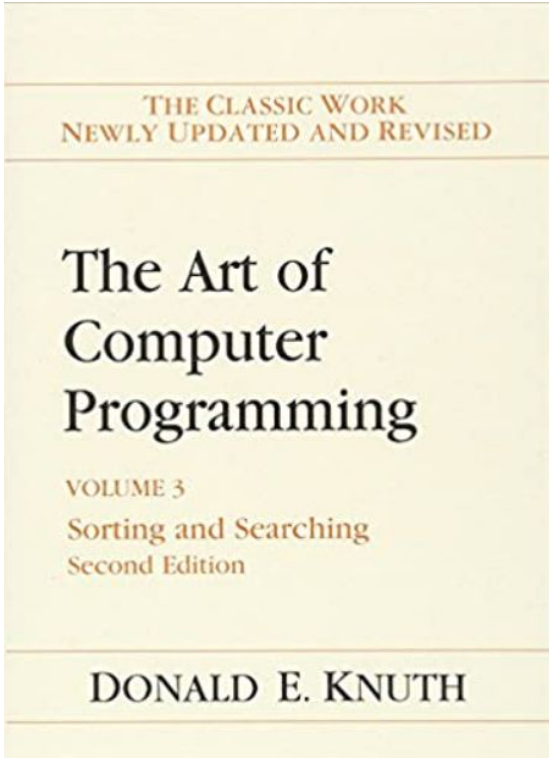
ECE 264 Spring 2023

Advanced C Programming

Aravind Machiry
Purdue University

Homework 1
Selection Sort



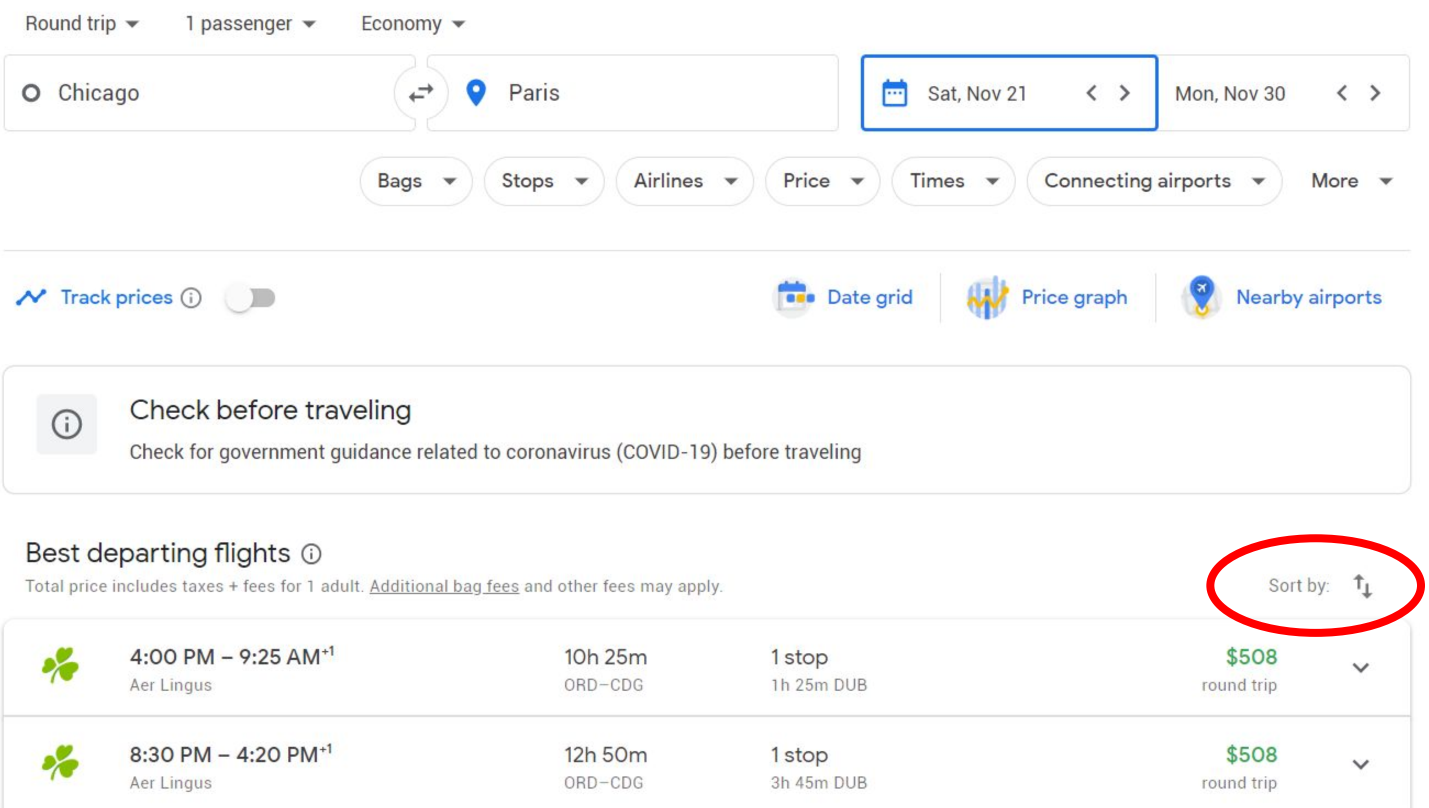
THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 3
Sorting and Searching
Second Edition

DONALD E. KNUTH

Where sorting is used?



Chicago

Paris

Sat, Nov 21

Mon, Nov 30

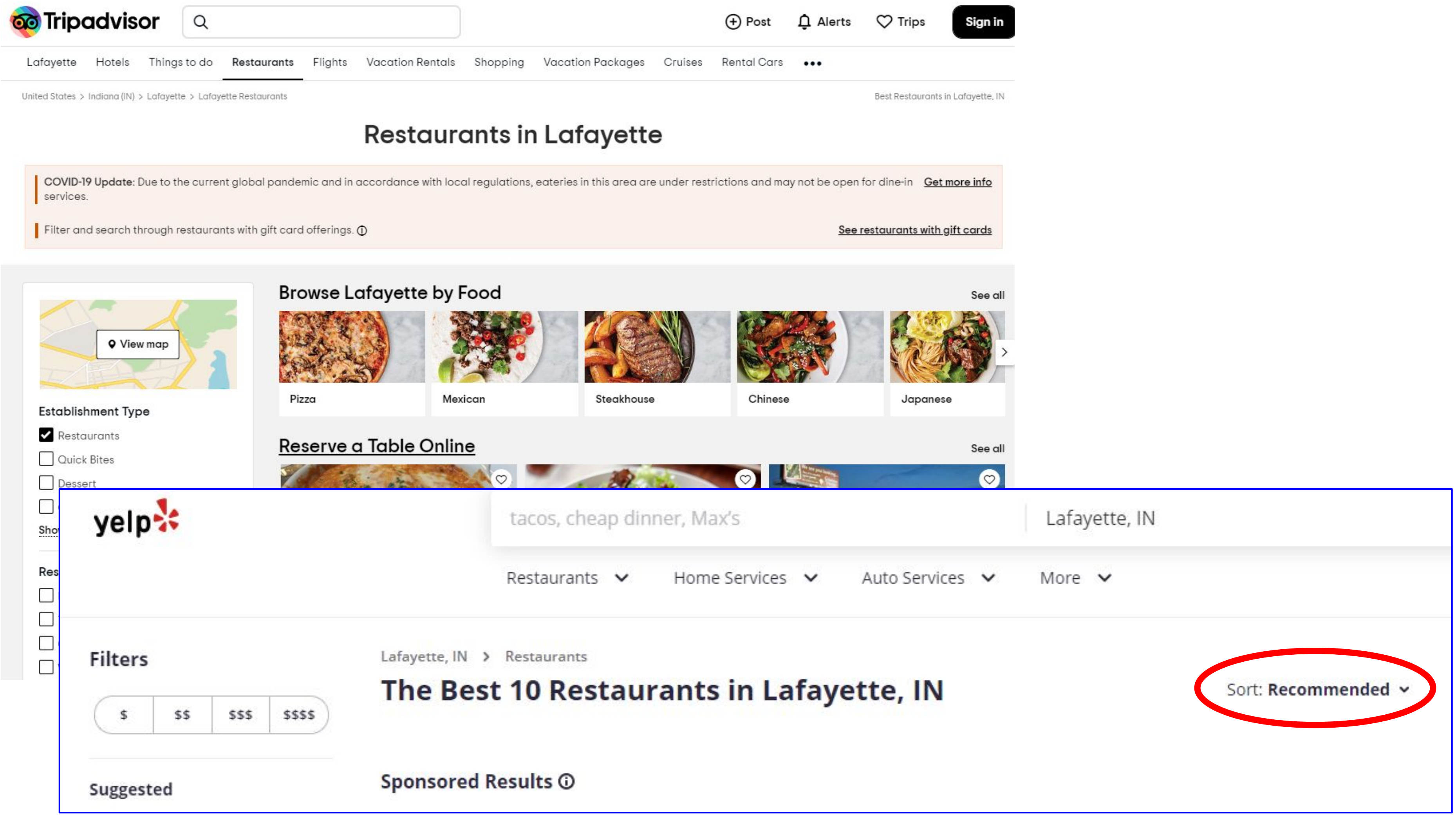
Check before traveling

Check for government guidance related to coronavirus (COVID-19) before traveling

Best departing flights

Total price includes taxes + fees for 1 adult. Additional bag fees and other fees may apply.

4:00 PM – 9:25 AM+1 Aer Lingus	10h 25m ORD–CDG	1 stop 1h 25m DUB	$508 round trip
8:30 PM – 4:20 PM+1 Aer Lingus	12h 50m ORD–CDG	1 stop 3h 45m DUB	$508 round trip



Tripadvisor

Post Alerts Trips **Sign in**

Lafayette Hotels Things to do **Restaurants** Flights Vacation Rentals Shopping Vacation Packages Cruises Rental Cars •••

United States > Indiana (IN) > Lafayette > Lafayette Restaurants

Best Restaurants in Lafayette, IN

Restaurants in Lafayette

COVID-19 Update: Due to the current global pandemic and in accordance with local regulations, eateries in this area are under restrictions and may not be open for dine-in services. Get more info

Filter and search through restaurants with gift card offerings. See restaurants with gift cards

