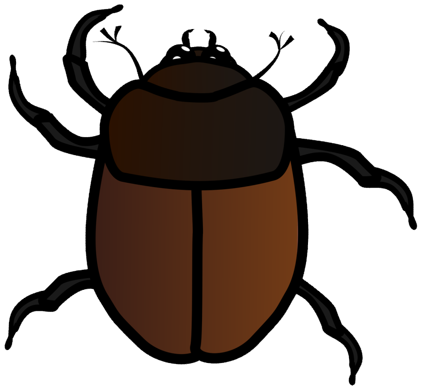
C Debugging

Benjamin Brewster

UNIX C Debugging

- Just a few notes on debugging…
  - http://en.wikipedia.org/wiki/Software_bug#Etymology



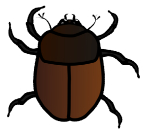
Debugging Process Review

1. Reproduce the problem reliably
    - Simplify input and environment until the problem can be replicated at will
        - e.g. Wolf Fence algorithm
    - Challenges:
        - Unique environment (space station, aunt Edna's PC in Hoboken, NJ, etc.)
        - Particular sequence of events leading up to the error are unknown or difficult to do more than once (lightning strike, aunt Edna tries to watch Netflix through her toaster, etc.)

2. Examine the process state at the time of error; we'll cover 3 types:
    1. Live Examination
    2. Post-mortem Debugging
    3. Trace Statement

Using a debugger with `gcc`

- Compile with the "-g" option.
```
$ gcc -g testit.c -o testit
```
- Then start the debugger on the program
```
$ gdb ./testit
```
- In the debugger, some key commands:
  - `run` :: (re)starts the program running; will stop at breakpoint (can add args, e.g.: `run 6 myfile`)
  - `break` :: sets a breakpoint where the debugger will stop and allow you to examine variables or single step
  - `step` :: executes a single line of C code; will enter a function call
  - `next` :: executes a single line of C code; will not enter a function call
  - `continue` :: continues execution again until another breakpoint is hit or the program completes
  - `print` :: prints out a variable
  - `quit` :: stop debugging (exit gdb)



Demo with testit.c

```
$ gcc -o testit testit.c
$ testit
Segmentation fault (core dumped)
```

Blast!

Demo with testit.c

−g flag compiles with debug symbols

```
$ gcc -g -o testit testit.c
$ gdb testit
[…]
Reading symbols from /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit...done.
(gdb) run
Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit
Program received signal SIGSEGV, Segmentation fault.
0x00000000004004e3 in main () at testit.c:12
12              temp[2]='F';
(gdb) break 12
Breakpoint 1 at 0x4004db: file testit.c, line 12.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit

Breakpoint 1, main () at testit.c:10
12              temp[2]='F';
```

Set a breakpoint at line 12

Execution pauses just before running line 12

Demo with testit.c

```
(gdb) print temp
$1 = 0x400628 "CS344"
(gdb) list 12
7               char* temp = "CS344";
8
9               int i;
10               i=0;
11
12              temp[2]='F';
13
14              for (i = 0; i < 5 ; i++ )
15                      printf("%c\n", temp[i]);
16
```

Show the contents and address of the `temp` variable

Display the five lines before and after line 12

Demo with testit.c

```
(gdb) print temp
$1 = 0x400628 "CS344"
(gdb) list 12
7               char* temp = "CS344";
8
9               int i;
10               i=0;
11
12              temp[2]='F';
13
14              for (i = 0; i < 5 ; i++ )
15                      printf("%c\n", temp[i]);
16
```

Oops – can’t modify a string literal!

Demo with testit.c

```
7           char* temp = "CS344";
8
9           int i;
10          i=0;
11
12          temp[2]='F';
13
14          for (i = 0; i < 5 ; i++ )
15                  printf("%c\n", temp[i]);
16
```

```
(gdb) jump 13
Continuing at 0x4004e6.
C
S
3
4
4
Adding 6 to 3: 10
Program exited with code 022.
```

Let's see if the rest of this works:

Jump to line 13, skipping line 12, and continue

???

