Systems and Internet Infrastructure Security

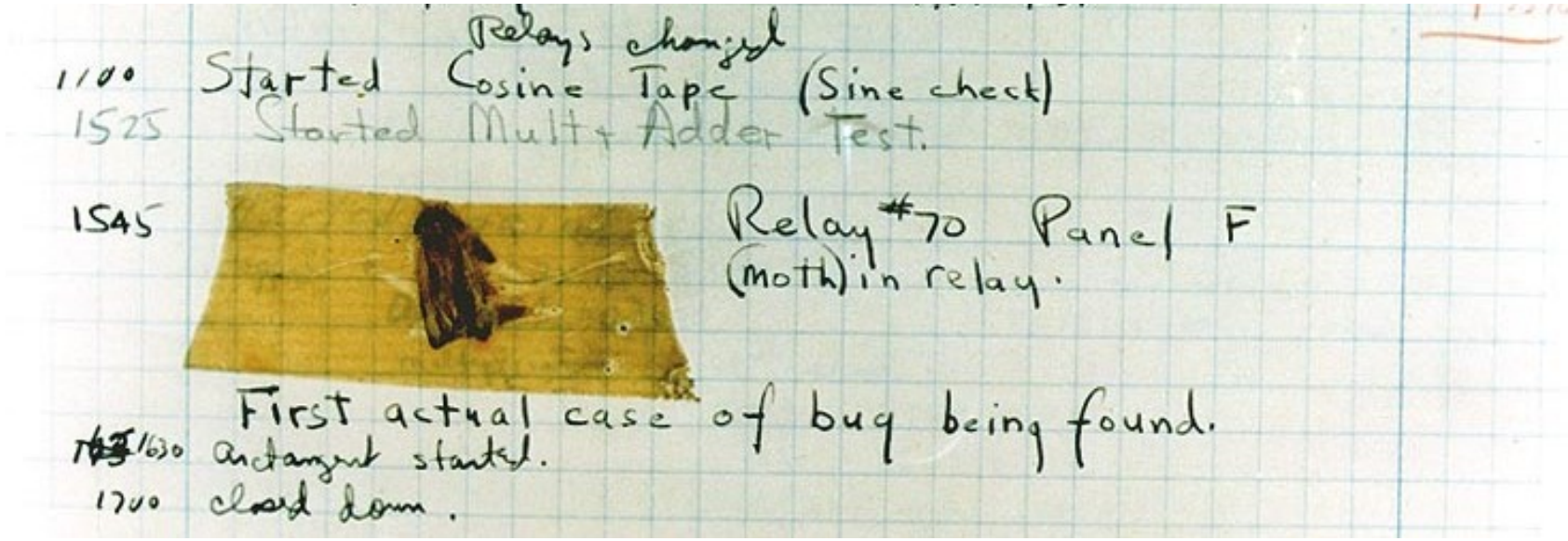
Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

Debugging

Devin J. Pohly <djpohly@cse.psu.edu>

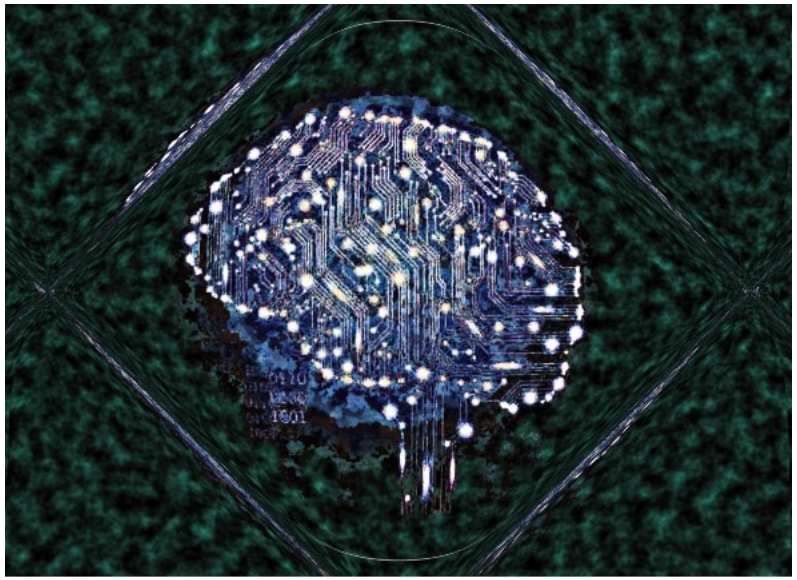
Debugging

- Often the most complicated and time-consuming part of developing a program is *debugging*
  - Figuring out where and why your program diverges from your idea of what the code should be doing
  - Confirming that your program is doing what you expect
  - Finding and fixing bugs…



