Debugging

9/9

0800 Arctan started

1000 " stopped - arctan ✓

13 wc (032) MP - MC ~~2.1304~~

(033) PRO 2 2.1304

correct 2.1306

Relays 6-2 in 033 failed

In relay

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 

Relay #70 Panel F

(moth) in relay.

First actual case of bug being found.

~~1700~~ 1630 Arctangent started.

1700 closed down.

Program debugging

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

- **One way to debug is to print out the values of variables and memory at different points**
  - e.g (“My variable value is %d”, myvar);
  - But this is very limited

Assert

- **assert()** is a function provided by C that allows you to place statements in code that must always be true, where the process Aborts if it is not

- This is a great tool for checking to make sure your assumptions about inputs/logic are always true
- Syntax: assert(expression);

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

int factorial( int i ) {

        assert( i>=0 ); // Breakpoint here
        if ( i == 1 ) {
                return( 1 );
        }
        return( factorial(i-1)*i );
}
```

```
cs257@cs257-VirtualBox:~/Desktop/debug$ ./debug_test
24Argumentts (1), last arg [./debug_test]
Factorial : 5! = 120
debug_test: debug_test.c:6: factorial: Assertion `i>=0' failed.
Aborted (core dumped)
cs257@cs257-VirtualBox:~/Desktop/debug$
```

Debugger

- **A *debugger* is a program that runs your program within a controlled environment:**
  - Control aspects of the environment that your program will run in.
  - Start your program, or connect up to an already-started process
  - Make your program stop for inspection or under specified conditions.
  - Step through your program one line at a time, or one machine instruction at a time.
  - Inspect the state of your program once it has stopped.
  - 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)**

Compiling for debugging

- Option –g produces debugging information to be used in gdb

gcc -Wall -g debug_test.c -o debug_test

- If you forget –g option when compiling, gdb will still work, but will be missing important information for debugging, such as line numbers.

- -ggdb is another option to produce useful information for debugging

Debugger

- You run the debugger by passing the program to gdb

$gdb [program name]

- This is an *interactive* terminal-based debugger
- Invoking the debugger does not start the program, but simply drops you into the gdb environment.

```
cs257@cs257-VirtualBox:~/Desktop/debug$ gdb debug_test
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 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 "i686-linux-gnu".
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 debug_test...done.
(gdb)
```

Running the program

- Once you enter the program, you must start running it, using the run command

```
(gdb) run
Starting program: /home/cs257/Desktop/debug/debug_test
Argumentts (1), last arg [/home/cs257/Desktop/debug/debug_test]
Factorial : 5! = 120
[Inferior 1 (process 2973) exited normally]
(gdb)
```

- If you have arguments to pass to the program, simply add them to the run command line

```
(gdb) run hello
Starting program: /home/cs257/Desktop/debug/debug_test hello
Argumentts (2), last arg [hello]
Factorial : 5! = 120
[Inferior 1 (process 2982) exited normally]
(gdb)
```

Looking at code

- **While in the debugger you often want to look at regions of code, so use the list command**
  - Shows 10 lines at a time
  - You can specify a line number (in the current file), or specify a function name
  - Most commands are aliased with single character (l for list)

```
(gdb) list
1       #include <stdio.h>
2       #include <assert.h>
3
4       int factorial( int i ) {
5
6       assert( i>=0 ); // Breakpoint h
7       if ( i == 1 ) {
8               return( 1 );
9       }
10      return( factorial(i-1)*i );
(gdb)
```

```
(gdb) l main
8               return( 1 );
9       }
10      return( factorial(i-1)*i );
11      }
12
13      int main( int argc, char *argv[] ) {
14
15      if ( argc > 0 ) {
16              printf( "Argumentts (%d), last arg [%s]\n",
17              argc, argv[argc-1] ); // Breakpoint here
(gdb)
```

Breakpoints

- **A *breakpoint* is a position in the code you wish for the debugger to stop and wait for your commands**
  - Breakpoints are set using the break (b) command

```
break [function_name | line_number]
```

  - Each one is assigned a number you can reference later
  - You can delete the breakpoint by using the delete (d)

