Debugging

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 5

Goals of this Lecture

• Help you learn about:
  • Strategies and tools for debugging your code

• Why?
  • Debugging large programs can be difficult
  • A power programmer knows a wide variety of debugging strategies
  • A power programmer knows about tools that facilitate debugging
    • Debuggers
    • Version control systems
Testing vs. Debugging

• Testing
  • What should I do to try to break my program?

• Debugging
  • What should I do to try to fix my program?

Debugging Heuristics

<table>
<thead>
<tr>
<th>Debugging Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Understand error messages</td>
</tr>
<tr>
<td>(2) Think before writing</td>
</tr>
<tr>
<td>(3) Look for familiar bugs</td>
</tr>
<tr>
<td>(4) Divide and conquer</td>
</tr>
<tr>
<td>(5) Add more internal tests</td>
</tr>
<tr>
<td>(6) Display output</td>
</tr>
<tr>
<td>(7) Use a debugger</td>
</tr>
<tr>
<td>(8) Focus on recent changes</td>
</tr>
</tbody>
</table>
Understand Error Messages

Debugging at **build-time** is easier than debugging at **run-time**, if and only if you...

Understand the error messages

```c
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and return 0. */
{
    printf("hello, world\n");
    return 0;
}
```

What's the error? (Don’t look at next slide.)

Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

```c
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and return 0. */
{
    printf("hello, world\n");
    return 0;
}
```

Which tool (preprocessor, compiler, or linker) reports the error(s)?

```bash
$ gcc hello.c -o hello
hello.c:1:20: stdio.h: No such file or directory
hello.c:3:1: unterminated comment
hello.c:2: error: syntax error at end of input
```
(1) Understand the error messages (cont.)

```c
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
return 0. */
{
    printf("hello, world\n");
    return 0;
}
```

What are the error(s)?
(Don't look at the next slide.)

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:7: error: 'retun' undeclared (first use in this function)
hello.c:7: error: (Each undeclared identifier is reported only once)
hello.c:7: error: for each function it appears in.)
hello.c:7: error: syntax error before numeric constant
```

Which tool (preprocessor, compiler, or linker) reports the error(s)?
(1) Understand error messages (cont.)

```c
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and return 0. */
{
    printf("hello, world\n");
    return 0;
}
```

What are the error(s)?
(Don't look at the next slide.)

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:6: warning: implicit declaration of function `printf'
/tmp/cc43abjk.o(.text+0x25): In function `main':
    : undefined reference to `printf'
collect2: ld returned 1 exit status
```
Think Before Writing

Inappropriate changes could make matters worse, so…

(2) Think before writing
- Draw pictures of the data structures
- Take a break
  - Sleep on it
  - Start early so you can sleep on it
- Explain the code to:
  - Yourself
  - Someone else

Look for Familiar Bugs

(3) Look for familiar bugs
- Some of our favorites:

```c
switch (i) {
    case 0:
      ...
      break;
    case 1:
      ...
    case 2:
      ...
}
```

```c
int i;
  ...
  scanf("%d", i);

char c;
  ...
  c = getchar();
```

```c
while (c = getchar() != EOF)
  ...
```

```c
if (i = 5)
  ...
```

```c
if (i & j)
  ...
```

```c
if (5 < i < 10)
  ...
```

What are the errors?
Divide and Conquer

(4) Divide and conquer

- Incrementally find smallest/simplest input that illustrates the bug
  - Example: Program fails on large input file filex

- Approach 1: Remove input
  - Start with filex
  - Incrementally remove lines of filex until bug disappears
    - Maybe in “binary search” fashion

- Approach 2: Add input
  - Start with small subset of filex
  - Incrementally add lines of filex until bug appears

Divide and Conquer (cont.)

(4) Divide and conquer (cont.)