View map

Establishment Type

☑ Restaurants

☐ Quick Bites

☐ Dessert

☐

Sho

Res

☐

☐

☐

☐

Browse Lafayette by Food See all

Pizza

Mexican

Steakhouse

Chinese

Japanese

Reserve a Table Online See all

yelp

tacos, cheap dinner, Max's

Lafayette, IN

Restaurants Home Services Auto Services More

Filters

$ $$ $$$ $$$$

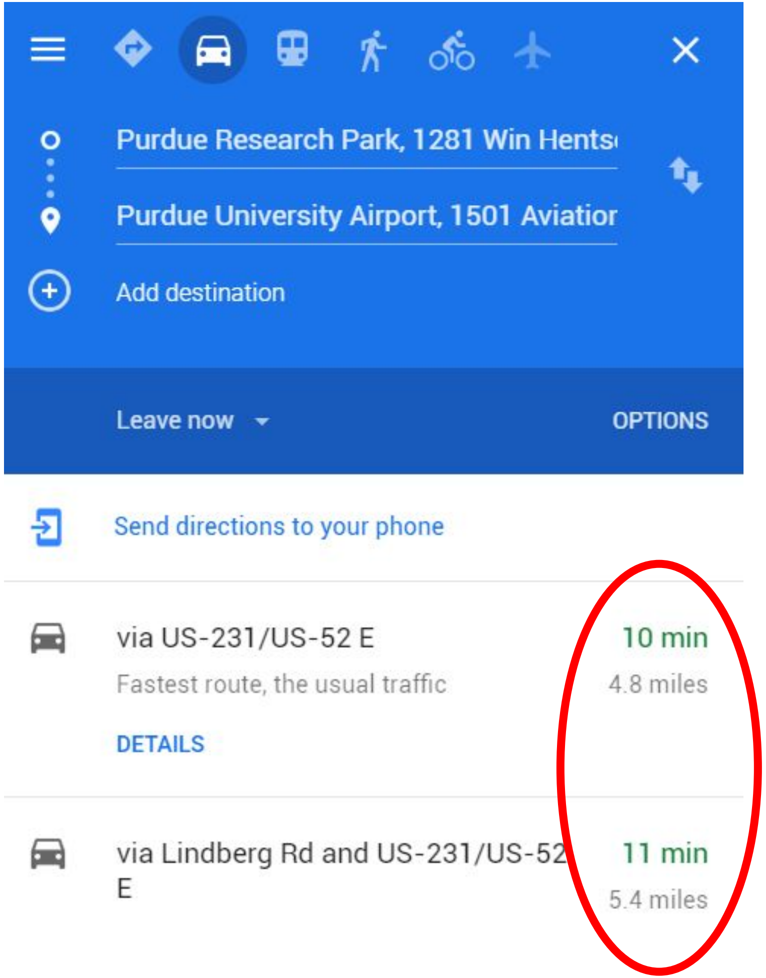
Suggested

Lafayette, IN > Restaurants

The Best 10 Restaurants in Lafayette, IN

Sort: **Recommended**

Sponsored Results



Purdue Research Park, 1281 Win Hents

Purdue University Airport, 1501 Aviatior

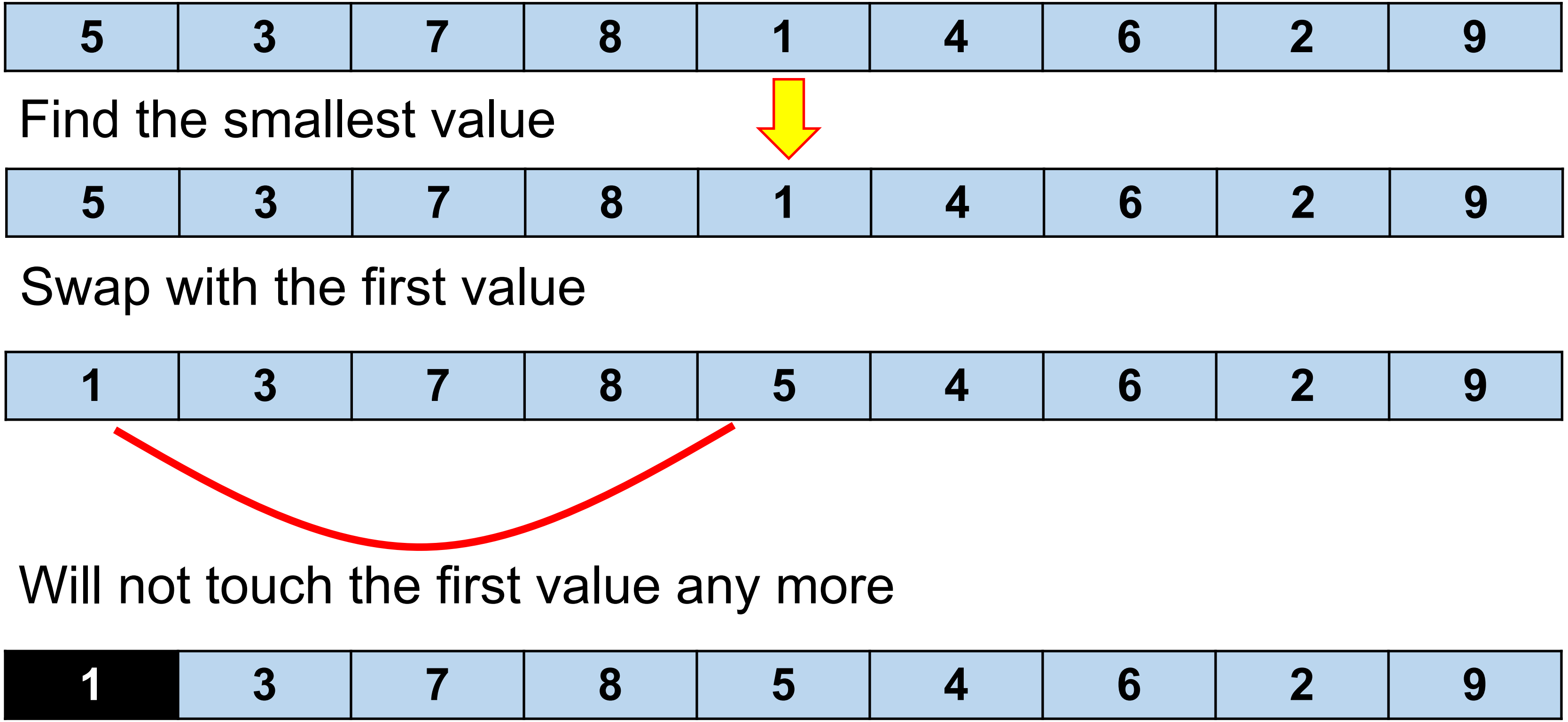
via US-231/US-52 E 10 min

Fastest route, the usual traffic 4.8 miles

via Lindberg Rd and US-231/US-52 E 11 min

5.4 miles

Selection Sort



5	3	7	8	1	4	6	2	9

Find the smallest value

5	3	7	8	1	4	6	2	9

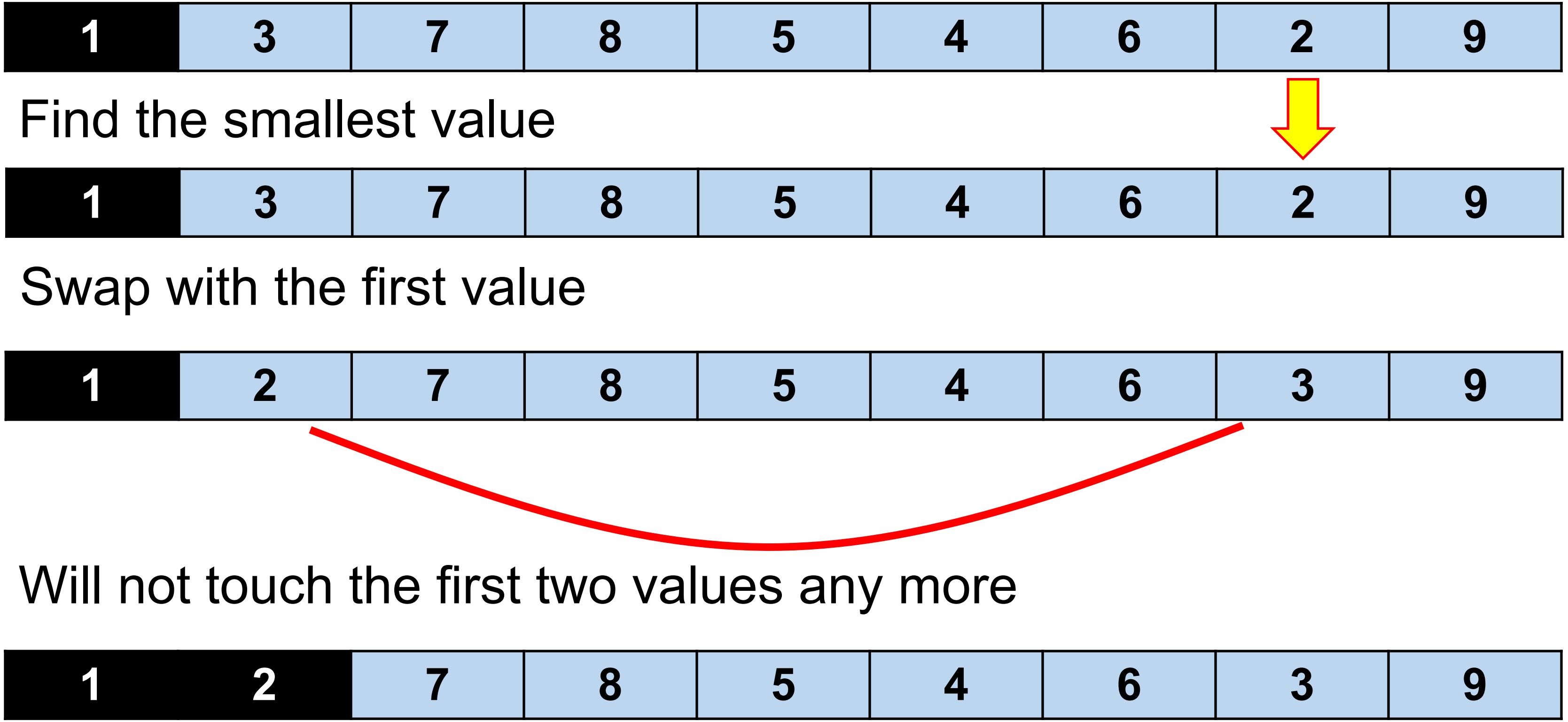
Swap with the first value

1	3	7	8	5	4	6	2	9

Will not touch the first value any more

1	3	7	8	5	4	6	2	9

Selection Sort



1	3	7	8	5	4	6	2	9

Find the smallest value

1	3	7	8	5	4	6	2	9

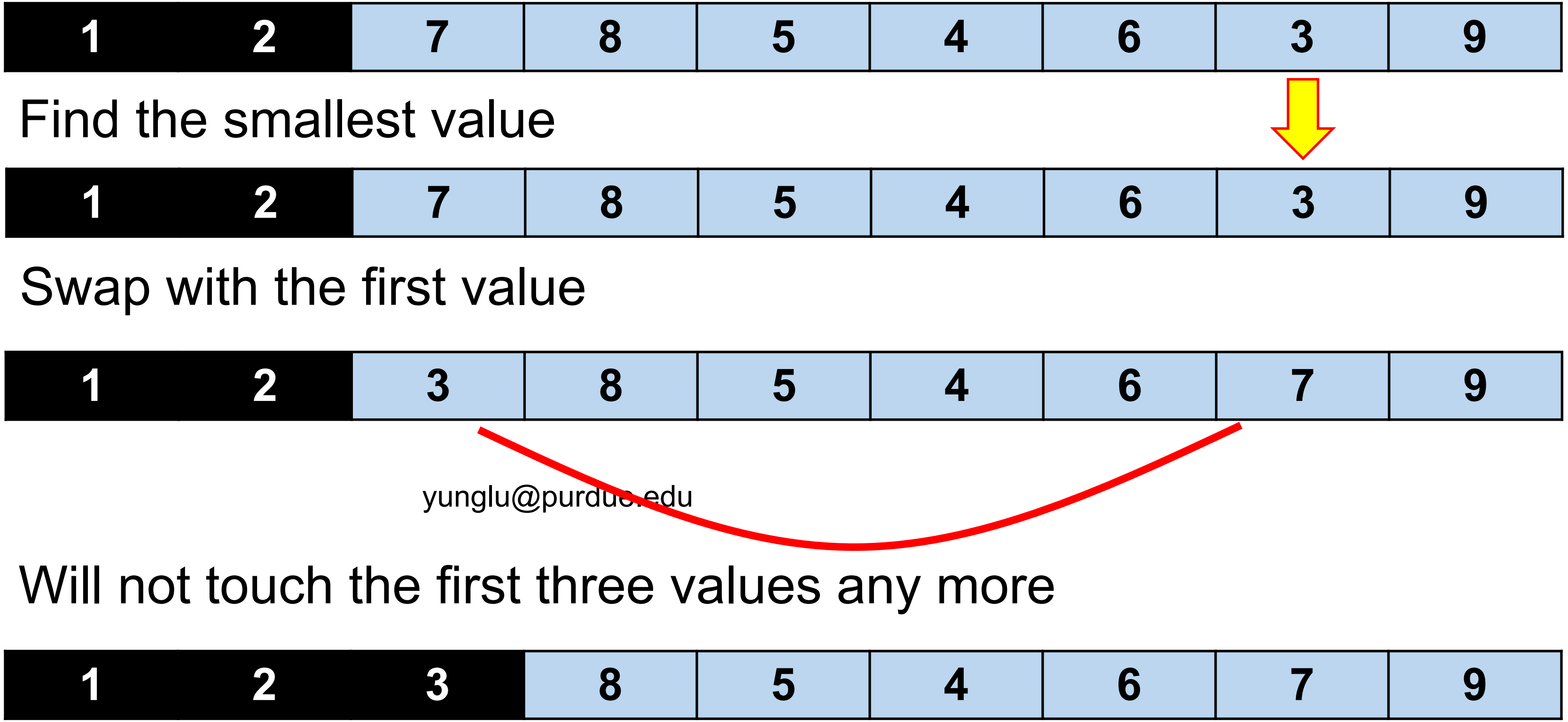
Swap with the first value

1	2	7	8	5	4	6	3	9

Will not touch the first two values any more

1	2	7	8	5	4	6	3	9

Selection Sort



1	2	7	8	5	4	6	3	9

Find the smallest value

1	2	7	8	5	4	6	3	9

Swap with the first value

1	2	3	8	5	4	6	7	9

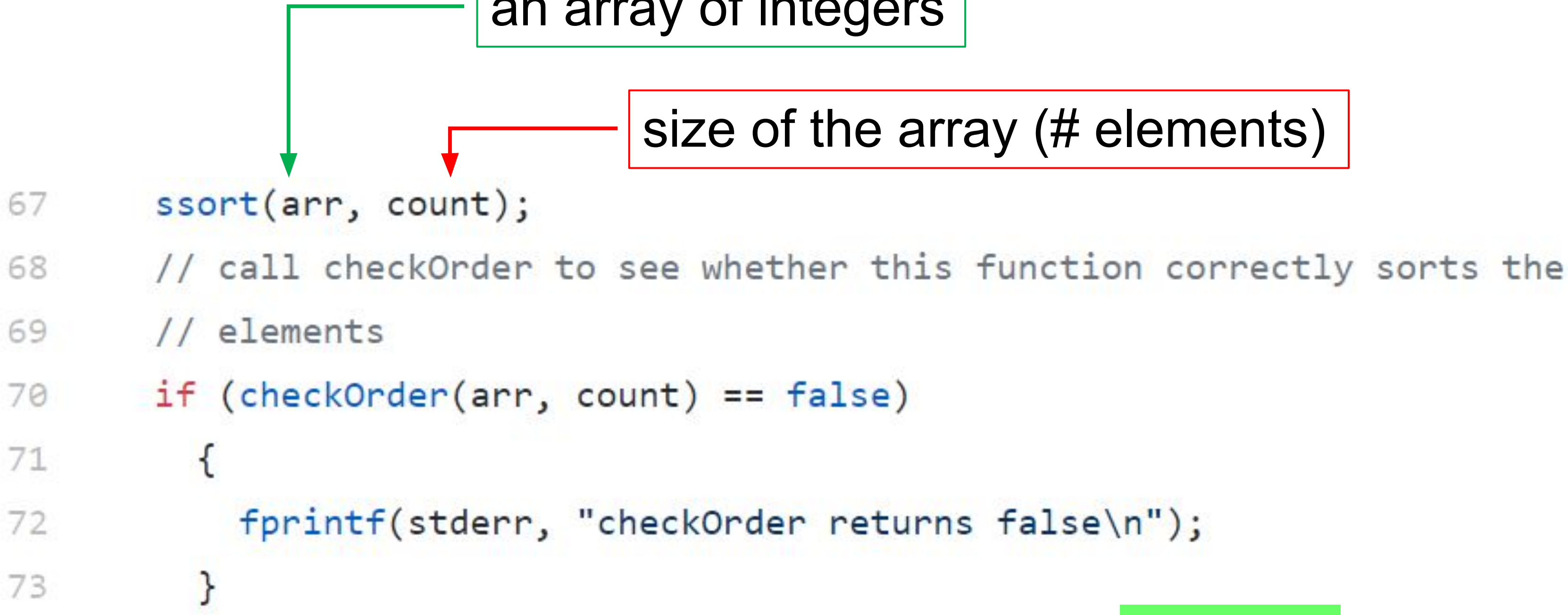
yunglu@purdue.edu

Will not touch the first three values any more

1	2	3	8	5	4	6	7	9

Selection Sort

- Two levels of iterations:
- Outer: from the first element to the second last element
    - Inner: from one after the outside to the last element
    - Select the smallest value
- If the smallest value is different from the outside value, swap
- If there are n elements, at most n swaps
- The number of comparisons is $\approx (n - 1) \times (n - 1) / 2 \approx n^2$

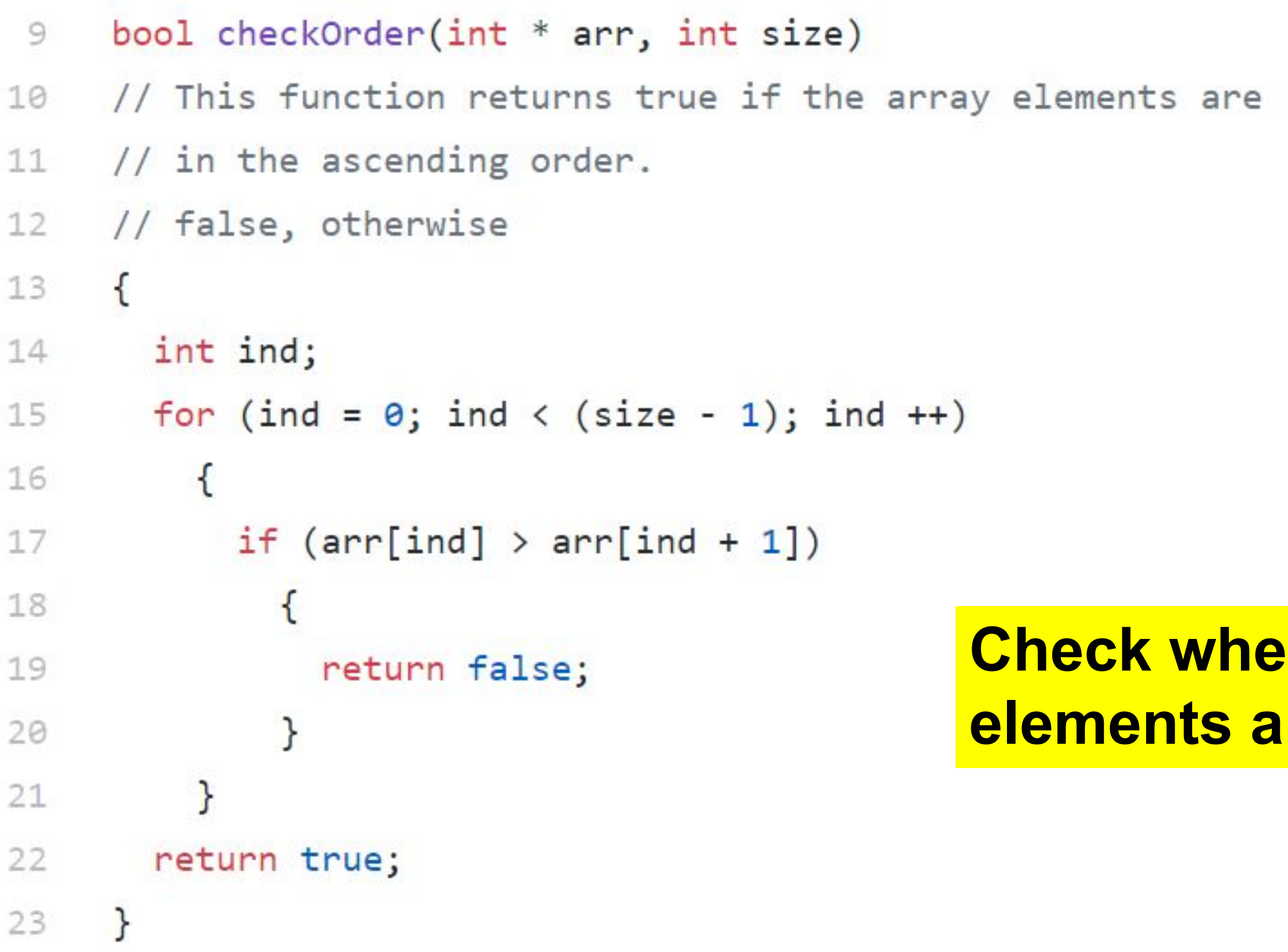


an array of integers

size of the array (# elements)

