CMPSC 311 - Introduction to Systems Programming

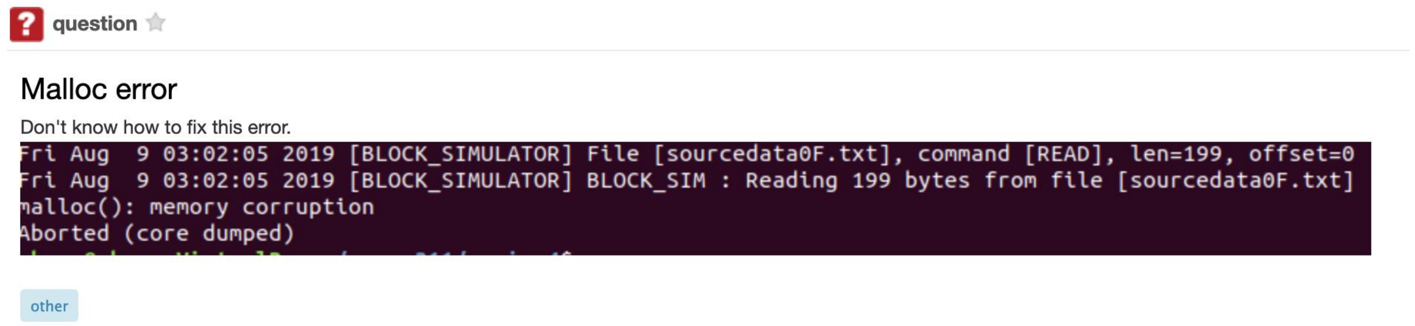
Debugging

Professors Sencun Zhu and Suman Saha

(Slides are mostly by Professors Patrick McDaniel and Abutalib Aghayev)

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 …



question

Malloc error

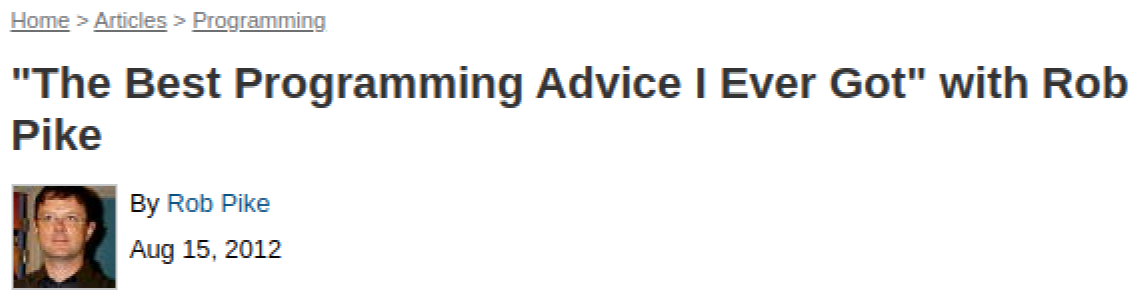
Don't know how to fix this error.

```
Fri Aug  9 03:02:05 2019 [BLOCK_SIMULATOR] File [sourcedata0F.txt], command [READ], len=199, offset=0
Fri Aug  9 03:02:05 2019 [BLOCK_SIMULATOR] BLOCK_SIM : Reading 199 bytes from file [sourcedata0F.txt]
malloc(): memory corruption
Aborted (core dumped)
```

other

Think before you debug

- When something went wrong, I'd reflexively start to dig in to the problem, examining stack traces, sticking in print statements, invoking a debugger, and so on. But Ken would just stand and think, ignoring me and the code we'd just written. After a while I noticed a pattern: Ken would often understand the problem before I would, and would suddenly announce, "I know what's wrong." He was usually correct. I realized that Ken was building a mental model of the code and when something broke it was an error in the model. By thinking about *how* that problem could happen, he'd intuit where the model was wrong or where our code must not be satisfying the model.


Home > Articles > Programming

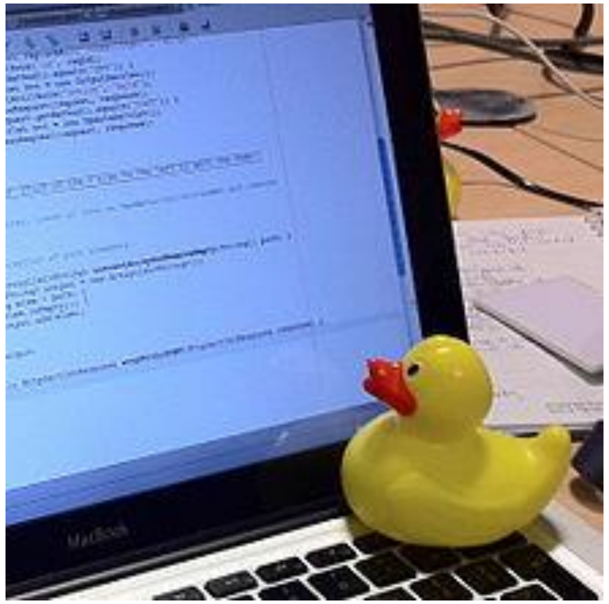
"The Best Programming Advice I Ever Got" with Rob Pike

By Rob Pike

Aug 15, 2012

Rubber duck debugging

- The name is a reference to a story in the book The Pragmatic Programmer in which a programmer would carry around a rubber duck and debug their code by forcing themself to explain it, line-by-line, to the duck.



