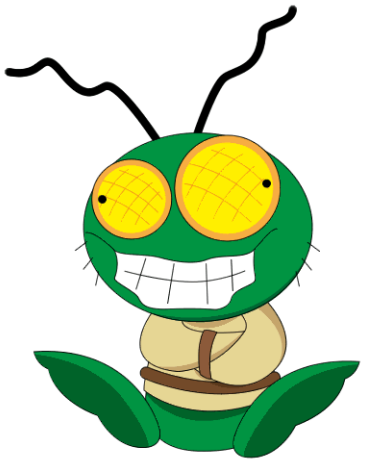
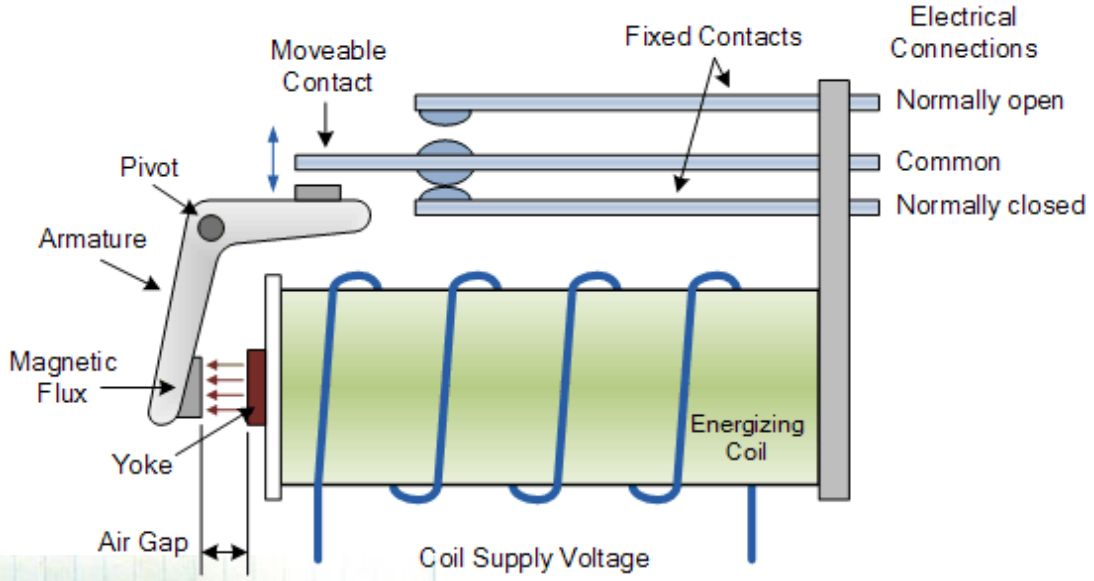
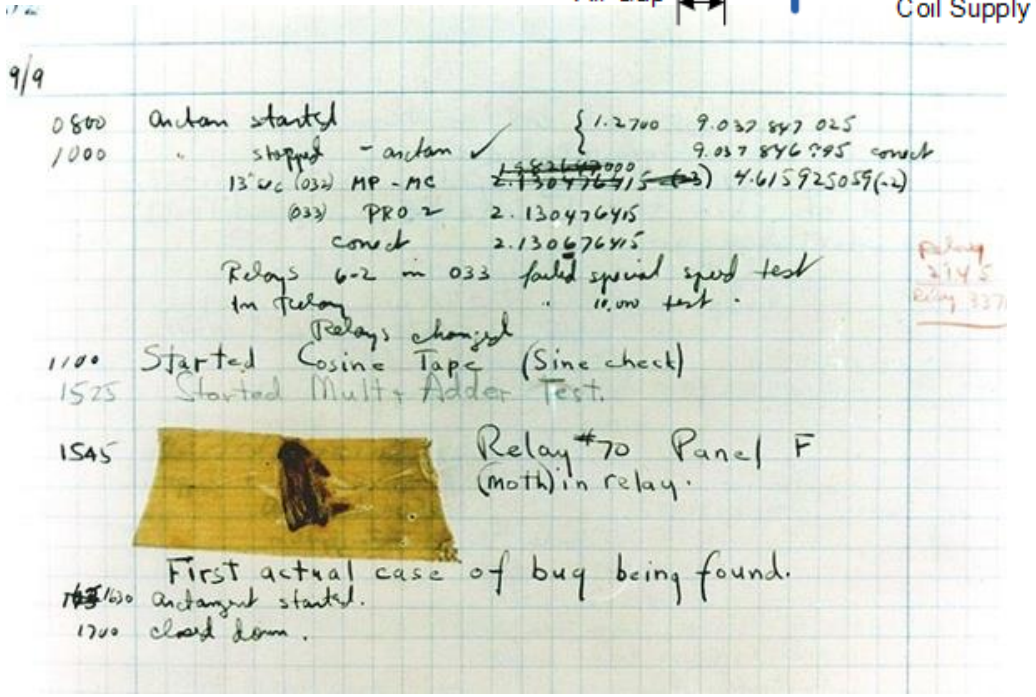
C Debugging Basics