```
delete [breakpoint_number]
```

```
(gdb) b factorial
Breakpoint 1 at 0x40056e: file debug_test.c, line 6.
(gdb) b 16
Breakpoint 2 at 0x4005ef: file debug_test.c, line 16.
(gdb) delete 1
(gdb) d 2
(gdb)
```

Conditional Breakpoints

- A conditional *breakpoint* is a point where you want the debugger pause only if the condition holds
- Breakpoints are set conditional using the **cond** command

```
cond [breakpoint_number] (expr)
```

```
(gdb) b 6
Breakpoint 3 at 0x40056e: file debug_test.c, line 6.
(gdb) cond 1 i<=1
No breakpoint number 1.
(gdb) cond 3 i<=1
(gdb) r
Starting program: /home/cs257/Desktop/debug/debug_test hello
Argumentts (2), last arg [hello]

Breakpoint 3, factorial (i=1) at debug_test.c:6
6       assert( i>=0 ); // Breakpoint here
(gdb) c
Continuing.
Factorial : 5! = 120
[Inferior 1 (process 2988) exited normally]
(gdb)
```

Conditional Breakpoints

- **Alternately, breakpoints can be set with if expression**

```
b [line | function] if (expr)
```

```
(gdb) l 6
1       #include <stdio.h>
2       #include <assert.h>
3
4       int factorial( int i ) {
5
6           assert( i>=0 ); // Breakpoint here
7           if ( i == 1 )
8           {
9                   return( 1 );
10          }
(gdb) b 7 if i==1
Breakpoint 1 at 0x592: file debug_test.c, line 7.
(gdb) r
Starting program: /home/cs257/Desktop/debug/debug_test

Breakpoint 1, factorial (i=1) at debug_test.c:7
7           if ( i == 1 )
(gdb) p i
$1 = 1
```

Seeing breakpoints

- If you want to see your breakpoints use the info breakpoints command

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
3       breakpoint     keep y   0x0040056e in factorial at debug_test.c:6
        stop only if i<=1
        breakpoint already hit 1 time
(gdb)
```

- The info command allows you see lots of information about the state of your environment and program

```
(gdb) help info
Generic command for showing things about the program being debugged.

List of info subcommands:

info address -- Describe where symbol SYM is stored
info all-registers -- List of all registers and their contents
info args -- Argument variables of current stack frame
```

Saving breakpoints

- You can save breakpoints to a file for later use using save command

```
(gdb) b factorial
Note: breakpoint 3 also set at pc 0x40056e.
Breakpoint 4 at 0x40056e: file debug_test.c, line 6.
(gdb) b 16
Breakpoint 5 at 0x4005ef: file debug_test.c, line 16.
(gdb) save breakpoint breakpoints.txt
Saved to file 'breakpoints.txt'.
(gdb) quit
```

- You can load the breakpoints from a file later using source command

```
(gdb) source breakpoints.txt
Breakpoint 1 at 0x56e: file debug_test.c, line 6.
Breakpoint 2 at 0x56e: file debug_test.c, line 6.
Breakpoint 3 at 0x5ef: file debug_test.c, line 16.
(gdb)
```

Watchpoints

- Watchpoints (also known as a data breakpoint) stop execution whenever the value of an variables changes, without having to predict a particular place where this may happen.
- The simplest form is simply waiting for a variable to change
- Variable should belong to current stack

```
(gdb) l 15
10          }
11          return( factorial(i-1)*i );
12      }
13
14      int main( int argc, char *argv[] ) {
15          int z;
16          z = 0;
17          z  = factorial (4);
18          printf("%d", z);
19          if ( argc > 0 ) {
(gdb) b 15
Breakpoint 1 at 0x5da: file debug_test.c, line 15.
(gdb) r
Starting program: /home/cs257/Desktop/debug/debug_tes

Breakpoint 1, main (argc=1, argv=0xbffff234) at debug
16          z = 0;
(gdb) watch z
Hardware watchpoint 2: z
(gdb) c
Continuing.
```

```
Hardware watchpoint 2: z

Old value = 4196027
New value = 0
main (argc=1, argv=0xbffff234) at debug_test.c:17
17          z  = factorial (4);
(gdb) c
Continuing.

Hardware watchpoint 2: z

Old value = 0
New value = 24
main (argc=1, argv=0xbffff234) at debug_test.c:18
18          printf("%d", z);
(gdb)
```