Printing/Logging

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

Assert

- assert() is a function provided by C in which you place statements in code that must always be true, where the process aborts if it is not
  - Check to make sure your assumptions about inputs/logic are always true
  - Syntax: `assert( expression );`

```
Examples:

        assert( i>= 0 );                       // Checks value
        assert( ptr != NULL );                 // Checks non-NULL pointer
        assert( (ptr = malloc(100)) != NULL ); // Confirms malloc successful
        assert( func(10, 20) );                // Confirms function returns 0
```

Note: These are sometimes called logic or program guards.

Demonstration function

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

int factorial( int i ) {

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

int main(int argc, char** argv) {
    int i = 5;
    if (argc > 1) {
        i = atoi(argv[1]);
    }
    printf("Factorial of %d: %d\n",i , factorial(i));
    return( 0 );
}
```

Demonstration function

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

int factorial( int i ) {

    assert( i>=0 );
    if ( i <= 1 ) {
```

```
user@311:~/project# gcc -g debugging.c -o debugging
user@311:~/project# ./debugging
Factorial of 5: 120
user@311:~/project# ./debugging 4
Factorial of 4: 24
user@311:~/project# ./debugging -1
debugging: debugging.c:7: factorial: Assertion `i >= 0' failed.
Aborted
```

```c
        i = atoi(argv[1]);
    }
    printf("Factorial of %d: %d\n",i , factorial(i));
    return( 0 );
}
```

The 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) and is `lldb` up and coming (on OSX, Linux, ...)

gdb

- 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.

```
$ gdb debugging
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
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-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/mcdaniel/src/debugging/debugging...done.
(gdb)
```

gdb

- 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.

```
$ gdb debugging
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License
This i
There
and "s
This G
For bug
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/mcdaniel/src/debugging/debugging...done.
(gdb)
```

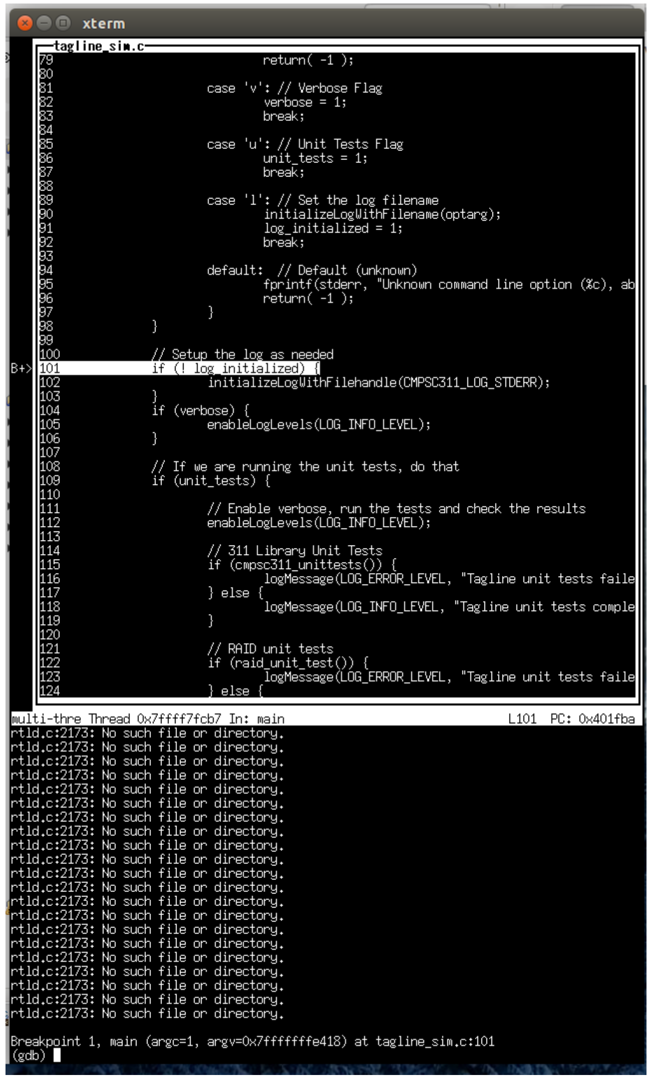
You can always get help for any command in `gdb` by typing `help [command]`

gdb with a user interface

- You can also get a simple terminal interface by starting the debugger …

```
gdb -tui
```