"Your brain and `printf`"

- One way to debug is to print out the values of variables and memory at critical points
  - e.g., `printf("myvar=%d\n", myvar);`
  - "The two oldest and most useful debugging aids are (1) your brain, (2) `printf`. Use them!" – Muli Ben-Yehuda, Linux kernel hacker



Logging

- Logging provides a more sophisticated interface than simply printing to the screen or writing to a file
  - Turning on and off "log levels"
  - Example levels: output, error, warning, info, debug, trace
  - Unix systems often provide system-wide logging utilities
    - Logfiles in `/var/log`

Assertions

- An *assertion* is a statement that will abort the program if a given condition is not true
  - Checks your assumptions about input or logic
  - Great for preconditions and postconditions
- Usage:

  `assert(condition);`
  - Part of the C standard, found in `<assert.h>`

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

long factorial(int i)
{
    // Precondition: factorial is not
    // defined for negative integers
    assert(i >= 0);

    if (i == 1) {
        return 1;
    }
    return i * factorial(i - 1);
}

int main(int argc, char **argv)
{
    printf("5! = %ld\n", factorial(5));
    printf("%s! = %ld\n", argv[1],
            factorial(atoi(argv[1])));
    return 0;
}
```

The debugger

PENNSTATE 1855

- A *debugger* is a program that controls the execution of a program, allowing you to:
  - Manipulate the environment that the program will run in
  - Start the program, or attach to an already-started process
  - Pause the program for inspection at a certain point or under specified conditions
  - Step through your program one line (or machine instruction) at a time
  - Examine the state of your program while it is paused
  - Change the state of your program and then allow it to resume execution
- In Unix/Linux environments, the debugger used most often is GDB (the GNU Debugger)

Debug symbols

- To make debugging easier, pass the compiler the `-g` option when building to insert "debug symbols"
  - Add to CFLAGS, either permanently in the Makefile or temporarily in the `make` command:

```
make clean; make CFLAGS=-g
```

    - Need to make clean and rebuild since changing flags doesn't change prerequisites (`make` will say "up-to-date")
  - Generates lots of debugging info, such as variable names and source code lines
  - Executable will be bigger, but symbols are extremely useful when debugging

Starting GDB

- Run the debugger by passing the program file to GDB:

  `gdb program`

- This is an *interactive* terminal-based debugger
  - Based on typed commands
  - Get command help by typing `help command`, or just `help` for a list of topics

Starting GDB

- Note that invoking the debugger does not start the program
  - Loads the program and debug symbols
  - Gives you control at the GDB prompt
  - To quit, use the `quit` command

```
$ gdb fact
GNU gdb (GDB) 7.6.2
Copyright (C) 2013 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-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/djpohly/fact...done.
(gdb)
```

Looking at code

- While in the debugger, you can view parts of the source code using the `list` command
  - Shows 10 lines around the current statement by default
  - You can specify a line number or function name to list

```
(gdb) list factorial
1   #include <stdio.h>
2   #include <assert.h>
3
4   long factorial(int i)
5   {
6       // Precondition: factorial is not
7       // defined for negative integers
8       assert(i >= 0);
9
10      if (i == 1) {
(gdb)
```

Repeating commands

- You can often repeat the last command by pressing Enter again
  - This works for `list` by showing the next 10 lines
  - Works for many other commands too