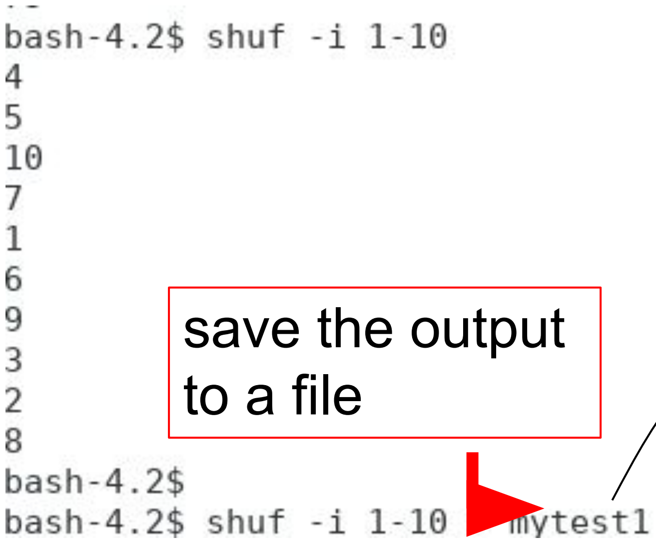
```
ssort(arr, count);
// call checkOrder to see whether this function correctly sorts the
// elements
if (checkOrder(arr, count) == false)
  {
    fprintf(stderr, "checkOrder returns false\n");
  }
```

main.c



```
bool checkOrder(int * arr, int size)
// This function returns true if the array elements are
// in the ascending order.
// false, otherwise
{
  int ind;
  for (ind = 0; ind < (size - 1); ind ++)
    {
      if (arr[ind] > arr[ind + 1])
        {
          return false;
        }
    }
  return true;
}
```

Check whether elements are sorted

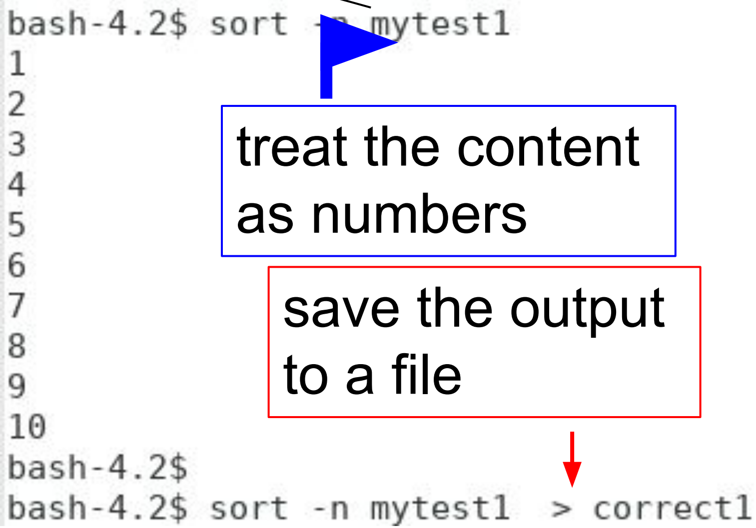


```
bash-4.2$ shuf -i 1-10
4
5
10
7
1
6
9
3
2
8
bash-4.2$
bash-4.2$ shuf -i 1-10   mytest1
```

save the output to a file

create test cases

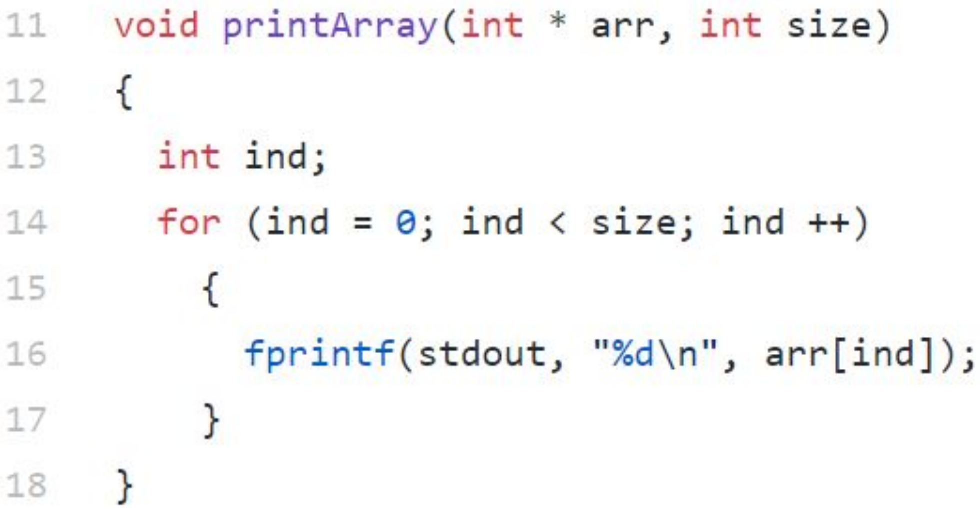
create correct answer



```
bash-4.2$ sort -n mytest1
1
2
3
4
5
6
7
8
9
10
bash-4.2$
bash-4.2$ sort -n mytest1  > correct1
```

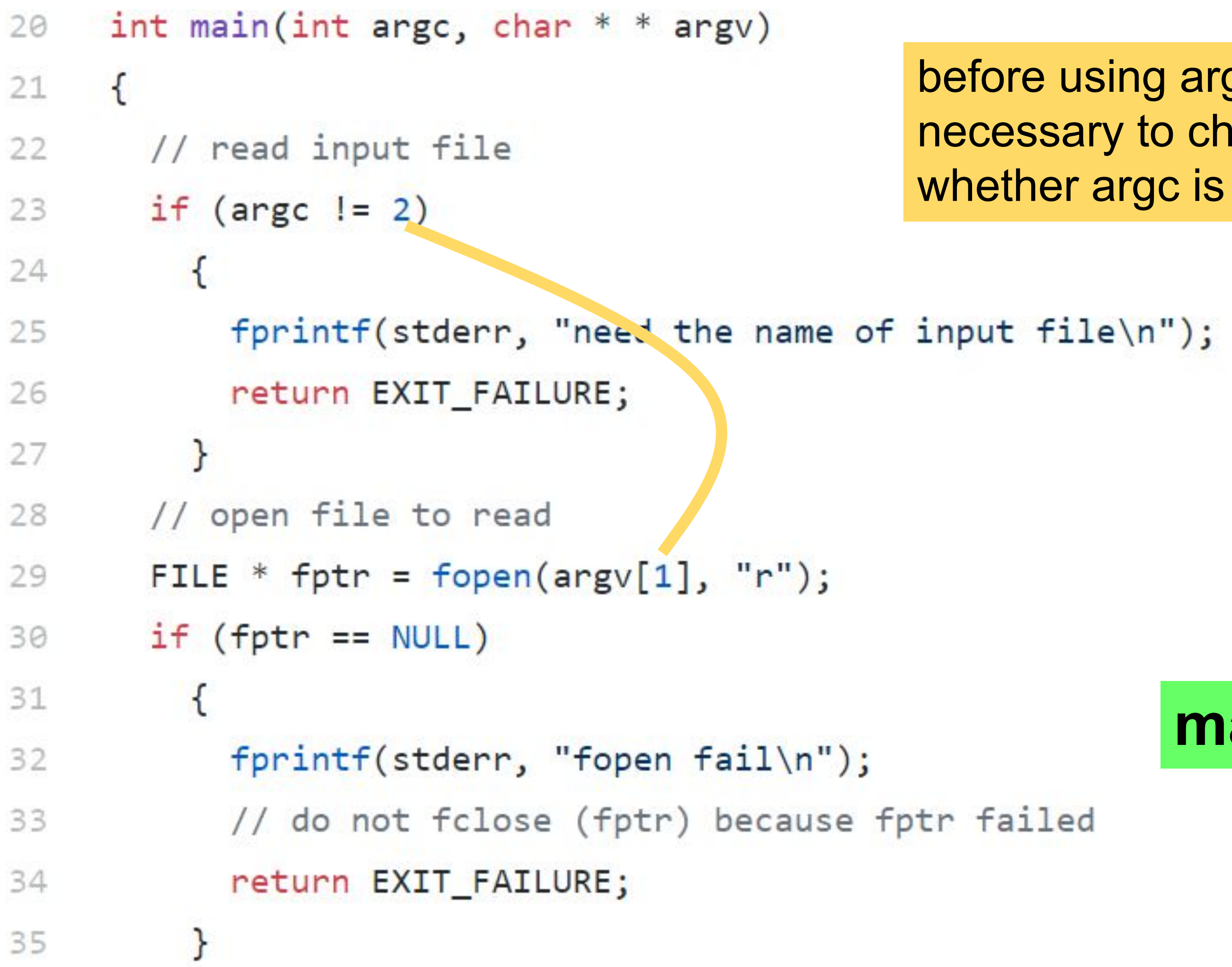
treat the content as numbers

save the output to a file



```c
void printArray(int * arr, int size)
{
  int ind;
  for (ind = 0; ind < size; ind ++)
    {
      fprintf(stdout, "%d\n", arr[ind]);
    }
}
```

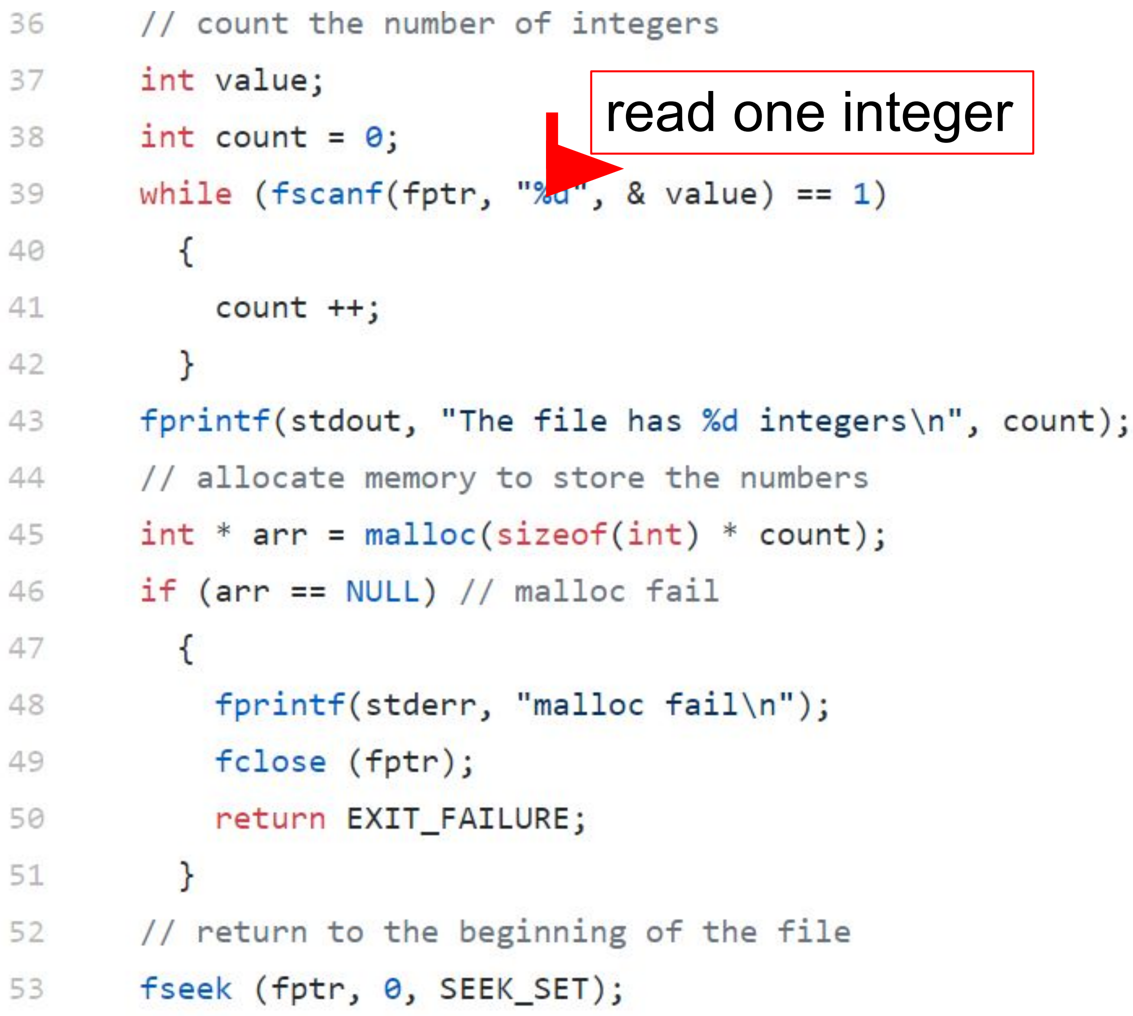
main.c



```c
int main(int argc, char * * argv)
{
  // read input file
  if (argc != 2)
    {
      fprintf(stderr, "need the name of input file\n");
      return EXIT_FAILURE;
    }
  // open file to read
  FILE * fptr = fopen(argv[1], "r");
  if (fptr == NULL)
    {
      fprintf(stderr, "fopen fail\n");
      // do not fclose (fptr) because fptr failed
      return EXIT_FAILURE;
    }
```