- This walks you through the code and makes debugging easier. The commands are the same.
- Tools like `VS Code` integrate the `gdb` function into the editor/IDE



```
xterm
tagline_sim.c
                        return( -1 );

                case 'v': // Verbose Flag
                        verbose = 1;
                        break;

                case 'u': // Unit Tests Flag
                        unit_tests = 1;
                        break;

                case 'l': // Set the log filename
                        initializeLogWithFilename(optarg);
                        log_initialized = 1;
                        break;

                default:  // Default (unknown)
                        fprintf(stderr, "Unknown command line option (%c), ab
                        return( -1 );
                }
        }

        // Setup the log as needed
B+>     if (! log_initialized) {
                initializeLogWithFilehandle(CMPSC311_LOG_STDERR);
        }
        if (verbose) {
                enableLogLevels(LOG_INFO_LEVEL);
        }

        // If we are running the unit tests, do that
        if (unit_tests) {

                // Enable verbose, run the tests and check the results
                enableLogLevels(LOG_INFO_LEVEL);

                // 311 Library Unit Tests
                if (cmpsc311_unittests()) {
                        logMessage(LOG_ERROR_LEVEL, "Tagline unit tests faile
                } else {
                        logMessage(LOG_INFO_LEVEL, "Tagline unit tests comple
                }

                // RAID unit tests
                if (raid_unit_test()) {
                        logMessage(LOG_ERROR_LEVEL, "Tagline unit tests faile
                } else {
multi-thre Thread 0x7ffff7fcb7 In: main                          L101   PC: 0x401fba
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.
rtld.c:2173: No such file or directory.

Breakpoint 1, main (argc=1, argv=0x7fffffffe418) at tagline_sim.c:101
(gdb) █
```

Running the program

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

```
(gdb) run
Starting program: /root/project/debugging
Factorial of 5: 120
 [Inferior 1 (process 149) exited normally]
(gdb)
```

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

```
(gdb) run 12
Starting program: /root/project/debugging 12
Factorial of 12: 479001600
 [Inferior 1 (process 153) exited normally]
(gdb)
```

Looking at code

- If want to look at regions of code, so use the `list` command
  - shows 10 lines at a time, centered around the target
  - you can specify a line number (in the current file),
  - or specify a function name

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

```
(gdb) l main
10                }
11                return (factorial(i - 1) * i);
12            }
13
14            int main(int argc, char** argv)
15            {
16                int i;
17                if (argc > 1) {
18                    i = atoi(argv[1]);
19                }
(gdb)
```

- Most commands are aliased with single character (l)

Breakpoints

- A breakpoint is a position in the code you wish for the debugger to stop and wait for your commands

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

  - Breakpoints are set using the break (b) command
  - Each one is assigned a number you can reference later
- You can delete the breakpoint by using the delete (d) command

```
delete [breakpoint_number]
```

```
(gdb) b factorial
Breakpoint 1 at 0x400587: file debugging.c, line 6.
(gdb) b 16
Breakpoint 2 at 0x4005db: file debugging.c, line 16.
(gdb) delete 1
(gdb) d 2
```

Conditional Breakpoints

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

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

```
(gdb) l 7
7               assert(i >= 0); // ** CHECK **
(gdb) b 7
Breakpoint 1 at 0x6e5: file debugging.c, line 7.
(gdb) cond 1 i<=1
(gdb) r
Starting program: /root/project/debugging

Breakpoint 1, factorial (i=1) at debugging.c:7
7               assert(i >= 0); // ** CHECK **
(gdb) c
Continuing.
Factorial of 5: 120
 [Inferior 1 (process 157) exited normally]
(gdb)
```

Conditional Breakpoints

- A conditional breakpoint is a point where you want the debugger only if the condition holds
  - Alternately, breakpoints can be set with `if` expression

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

```
(gdb) l 7
7               assert(i >= 0); // ** CHECK **
(gdb) b 7 if i<=1
Breakpoint 1 at 0x6e5: file debugging.c, line 7.
(gdb) r
Starting program: /root/project/debugging

Breakpoint 1, factorial (i=1) at debugging.c:7
7               assert(i >= 0); // ** CHECK **
(gdb) c
Continuing.
Factorial of 5: 120
 [Inferior 1 (process 165) exited normally]
(gdb)
```

Seeing breakpoints

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

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000000006e5 in factorial at debugging.c:7
2       breakpoint     keep y   0x000000000000075c in main at debugging.c:22
(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 use later using `save` command

```
(gdb) b factorial
Breakpoint 1 at 0x6e5: file debugging.c, line 7.
(gdb) b 22
Breakpoint 2 at 0x75c: file debugging.c, line 20.
(gdb) save breakpoints bpoints.txt
Saved to file 'bpoints.txt'.
(gdb)
```

