Lecture - XV
Debugging

Good Programming Habits

- More important than debugging: do not write bugs!
- Write simple code!

```c
/* How is anyone supposed to understand this system? */
for(i=0; i<10; i++)
for(j=0; j<10; j++)
	for(k=0; k<10; k++)
					cout<<i<<j<<k;
```
- Always use {} around compounds:

```c
/* This code probably does not do what you expect */
while (i < 4) {
	found = find(i);
	++i;
```

Check Function Return Values

- Most functions from the C library return values
  - Most often: >= 0 if everything went fine, < 0 in case of error
- Always check these return values!
  - I often don’t write it in my slides by lack of space
  - But you do not have any excuse for not doing it...

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
	// error...
}
```

Use perror()

- There is a standard global variable called errno
  - It is defined in <errno.h>
- When standard functions fail, they store an error code in errno
- You should look at errno for the cause of the problem
- To convert int errno into a human-readable string:

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
	 perror("Error while opening socket");
	 exit(1);
}
```

Use Assertions

- Often in a program you know that a given property should normally be true:
  - This variable’s value should always between 0 and 10
  - This pointer should not be null
  - max_data_rate should always be lower than max_data_rate
  - etc...
- Use assert() to check if those properties are true:
  - If the property is true, assert will do nothing
  - Otherwise, it will display a message, stop the program and dump a core
- Use GDB to read the core file and see what happened!

```c
#include <cassert.h>
#define assert(expr)

assert((expr));
```

Use Assertions

```sh
$ cat prog.c
#include <assert.h>
int main(int argc, char **argv) {
	/* this program should never take any command-line parameter */
	assert(argc==1);
	return 0;
}
$ prog
$ prog0 wrongparameter
prog: prog0:0: main: Assertion 'argc==1' failed: (core dumped)
```

- Use:
- `assert(expr)`
Avoid These Functions!

- Certain standard C functions do not let you control buffer boundaries
  - You should never use them
  - There is always a good replacement for these

<table>
<thead>
<tr>
<th>Do not use</th>
<th>Use instead</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy</td>
<td>strncpy</td>
</tr>
<tr>
<td>gets</td>
<td>fgets</td>
</tr>
</tbody>
</table>

Use Proper Formatting

- If you want to display a string:
  ```c
  printf("%s", string);  // This is correct
  print(string);         // This is wrong: wrong function
  ```

- Try this program (`echo`):
  ```c
  int main(int argc, char **argv) {
    int i;
    for (i = 0; i < argc; i++) printf("%s ", argv[i]);  // No format string here
    printf("\n");
  }
  ```

  ```
  $ ./a.out foobars
  foobars
  $ ./a.out foo bars
  foo bars
  ```

GDB: The GNU Debugger

- A debugger can do two things for you:
  - Run a program step by step, let you follow what it is doing, examine the content of the memory
  - After a program has crashed, load the core file and let you examine what has happened
- GDB can debug programs written in C, C++, Pascal, ADA, etc.
- Current version: 6.6
- `http://www.gnu.org/software/gdb/`

Compiling with Debugging Info

- GDB can debug any program
  - But when it executes an instruction, you probably want to see the source code of the instruction being executed
  - This information is normally not present in executable files
- To get them, you must add a flag at compile time
  - This is not necessary at link time (but it cannot hurt)

```c
$ gcc -g -Wall foobar.c
$ gcc -o foo foobar.o
```

- This includes line-number information in your compiled programs

GDB Basic Commands

- Basic commands:
  - To run GDB: gdb [program name]
  - To set a breakpoint: break [function name]
  - w: [function name]
  - b: [function name] (time-arg)
  - To display the source around the current instruction: list (or l)
  - To start running the program: run [command-line param]
  - To continue the execution after a breakpoint: c
  - To execute one instruction:
    - next or n (nexts a function call as a single instruction)
    - step or s (steps inside a function when it is called)
  - To print the value of a variable: print [var] or p [var]
  - To use the function stack:
    - To reexecute the last command: <error>
    - To quit: quit

Example

```c
#include <stdio.h>

void foo() {
    printf("This is function foo()\n");
}

int main() {
    int i;
    while (i++) {
        foo();
        foo();
        return 0;
    }
} ```

GDB Can Show More...

```c
struct complex {
    float real;
    float imag;
};

struct timespec {
    struct timespec next;
    struct timespec prev;
};

int main() {
    struct timespec m1 = (1.5, 1.0), m2;
    struct timespec m3 = m1
    return 0;
}
```

Debugging After Core Dump

- Did you ever wonder what “core dump” means?
  - When a program crashes, your operating system saves the whole state of the program's memory into a file.
  - So that you can have a look and identify what went wrong
    - Which instruction caused the crash
    - What was the state of the function stack
    - What was the contents of variables
- It is up to you to figure out why the program reached that state!

Debugging After Core Dump

- Programs dump a core:
  - Upon a segmentation fault (your program tried to access a protected piece of memory)
  - Upon a bus error (your program tried to make a non-aligned memory access)
  - E.g., integer's memory addresses must be multiples of 4
  - When a program calls abort()
  - When an assert() fails
- Sometimes the system will not dump any core
  - Type this command, then run your program again in the same terminal

```bash
(gdb) list
1 int main() {
2     ...
19 }
(gdb) continue
This is function foo()
Program exited normally.
(gdb) quit
```

```bash
$ gdb prog
GDB 7.10.1-6ubuntu2 (Ubuntu 7.10.1-6ubuntu2)
Copyright 2012-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
(gdb) run
Starting program: /home/guerrero/prog1
Breakpoint 1, foo () at prog1.c:4
4   printf("This is function foo()");
(gdb) where
#0  foo () at prog1.c
#1  0x0000000004040404 in main () at prog1.c:11
(gdb) up
#1  0x0000000004040404 in main () at prog1.c:11
#2  foo();
```
DDD: The Data Display Debugger

- When you have complex data structures it can be tedious to explore them with gdb
  - DDD is especially good at displaying them graphically
- DDD is not a debugger but just a graphical interface
  - It starts GDB for you
  - Every action you make is translated into a GDB command
  - It displays the result graphically
- It can also interface to the Java debugger, perl, bash, etc.
- Current version: 3.3.11
  - http://www.gnu.org/software/ddd/

Valgrind

- GDB does little to detect memory leaks
  - It merely shows you what is going on
  - It does not "know" what is good or bad programming
  - Memory leaks do not directly produce an error
  - They are hard to locate with GDB
- Valgrind is specialized in memory-related bugs
  - Current version: 3.6.0
  - http://valgrind.org/
- Valgrind is a set of tools
  - Two memory error detectors, a thread error detector, a cache profiler and a heap profiler
  - The most important one: Memcheck (memory debugger)
Splint

- Very long ago, somebody wrote a program called lint
  - It took a C source file as input
  - And checked for common mistakes

- Even better: splint
  - http://www.splint.org/
  - It checks for common bugs
  - Focuses mostly on security holes (but not only)

- splint will issue warnings
  - Some warnings you may decide to ignore (at your own risk)
  - Remember, even if splint does not display anything, this does not mean that your program is correct!

Example

- Let us write a very bad program:

```c
#include <stdio.h>

int main() {
  char buf[16];
  printf(buf);
  return 0;
}
```

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), and J.Kim (KAIST).