Programming in C and C++

Lecture 9: Debugging

Neel Krishnaswami and Alan Mycroft

What is Debugging?

> *Debugging is a methodical process of finding and reducing the number of bugs (or defects) in a computer program, thus making it behave as originally expected.*

There are two main types of errors that need debugging:

- Compile-time: These occur due to misuse of language constructs, such as syntax errors. Normally fairly easy to find by using compiler tools and warnings to fix reported problems, e.g.: `gcc -Wall -pedantic -c main.c`
- Run-time: These are much harder to figure out, as they cause the program to generate incorrect output (or crash) during execution. This lecture will examine how to methodically debug a run-time error in your C code.

The Runtime Debugging Process

A typical lifecycle for a C/C++ bug is:

- a program fails a *unit test* included with the source code, or a bug is reported by a user, or observed by the programmer.
- given the failing input conditions, a programmer *debugs* the program until the offending source code is located.
- the program is recompiled with a source code fix and a *regression test* is run to confirm that the behaviour is fixed.

Unit tests are short code fragments written to test code modules in isolation, typically written by the original developer.

Regression testing ensures that changes do not uncover new bugs, for example by causing unit tests to fail in an unrelated component.

What is a Bug?

The program contains a *defect* in the source code, either due to a design misunderstanding or an implementation mistake. This defect is manifested as a *runtime failure*. The program could:

- crash with a memory error or segmentation fault
- return an incorrect result
- have unintended side-effects like corrupting persistent storage

The art of debugging arises because defects *do not materialise predictably*.

- The program may require specific inputs to trigger a bug
- Undefined or implementation-defined behaviour may cause it to only crash on a particular OS or hardware architecture
- The issue may require separate runs and many hours (or months!) of execution time to manifest

Finding Defects

Defects are not necessarily located in the source code near a particular runtime failure.

- A variable might be set incorrectly that causes a timer to fire a few seconds too late. The *defect* is the assignment point, but the *failure* is later in time.
- A configuration file might be parsed incorrectly, causing the program to subsequently choose the wrong control path. The *defect* is the parse logic, but the *failure* is observing the program perform the wrong actions.
- A function `foo()` may corrupt a data structure, and then `bar()` tries to use it and crashes. The *defect* is in `foo()` but the *failure* is triggered at `bar()`.

The greater the distance between the original defect and the associated failure, the more difficult it is to debug.

Finding Defects (cont.)

Sometimes the defect directly causes the failure and is easy to debug. Consider this NULL pointer error:

```c
#include <netdb.h>
#include <stdio.h>

int main(int argc, char **argv) {
  struct hostent *hp;
  hp = gethostbyname("doesntexist.abc");
  printf("%s\n", hp->h_name);
  return 0;
}
```

Just fix the code to add a NULL pointer check before printing hp.

Debugging via printing values

A very common approach to debugging is via printing values as the program executes.

```
1  #include <netdb.h>
2  #include <stdio.h>
3
4  int main(int argc, char **argv) {
5    struct hostent *hp;
6    printf("argc: %d\n", argc);
7    for (int i=1; i<argc; i++) {
8      hp = gethostbyname(argv[i]);
9      printf("hp: %p\n", hp);
10     if (hp)
11      printf("%s\n", hp->h_name);
12   }
13   return 0;
14 }
```

Debugging via printing values (cont.)

Executing this will always show the output as the program runs.

```
./lookup google.org recoil.org
argc: 3
hp: 0x7fd87ae00420
google.org
hp: 0x7fd87ae00490
recoil.org
```

Some tips on debug printing:

- Put in as much debugging information as you can to help gather information in the future
- Make each entry as unique as possible so that you tie the output back to the source code
- Flush the debug output so that it reliably appears in the terminal

Debugging via printing values (cont.)

```c
#include <netdb.h>
#include <stdio.h>

int main(int argc, char **argv) {
  struct hostent *hp;
  printf("%s:%2d argc: %d\n", __FILE__, __LINE__, argc);
  for (int i=1; i<argc; i++) {
    hp = gethostbyname(argv[i]);
    printf("%s:%2d hp: %p\n", __FILE__, __LINE__, hp);
    fflush(stdout);
    printf("%s:%2d %s\n", __FILE__, __LINE__, hp->h_name);
    fflush(stdout);
  }
  return 0;
}
```

Debugging via printing values (cont.)