Examining the stack

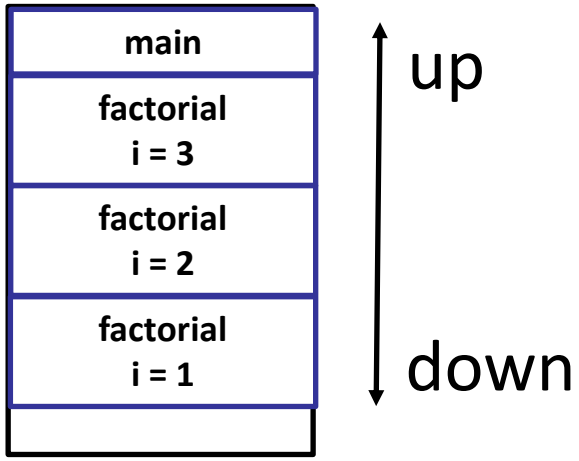
- You can always tell where you are in the program by using the **where** command, which gives you a stack and the specific line number you are on

```
(gdb) where
#0  factorial (i=1) at debug_test.c:6
#1  0x004005ae in factorial (i=2) at debug_test.c:10
#2  0x004005ae in factorial (i=3) at debug_test.c:10
#3  0x004005ae in factorial (i=4) at debug_test.c:10
#4  0x004005eb in main (argc=1, argv=0xbffff234) at debug_test.c:15
(gdb)
```

Climbing and descending the stack

- You can move up and down the stack and see variables by using the up and down commands

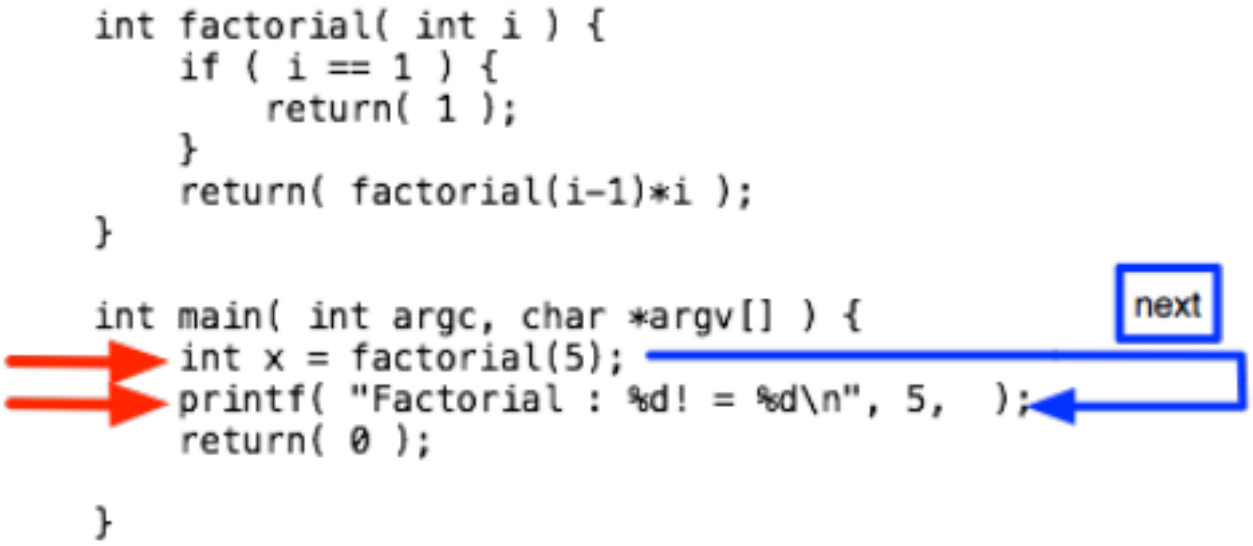
```
(gdb) p i
$1 = 1
(gdb) up
#1  0x004005ae in factorial (i=2) at debug_test.c:10
10      return( factorial(i-1)*i );
(gdb) p i
$2 = 2
(gdb) up
#2  0x004005ae in factorial (i=3) at debug_test.c:10
10      return( factorial(i-1)*i );
(gdb) p i
$3 = 3
(gdb) down
#1  0x004005ae in factorial (i=2) at debug_test.c:10
10      return( factorial(i-1)*i );
(gdb) p i
$4 = 2
```




```
(gdb) list 5
1       #include <stdio.h>
2       #include <assert.h>
3
4       int factorial( int i ) {
5
6       assert( i>=0 ); // Breakpoint here
7       if ( i == 1 ) {
8               return( 1 );
9       }
10      return( factorial(i-1)*i );
(gdb)
```