```
(gdb) list factorial
1   #include <stdio.h>
2   #include <assert.h>
3
4   long factorial(int i)
5   {
6       // Precondition: factorial is not
7       // defined for negative integers
8       assert(i >= 0);
9
10      if (i == 1) {
(gdb) <Enter>
11          return 1;
12      }
13      return i * factorial(i - 1);
14  }
15
16  int main(int argc, char **argv)
17  {
18      printf("5! = %ld\n", factorial(5));
19      printf("%s! = %ld\n", argv[1],
20              factorial(atoi(argv[1])));
(gdb)
```

Running the program

- Once you load the program, you can start it by using the `run` command
  - If you have arguments to pass to the program or redirections for stdin/stdout, simply add them to the run command

```
(gdb) run 8
Starting program: /home/djpohly/fact 8
5! = 120
8! = 40320
[Inferior 1 (process 12477) exited normally]
(gdb)
```

Pausing the program

- The debugger will automatically pause the program and take control if it receives a fatal signal:
  - The program segfaults (signal SIGSEGV)
  - The program aborts, such as when an assertion fails (signal SIGABRT)
  - You interrupt the program with Ctrl-C (signal SIGINT)
- You can resume execution with the `continue` command

```
(gdb) run 0
Starting program: /home/djpohly/fact 0
5! = 120
fact: fact.c:8: factorial: Assertion `i >= 0' failed.

Program received signal SIGABRT, Aborted.
0x00007ffff7a68369 in raise () from /usr/lib/libc.so.6
(gdb) continue
Continuing.

Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
(gdb)
```

Breakpoints

- A *breakpoint* allows you to manually specify a place in your code where the debugger should pause
  - Breakpoints are set using the `break` command:

```
break funcname
break linenum
```

  - Each breakpoint is assigned an ID so you can refer to it later
- You can delete a breakpoint by using the delete command:

```
delete id
```

Breakpoints

```
(gdb) list 10,12
10      if (i == 1) {
11          return 1;
12      }
(gdb) break 11
Breakpoint 1 at 0x4005be: file fact.c, line 11.
(gdb) run 8
Starting program: /home/djpohly/fact 8

Breakpoint 1, factorial (i=1) at fact.c:11
11          return 1;
(gdb) continue
Continuing.
5! = 120

Breakpoint 1, factorial (i=1) at fact.c:11
11          return 1;
(gdb) continue
Continuing.
8! = 40320
[Inferior 1 (process 17007) exited normally]
(gdb)
```

Conditional breakpoints

- A *conditional breakpoint* is a point where you want to pause the program only if a certain condition holds
  - Specified by adding an "if" clause to the break command:

`break location if condition`

```
(gdb) break factorial if i < 0
Breakpoint 1 at 0x400599: file fact.c, line 8.
(gdb) r 0
Starting program: /home/djpohly/fact 0
5! = 120

Breakpoint 1, factorial (i=-1) at fact.c:8
8       assert(i >= 0);
(gdb)
```

Listing breakpoints

- If you want to see the current breakpoints, use the `info breakpoints` command
- The `info` command allows you to see lots of other information about the state of your program
  - Try `help info` if you feel adventurous…

```
(gdb) break factorial if i < 0
Breakpoint 1 at 0x400599: file fact.c, line 8.
(gdb) break 11
Breakpoint 2 at 0x4005be: file fact.c, line 11.
(gdb) break 21
Breakpoint 3 at 0x40064d: file fact.c, line 21.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400599 in factorial at fact.c:8
    stop only if i < 0
2       breakpoint     keep y   0x00000000004005be in factorial at fact.c:11
3       breakpoint     keep y   0x000000000040064d in main at fact.c:21
(gdb)
```

Examining the stack

- You can see exactly where the program is currently executing by using the `where` command, which prints a list of stack frames and corresponding line numbers
  - The official name of this command is `backtrace`, but the `where` alias is easy to remember and does the same thing.