Demo with testit.c

Set a breakpoint on line 13, because jump starts ongoing execution again

```
(gdb) break 13
Breakpoint 2 at 0x4004e6: file testit.c, line 13.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004db in main at testit.c:12
        breakpoint already hit 1 time
2       breakpoint     keep y   0x00000000004004e6 in main at testit.c:13
(gdb) run
Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit

Breakpoint 1, main () at testit.c:12
12              temp[2]='F';
(gdb) jump 13
Continuing at 0x4004e6.

Breakpoint 2, main () at testit.c:14
14              for (i = 0; i < 5 ; i++ )
```

Skip line 13, stopping at the next breakpoint

Demo with testit.c

Run the next line of code (14), then display the *next* one (15)

```
(gdb) step
15                              printf("%c\n", temp[i]);
(gdb) print i
$2 = 0
(gdb) step
C
14                      for (i = 0; i < 5 ; i++ )
(gdb) print i
$3 = 0
(gdb) step
15                              printf("%c\n", temp[i]);
(gdb) print i
$4 = 1
(gdb) where
#0  main () at testit.c:15
```

`i` has finally updated

Demo with testit.c

```
(gdb) break 17
Breakpoint 3 at 0x40051c: file testit.c, line 17.
(gdb) continue
Continuing.
S
3
4
4
Breakpoint 3, main () at testit.c:17
17              printf("Adding 6 to 3: %d\n", Add6(3));
(gdb) next
Adding 6 to 3: 10
18      }
(gdb) where
#0  main () at testit.c:18
```

This is getting boring iterating through this for loop, so just set a breakpoint in the future (at line 17; we're currently at 15) to skip ahead to find this math bug

Floor it

A function! Let's go in!

Oops: **next** goes to the next line but *won't* enter functions! I should have used **step**

Demo with testit.c

Alright, restart the whole thing, since we missed our function

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit


Breakpoint 1, main () at testit.c:12
12                temp[2]='F';
(gdb) jump 13
Continuing at 0x4004e6.

Breakpoint 2, main () at testit.c:14
14               for (i = 0; i < 5 ; i++ )
```

Skip past the known seg fault

Demo with testit.c

```
(gdb) continue
Continuing.
C
S
3
4
4

Breakpoint 3, main () at testit.c:17
17              printf("Adding 6 to 3: %d\n", Add6(3));
```

Continue on to the function call

Demo with testit.c

```
(gdb) step
Add6 (in=3) at testit.c:22
22              int six = 7;
(gdb) watch six
Hardware watchpoint 8: six
(gdb) next
Hardware watchpoint 8: six
Old value = 52
New value = 7
Add6 (in=3) at testit.c:24
24              return six + in;
(gdb) continue
Continuing.

Watchpoint 8 deleted because the program has left the block in
which its expression is valid.
0x0000000000400526 in main () at testit.c:17
17              printf("Adding 6 to 3: %d\n", Add6(3));
(gdb) continue
Continuing.
Adding 6 to 3: 10

Program exited with code 022.
(gdb) quit
```

Step into the function!

gdb tells us it just entered a function

Oops! six = 6!

`watch` causes `gdb` to pause execution if the variable `six` changes. This could also have been an expression about it's value:

```
(gdb) watch six if six > 6
```

`gdb` pauses and tells us it deleted a watchpoint

Visual Studio Destroys `gdb`

- Any Integrated Development Environment destroys `gcc` and `gdb`
  - IDEs have code generation, compiling, optimization, organization, debugging, live code step-through, and documenting all built in
- Visual Studio 20XX rocks
- But we don’t have access to that in UNIX, so how do we find nasty bugs like memory leaks?

valgrind

- valgrind helps us to find memory leaks in C programs
- Compile with –g to add better diagnostics

```
$ cat leaky.c
// leaky.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{
        char* dynamicBuffer;
        dynamicBuffer = malloc(10);
}
$ gcc -o leaky -g leaky.c
```

`malloc()` but no `free()`

Compile with debug symbols (line numbers, function & variable names, etc.

valgrind – leaky example