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

```
root@a2354b724f6e:~/project# gdb debugging
(gdb) source bpoints.txt
Breakpoint 1 at 0x6e5: file debugging.c, line 7.
Breakpoint 2 at 0x75c: file debugging.c, line 20.
(gdb)
```

Watchpoints

- Watchpoints (also known as a data breakpoint) stop execution whenever the value of an variable changes, *without* a particular place where it happens.
  - The simplest form is simply waiting for a variable to change

```
(gdb) b main
Breakpoint 1 at 0x737: file debugging.c, line 17.
(gdb) run 4
Starting program: /root/project/debugging 4

Breakpoint 1, main (argc=2, argv=0x7fffffffe718) at debugging.c:17
warning: Source file is more recent than executable.
17              if (argc > 1) {
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 5
New value = 4
0x0000555555554753 in main (argc=2, argv=0x7fffffffe718) at debugging.c:18
18                  i = atoi(argv[1]);
(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 one

```
(gdb) b 7 if i<=1
Breakpoint 1 at 0x6e5: file debugging.c, line 7.
(gdb) run 4
Starting program: /root/project/debugging 4

Breakpoint 1, factorial (i=1) at debugging.c:7
7              assert(i >= 0); // ** CHECK **
(gdb) where
#0  factorial (i=1) at debugging.c:7
#1  0x0000555555554722 in factorial (i=2) at debugging.c:11
#2  0x0000555555554722 in factorial (i=3) at debugging.c:11
#3  0x0000555555554722 in factorial (i=4) at debugging.c:11
#4  0x0000555555554764 in main (argc=2, argv=0x7fffffffe718) at debugging.c:20
(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  0x0000555555554722 in factorial (i=2) at debugging.c:11
11              return (factorial(i - 1) * i);
(gdb) p i
$2 = 2
(gdb) up
#2  0x0000555555554722 in factorial (i=3) at debugging.c:11
11              return (factorial(i - 1) * i);
(gdb) p i
$3 = 3
(gdb) down
#1  0x0000555555554722 in factorial (i=2) at debugging.c:11
11              return (factorial(i - 1) * i);
(gdb) p i
$4 = 2
(gdb) down
#0  factorial (i=1) at debugging.c:7
7               assert(i >= 0); // ** CHECK **
(gdb) p i
$5 = 1
(gdb)
```

Printing variables

- At any point in the debug session can print the value of any variable you want by printing its value using

```
print[/<format>] variable
```

- Dictate the output formatted with `o`(octal), `x`(hex), `d`(decimal), `u`(unsigned decimal), `t`(binary), `f`(float), `a`(address), `i`(instruction), and `s`(string)

```
(gdb) p values
$1 = "\001\002\003\004"
(gdb) p/x values
$2 = {0x1, 0x2, 0x3, 0x4}
(gdb) p val1
$3 = 4283787007
(gdb) p/x val1
$4 = 0xff5566ff
(gdb) p val2
$5 = 2.45677996
(gdb)
```

```
int myvalues() {
    char values[] = { 0x1, 0x2, 0x3, 0x4 };
    uint32_t val1 = 0xff5566ff;
    float val2 = 2.45678;
    return 0; // breakpoint here
}
```

Examining memory

- You examine memory regions using the x command

```
x [/<num><format><size>] address
```

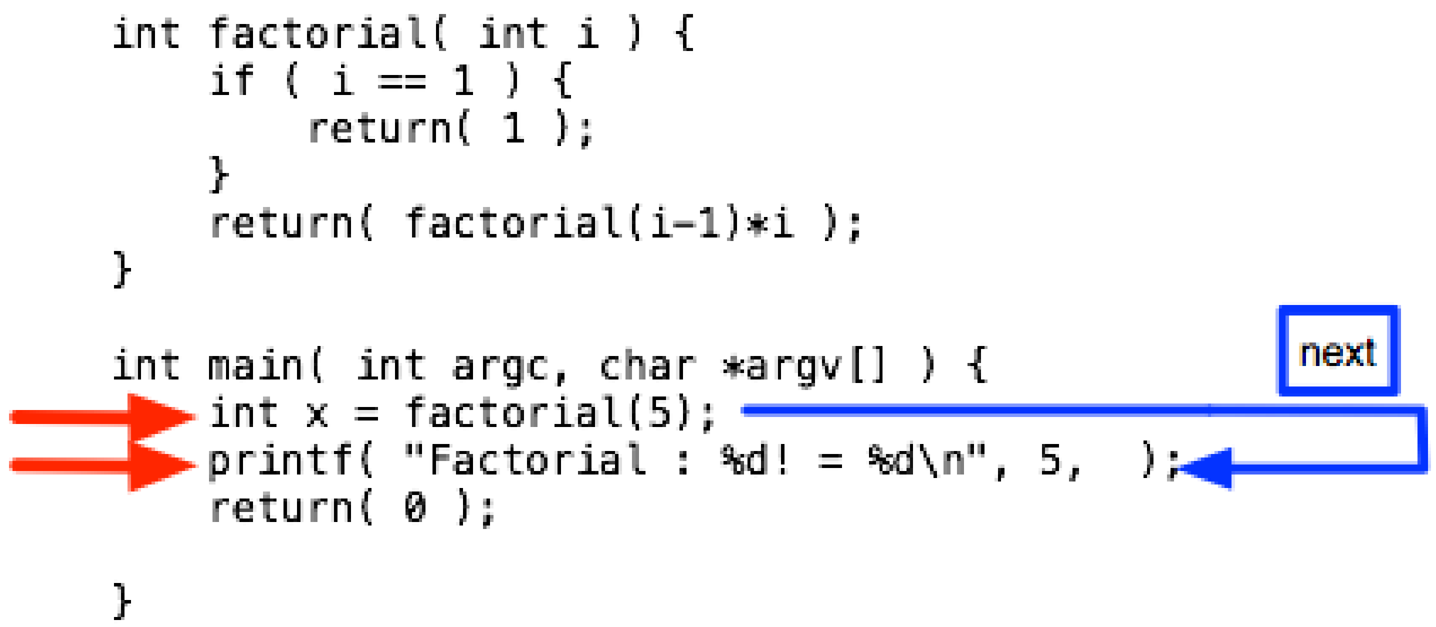
- Modify the output using a number of values formatted with `[oxdutfais]` type and size are `b`(byte), `h`(halfword), `w`(word), `g`(giant, 8 bytes).