```
(gdb) break 11
Breakpoint 1 at 0x4005be: file fact.c, line 11.
(gdb) run 8
Starting program: /home/djpohly/fact 8

Breakpoint 1, factorial (i=1) at fact.c:11
11          return 1;
(gdb) where
#0  factorial (i=1) at fact.c:11
#1  0x00000000004005d8 in factorial (i=2) at fact.c:13
#2  0x00000000004005d8 in factorial (i=3) at fact.c:13
#3  0x00000000004005d8 in factorial (i=4) at fact.c:13
#4  0x00000000004005d8 in factorial (i=5) at fact.c:13
#5  0x00000000004005fc in main (argc=2, argv=0x7fffffffe7b8) at fact.c:18
(gdb)
```

Navigating the stack

- You can move up and down a stack frame at a time using the `up` and `down` commands
  - You can also use the `frame` command to go directly to a specific stack frame

```
(gdb) where
#0  factorial (i=1) at fact.c:11
#1  0x00000000004005d8 in factorial (i=2) at fact.c:13
#2  0x00000000004005d8 in factorial (i=3) at fact.c:13
#3  0x00000000004005d8 in factorial (i=4) at fact.c:13
#4  0x00000000004005d8 in factorial (i=5) at fact.c:13
#5  0x00000000004005fc in main (argc=2, argv=0x7fffffffe7b8) at fact.c:18
(gdb) up
#1  0x00000000004005d8 in factorial (i=2) at fact.c:13
13      return i * factorial(i - 1);
(gdb) down
#0  factorial (i=1) at fact.c:11
11          return 1;
(gdb) frame 5
#5  0x00000000004005fc in main (argc=2, argv=0x7fffffffe7b8) at fact.c:18
18      printf("5! = %ld\n", factorial(5));
(gdb)
```

Printing variables

- At any point in the debug session, you can print the value of a variable using the `print` command:

`print var`

`print/fmt var`

  - `fmt` is an optional single character that specifies the output format: **d**ecimal, **u**nsigned, he**x**, bi**t**s (binary), **o**ctal, **a**ddress, **f**loat, **c**har, **s**tring, or **i**nstruction

```
(gdb) print argv
$3 = (char **) 0x7fffffffe7b8
(gdb) print/d argv
$4 = 140737488349112
(gdb) print *argv
$5 = 0x7fffffffeac5 "/home/djpohly/fact"
(gdb)
```

Examining memory

- You can also examine raw memory regions using the `x` command:

```
x address
x/fmt address
```

  - Here the `fmt` parameter has three parts, all of which are optional:
    - First, a repeat count: how many things to print?
    - Second, a format letter: how to print each thing? These are the same as for the `print` command.
    - Third, a size letter: how big is a thing? Options are **b**yte, **h**alfword (2 bytes), **w**ord (4 bytes), or **g**iant (8 bytes).
  - Useful for pointers and arrays

Examining memory

- For example, `argv` is an array of strings in memory, and the length of the array is `argc`
  - So we can first print the value of `argc`...
  - ...then fill that in as the repeat count, with `s` for the format letter, and voila!