before using argv[1], necessary to check whether argc is at least two

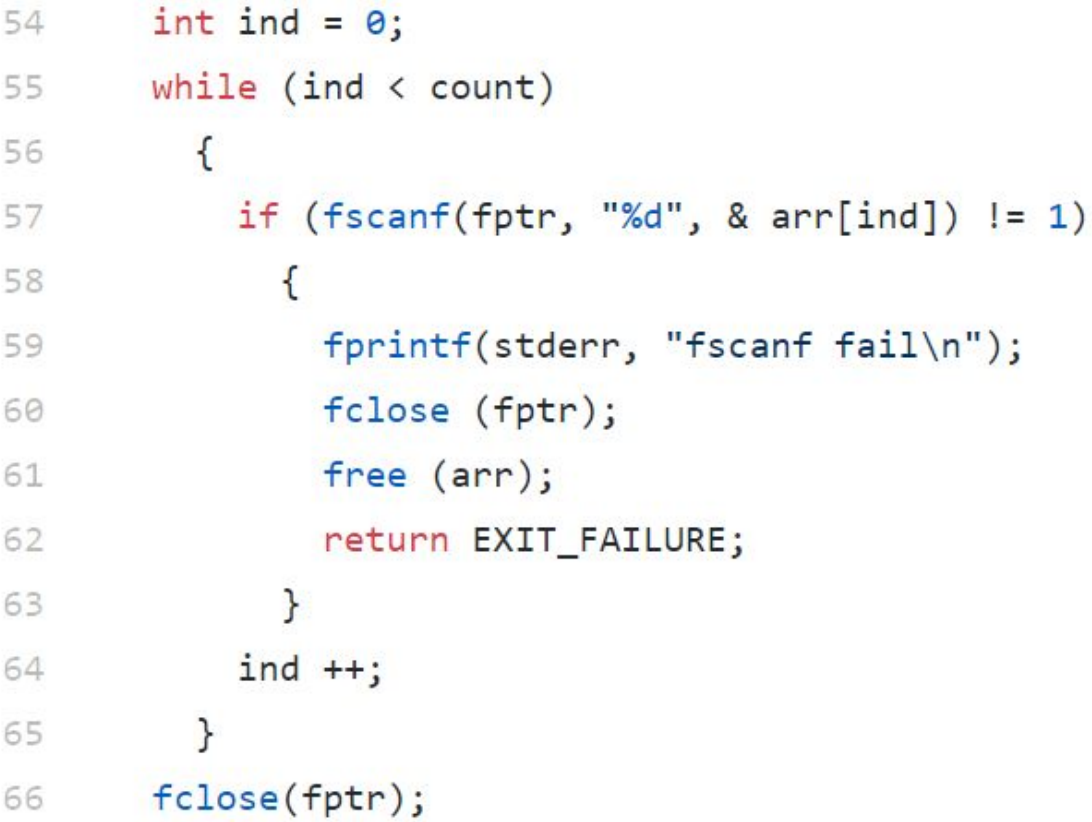
main.c



```
// count the number of integers
int value;
int count = 0;
while (fscanf(fptr, "%d", & value) == 1)
  {
    count ++;
  }
fprintf(stdout, "The file has %d integers\n", count);
// allocate memory to store the numbers
int * arr = malloc(sizeof(int) * count);
if (arr == NULL) // malloc fail
  {
    fprintf(stderr, "malloc fail\n");
    fclose (fptr);
    return EXIT_FAILURE;
  }
// return to the beginning of the file
fseek (fptr, 0, SEEK_SET);
```

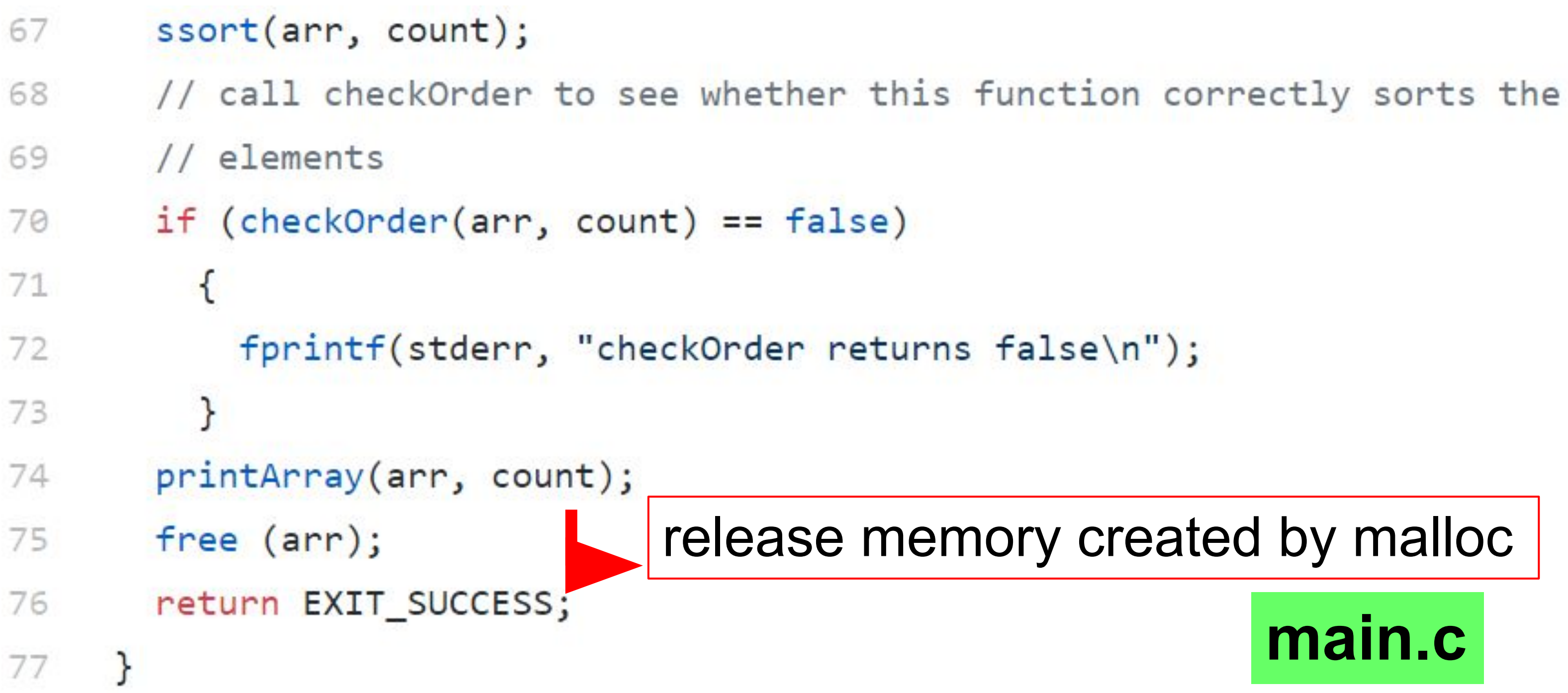
read one integer

main.c



```c
int ind = 0;
while (ind < count)
 {
  if (fscanf(fptr, "%d", & arr[ind]) != 1)
   {
    fprintf(stderr, "fscanf fail\n");
    fclose (fptr);
    free (arr);
    return EXIT_FAILURE;
   }
  ind ++;
 }
fclose(fptr);
```

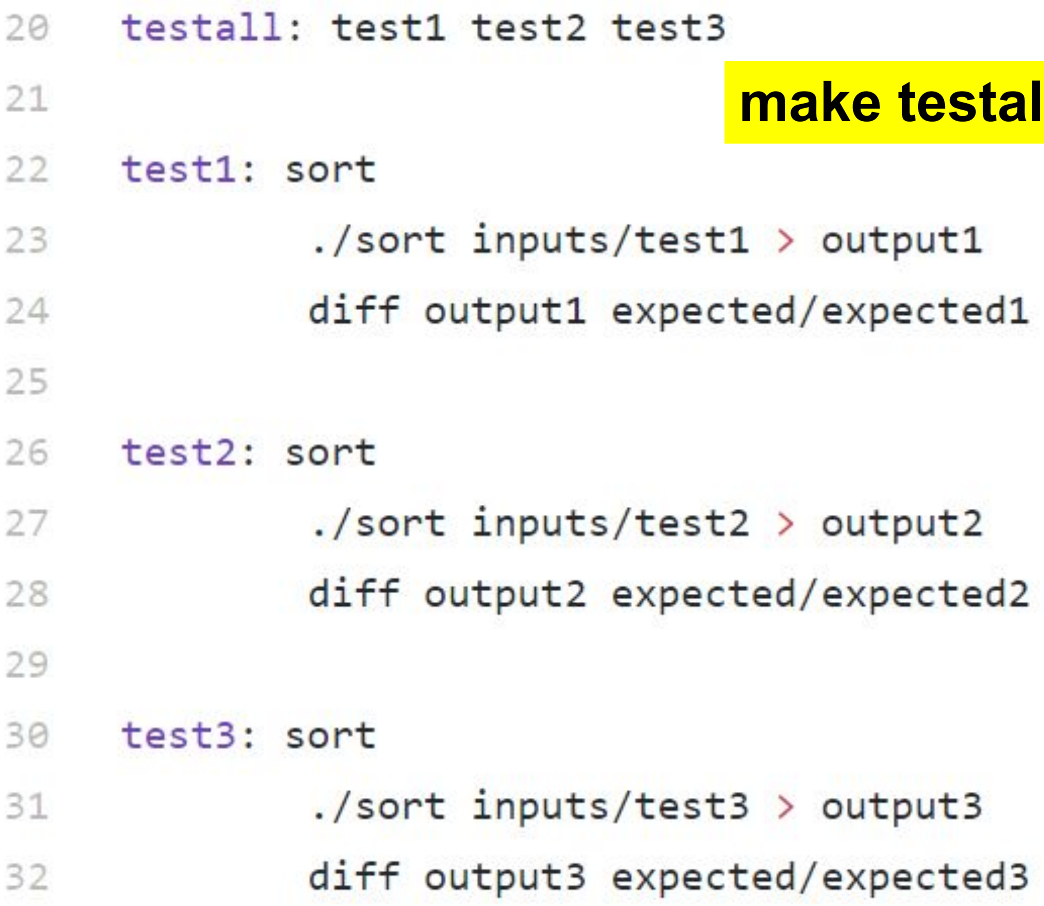
main.c



```c
    ssort(arr, count);
    // call checkOrder to see whether this function correctly sorts the
    // elements
    if (checkOrder(arr, count) == false)
      {
        fprintf(stderr, "checkOrder returns false\n");
      }
    printArray(arr, count);
    free (arr);
    return EXIT_SUCCESS;
  }
```

release memory created by malloc

main.c

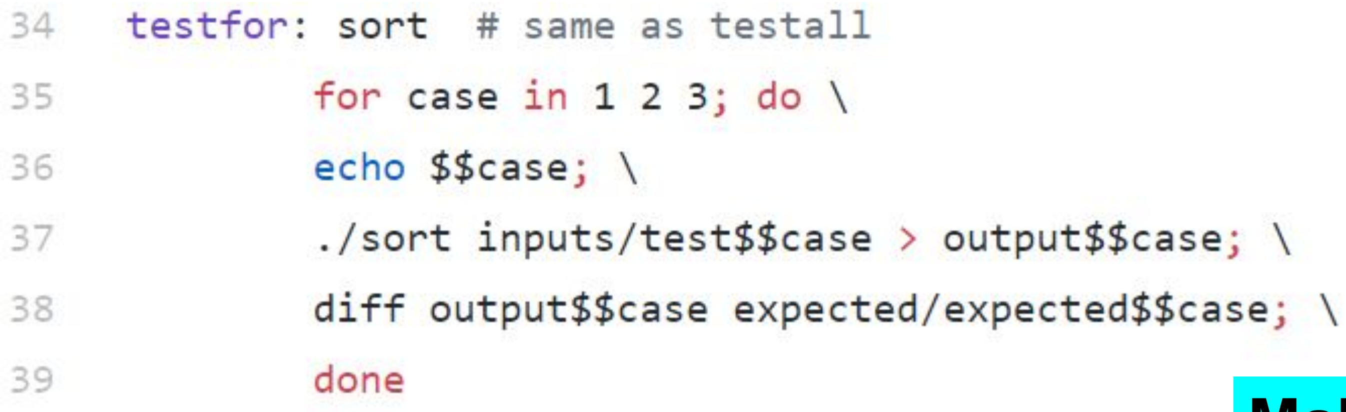


```
testall: test1 test2 test3

test1: sort
	./sort inputs/test1 > output1
	diff output1 expected/expected1

test2: sort
	./sort inputs/test2 > output2
	diff output2 expected/expected2

test3: sort
	./sort inputs/test3 > output3
	diff output3 expected/expected3
```

make testall: run all three test cases

Makefile

make testfor: run all three test cases



```
testfor: sort  # same as testall
	for case in 1 2 3; do \
	echo $$case; \
	./sort inputs/test$$case > output$$case; \
	diff output$$case expected/expected$$case; \
	done
```

Makefile

How to test code (and not)?

Separate “Product” from “Development” code

Product Code	**Development Code**
Create products	Internal use
Polished	Experimental
Only necessary for product	May include additional for instrumentation
No assert	May use assert in testing
No debugging message	May include debugging messages

homework submission

1:5 rule: for each line of product code, write 5 lines of development code

How to test your code correctly?

Product Code

X = A function your write

ssort

Development Code

Prepare data for testing X
call X with the proper data
check results
print debugging messages
(use assert here if you wish)

main

How to test your code incorrectly? (mix product code and testing code)

```
X = A function your write (Product Code)
{
    ….
    necessary code
    check results
    debugging messages
    assert
    ….
}
```

Linux Tools for C Programming

Many Linux tools for C Programming

GDB

The GNU Project Debugger

eclipse

Valgrind

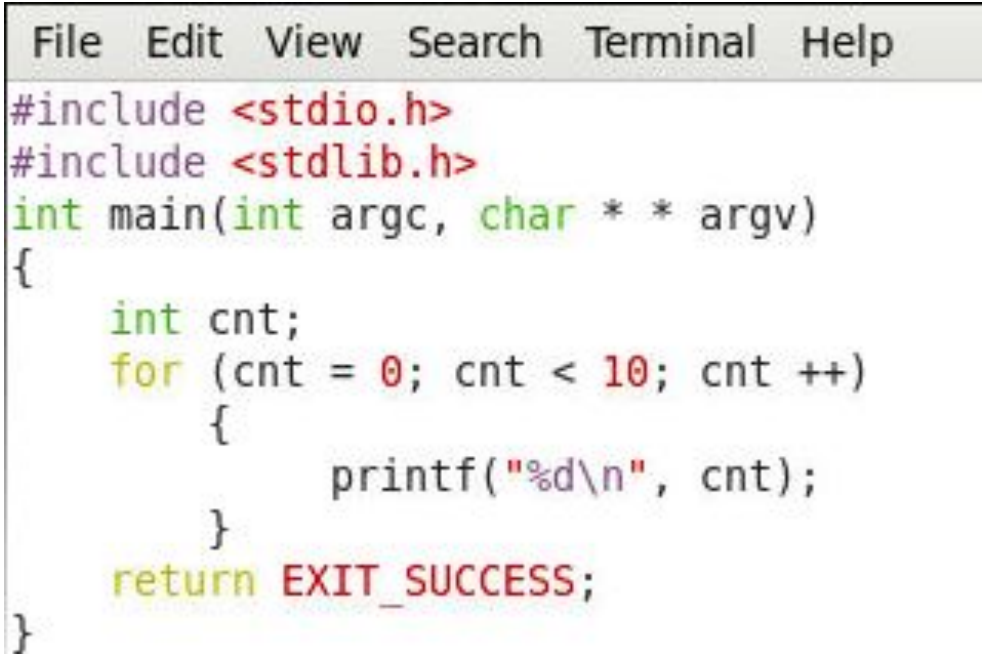
GCC

DDD

v3.3

DataDisplayDebugger

A Simple C Program

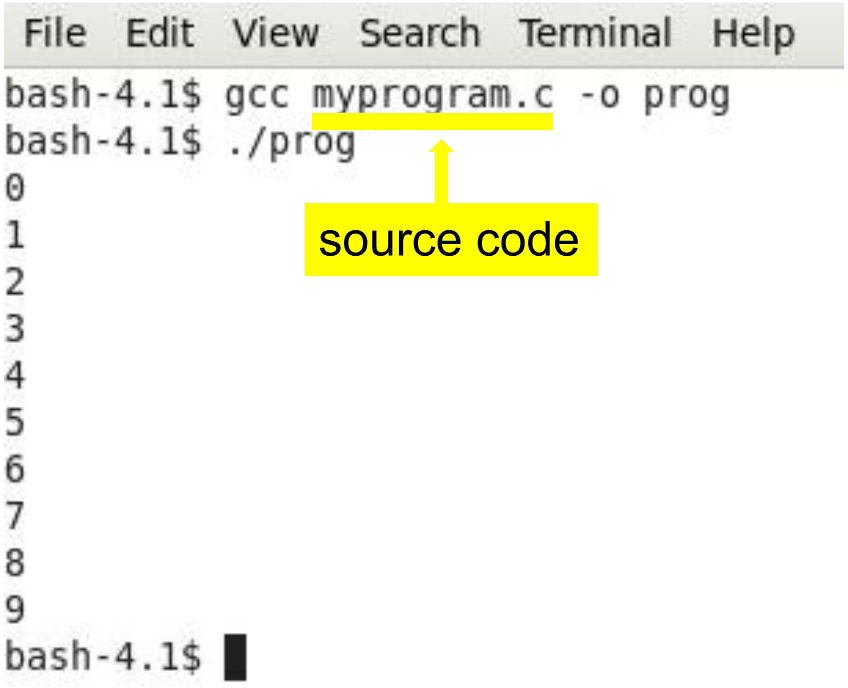


```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
    int cnt;
    for (cnt = 0; cnt < 10; cnt ++)
        {
            printf("%d\n", cnt);
        }
    return EXIT_SUCCESS;
}
```