No relevant text



First Computer Bug





The “printf” debugger

- Insert printf statements to print debug information
- Build/Run
- Modify to print new information

Advantages

- Simple
- Complete
- Available anywhere

Disadvantages

- Time consuming
- TMI
- Side effects change bug
- Forgetting to remove debug

Assert

- Method to check specific “assertions”
- An assertion is any logical expression, evaluated to true or false
- Expression passed as an argument to the “assert” function
  - If the expression is true, “assert” does nothing.
  - If the expression is false, “assert”…
    - writes an error message to stderr which contains
      - function name, assertion, and line number
    - aborts the C program (including a “core dump” if enabled)
- Assertion evaluation can be turned off once program is debugged
  - #define NDEBUG

Assertion example

```
#inclue <assert.h>
#include <stdio.h>

int main() {
        int j;
        for(j=0;j!=100;j++) {
                assert(j<100);
                j=factorial(j);
        }
}
```

GDB – A Source Level Debugger!

- Compiling with "-g" option causes compiler/linker to track:
  - C file names
  - C instructions/line numbers
  - Variable names and types
  - Function names and line numbers and stack frame layout
- Compiling with –g required for GDB
- Compiling with –g limits optimization and increases executable size

Command Line Mentality

Start w/ gdb command, specifying executable file as argument

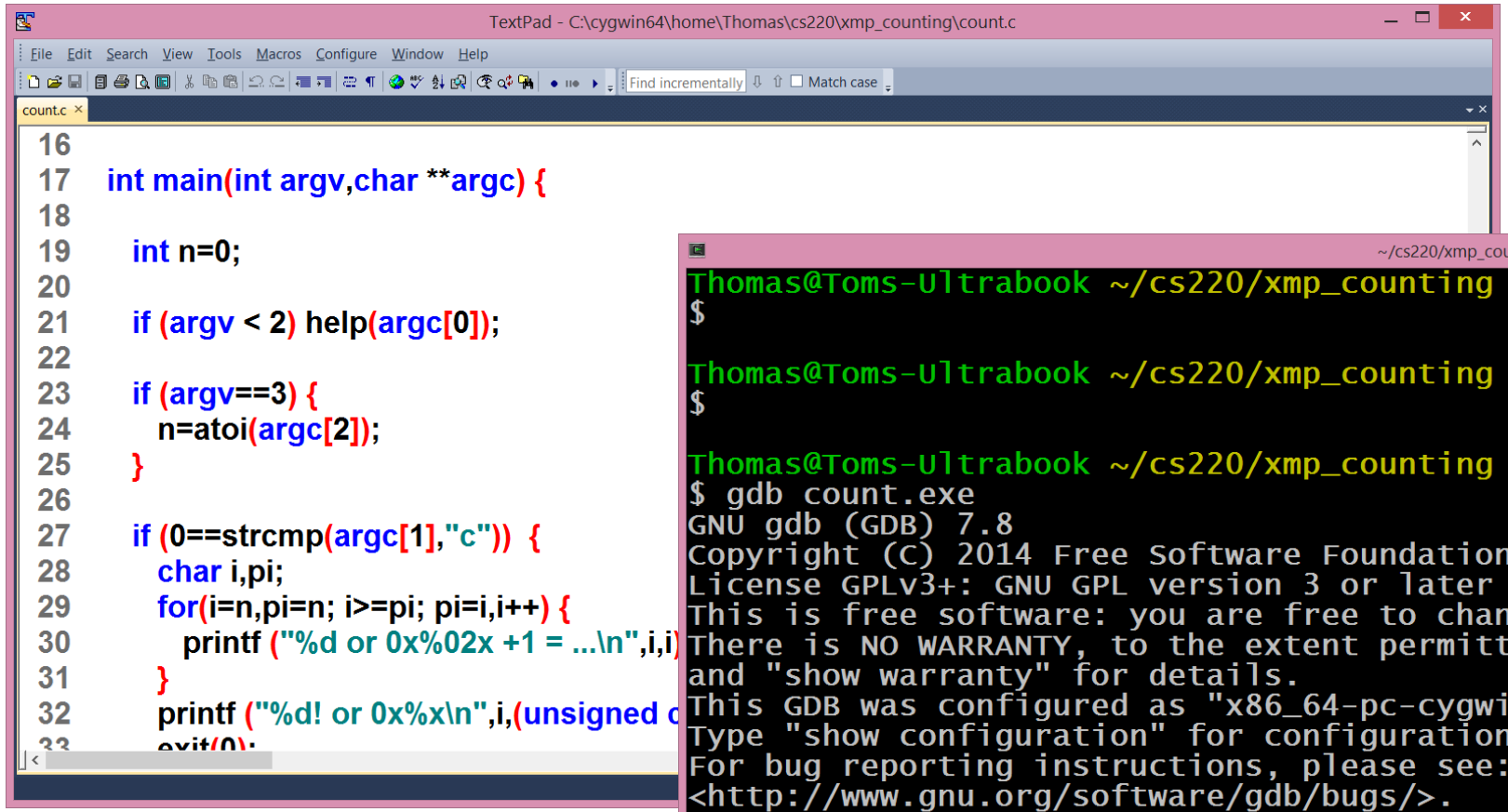
gdb prints boilerplate info

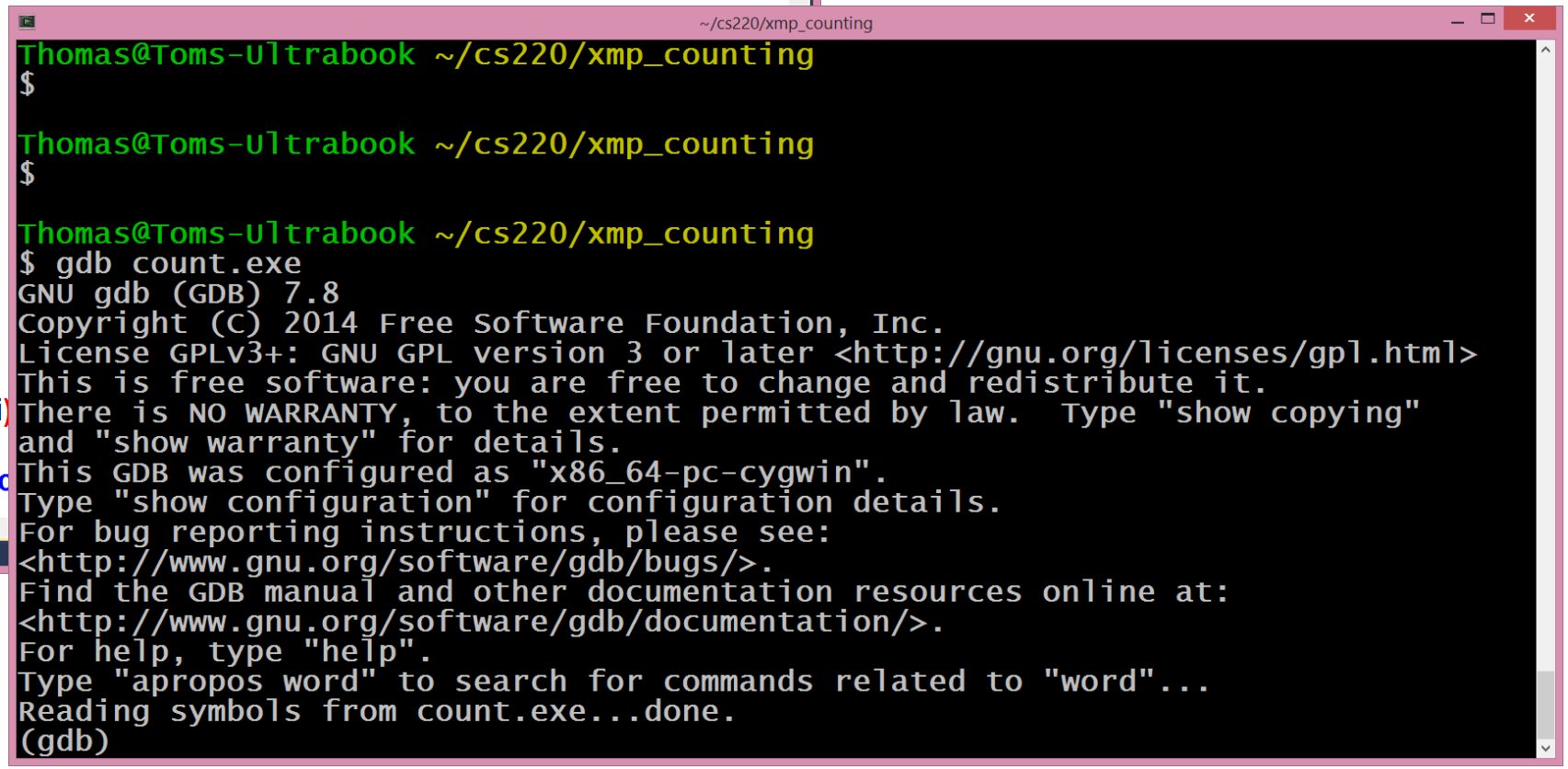
gdb loads the executable, but does NOT transfer control

gdb opens a gdb prompt

```
$ gdb count.exe
GNU gdb (GDB) 7.8
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-cygwin".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from count.exe...done.
(gdb)
```

Hint: Open Editor/Code window first





GDB "Context"

- Current source file
- Current instruction: Line number and instruction about to execute
- Call stack: List of function calls, starting at "main" that got you to the current instruction
  - Each entry is a "frame"
  - main's frame is on "top"
  - Current "frame" is a pointer into the call stack, typically "bottom" frame
  - GDB Variable scope is scope of the current frame
- Breakpoint list: List of files/lines/functions where gdb should interrupt execution and open command line prompt
- …

Initial GDB prompt

- Executable is loaded, but gdb is still in control
- Current GDB context…
  - Current file: C file which contains “main” function
  - Current instruction: None
  - No breakpoints set