```
(gdb) p argc
$6 = 2
(gdb) x/2s *argv
0x7fffffffeac5:  "/home/djpohly/fact"
0x7fffffffead8:  "8"
(gdb)
```

Walking the program

- There are several ways to advance the program manually in GDB
  - `next`: walks the program forward one full statement to the next line of code (VS "step over")
  - `step`: walks the program forward one statement but "steps into" a function if it encounters one (VS "step into")
  - `finish`: continues until the current stack frame finishes, i.e., until the function returns (VS "step out")
  - Look at `help running` for a full list

OK, who's next?

Current line

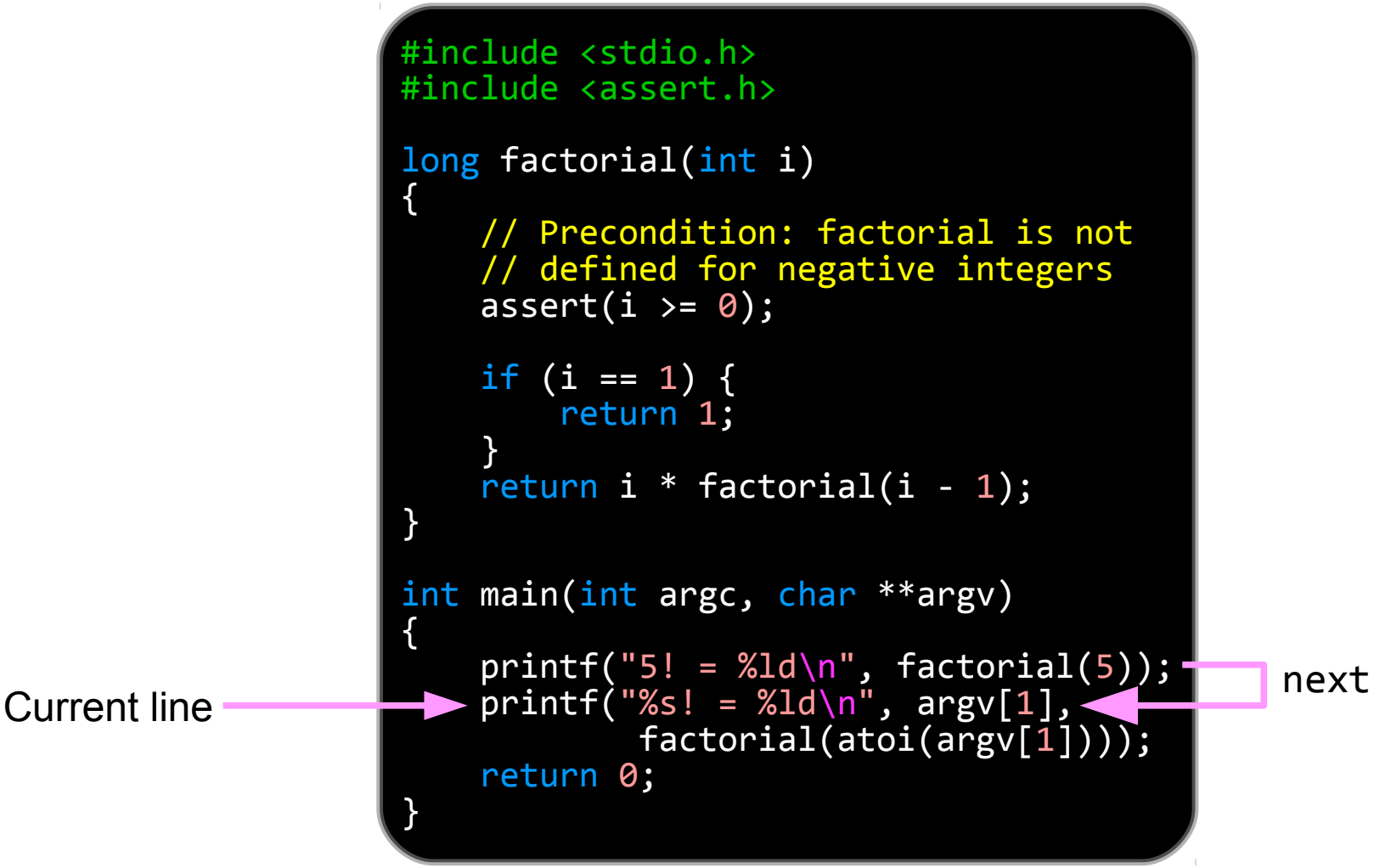
```c
#include <stdio.h>
#include <assert.h>

long factorial(int i)
{
    // Precondition: factorial is not
    // defined for negative integers
    assert(i >= 0);