```
(gdb) x buf
0x555555756260:     0xefefefef
(gdb) x/8xb buf
0x555555756260:     0xef      0xef      0xef      0xef      0xef      0xef      0xef      0xef
(gdb) x/xg buf
0x555555756260:     0xefefefefefefefef
(gdb) x buf
0x555555756260:     0xefefefefefefefef
(gdb) x &buf
0x7fffffffe5f8:     0x0000555555756260
(gdb)
```

```
int myexamine() {
    char *buf = NULL;
    buf = malloc( 8 );
    memset( buf, 0xef, 8 );
    return 0; // breakpoint here
}
```

Walking the program

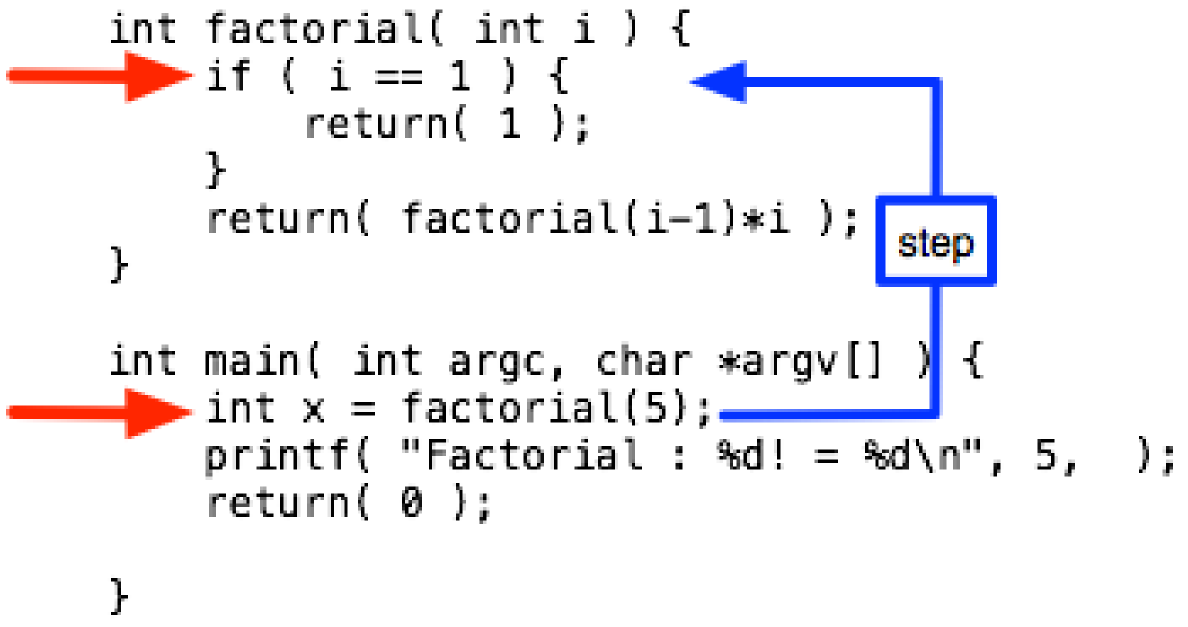
- There are four ways to advance the program in gdb
  - next (n) steps the program forward one statement



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

int main( int argc, char *argv[] ) {
    int x = factorial(5);
    printf( "Factorial : %d! = %d\n", 5,   );
    return( 0 );

}
```

