



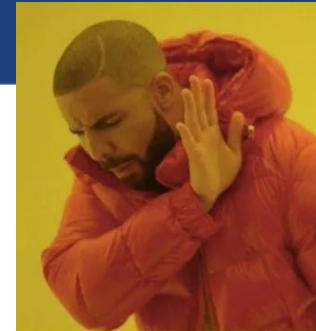
PennState

CMPSC 311 - Introduction to Systems Programming

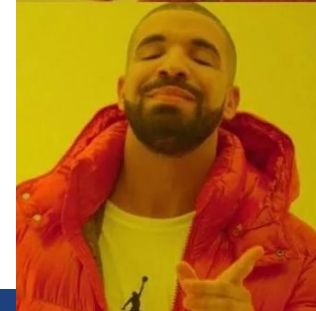
Debugging

Suman Saha

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



Using a debugger to find where your program crashed



`print("got this far")`

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



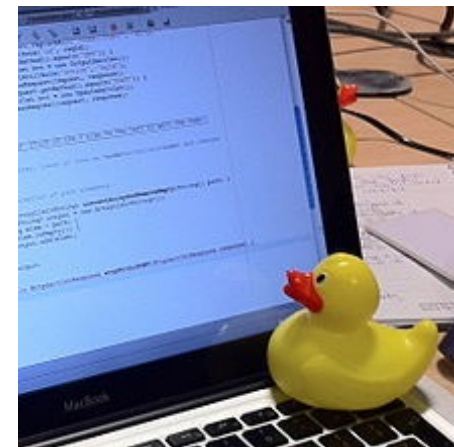
By Rob Pike

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 themselves 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



PennState

```
#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



PennState

```
#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
```

```
        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, ...)



- You run the debugger by passing the program to gdb

```
$ gdb [program name]
```

- This is an **interactive** terminal-based debugger
- It can be launched inside Emacs->Tools->Debugger(GDB)
- 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)
```



- 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



PennState

- 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** and emacs integrate the **gdb** functionality into the editor/IDE

```
xterm  
#define sln.c  
7  
8         return(-1));  
9  
10        case 'v': // Verbose Flag  
11            verbose = 1;  
12            break;  
13  
14        case 'u': // Unit Tests Flag  
15            unit_tests = 1;  
16            break;  
17  
18        case 'l': // Set the log filename  
19            initializeLogUnitFileHandle(optarg);  
20            log_initialized = 1;  
21            break;  
22  
23        default: // Default (unknown)  
24            fprintf(stderr, "Unknown command line option (%c), ab  
25            return(-1));  
26    }  
27  
28    )  
29  
30    // Setup the log as needed  
31    if (!log_initialized) {  
32        initializeLogUnitFileHandle(CPFS3C11_LOG_STEREO);  
33    }  
34    if (verbose) {  
35        enableLogLevel(LDG_INFO_LEVEL);  
36    }  
37  
38    // If we are running the unit tests, do that  
39    if (unit_tests) {  
40  
41        // Enable verbose, run the tests and check the results  
42        enableLogLevel(LDG_INFO_LEVEL);  
43  
44        // 311 Library Unit Tests  
45        if (cpm3c11_unit_tests()) {  
46            logMessage(LDG_ERROR_LEVEL, "Tagline unit tests failed");  
47        } else {  
48            logMessage(LDG_INFO_LEVEL, "Tagline unit tests complete");  
49        }  
50  
51        // RMD unit tests  
52        if (rmd_unit_test()) {  
53            logMessage(LDG_ERROR_LEVEL, "Tagline unit tests failed");  
54        } else {  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782
```

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) 1 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   0x000000000000006e5 in factorial at debugging.c:7
2        breakpoint     keep y   0x0000000000000075c 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=0x7fffffff718) 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
0x7fffffff5f8: 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 );  
}
```

The diagram illustrates the 'next' command in a debugger. A blue box labeled 'next' is positioned above the line `printf("Factorial : %d! = %d\n", 5,);`. A blue arrow points from the box to the line. Two red arrows point to the lines `int x = factorial(5);` and `printf("Factorial : %d! = %d\n", 5,);`.

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 );  
}
```

A diagram illustrating the 'step' command in a debugger. A blue box labeled 'step' has two arrows originating from it. One arrow points to the 'if' statement in the 'factorial' function, and the other points to the 'int x = factorial(5);' line in the 'main' function. Red arrows point to the 'if' statement and the 'int x = factorial(5);' line, indicating the current execution point.

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 );  
}
```

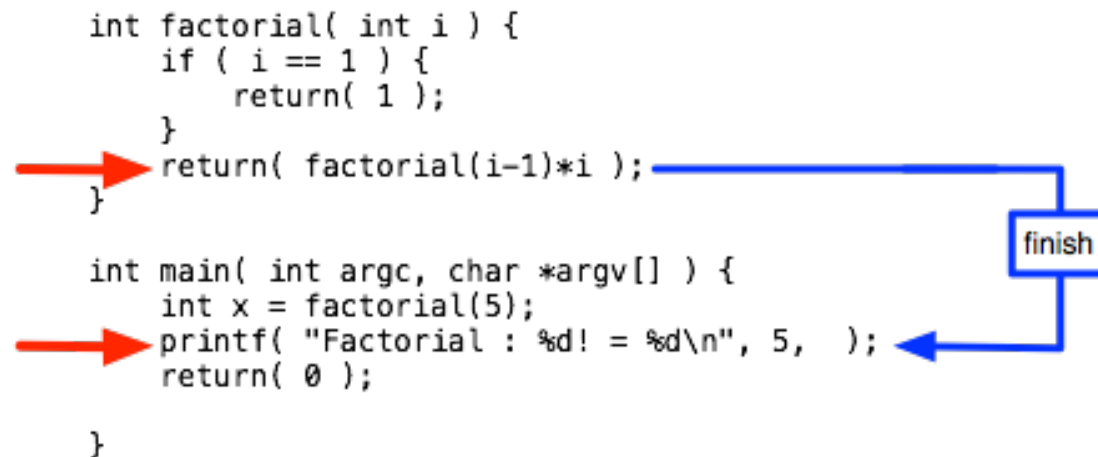
continue

A diagram illustrating the effect of the 'continue' command in a debugger. A red arrow points to the line 'int x = factorial(5);' in the code. A blue box labeled 'continue' has a blue arrow pointing down from it, indicating that the program will skip the remaining statements in the current function and jump to the next breakpoint or the end of the program.

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



Putting it all together



PennState

```
#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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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



PennState

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

```
Breakpoint 2, main (argc=2, argv=0x7fffffff718) 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 );
}
```