- Incrementally find smallest code subset that illustrates the bug
  - Example: Test client for your module fails

- Approach 1: Remove code
  - Start with test client
  - Incrementally remove lines of test client until bug disappears

- Approach 2: Add code
  - Start with minimal client
  - Incrementally add lines of test client until bug appears
Add More Internal Tests

(5) Add more internal tests

- Internal tests help **find** bugs (see “Testing” lecture)
- Internal test also can help **eliminate** bugs
  - Checking invariants and conservation properties can eliminate some functions from the bug hunt

Display Output

(6) Display output

- Print values of important variables at critical spots

- Poor:
  
  ```c
  printf("%d", keyvariable);
  ```

  `stdout` is buffered; program may crash before output appears

- Maybe better:

  ```c
  printf("%d\n", keyvariable);
  ```

  Printing '\n' flushes the stdout buffer, but not if stdout is redirected to a file

- Better:

  ```c
  printf("%d", keyvariable);
  fflush(stdout);
  ```

Call **fflush**() to flush stdout buffer explicitly
(6) Display output (cont.)

• Maybe even better:

```c
fprintf(stderr, "%d", keyvariable);
```

• Maybe better still:

```c
FILE *fp = fopen("logfile", "w");
    fprintf(fp, "%d", keyvariable);
    fflush(fp);
```

Write debugging output to `stderr`; debugging output can be separated from normal output via redirection.

Bonus: `stderr` is unbuffered.

Write to a log file.

(7) Use a debugger

• Alternative to displaying output

• Bonuses:
  • Debugger can load “core dumps”
  • Examine state of program when it terminated
  • Debugger can “attach” to a running program
The GDB Debugger

- GNU Debugger
  - Part of the GNU development environment
  - Integrated with Emacs editor
  - Allows user to:
    - Run program
    - Set breakpoints
    - Step through code one line at a time
    - Examine values of variables during run
    - Etc.

- See Appendix 1 for details

Focus on Recent Changes

(8) Focus on recent changes

- Corollary: Debug now, not later

**Difficult:**
1. Write entire program
2. Test entire program
3. Debug entire program

**Easier:**
1. Write a little
2. Test a little
3. Debug a little
4. Write a little
5. Test a little
6. Debug a little
...


Focus on Recent Changes (cont.)

(8) Focus on recent changes (cont.)

• Corollary: Maintain old versions

<table>
<thead>
<tr>
<th>Difficult:</th>
<th>Easier:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Change code</td>
<td>(1) Backup working version</td>
</tr>
<tr>
<td>(2) Note bug</td>
<td>(2) Change code</td>
</tr>
<tr>
<td>(3) Try to remember what changed since last working version</td>
<td>(3) Note bug</td>
</tr>
<tr>
<td></td>
<td>(4) Compare code with working version to determine what changed</td>
</tr>
</tbody>
</table>

Maintaining Previous Versions

• To maintain old versions
  • Approach 1: Manually copy project directory

```
$ mkdir myproject
$ cd myproject
    Create project files here.
$ cd..
$ cp -r myproject myprojectDateTime
$ cd myproject
    Continue creating project files here.
...
```

• Repeat occasionally
• Approach 2: Use RCS...
RCS

Revision Control System
• A simple personal version control system
• Provided with many Linux distributions
  • Available on hats
• Allows developer to:
  • Create a source code repository
  • Check source code files into repository
    • RCS saves old versions
  • Check source code files out of repository
  • Appropriate for one-developer projects
  • Not appropriate for multi-developer projects
    • Use CVS or Subversion instead
• See Appendix 2 for details

Summary

<table>
<thead>
<tr>
<th>Debugging Heuristic</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Understand error messages</td>
<td></td>
</tr>
<tr>
<td>(2) Think before writing</td>
<td></td>
</tr>
<tr>
<td>(3) Look for familiar bugs</td>
<td></td>
</tr>
<tr>
<td>(4) Divide and conquer</td>
<td></td>
</tr>
<tr>
<td>(5) Add more internal tests</td>
<td></td>
</tr>
<tr>
<td>(6) Display output</td>
<td></td>
</tr>
<tr>
<td>(7) Use a debugger *</td>
<td></td>
</tr>
<tr>
<td>(8) Focus on recent changes **</td>
<td></td>
</tr>
</tbody>
</table>