The source code is now very ugly and littered with debugging statements. The C preprocessor comes to the rescue.

- Define a DEBUG parameter to compile your program with.
- `#define` a debug printf that only runs if DEBUG is non-zero.
- Disabling DEBUG means debugging calls will be optimised away at compile time.

```
1  #ifndef DEBUG
2  #define DEBUG 0
3  #endif
4  #define debug_printf(fmt, ...) \
5    do { if (DEBUG) { \
6      fprintf(stderr, fmt, __VA_ARGS__); \
7      fflush(stderr); } } \
8    while (0)
```

Debugging via printing values (cont.)

```
1  #include <netdb.h>
2  #include <stdio.h>
3  #ifndef DEBUG
4  #define DEBUG 0
5  #endif
6  #define debug_printf(fmt, ...) \
7    do { if (DEBUG) { fprintf(stderr, fmt, __VA_ARGS__); \
8                      fflush(stderr); } } while (0)
9  int main(int argc, char **argv) {
10   debug_printf("argc: %d\n", argc);
11   for (int i=1; i<argc; i++) {
12     struct hostent *hp = gethostbyname(argv[i]);
13     debug_printf("hp: %p\n", hp);
14     printf("%s\n", hp->h_name);
15   }
16   return 0;
17 }
```

Debugging via Assertions

Defects can be found more quickly than printing values by using assertions to encode invariants through the source code.

```
1  #include <netdb.h>
2  #include <stdio.h>
3  #include <assert.h> // new header file
4
5  int main(int argc, char **argv) {
6    struct hostent *hp;
7    hp = gethostbyname("doesntexist.abc");
8    assert(hp != NULL); // new invariant
9    printf("%s\n", hp->h_name);
10   return 0;
11 }
```

Debugging via Assertions (cont.)

The original program without assertions will crash:

```
cc -Wall debug-s6.c && ./lookup
Segmentation fault: 11
```

Running with assertions results in a much more friendly error message.

```
cc -Wall debug-s12.c && ./lookup
Assertion failed: (hp != NULL),
function main, file debug2.c, line 10.
```

Debugging via Assertions (cont.)

Using assert is a cheap way to ensure an invariant remains true.

- A failed assertion will immediately exit a program.
- Assertions can be disabled by defining the NDEBUG preprocessor flag.

Never cause side-effects in assertions as they may not be active!

```
cc -Wall -DNDEBUG debug-s12.c && ./lookup
Segmentation fault: 11
```

Fault Isolation

Debugging is the process of fault isolation to find the cause of the failure.

- Never try to guess the cause randomly. This is time consuming.
- Stop making code changes incrementally to fix the bug.
- Do think like a detective and find clues to the cause.

Remember that you are trying to:

- Reproduce the problem so you can observe the failure.
- Isolate the failure to some specific inputs.
- Fix the issue and confirm that there are no regressions.

Reproducing the Bug

Consider this revised program that performs an Internet name lookup from a command-line argument.

```c
#include <netdb.h>
#include <stdio.h>
#include <assert.h>
int main(int argc, char **argv) {
  struct hostent *hp;
  hp = gethostbyname(argv[1]);
  printf("%s\n", hp->h_name);
  return 0;
}
```

This program can crash in at least two ways. How can we reproduce both?

Reproducing the Bug (cont.)

This program can crash in at least two ways.

`cc -Wall -o lookup debug-s16.c`

First crash: if we do not provide a command-line argument:

```
./lookup
Segmentation fault: 11
```

Second crash: if we provide an invalid network hostname:

```
./lookup doesntexist.abc
Segmentation fault: 11
```

It does work if we provide a valid hostname:

```
./lookup www.recoil.org
bark.recoil.org
```

Both positive and negative results are important to give you more hints about how many distinct bugs there are, and where their source is.