gcc: GNU C Compiler

```
bash-4.1$ gcc myprogram.c -o prog
bash-4.1$ ./prog
0
1
2
3
4
5
6
7
8
9
bash-4.1$ █
```

gcc: GNU C Compiler



```
bash-4.1$ gcc myprogram.c -o prog
bash-4.1$ ./prog
0
1
2
3
4
5
6
7
8
9
bash-4.1$ 
```

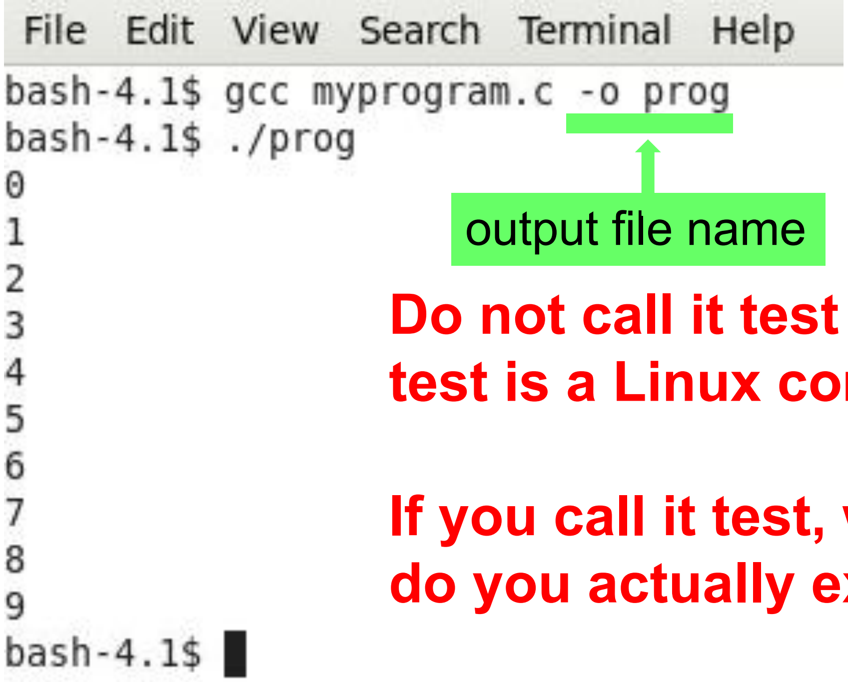
source code

gcc -o output

```
bash-4.1$ gcc myprogram.c -o prog
bash-4.1$ ./prog
0
1
2
3
4
5
6
7
8
9
bash-4.1$
```

output file name

gcc -o output



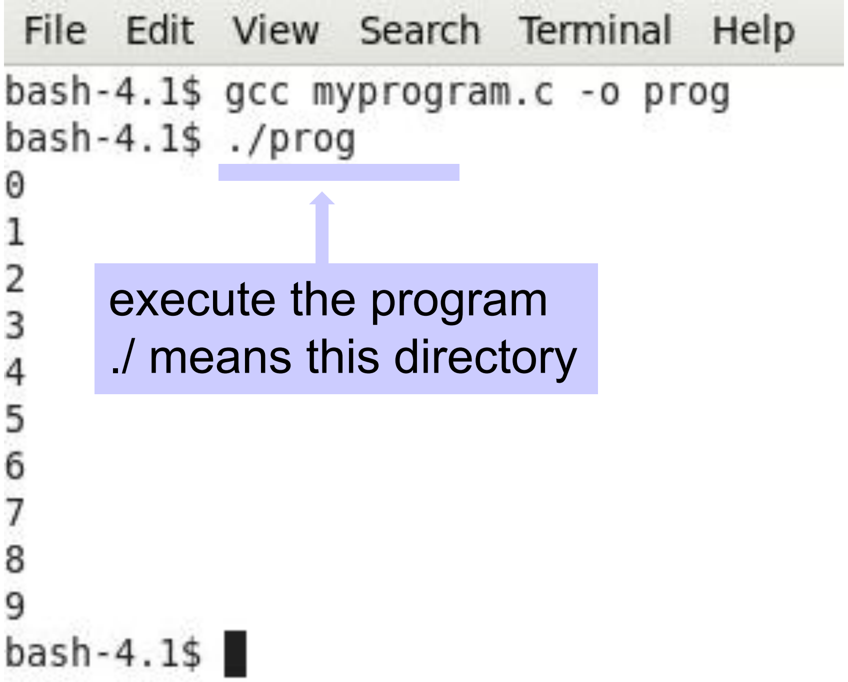
```
bash-4.1$ gcc myprogram.c -o prog
bash-4.1$ ./prog
0
1
2
3
4
5
6
7
8
9
bash-4.1$ █
```

output file name

Do not call it test because test is a Linux command.

If you call it test, which program do you actually execute?

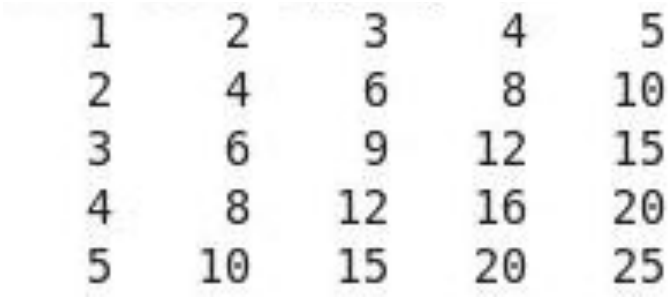
Execute the program



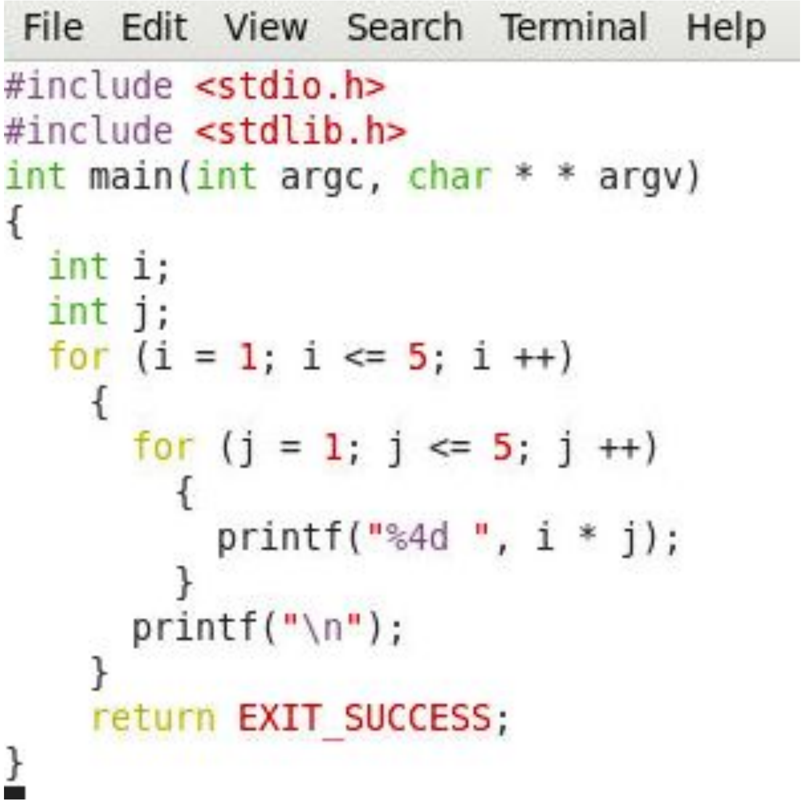
```
bash-4.1$ gcc myprogram.c -o prog
bash-4.1$ ./prog
0
1
2
3
4
5
6
7
8
9
bash-4.1$ █
```

execute the program
./ means this directory

Print 5 x 5 multiplication



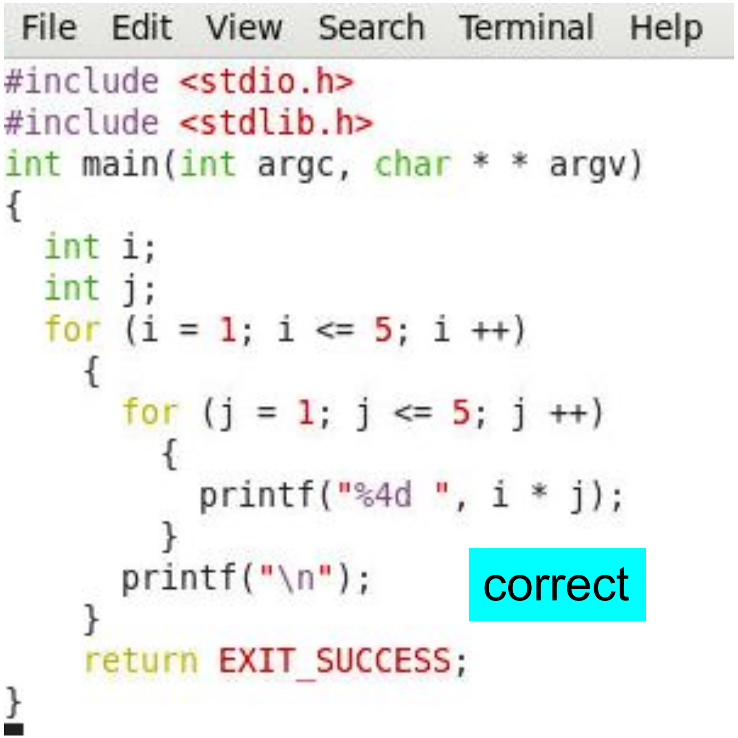
```
   1    2    3    4    5
   2    4    6    8   10
   3    6    9   12   15
   4    8   12   16   20
   5   10   15   20   25
```



```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
 int i;
 int j;
 for (i = 1; i <= 5; i ++)
  {
   for (j = 1; j <= 5; j ++)
    {
     printf("%4d ", i * j);
    }
   printf("\n");
  }
  return EXIT_SUCCESS;
}
```

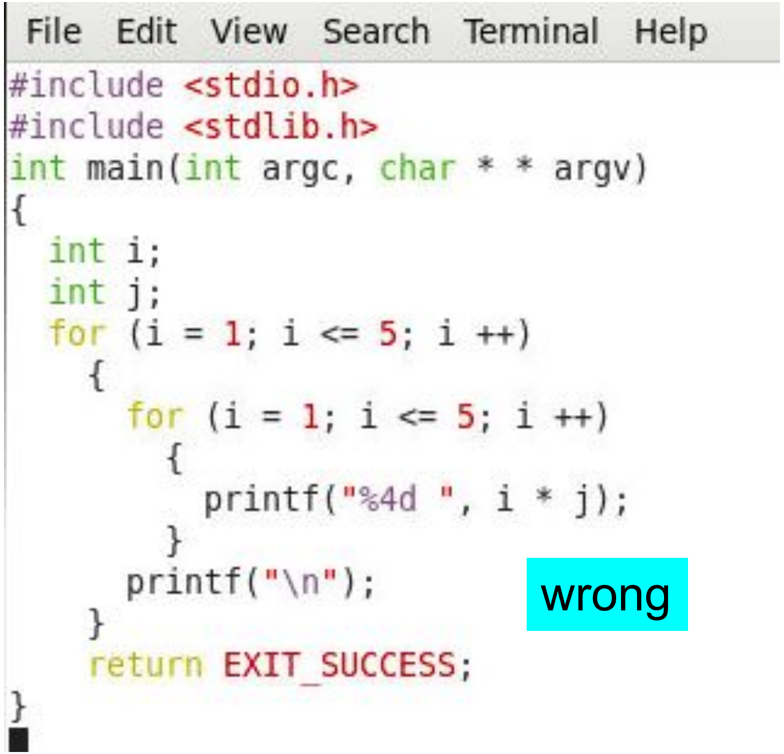
Compare correct and wrong answers

```
   0    0    0    0    0
```



```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  int i;
  int j;
  for (i = 1; i <= 5; i ++)
    {
      for (j = 1; j <= 5; j ++)
        {
          printf("%4d ", i * j);
        }
      printf("\n");
    }
    return EXIT_SUCCESS;
}
```

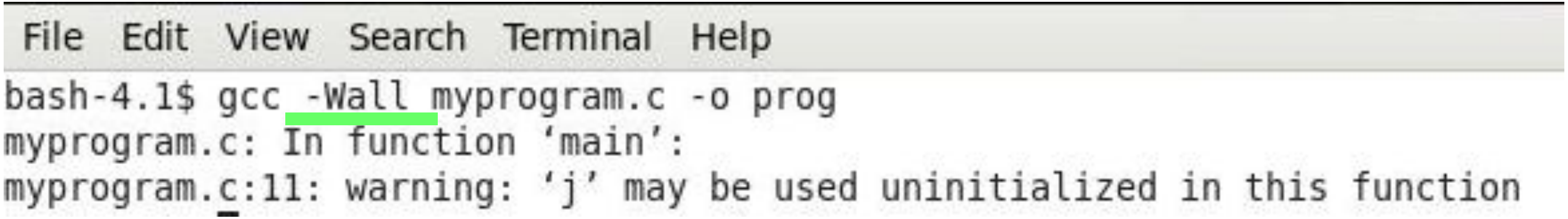
correct



```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  int i;
  int j;
  for (i = 1; i <= 5; i ++)
    {
      for (i = 1; i <= 5; i ++)
        {
          printf("%4d ", i * j);
        }
      printf("\n");
    }
    return EXIT_SUCCESS;
}
```

wrong

Enable gcc warnings



```
bash-4.1$ gcc -Wall myprogram.c -o prog
myprogram.c: In function ‘main’:
myprogram.c:11: warning: ‘j’ may be used uninitialized in this function
```

-Wall enables warnings

gcc warnings can help you identify problems early.