* Use GDB
** Use RCS
Appendix 1: Using GDB

• An example program
  File testintmath.c:

  ```c
  #include <stdio.h>
  int gcd(int i, int j) {
    int temp;
    while (j != 0) {
      temp = i % j;
      i = j;
      j = temp;
    }
    return i;
  }

  int lcm(int i, int j) {
    return (i / gcd(i, j)) * j;
  }
  ```

  The program is correct
  But let’s pretend it has a runtime error in `gcd()`…

  ```c
  int main(void) {
    int iGcd;
    int iLcm;
    iGcd = gcd(8, 12);
    iLcm = lcm(8, 12);
    printf("%d %d\n", iGcd, iLcm);
    return 0;
  }
  ```

Euclid’s algorithm; Don’t be concerned with details

Appendix 1: Using GDB (cont.)

• General GDB strategy:

  • Execute the program to the point of interest
    • Use breakpoints and stepping to do that
  
  • Examine the values of variables at that point
Appendix 1: Using GDB (cont.)

• Typical steps for using GDB:

  (1) Build with \texttt{-g}

  \texttt{gcc testintmath.c \texttt{-o testintmath}}

  • Adds extra information to executable file that GDB uses

  (2) Run Emacs, with no arguments

  \texttt{emacs}

  (3) Run GDB on executable file from within Emacs

  \texttt{<Esc key> x gdb <Enter key> testintmath <Enter key>}

  (4) Set breakpoints, as desired

  \texttt{break main}

  • GDB sets a breakpoint at the first executable line of main()

  \texttt{break gcd}

  • GDB sets a breakpoint at the first executable line of gcd()

Appendix 1: Using GDB (cont.)

• Typical steps for using GDB (cont.):

  (5) Run the program

  \texttt{run}

  • GDB stops at the breakpoint in main()

  • Emacs opens window showing source code

  • Emacs highlights line that is to be executed next

  \texttt{continue}

  • GDB stops at the breakpoint in gcd()

  • Emacs highlights line that is to be executed next

  (6) Step through the program, as desired

  \texttt{step (repeatedly)}

  • GDB executes the next line (repeatedly)

  • Note: When next line is a call of one of your functions:

  • \texttt{step} command \texttt{steps into} the function

  • \texttt{next} command \texttt{steps over} the function, that is, executes the next line

    without stepping into the function
Appendix 1: Using GDB (cont.)

• Typical steps for using GDB (cont.):

(7) Examine variables, as desired
   \texttt{print } i
   \texttt{print } j
   \texttt{print } temp
   \textbullet{} GDB prints the value of each variable
(8) Examine the function call stack, if desired
   \texttt{where}
   \textbullet{} GDB prints the function call stack
   \textbullet{} Useful for diagnosing crash in large program
(9) Exit gdb
   \texttt{quit}
(10) Exit Emacs
   \texttt{<Ctrl-x key> <Ctrl-c key>}

Appendix 1: Using GDB (cont.)

• GDB can do much more:
  • Handle command-line arguments
    \texttt{run arg1 arg2}
  • Handle redirection of stdin, stdout, stderr
    \texttt{run < somefile > someotherfile}
  • Print values of expressions
  • Break conditionally
  • Etc.