    if (i == 1) {
        return 1;
    }
    return i * factorial(i - 1);
}

int main(int argc, char **argv)
{
    printf("5! = %ld\n", factorial(5));
    printf("%s! = %ld\n", argv[1],
           factorial(atoi(argv[1])));
    return 0;
}
```

OK, who's next?


#include <stdio.h>
#include <assert.h>
long factorial(int i)
{
// Precondition: factorial is not
// defined for negative integers
assert(i >= 0);
if (i == 1) {
return 1;
}
return i * factorial(i - 1);
}
int main(int argc, char **argv)
{
printf("5! = %ld\n", factorial(5));
printf("%s! = %ld\n", argv[1],
factorial(atoi(argv[1])));
return 0;
}
Current line
next

One step at a time

PENNSTATE 1855

Current line

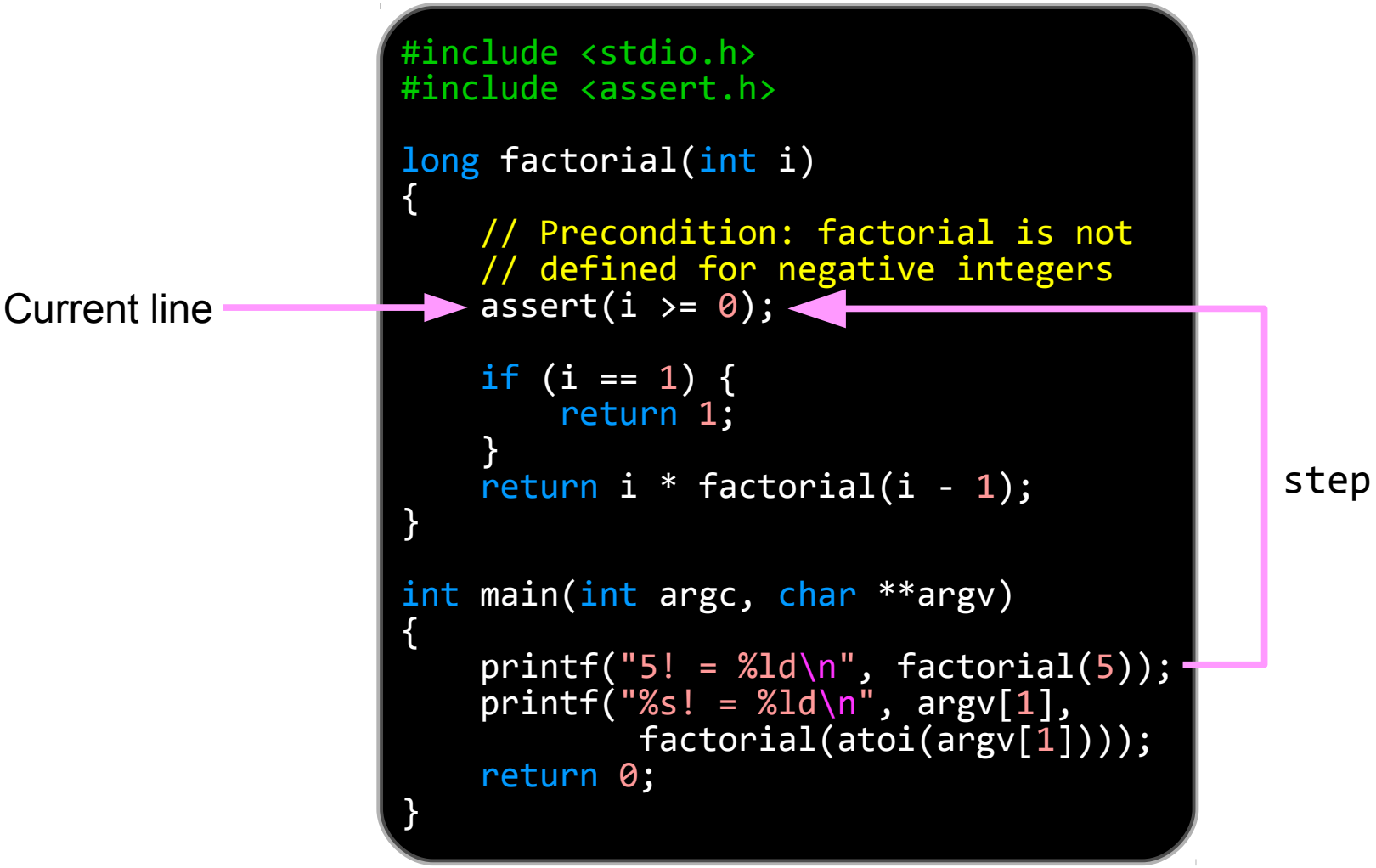
```c
#include <stdio.h>
#include <assert.h>

long factorial(int i)
{
    // Precondition: factorial is not
    // defined for negative integers
    assert(i >= 0);