Walking the program

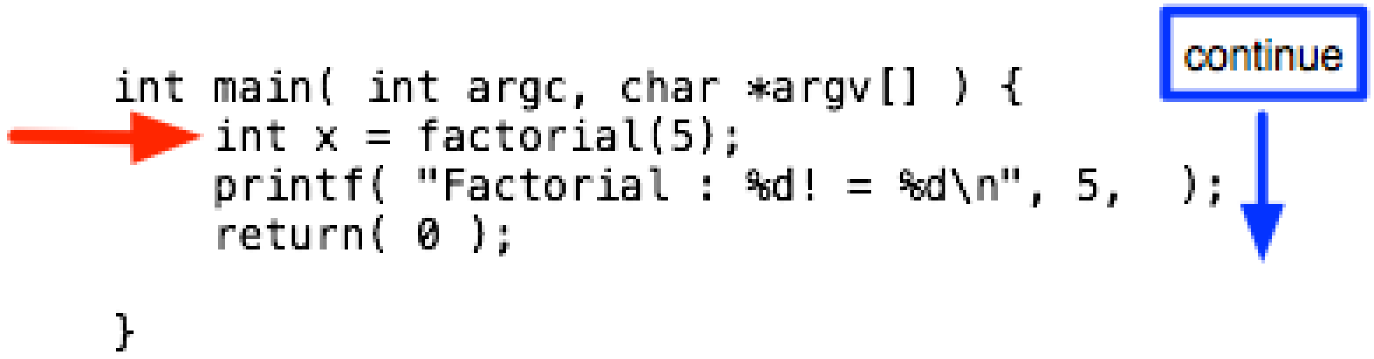
- There are four ways to advance the program in gdb
  - next (n) steps the program forward one statement
  - step (s) moves the program forward one statement, but “steps into” a function



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

int main( int argc, char *argv[] ) {
    int x = factorial(5);
    printf( "Factorial : %d! = %d\n", 5,   );
    return( 0 );

}
```

Walking the program

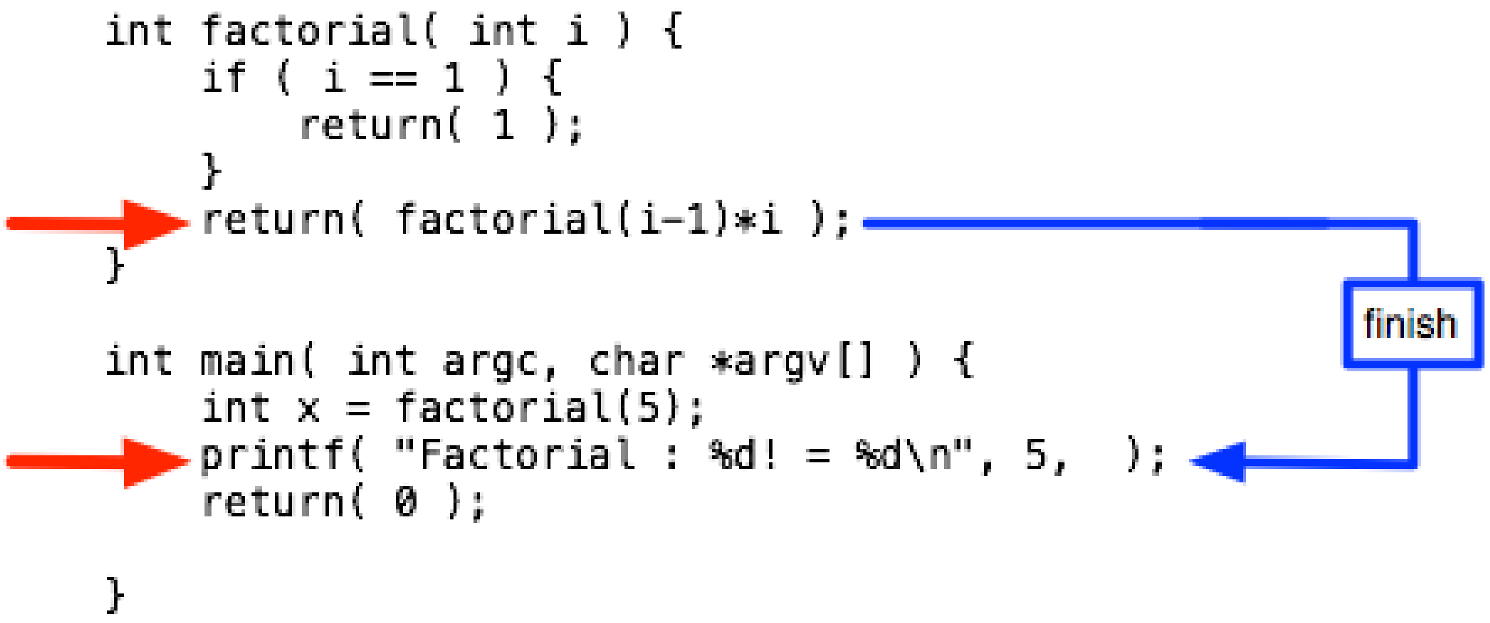
- There are four ways to advance the program in gdb
  - next (n) steps the program forward one statement
  - step (s) moves the program forward one statement, but “steps into” a function
  - continue (c) continues running the program from that point till it terminates or hits another breakpoint