• See \textit{Programming with GNU Software} (Loukides and Oram) Chapter 6
Appendix 2: Using RCS

- Typical steps for using RCS:
  1. Create project directory, as usual
     ```
     mkdir helloproj
     cd helloproj
     ```
  2. Create RCS directory in project directory
     ```
     mkdir RCS
     ```
     RCS will store its repository in that directory
  3. Create source code files in project directory
     ```
     emacs hello.c ...
     ```
  4. Check in
     ```
     ci hello.c
     ```
     - Adds file to RCS repository
     - Deletes local copy (don’t panic!)
     - Can provide description of file (1st time)
     - Can provide log message, typically describing changes
  5. Check out most recent version for reading
     ```
     co hello.c
     ```
     - Copies file from repository to project directory
     - File in project directory has read-only permissions
  6. Check out most recent version for reading/writing
     ```
     co -l hello.c
     ```
     - Copies file from repository to project directory
     - File in project directory has read/write permissions
  7. List versions in repository
     ```
     rlog hello.c
     ```
     - Shows versions of file, by number (1.1, 1.2, etc.), with descriptions
  8. Check out a specified version
     ```
     co -l -r versionnumber hello.c
     ```
Appendix 2: Using RCS (cont.)

- RCS can do much more:
  - Merge versions of files
  - Maintain distinct development branches
  - Place descriptions in code as comments
  - Assign symbolic names to versions
  - Etc.

- See *Programming with GNU Software* (Loukides and Oram) Chapter 8

- Recommendation: Use RCS
  - `ci` and `co` can become automatic!

Appendix 3: Debugging Mem Mgmt

- Some debugging techniques are specific to *dynamic memory management*
  - That is, to memory managed by `malloc()`, `calloc()`, `realloc()`, and `free()`

- Soon will be pertinent in the course

- For future reference...
Appendix 3: Debugging Mem Mgmt (cont.)

(9) Look for familiar dynamic memory management bugs

• Some of our favorites:

```c
int *p; /* value of p undefined */
    ... *p = somevalue;
```

Dangling pointer

```c
int *p; /* value of p undefined */
    ... fgets(p, 1024, stdin);
```

Dangling pointer

```c
int *p;
    ... p = (int*)malloc(sizeof(int));
    ... free(p);
    *p = 5;
```

Dangling pointer

---

Appendix 3: Debugging Mem Mgmt (cont.)

(9) Look for familiar dynamic memory management bugs (cont.)

• Some of our favorites (cont.):

```c
int *p;
    ...
    p = (int*)malloc(sizeof(int));
    ...
```

Memory leak

alias

Garbage creation

Detection: Valgrind, etc.

```c
int *p;
    ...
    p = (int*)malloc(sizeof(int));
    ...
    free(p);
    ...
    free(p);
```

Multiple free

Detection: man malloc, MALLOC_CHECK_
Appendix 3: Debugging Mem Mgmt (cont.)

(9) Look for familiar dynamic memory management bugs (cont.)
  • Some of our favorites (cont.):

```c
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(strlen(s1));
strcpy(s2, s1);
```

Allocating too few bytes

```c
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(sizeof(s1));
strcpy(s2, s1);
```

Allocating too few bytes

```c
double *p;
p = (double*)malloc(sizeof(double));
```

Allocating too few bytes

Appendix 3: Debugging Mem Mgmt (cont.)

(10) Segmentation fault? Make it happen within gdb, and then issue the `gdb where` command. The output will lead you to the line that caused the fault. (But that line may not be where the error resides.)

(11) Call `assert()` to make sure value returned by `malloc()`, `calloc()`, and `realloc()` is not NULL.

(12) Manually inspect each call of `malloc()`, `calloc()`, and `realloc()` in your code, making sure that it allocates enough memory.

(13) Temporarily hardcode each call of `malloc()`, `calloc()`, and `realloc()` such that it requests a large number of bytes. If the error disappears, then you'll know that at least one of your calls is requesting too few bytes.
(14) Temporarily comment-out each call of free() in your code. If the error disappears, then you'll know that you're freeing memory too soon, or freeing memory that already has been freed, or freeing memory that should not be freed, etc.

(15) Use the Meminfo tool. Programs built with gcc217m are much more sensitive to dynamic memory management errors than are programs built with gcc217. So the error might manifest itself earlier, and thereby might be easier to diagnose.