```
$ valgrind --leak-check=yes --show-reachable=yes ./leaky
…
==31186== HEAP SUMMARY:
==31186==     in use at exit: 10 bytes in 1 blocks
==31186==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==31186==
==31186== 10 bytes in 1 blocks are still reachable in loss record 1 of 1
==31186==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==31186==    by 0x4004D5: main (leaky.c:8)
==31186==
==31186== LEAK SUMMARY:
==31186==    definitely lost: 0 bytes in 0 blocks
==31186==    indirectly lost: 0 bytes in 0 blocks
==31186==      possibly lost: 0 bytes in 0 blocks
==31186==    still reachable: 10 bytes in 1 blocks
==31186==         suppressed: 0 bytes in 0 blocks
==31186==
==31186== For counts of detected and suppressed errors, rerun with: -v
==31186== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Still reachable refers to `dynamicBuffer`, `malloc`'d on line 8, whose dynamically allocated memory pointer was not overwritten but simply didn't get freed before the program terminated.

Because the OS frees all memory when your process terminates, these can often be safely ignored without consequence, as long as you aren't allocating and forgetting about the memory in a loop...

But it's safest to de-allocate memory as needed to facilitate safe code revisions *later*!

valgrind – leaky2

- A note about valgrind and `printf()`
- This program is the same as leaky.c except for the `printf` statement

```
// leaky2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{
        char* dynamicBuffer;
        dynamicBuffer = malloc(10);
        printf("This printf causes valgrind to think the malloc pointer is lost\n");
}
```

`malloc()` but no `free()`

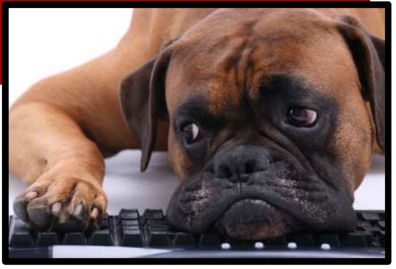
`printf()` uses all kinds of internal variables that confuse valgrind into thinking things are worse than they are

valgrind – leaky2

```
$ valgrind --leak-check=yes --show-reachable=yes ./leaky2
…
==8303== Command: ./leaky2
…
This printf causes valgrind to think the malloc pointer is lost
==8303==
==8303== HEAP SUMMARY:
==8303==     in use at exit: 10 bytes in 1 blocks
==8303==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==8303==
==8303== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8303==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==8303==    by 0x400515: main (leaky2.c:8)
==8303==
==8303== LEAK SUMMARY:
==8303==    definitely lost: 10 bytes in 1 blocks
==8303==    indirectly lost: 0 bytes in 0 blocks
==8303==      possibly lost: 0 bytes in 0 blocks
==8303==    still reachable: 0 bytes in 0 blocks
==8303==         suppressed: 0 bytes in 0 blocks
==8303==
==8303== For counts of detected and suppressed errors, rerun with: -v
==8303== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Despite this normally bad warning, you *can* still fix your code and `free()` it, but there's definitely still a memory leak if you don't.

Bad printf!



valgrind – leaky3

- valgrind can also help you discover when you use variables that are uninitialized:

```
// leaky3.c
#include <stdio.h>
void main()
{
        int six;
        printf("six: %d\n", six);
}
```

`six` isn't initialized!

valgrind – leaky3

```
$ valgrind --leak-check=yes --show-reachable=yes ./leaky3
…
==10122== Use of uninitialised value of size 8
==10122==    at 0x334E843A5B: _itoa_word (in /lib64/libc-2.12.so)
==10122==    by 0x334E846612: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
==10122==
==10122== Conditional jump or move depends on uninitialised value(s)
==10122==    at 0x334E843A65: _itoa_word (in /lib64/libc-2.12.so)
==10122==    by 0x334E846612: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
==10122==
==10122== Conditional jump or move depends on uninitialised value(s)
==10122==    at 0x334E8450A3: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
==10122==
==10122== Conditional jump or move depends on uninitialised value(s)
==10122==    at 0x334E8450C1: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
…
```

Knowing that the error happened on line 6 is priceless

That’s a lot of whining