member variables.

`print variable  - print the value of a variable`

```
(gdb) list 88
83
84          Stack stack;
85
86          stack.Push(4);
87          stack.Push(3);
88          cout << stack.Pop() << endl;
89          stack.Push(5);
90          stack.Push(6);
91
92        return 0;
(gdb) break 86
Breakpoint 1 at 0x100000931: file test1.cpp, line 86.
(gdb) break 87
Breakpoint 2 at 0x100000945: file test1.cpp, line 87.
(gdb) run
Starting program: /Users/marvin_smith1/Documents/Personal_Sandbox/Personal_Files/Slideshows/Debuggers
/a.out
Reading symbols for shared libraries ++. done

Breakpoint 1, main () at test1.cpp:86
86          stack.Push(4);
(gdb) print stack.len
$1 = 0
(gdb) continue
Continuing.

Breakpoint 2, main () at test1.cpp:87
87          stack.Push(3);
(gdb) print stack.len
$2 = 1
(gdb)
```

(gdb) step (gdb) next (gdb) continue

At some point, we need to step through the program and analyze it line by line.

```
next      - process the next line of code in function
step      - if code is function, go into it
continue  - run program until next breakpoint or finish
```

```
Breakpoint 1, main () at test1.cpp:86
86          stack.Push(4);
(gdb) next
87          stack.Push(3);
(gdb)
```

Notice that next skips the Push function.

```
Breakpoint 1, main () at test1.cpp:86
86          stack.Push(4);
(gdb) step
Stack::Push (this=0x7fff5fbff260, value=@0x7fff5fbff27c) at test1.cpp:54
54              NODE* tnode = new NODE(value);
(gdb)
```

Notice that step enters into the Push function.

(gdb) backtrace

Segmentation faults are difficult as it can be annoying and difficult to step until a failure. Rather than step through code, just let the program fail and run a backtrace.

`backtrace` - `print the functions leading up to the current line`

```
(gdb) run
Starting program: /Users/marvin_smith1/Documents/Personal_Sandbox/Personal_Files/Slideshows/Debuggers
/a.out
Reading symbols for shared libraries ++. done
3
Remaining elements
6
5
4
a.out(47404) malloc: *** error for object 0x100100080: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug

Program received signal SIGABRT, Aborted.
0x00007fff8230c0b6 in __kill ()
(gdb) backtrace
#0  0x00007fff8230c0b6 in __kill ()
#1  0x00007fff823ac9f6 in abort ()
#2  0x00007fff822c4195 in free ()
#3  0x0000000100000ba0 in Stack::~Stack (this=0x7fff5fbff260) at test1.cpp:50
#4  0x00000001000009b5 in main () at test1.cpp:92
(gdb)
```

Segmentation fault handling

This tells us that the last function the program entered which our code was the destructor for the Stack. We should enter the destructor and find the line which caused the failure.

Start Demo