  - No function stack
- Need a “(gdb) run <cmd_args>” command to transfer execution
  - Parses <cmd_args> into arguments to main
  - Transfers execution to main
  - WARNING: program will run to completion if no breakpoints are set!

GDB Command Line Processing

- (gdb) <enter> - Repeat previous command
- (gdb) <command> <arguments>
  - <command> can be the shortest non-ambiguous gdb command prefix
    - e.g. “(gdb) n” for “next”
    - e.g. “(gdb) where” for “where”
    - (gdb) w
      Ambiguous command "w": watch, wh, whatis, where, while, while-stepping, winheight, ws.
- Up/Down arrow to scroll through command history
- Left/Right arrow to edit command
- (gdb) help <cmd>

Managing Breakpoints

- Setting breakpoints: (gdb) b <location> if <condition>
  - Location:
    - <##> - break before instruction at line number <##> in current file
    - <file>:<##> - break before instruction at line number <##> in file <file>
    - <function_name> - break before first instruction in <function_name> function
    - No location – break before next instruction in current frame of call stack
  - Condition:
    - Any valid C expression… like "(*stringPtr)!=0x00"
- Listing breakpoints: (gdb) info breakpoints
- Commands at breakpoints: (gdb) commands
- Removing breakpoints: (gdb) clear <location>