Isolating the Bug

We now know of two failing inputs, but need to figure out where in the source code the defect is. From earlier, one solution is to put assert statements everywhere that we suspect could have a failure.

```
1  #include <netdb.h>
2  #include <stdio.h>
3  #include <assert.h>
4  int main(int argc, char **argv) {
5    struct hostent *hp;
6    assert(argv[1] != NULL);
7    hp = gethostbyname(argv[1]);
8    assert(hp != NULL);
9    printf("%s\n", hp->h_name);
10   return 0;
11 }
```

Reproducing the Bug with Assertions

Recompile the program with the assertions enabled.

```
cc -Wall -o lookup debug-s18.c
```

First crash: if we do not provide a command-line argument:

```
./lookup
Assertion failed: (argv[1] != NULL),
function main, file debug-s18.c, line 7.
```

Second crash: if we provide an invalid network hostname:

```
./lookup doesntexist.abc
Assertion failed: (hp != NULL), function main,
file debug-s18.c, line 9.
```

Reproducing the Bug with Assertions (cont.)

It does work if we provide a valid hostname:

```
./lookup www.recoil.org
bark.recoil.org
```

The assertions show that there are two distinct failure points in application, triggered by two separate inputs.

Using Debugging Tools

While assertions are convenient, they do not scale to larger programs. It would be useful to:

- Observe the value of a C variable *during* a program execution
- Stop the execution of the program if an assertion is violated.
- Get a *trace* of the function calls leading up to the failure.

These features are provided by **debuggers**, which let a programmer to monitor the memory state of a program during its execution.

- *Interpretive* debuggers work by simulating program execution one statement at a time.
- More common for C code are *direct execution* debuggers that use hardware and operating system features to inspect the program memory and set breakpoints to pause execution.

Example: Using lldb from LLVM

Let's use the lldb debugger from LLVM to find the runtime failure without requiring assertions.

```
cc -Wall -o lookup -DNDEBUG -g debug-s18.c
```

Run the binary using lldb instead of executing it directly.

```
lldb ./lookup
(lldb) target create "./lookup"
Current executable set to ./lookup (x86_64).
```

At the (lldb) prompt use run to start execution.

```
(lldb) run www.recoil.org
Process 9515 launched: ./lookup (x86_64)
bark.recoil.org
Process 9515 exited with status = 0 (0x00000000)
```

Example: Using lldb from LLVM (cont.)

Now try running the program with inputs that trigger a crash:

```
(lldb) run doesntexist.abc
frame #0: 0x0000000100000f52 lookup
main(argc=2, argv=0x00007fff5fbff888) + 50 at debug-s18.c:12
    9     assert(argv[1] != NULL);
   10     hp = gethostbyname(argv[1]);
   11     assert(hp != NULL);
-> 12     printf("%s\n", hp->h\_name);
   13
return 0;
```

The program has halted at line 12 and lets us inspect the value of variables that are in scope, confirming that the hp pointer is NULL.

```
(lldb) print hp
(hostent *) $1 = 0x0000000000000000
```

Example: Using lldb from LLVM (cont.)

We do not have to wait for a crash to inspect variables.
Breakpoints allow us to halt execution at a function call.

```
(lldb) break set --name main
(lldb) run www.recoil.org
    7    {
    8      struct hostent *hp;
    9      assert(argv[1] != NULL);
-> 10     hp = gethostbyname(argv[1]);
```

The program has run until the main function is encountered, and stopped at the first statement.

Example: Using lldb from LLVM (cont.)

We can set a watchpoint to inspect when variables change state.

```
(lldb) watchpoint set variable hp
```

This will pause execution right after the hp variable is assigned to. We can now resume execution and see what happens:

```
(lldb) continue
Process 9661 resuming
Process 9661 stopped
* thread #1: tid = 0x3c2fd3 <..> stop reason = watchpoint 1
frame #0: 0x0000000100000f4e <...> debug-s18.c:12
    9    assert(argv[1] != NULL);
   10    hp = gethostbyname(argv[1]);
   11    assert(hp != NULL);
-> 12    printf("%s\n", hp->h_name);
   13    return 0;
   14  }
```

Example: Using lldb from LLVM (cont.)