General Rule in Programming:

The earlier you can identify problems, the better.

Do not wait until testing. It requires much more effort.

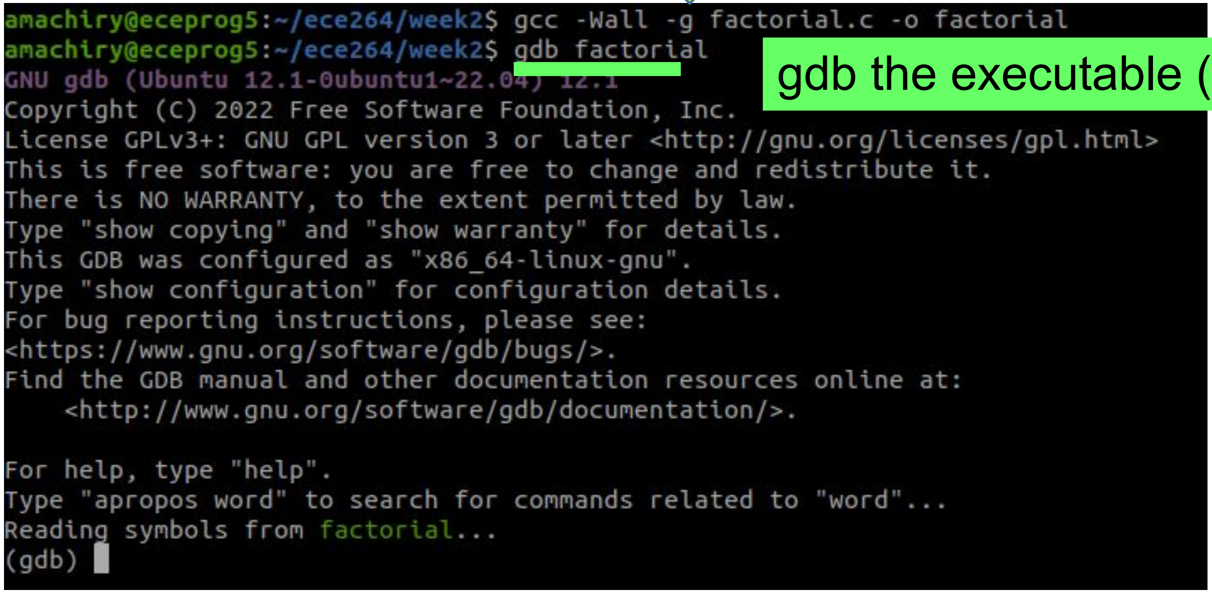
gdb: interactive debugging

- breakpoint: stop at specific line (can be conditional)
- print: see the value of a variable
- see stack memory

Program to compute factorial

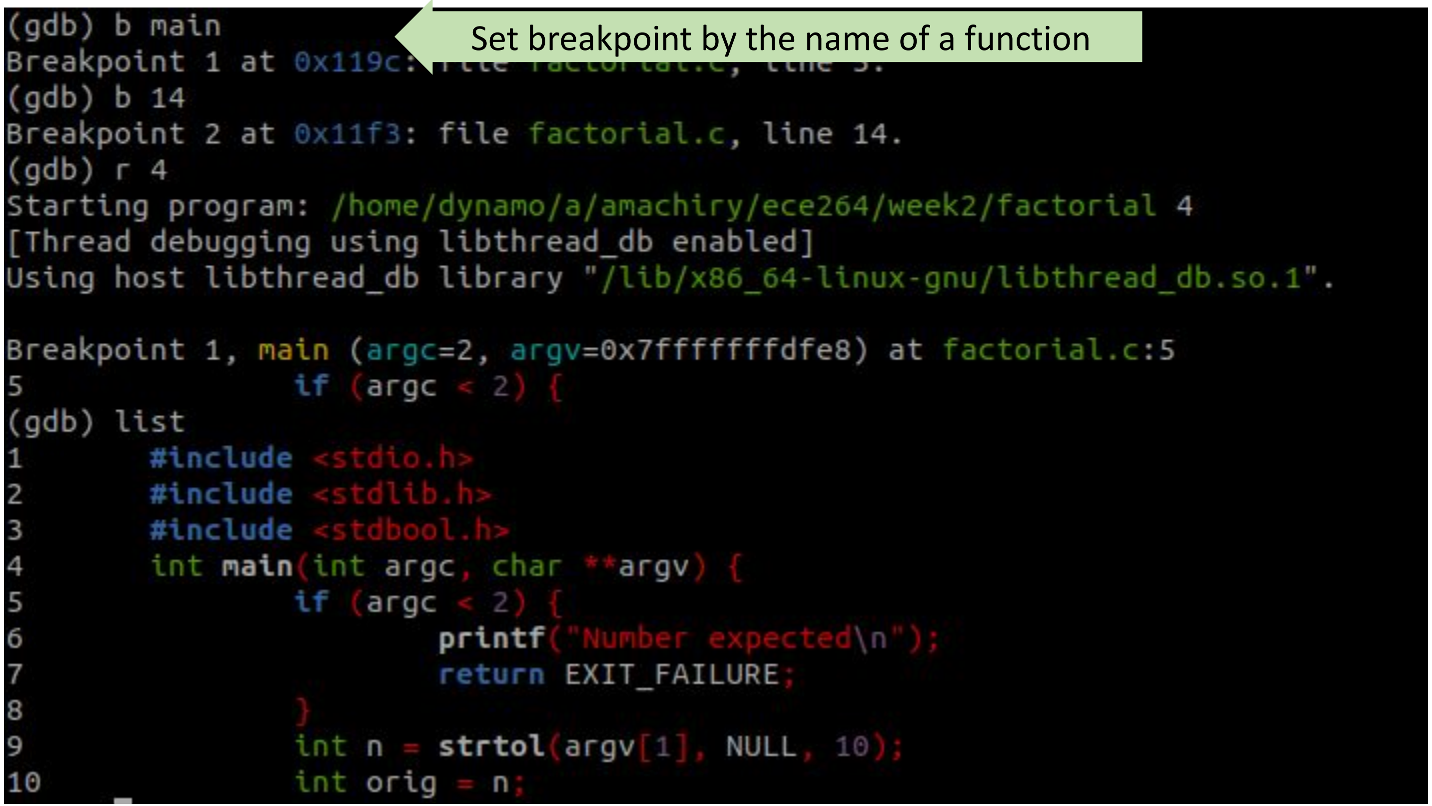
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Number expected\n");
        return EXIT_FAILURE;
    }
    int n = strtol(argv[1], NULL, 10);
    int orig = n;
    unsigned int f = 1;
    while (n >= 0) {
        f = f * n;
        n--;
    }
    printf("Number=%d, Factorial=%u\n", orig, f);
    return EXIT_SUCCESS;
}
```

-g after gcc enables debugging



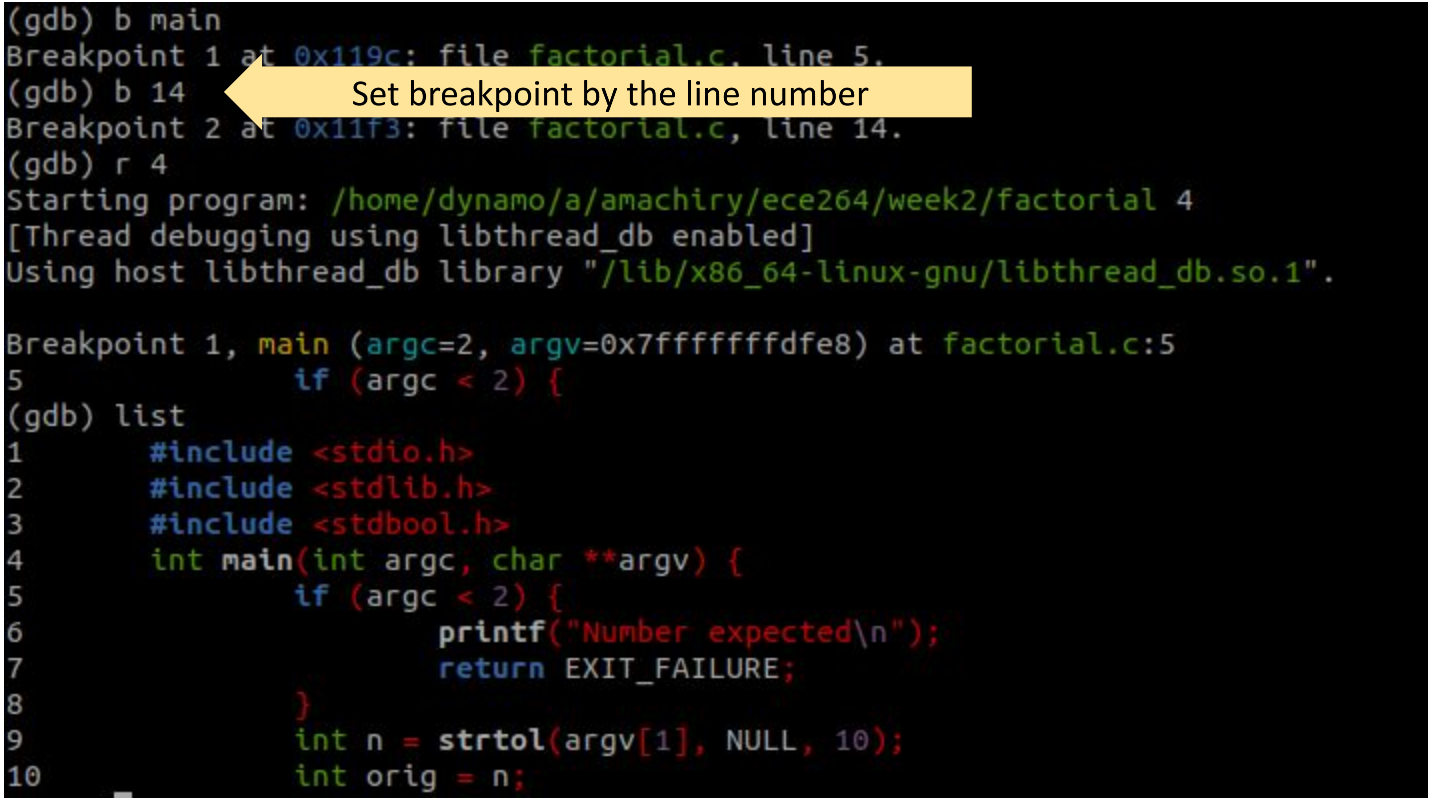
```
amachiry@eceprog5:~/ece264/week2$ gcc -Wall -g factorial.c -o factorial
amachiry@eceprog5:~/ece264/week2$ gdb factorial
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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 factorial...
(gdb) 
```

gdb the executable (not .c)



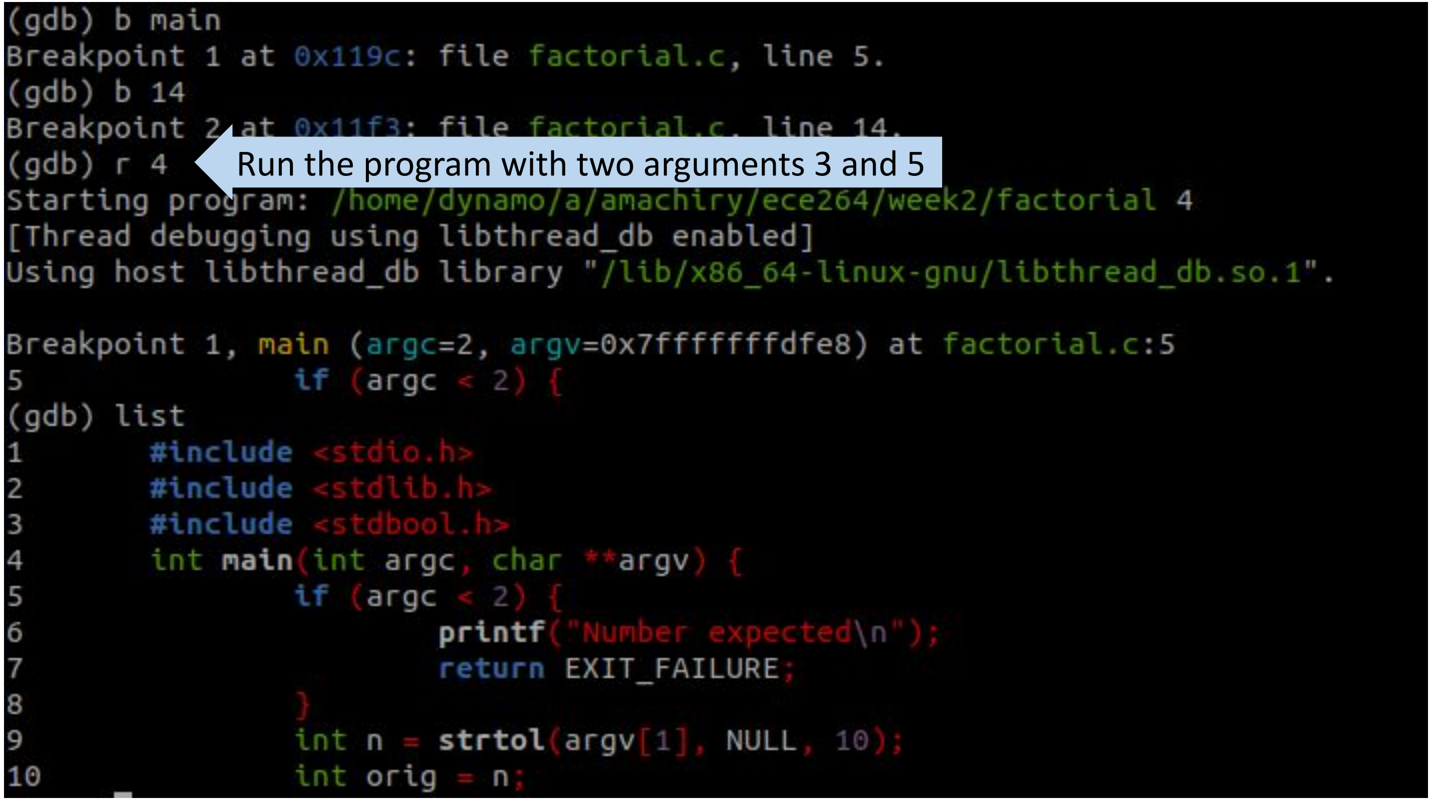
Set breakpoint by the name of a function