```
int main( int argc, char *argv[] ) {
    int x = factorial(5);
    printf( "Factorial : %d! = %d\n", 5,   );
    return( 0 );

}
```

Walking the program

- There are four ways to advance the program in gdb
  - next (n), step (s), continue (c), … and
  - finish (fin) continues until the function returns



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

int main( int argc, char *argv[] ) {
    int x = factorial(5);
    printf( "Factorial : %d! = %d\n", 5,   );
    return( 0 );

}
```

Putting it all together

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

```
(gdb) b factorial
Breakpoint 1 at 0x6e5: file debugging.c, line 7.
(gdb) b 20
Breakpoint 2 at 0x75a: file debugging.c, line 20.
(gdb)
```

Putting it all together

```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20              printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7               assert(i >= 0);
(gdb) n
8               if (i <= 1) {
(gdb) n
11              return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```

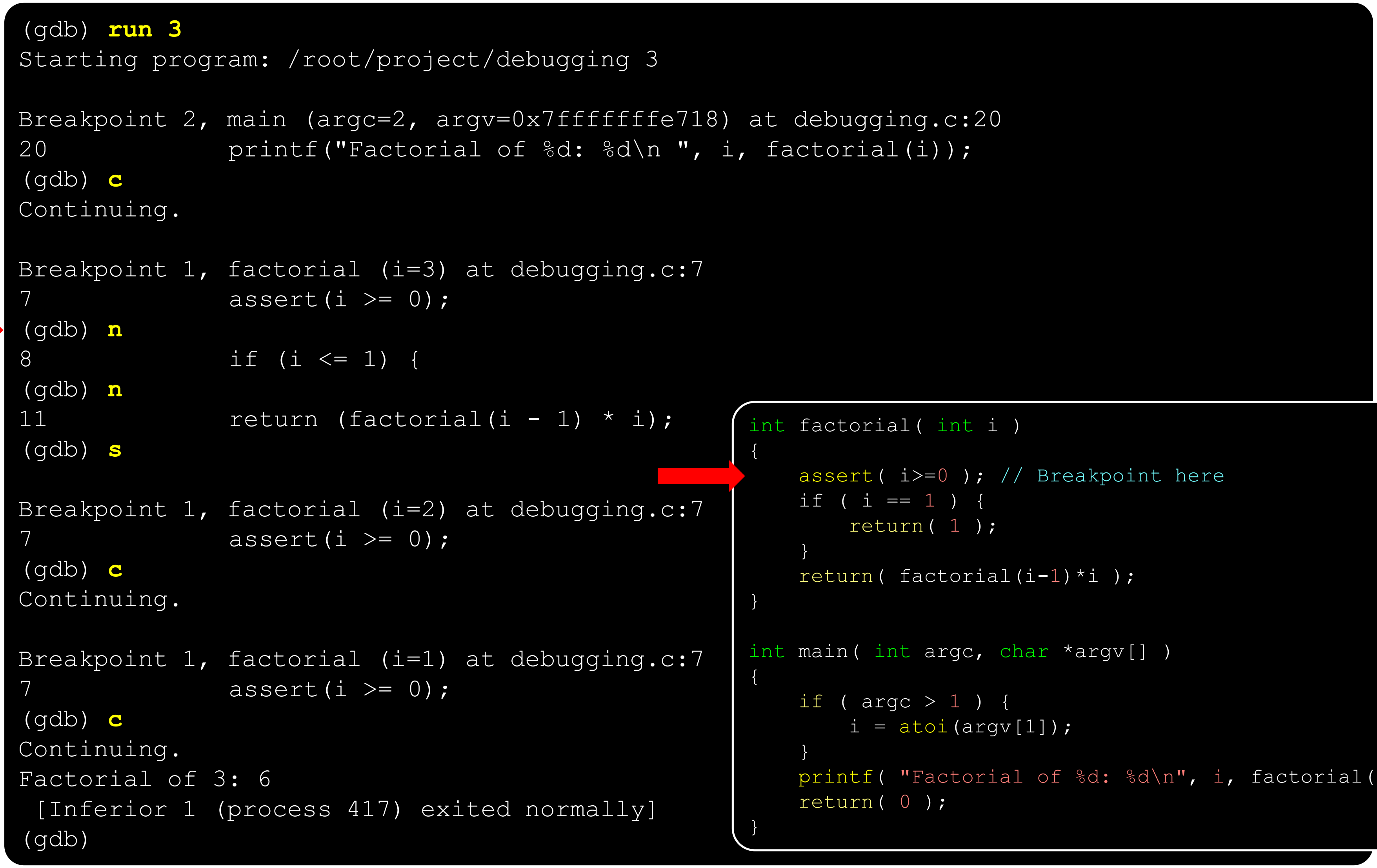
```c
int factorial( int i )
{
    assert( i>=0 ); // Breakpoint here
    if ( i == 1 ) {
        return( 1 );
    }
    return( factorial(i-1)*i );
}

int main( int argc, char *argv[] )
{
    if ( argc > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