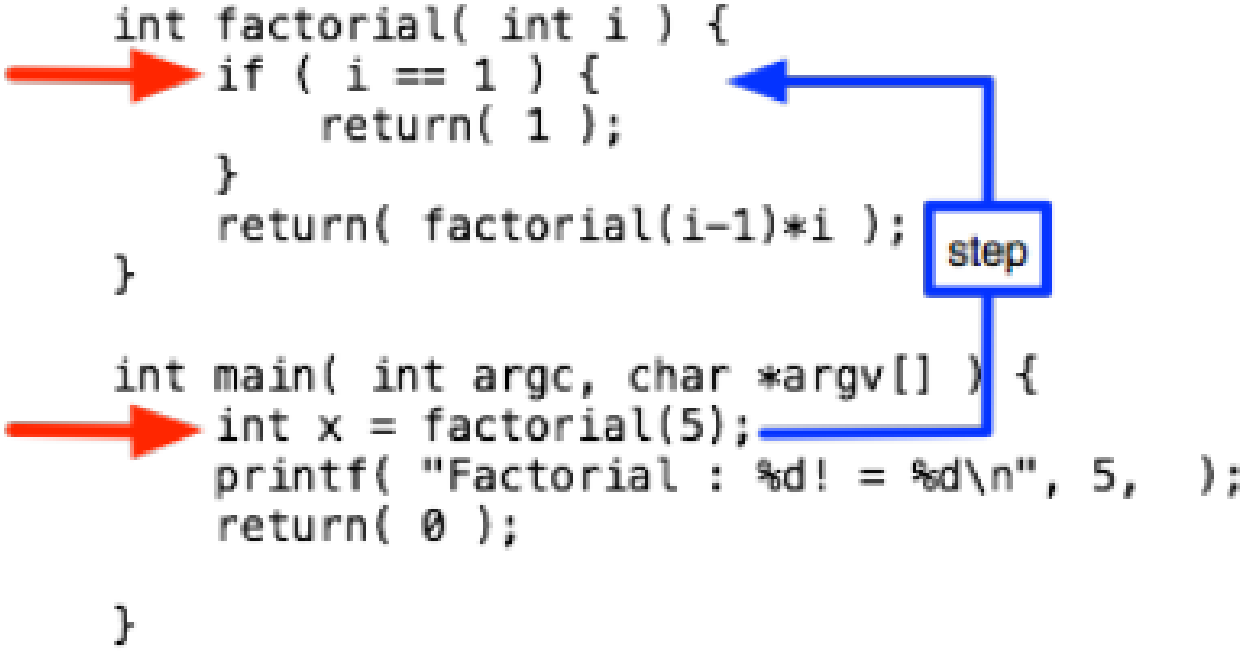
Walking the program

- **There are four ways to advance the program in gdb**
  - next (n) steps the program forward one statement, regardless of the kind of statement it is on