```
(gdb) b main
Breakpoint 1 at 0x119c: file factorial.c, line 5.
(gdb) b 14
Breakpoint 2 at 0x11f3: file factorial.c, line 14.
(gdb) r 4
Starting program: /home/dynamo/a/amachiry/ece264/week2/factorial 4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffdfe8) at factorial.c:5
5               if (argc < 2) {
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <stdbool.h>
4       int main(int argc, char **argv) {
5               if (argc < 2) {
6                       printf("Number expected\n");
7                       return EXIT_FAILURE;
8               }
9               int n = strtol(argv[1], NULL, 10);
10              int orig = n;
```



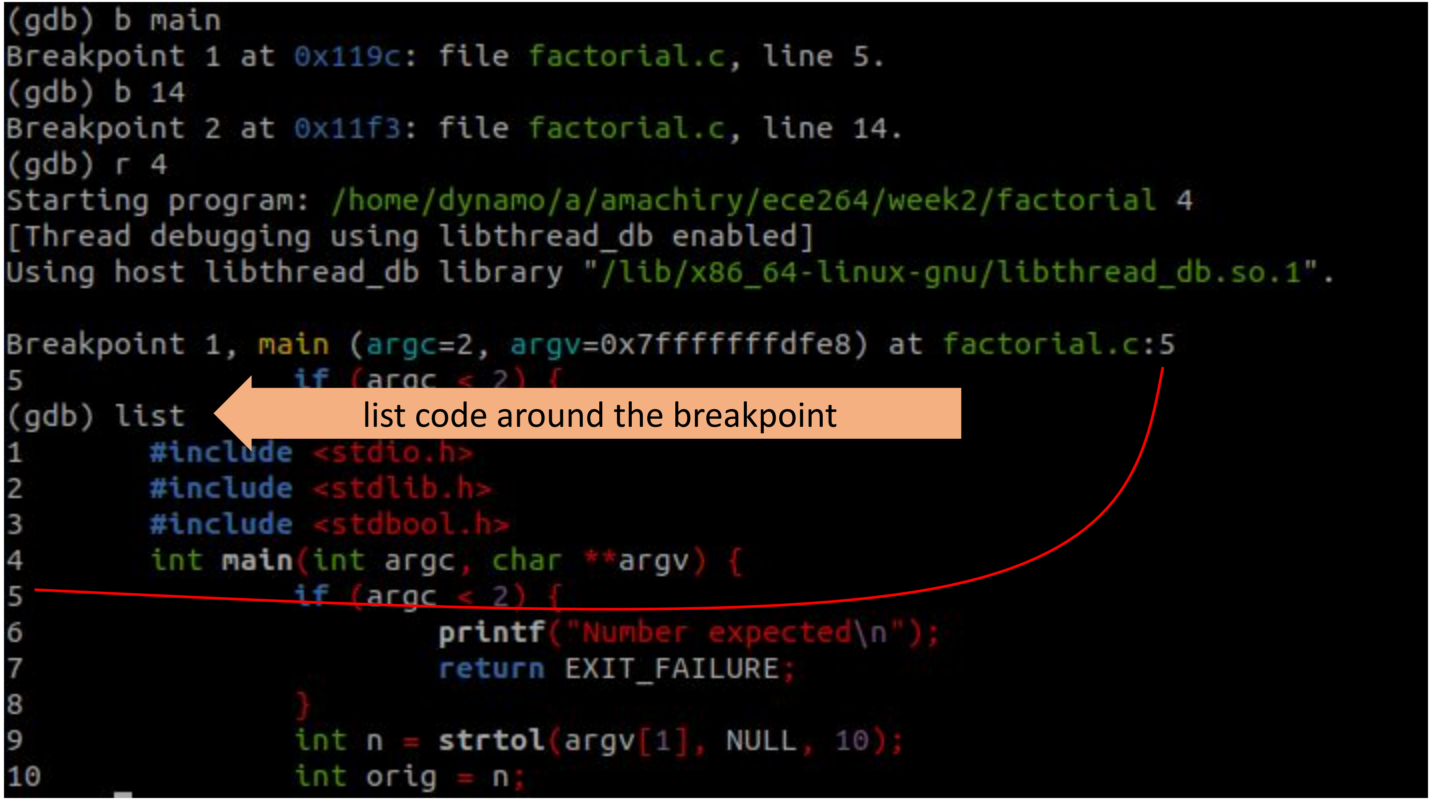
```
(gdb) b main
Breakpoint 1 at 0x119c: file factorial.c, line 5.
(gdb) b 14
Breakpoint 2 at 0x11f3: file factorial.c, line 14.
(gdb) r 4
Starting program: /home/dynamo/a/amachiry/ece264/week2/factorial 4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffdfe8) at factorial.c:5
5               if (argc < 2) {
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <stdbool.h>
4       int main(int argc, char **argv) {
5               if (argc < 2) {
6                       printf("Number expected\n");
7                       return EXIT_FAILURE;
8               }
9               int n = strtol(argv[1], NULL, 10);
10              int orig = n;
```

Set breakpoint by the line number



```
(gdb) b main
Breakpoint 1 at 0x119c: file factorial.c, line 5.
(gdb) b 14
Breakpoint 2 at 0x11f3: file factorial.c, line 14.
(gdb) r 4
Starting program: /home/dynamo/a/amachiry/ece264/week2/factorial 4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffdfe8) at factorial.c:5
5               if (argc < 2) {
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <stdbool.h>
4       int main(int argc, char **argv) {
5               if (argc < 2) {
6                       printf("Number expected\n");
7                       return EXIT_FAILURE;
8               }
9               int n = strtol(argv[1], NULL, 10);
10              int orig = n;
```

Run the program with two arguments 3 and 5



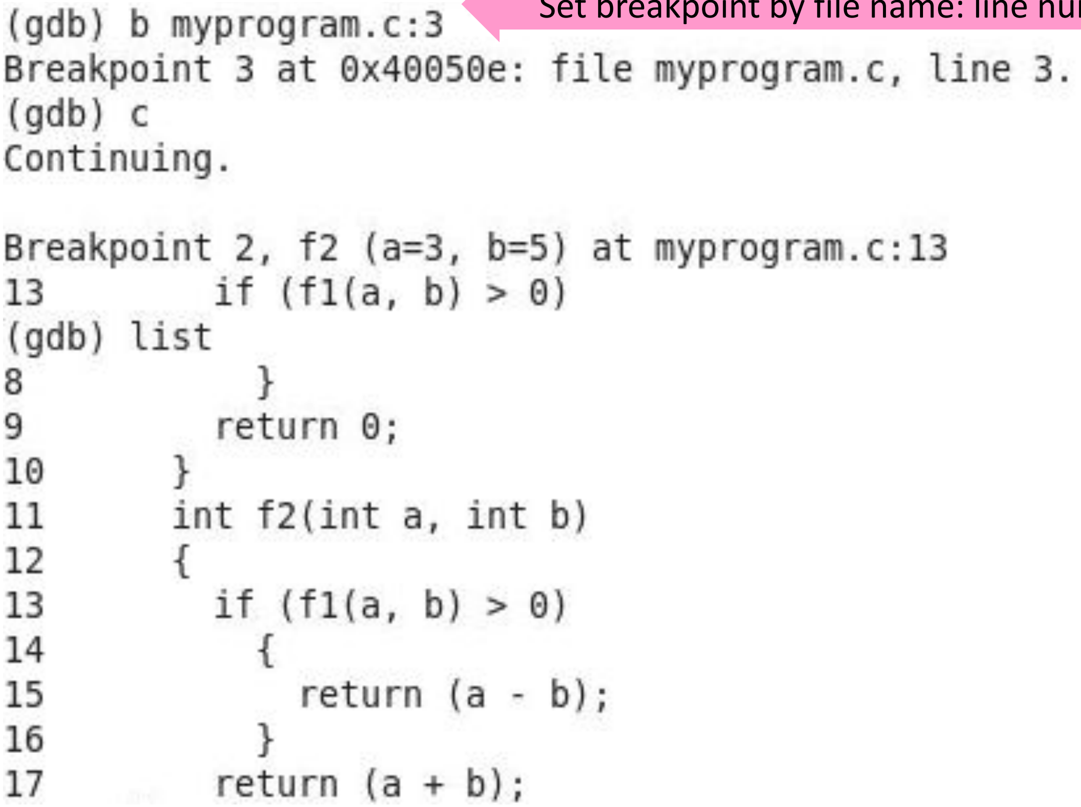
```
(gdb) b main
Breakpoint 1 at 0x119c: file factorial.c, line 5.
(gdb) b 14
Breakpoint 2 at 0x11f3: file factorial.c, line 14.
(gdb) r 4
Starting program: /home/dynamo/a/amachiry/ece264/week2/factorial 4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffdfe8) at factorial.c:5
5               if (argc < 2) {
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <stdbool.h>
4       int main(int argc, char **argv) {
5               if (argc < 2) {
6                       printf("Number expected\n");
7                       return EXIT_FAILURE;
8               }
9               int n = strtol(argv[1], NULL, 10);
10              int orig = n;
```

list code around the breakpoint

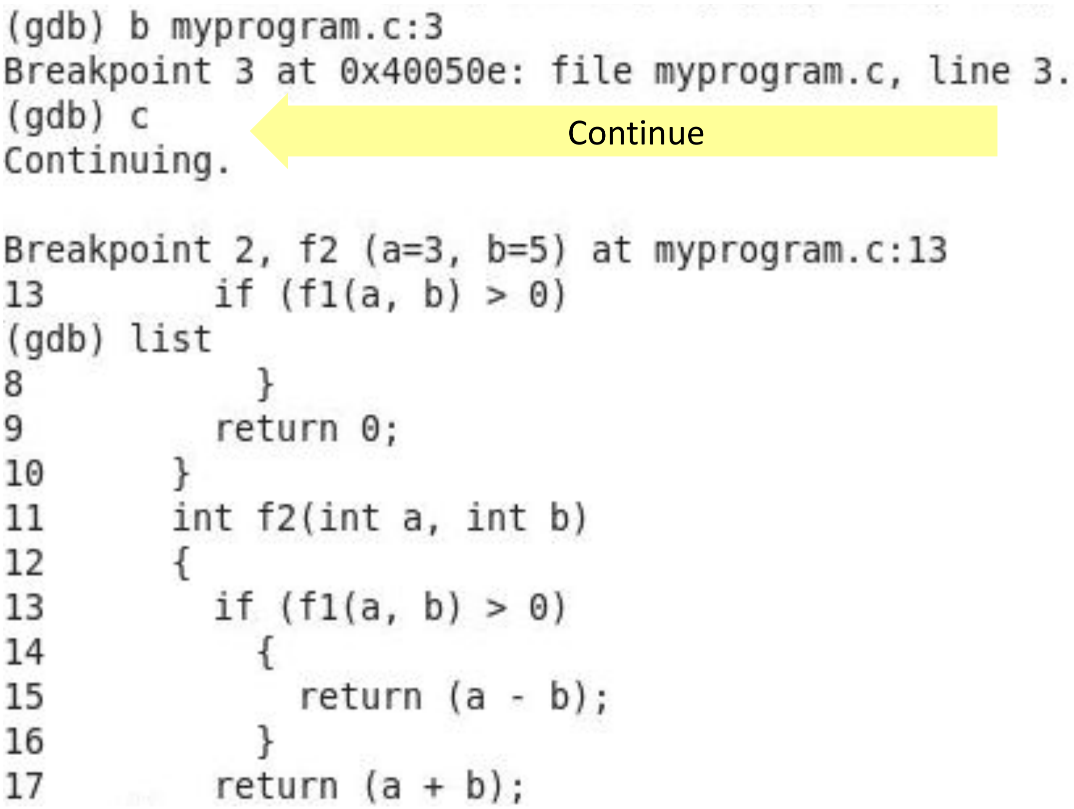
gdb commands

- b: set a breakpoint
- r: run the program
- list: list the code

Set breakpoint by file name: line number

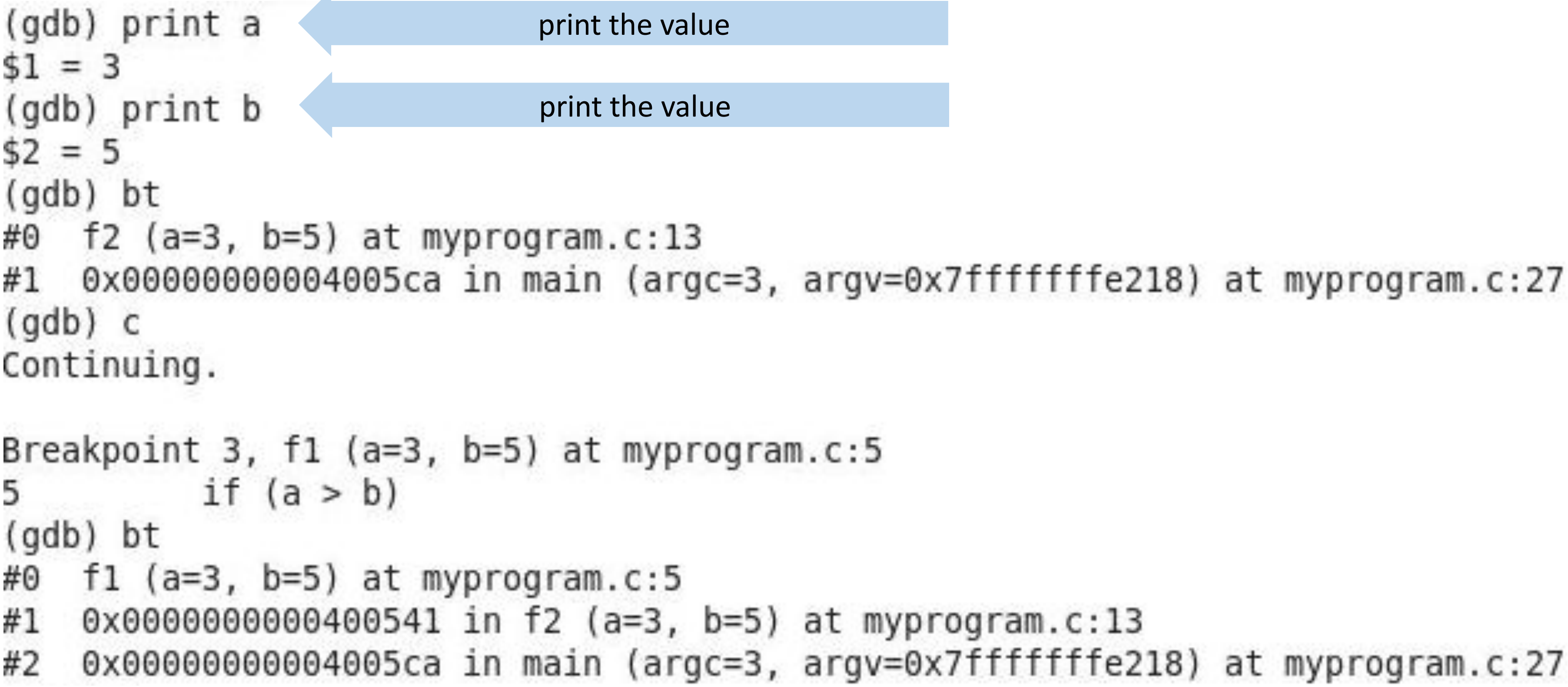


```
(gdb) b myprogram.c:3
Breakpoint 3 at 0x40050e: file myprogram.c, line 3.
(gdb) c
Continuing.

Breakpoint 2, f2 (a=3, b=5) at myprogram.c:13
13          if (f1(a, b) > 0)
(gdb) list
8             }
9           return 0;
10        }
11        int f2(int a, int b)
12        {
13          if (f1(a, b) > 0)
14            {
15              return (a - b);
16            }
17          return (a + b);
```



```
(gdb) b myprogram.c:3
Breakpoint 3 at 0x40050e: file myprogram.c, line 3.
(gdb) c
Continuing.