Single Stepping Code

- (gdb) s[tep]
  - Run next C instruction
  - If there is a function call, step into function (includes library functions)
  - If function in a file not compiled w/ -g, stop after function
  - Leaky abstractions: implicit returns and macros
- (gdb) n[ext]
  - Run next C instruction
  - If there is a function call, execute the function, but do not break inside the function
- (gdb) c[ontinue]
  - Continue execution to next breakpoint

Dealing w/ Data

- (gdb) p[rint] <expression>
  - Uses expression type to format result
  - <expression> may be most valid C expressions
  - <expression> may contain any variable visible in the current call stack frame scope
- (gdb) set <variable> = <expression>

Where Are You?

```
1 int fnB(int x, int y);
2 int fnC(int x, int y);
3
4 int main(int argc,char **argv) {
5               int i=fnb(3,4);
6               int j=fnb(3,6);
7               int k=fnb(3,1);
8               return 0;
9 }
10 int fnb(int x,int y) {
11              int m=fnc(x,1);
12              int n=fnc(1,y);
13              return m+n;
14 }
15 int fnc(int x, int y) {
16              return x%y;
17 }
```

```
(gdb) b 16
(gdb) run
...

Breakpoint 1, fnc (x=3, y=1) at xmp_where.c:16
16          return x%y;
(gdb)
```

You are here, but how did you get here?

see xmp_where

Where Are You?

```
1 int fnB(int x, int y);
2 int fnC(int x, int y);
3
4 int main(int argc,char **argv) {
5               int i=fnb(3,4);
6               int j=fnb(3,6);
7               int k=fnb(3,1);
8               return 0;
9 }
10 int fnb(int x,int y) {
11              int m=fnc(x,1);
12              int n=fnc(1,y);
13              return m+n;
14 }
15 int fnc(int x, int y) {
16              return x%y;
17 }
```

```
(gdb) b 16
(gdb) run
...

Breakpoint 1, fnc (x=3, y=1) at xmp_where.c:16
16          return x%y;
(gdb) where
#0  fnc (x=3, y=1) at xmp_where.c:16
#1  0x0000000100401150 in fnb (x=3, y=4) at xmp_where.c:11
#2  0x0000000100401103 in main (argc=1, argv=0x23cb20) at xmp_where.c:5
(gdb)
```

Managing the call stack

- (gdb) where
  - prints the call stack
  - Each frame: [#]<nn> <function> (<args>) at <file>:<line>
- (gdb) up / (gdb) down
  - Move the current frame up or down in the stack list
- (gdb) frame
  - More gory details about current stack frame

Where Are You?

```
1 int fnB(int x, int y);
2 int fnC(int x, int y);
3
4 int main(int argc,char **argv) {
5               int i=fnb(3,4);
6               int j=fnb(3,6);
7               int k=fnb(3,1);
8               return 0;
9 }
10 int fnb(int x,int y) {
11              int m=fnc(x,1);
12              int n=fnc(1,y);
13              return m+n;
14 }
15 int fnc(int x, int y) {
16              return x%y;
17 }
```

```
(gdb) b 16
(gdb) run
...

Breakpoint 1, fnc (x=3, y=1) at xmp_where.c:16
16          return x%y;
(gdb) where
#0  fnc (x=3, y=1) at xmp_where.c:16
#1  0x0000000100401150 in fnb (x=3, y=4) at xmp_where.c:11
#2  0x0000000100401103 in main (argc=1, argv=0x23cb20) at xmp_where.c:5
(gdb) print y
$1 = 1
(gdb) up
#1  0x0000000100401150 in fnb (x=3, y=4) at xmp_where.c:11
11          int m=fnc(x,1);
(gdb) where
#0  fnc (x=3, y=1) at xmp_where.c:16
#1  0x0000000100401150 in fnb (x=3, y=4) at xmp_where.c:11
#2  0x0000000100401103 in main (argc=1, argv=0x23cb20) at xmp_where.c:5
(gdb) print y
$2 = 4
```