Putting it all together

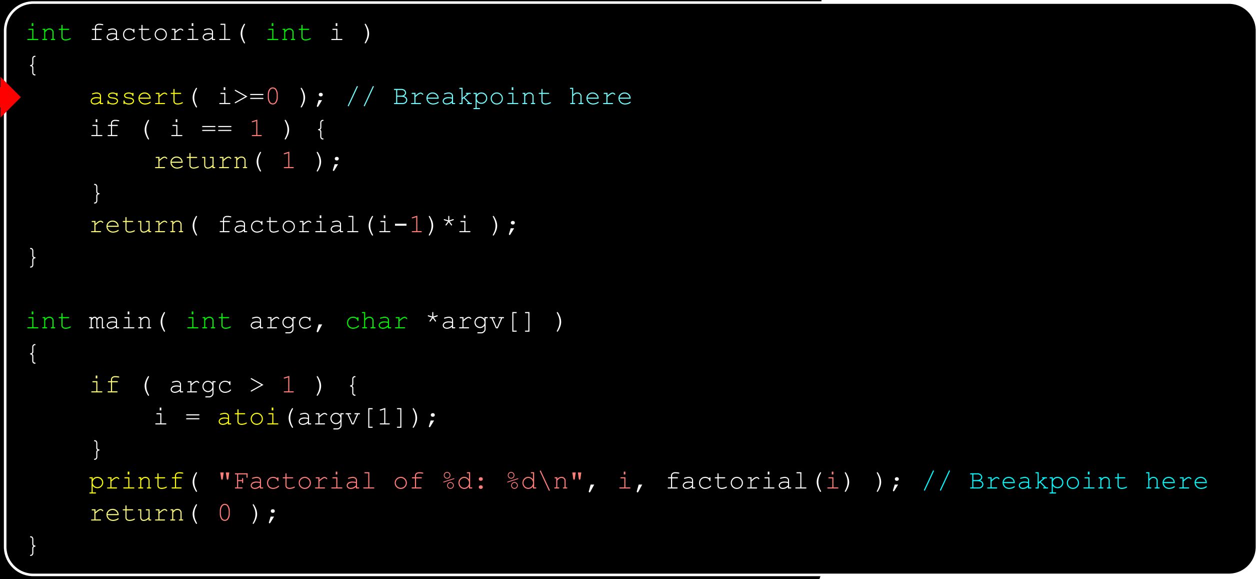
```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20              printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7               assert(i >= 0);
(gdb) n
8               if (i <= 1) {
(gdb) n
11              return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```

```c
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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

Putting it all together



```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20          printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7           assert(i >= 0);
(gdb) n
8           if (i <= 1) {
(gdb) n
11          return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7           assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7           assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```



```c
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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

Putting it all together

```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20              printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7               assert(i >= 0);
(gdb) n
8               if (i <= 1) {
(gdb) n
11              return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```

```c
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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

Putting it all together

```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20              printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7               assert(i >= 0);
(gdb) n
8               if (i <= 1) {
(gdb) n
11              return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```

```
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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

Putting it all together

```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20              printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7               assert(i >= 0);
(gdb) n
8               if (i <= 1) {
(gdb) n
11              return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7               assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```

```
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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

Putting it all together

```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20          printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7           assert(i >= 0);
(gdb) n
8           if (i <= 1) {
(gdb) n
11          return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7           assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7           assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```

```c
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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```

Putting it all together

```
(gdb) run 3
Starting program: /root/project/debugging 3

Breakpoint 2, main (argc=2, argv=0x7fffffffe718) at debugging.c:20
20            printf("Factorial of %d: %d\n ", i, factorial(i));
(gdb) c
Continuing.

Breakpoint 1, factorial (i=3) at debugging.c:7
7             assert(i >= 0);
(gdb) n
8             if (i <= 1) {
(gdb) n
11            return (factorial(i - 1) * i);
(gdb) s

Breakpoint 1, factorial (i=2) at debugging.c:7
7             assert(i >= 0);
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:7
7             assert(i >= 0);
(gdb) c
Continuing.
Factorial of 3: 6
 [Inferior 1 (process 417) exited normally]
(gdb)
```

```c
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 > 1 ) {
        i = atoi(argv[1]);
    }
    printf( "Factorial of %d: %d\n", i, factorial(i) ); // Breakpoint here
    return( 0 );
}
```