    if (i == 1) {
        return 1;
    }
    return i * factorial(i - 1);
}

int main(int argc, char **argv)
{
    printf("5! = %ld\n", factorial(5));
    printf("%s! = %ld\n", argv[1],
           factorial(atoi(argv[1])));
    return 0;
}
```

One step at a time



finish him!!

Current line

Called here

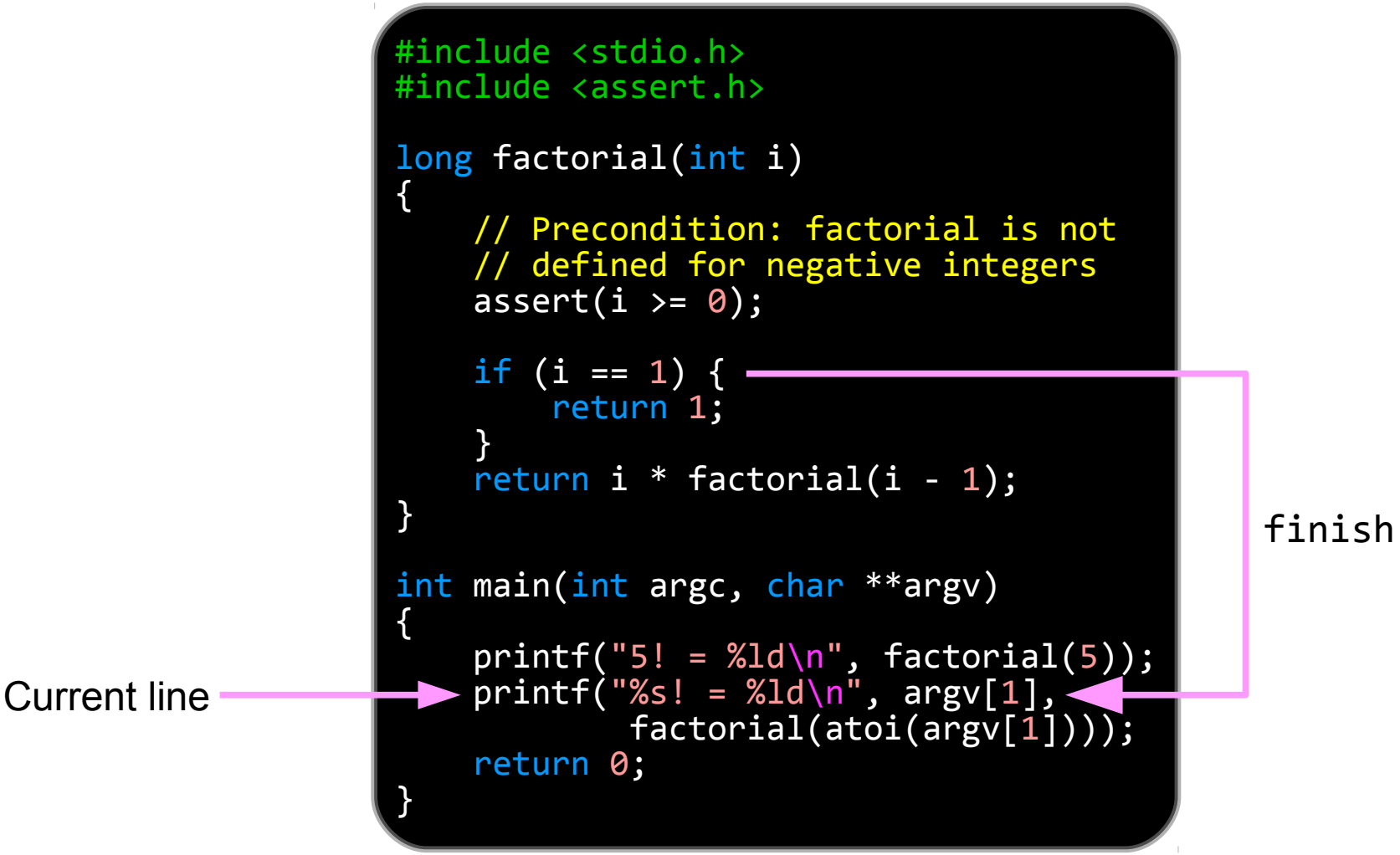
```c
#include <stdio.h>
#include <assert.h>

long factorial(int i)
{
    // Precondition: factorial is not
    // defined for negative integers
    assert(i >= 0);