When program execution is paused in lldb, we can inspect the local variables using the print command.

```
(lldb) print hp
(hostent *) $0 = 0x0000000100300460
(lldb) print hp->h_name
(char *) $1 = 0x0000000100300488 "bark.recoil.org"
```

We can thus:

- confirm that hp is non-NULL even when the program does not crash
- print the contents of the `hp->h` name value, since the debugger can follow pointers
- see the C types that the variables had (e.g. `hostent *`)

Debugging Symbols

How did the debugger find the source code in the compiled executable? Compile it without the `-g` flag to see what happens.

```
cc -Wall -DNDEBUG debug-s18.c
```

```
(lldb) run doesnotexist.abc
loadermain + 50:
-> 0x100000f52: movq  (%rax), %rsi
   0x100000f55: movb  $0x0, %al
   0x100000f57: callq 0x100000f72 ; symbol stub for: printf
   0x100000f5c: movl  $0x0, %ecx
```

We now only have assembly language backtrace, with some hints in the output about `printf`.

Debugging Symbols (cont.)

The compiler emits a *symbol table* that records the mapping between a programs variables and their locations in memory.

- Machine code uses memory addresses to reference memory, and has no notion of variable names. For example, `0x100000f72` is the address of printf earlier
- The symbol table records an association from `0x100000f72` and the printf function
- The `-g` compiler flag embeds additional debugging information into the symbol tables
- This debugging information also maps the source code to the program counter register, keeping track of the control flow

Debugging Symbols (cont.)

Debugging information is not included by default because:

- The additional table entries take up a significant amount of extra disk space, which would be a problem on embedded systems like a Raspberry Pi
- They are not essential to run most applications, unless they specifically need to modify their own code at runtime
- Advanced users can still use debuggers with just the default symbol table, although relying on the assembly language is more difficult
- Modern operating systems such as Linux, MacOS X and Windows support storing the debugging symbols in a separate file, making disk space and compilation time the only overhead to generating them.

Debugging Tools

`lldb` is just one of a suite of debugging tools that are useful in bug hunting.

- The LLVM compiler suite (`https://llvm.org`) also has the `clang-analyzer` static analysis engine that inspects your source code for errors.
- The GCC compiler (`https://gcc.gnu.org`) includes the `gdb` debugger, which has similar functionality to `lldb` but with a different command syntax.
- Valgrind (`http://valgrind.org`) (seen in an earlier lecture!) is a dynamic analysis framework that instruments binaries to detect many classes of memory management and threading bugs.

Unit and Regression Test

We have used many techniques to find our bugs, but it is equally important to make sure they do not return. Create a unit test:

```
1  #include <stdio.h>
2  #include <netdb.h>
3  #include <assert.h>
4  void lookup(char *buf) {
5    assert(buf != NULL);
6    struct hostent *hp = gethostbyname(buf);
7    printf("%s -> %s\n", buf, hp ? hp->h_name : "unknown");
8  }
9  void lookup_test(void) {
10   lookup("google.com");
11   lookup("doesntexist.abc");
12   lookup("");
13   lookup(NULL);
14 }
```

Unit and Regression Test (cont.)

We can now invoke `lookup()` for user code, or lookup `test()` to perform the self-test.

```
1  #include <stdlib.h>
2
3  void lookup(char *buf);
4  void lookup_test(void);
5
6  int main(int argc, char **argv) {
7    if (getenv("SELFTEST"))
8      lookup_test ();
9    else
10     lookup(argv[1]);
11 }
```

Unit and Regression Test (cont.)

Can now run this code as a test case or for live lookups.

```
cc -Wall -g lookup_logic.c lookup_main.c -o lookup
./lookup google.com
# for live operation
env SELFTEST=1 ./lookup # for unit tests
```

Unit and Regression Tests (cont.)

Building effective unit tests requires methodical attention to detail:

- The use of `assert` in the lookup logic is a poor interface, since it terminates the entire program. How could this be improved?
- The unit tests in lookup test are manually written to enumerate the allowable inputs. How can we improve this coverage?
- C and C++ have many *open-source unit test frameworks* available. Using them gives you access to widely used conventions such as `xUnit` that helps you structure your tests.
- Take the opportunity to run your unit tests after every significant code change to spot unexpected failures (dubbed *regression testing*).
- *Continuous integration* runs unit tests against every single code commit. If using GitHub, then Travis CI (`https://travis-ci.org`) will be useful for your projects in any language.