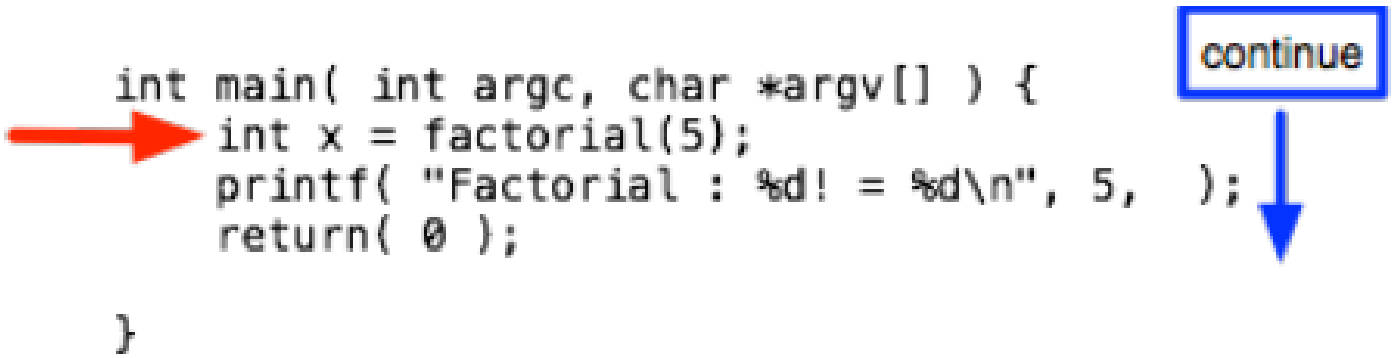
Walking the program

- **step (s) moves the program forward one statement, but "steps into" a program-defined function**



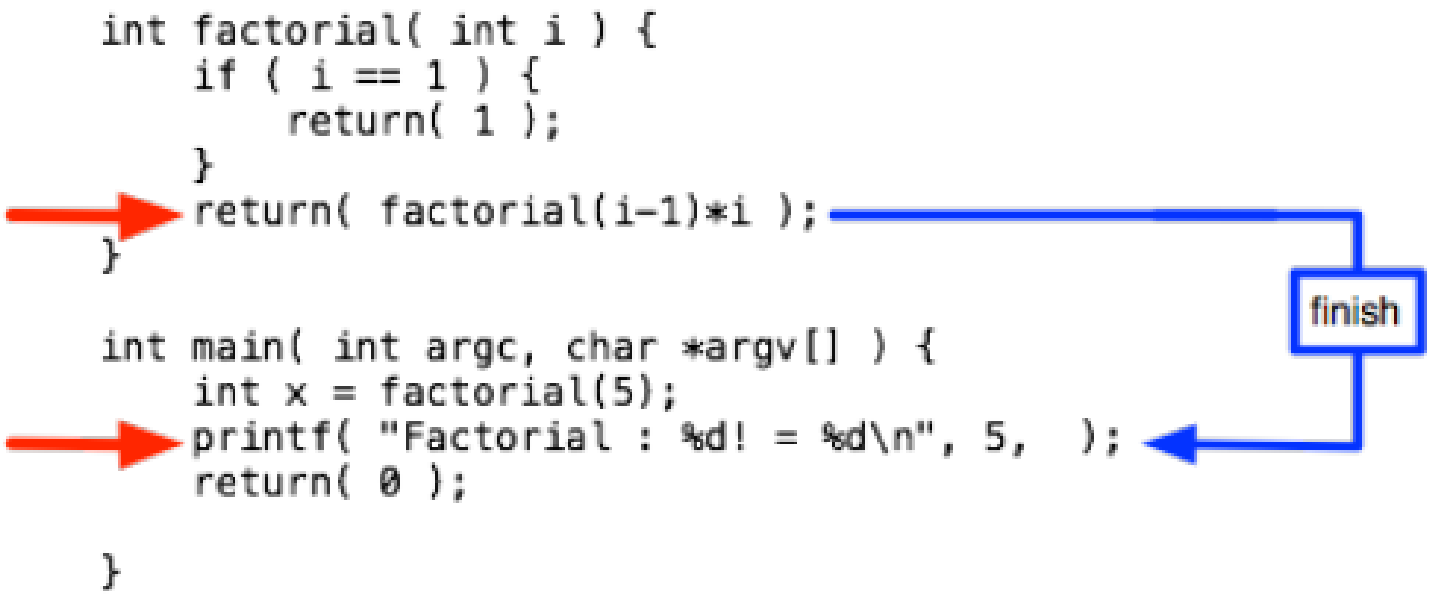
Walking the program

- **continue (c) continues running the program from that point till it terminates or hits another breakpoint**



Walking the program

- **Finish (fin)** continues until the function returns



Test Yourself

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

int factorial( int i ) {

        assert( i>=0 ); // Breakpoint here
        if ( i == 1 ) {
                return( 1 );
        }
        return( factorial(i-1)*i );
}

int main( int argc, char *argv[] ) {

        if ( argc > 0 ) {
                printf( "Arguments (%d), last arg [%s]\n",
                argc, argv[argc-1] ); // Breakpoint here
        }

        printf( "Factorial : %d! = %d\n", 5, factorial(5) );
        // factorial( -1 );
        return( 0 );
}
```