    if (i == 1) {
        return 1;
    }
    return i * factorial(i - 1);
}

int main(int argc, char **argv)
{
    printf("5! = %ld\n", factorial(5));
    printf("%s! = %ld\n", argv[1],
           factorial(atoi(argv[1])));
    return 0;
}
```

finish him!!


#include <stdio.h>
#include <assert.h>
long factorial(int i)
{
// Precondition: factorial is not
// defined for negative integers
assert(i >= 0);
if (i == 1) {
return 1;
}
return i * factorial(i - 1);
}
int main(int argc, char **argv)
{
printf("5! = %ld\n", factorial(5));
printf("%s! = %ld\n", argv[1],
factorial(atoi(argv[1])));
return 0;
}
Current line
finish

Abbreviating commands

- Almost every GDB command we learned today can be abbreviated:
  - `q` = `quit`
  - `h` = `help`
  - `l` = `list`
  - `r` = `run`
  - `c` = `continue`
  - `b` = `break`
  - `d` = `delete`
  - `i` = `info`
  - `bt` = `backtrace`
  - `f` = `frame`
  - `p` = `print`
  - `n` = `next`
  - `s` = `step`
  - `fin` = `finish`

Putting it all together

```
$ gdb fact
Reading symbols from /home/djpohly/fact...done.
(gdb) r 0
Starting program: /home/djpohly/fact 0
5! = 120
fact: fact.c:8: factorial: Assertion `i >= 0' failed.

Program received signal SIGABRT, Aborted.
0x00007ffff7a68369 in raise () from /usr/lib/libc.so.6
(gdb) bt
#0  0x00007ffff7a68369 in raise () from /usr/lib/libc.so.6
#1  0x00007ffff7a69768 in abort () from /usr/lib/libc.so.6
#2  0x00007ffff7a61456 in __assert_fail_base () from /usr/lib/libc.so.6
#3  0x00007ffff7a61502 in __assert_fail () from /usr/lib/libc.so.6
#4  0x00000000004005b8 in factorial (i=-1) at fact.c:8
#5  0x00000000004005d8 in factorial (i=0) at fact.c:13
#6  0x000000000040062d in main (argc=2, argv=0x7fffffffe7b8) at fact.c:19
(gdb) f 4
#4  0x00000000004005b8 in factorial (i=-1) at fact.c:8
8       assert(i >= 0);
(gdb) p i
$1 = -1
(gdb) up
#5  0x00000000004005d8 in factorial (i=0) at fact.c:13
13      return i * factorial(i - 1);
(gdb) p i
$2 = 0
```

Putting it all together

```
(gdb) c
Continuing.

Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
(gdb) l main
12          }
13          return i * factorial(i - 1);
14      }
15
16      int main(int argc, char **argv)
17      {
18          printf("5! = %ld\n", factorial(5));
19          printf("%s! = %ld\n", argv[1],
20                  factorial(atoi(argv[1])));
21          return 0;
(gdb) b 19
Breakpoint 1 at 0x400616: file fact.c, line 19.
(gdb) r 0
Starting program: /home/djpohly/fact 0
5! = 120

Breakpoint 1, main (argc=2, argv=0x7fffffffe7b8) at fact.c:19
19          printf("%s! = %ld\n", argv[1],
```

Putting it all together

```
(gdb) s
factorial (i=0) at fact.c:8
8       assert(i >= 0);
(gdb) bt
#0  factorial (i=0) at fact.c:8
#1  0x000000000040062d in main (argc=2, argv=0x7fffffffe7b8) at fact.c:19
(gdb) s
10      if (i == 1) {
(gdb)
13      return i * factorial(i - 1);
(gdb)
factorial (i=-1) at fact.c:8
8       assert(i >= 0);
(gdb) bt
#0  factorial (i=-1) at fact.c:8
#1  0x00000000004005d8 in factorial (i=0) at fact.c:13
#2  0x000000000040062d in main (argc=2, argv=0x7fffffffe7b8) at fact.c:19
(gdb) q
A debugging session is active.

    Inferior 1 [process 3683] will be killed.

Quit anyway? (y or n) y
$
```

On Friday

- No Hands-On Friday this week
  - Assignment review – bring your questions
- I will be a bit less accessible over break, so please use this opportunity wisely!