Breakpoint 2, f2 (a=3, b=5) at myprogram.c:13
13	  if (f1(a, b) > 0)
(gdb) list
8	    }
9	  return 0;
10	}
11	int f2(int a, int b)
12	{
13	  if (f1(a, b) > 0)
14	    {
15	      return (a - b);
16	    }
17	  return (a + b);
```

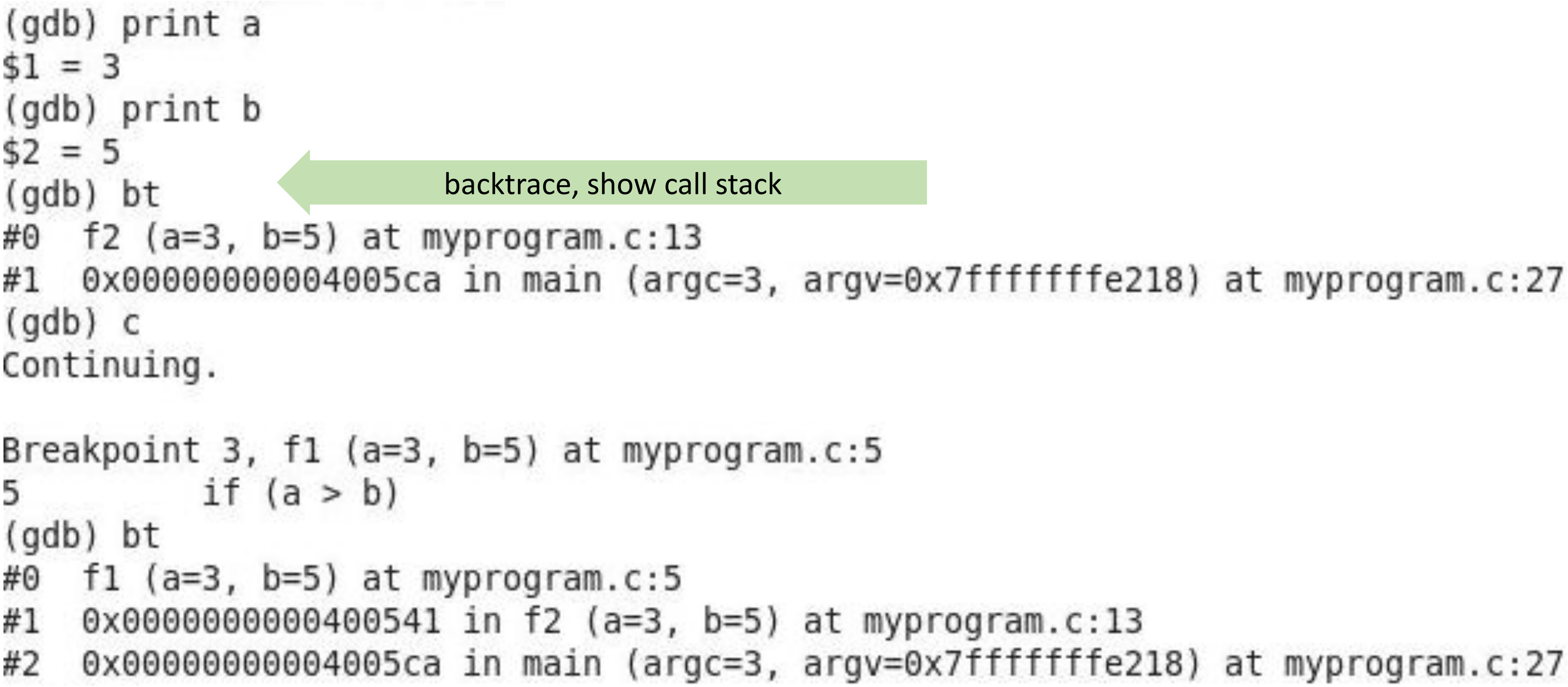
Continue



```
(gdb) print a
$1 = 3
(gdb) print b
$2 = 5
(gdb) bt
#0  f2 (a=3, b=5) at myprogram.c:13
#1  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
(gdb) c
Continuing.

Breakpoint 3, f1 (a=3, b=5) at myprogram.c:5
5           if (a > b)
(gdb) bt
#0  f1 (a=3, b=5) at myprogram.c:5
#1  0x0000000000400541 in f2 (a=3, b=5) at myprogram.c:13
#2  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
```

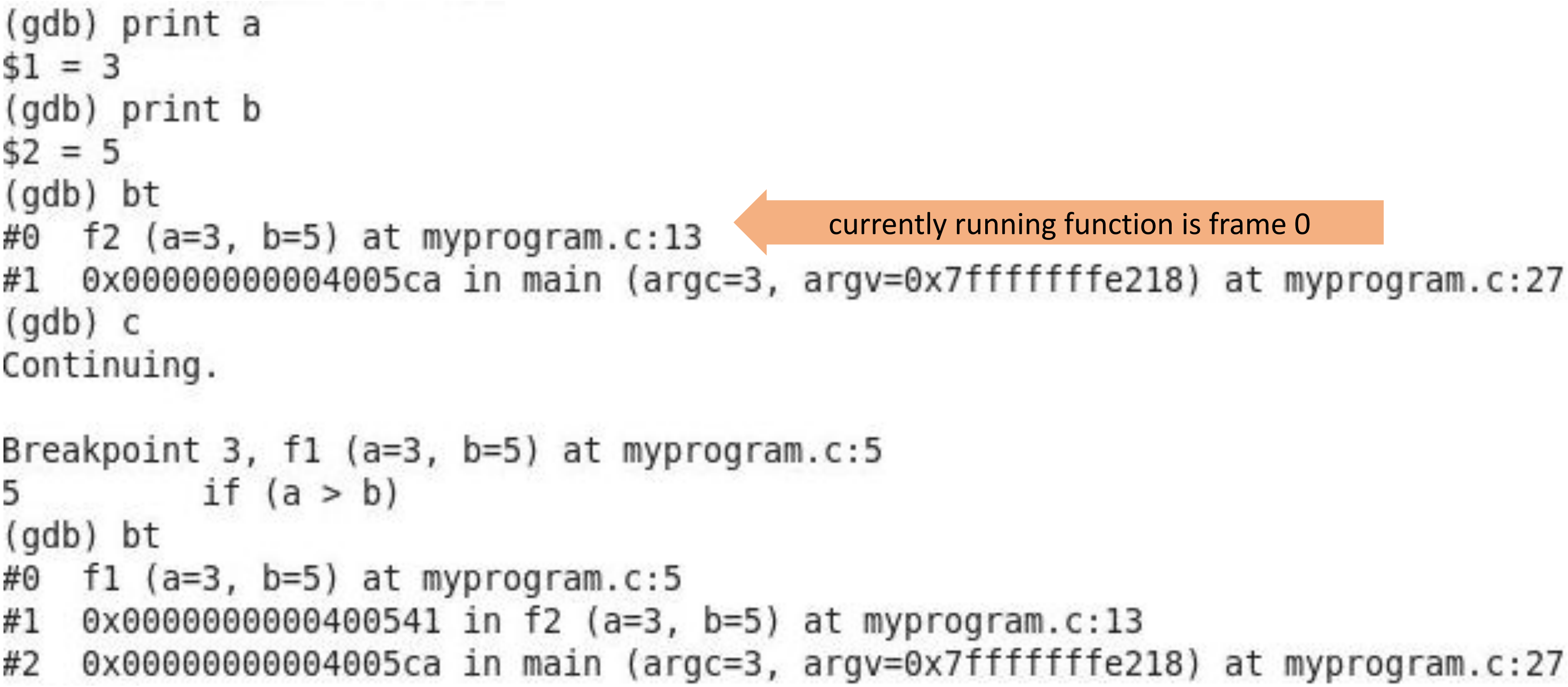
print the value

print the value



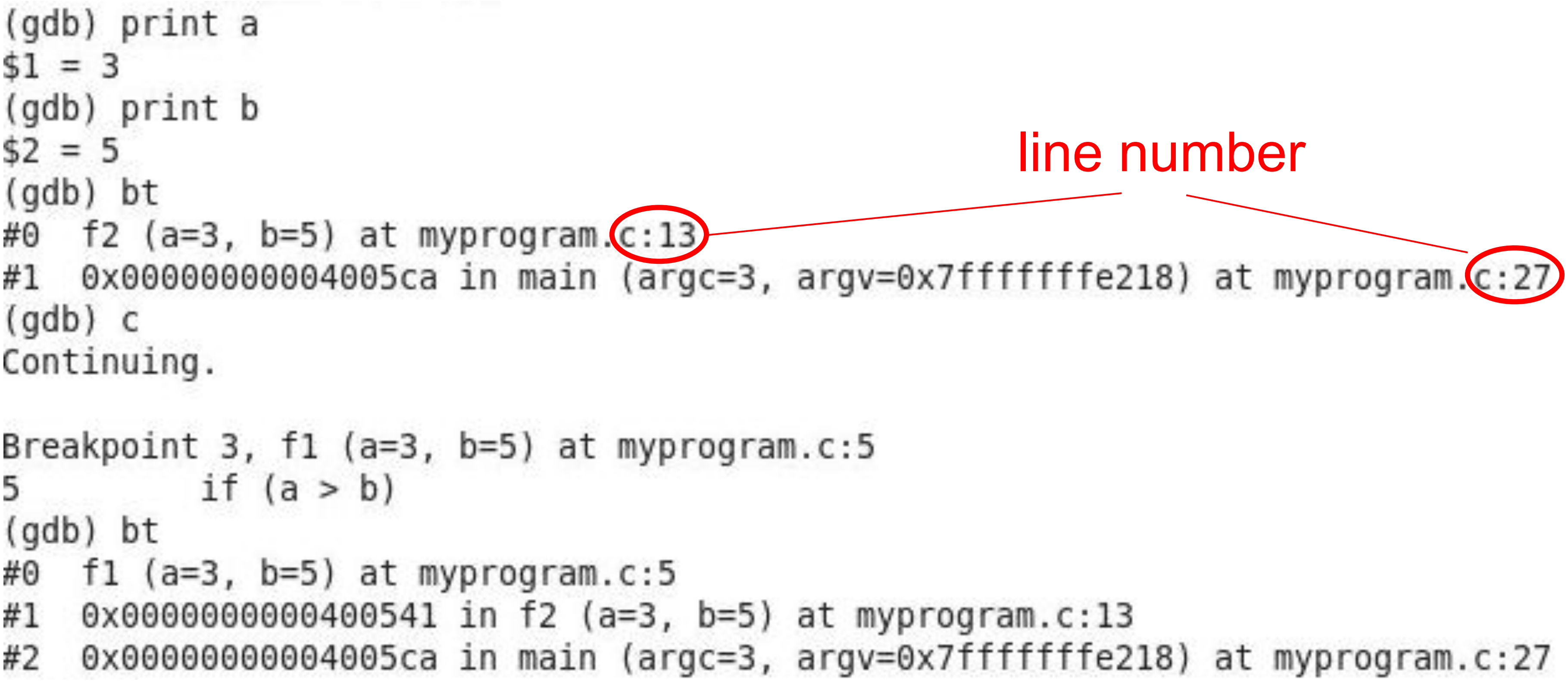
```
(gdb) print a
$1 = 3
(gdb) print b
$2 = 5
(gdb) bt
#0  f2 (a=3, b=5) at myprogram.c:13
#1  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
(gdb) c
Continuing.

Breakpoint 3, f1 (a=3, b=5) at myprogram.c:5
5         if (a > b)
(gdb) bt
#0  f1 (a=3, b=5) at myprogram.c:5
#1  0x0000000000400541 in f2 (a=3, b=5) at myprogram.c:13
#2  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
```

backtrace, show call stack

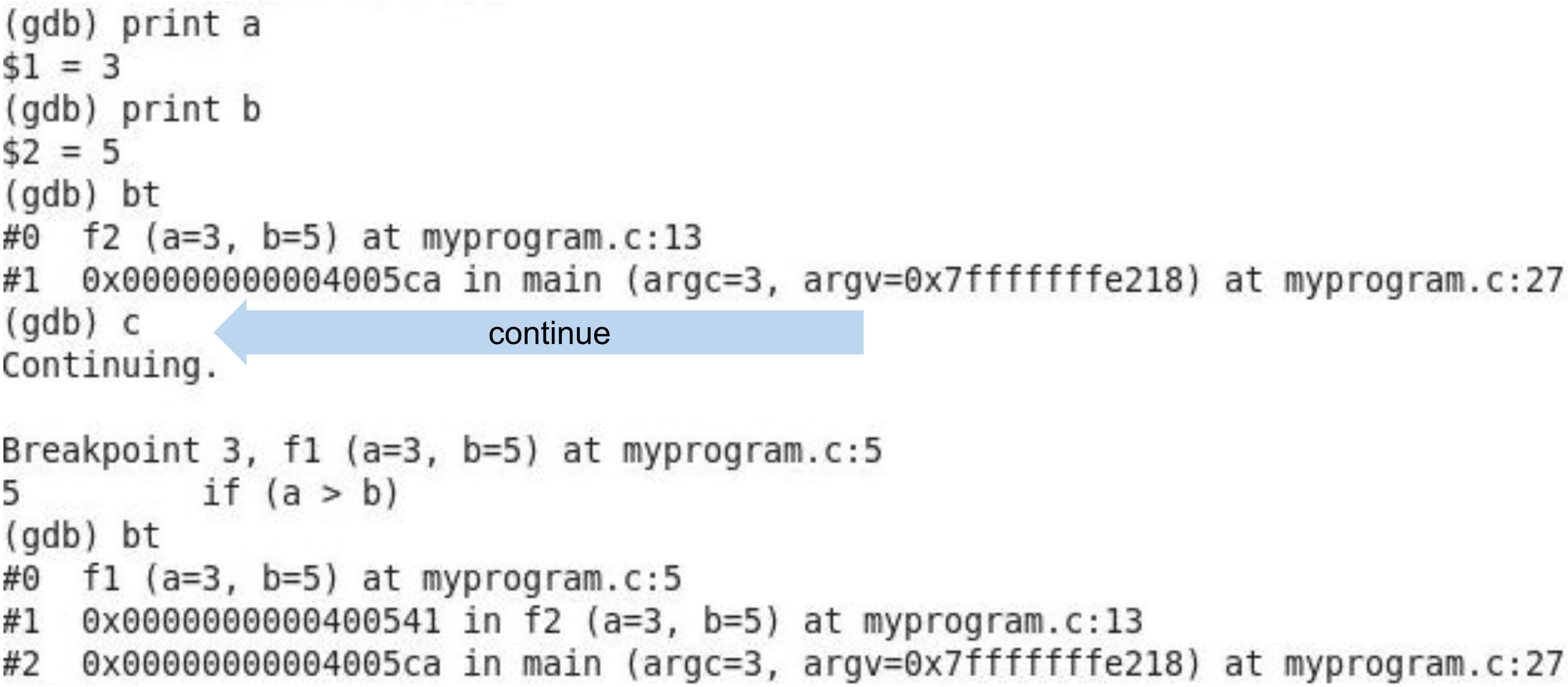


```
(gdb) print a
$1 = 3
(gdb) print b
$2 = 5
(gdb) bt
#0  f2 (a=3, b=5) at myprogram.c:13
#1  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
(gdb) c
Continuing.

Breakpoint 3, f1 (a=3, b=5) at myprogram.c:5
5         if (a > b)
(gdb) bt
#0  f1 (a=3, b=5) at myprogram.c:5
#1  0x0000000000400541 in f2 (a=3, b=5) at myprogram.c:13
#2  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
```

currently running function is frame 0

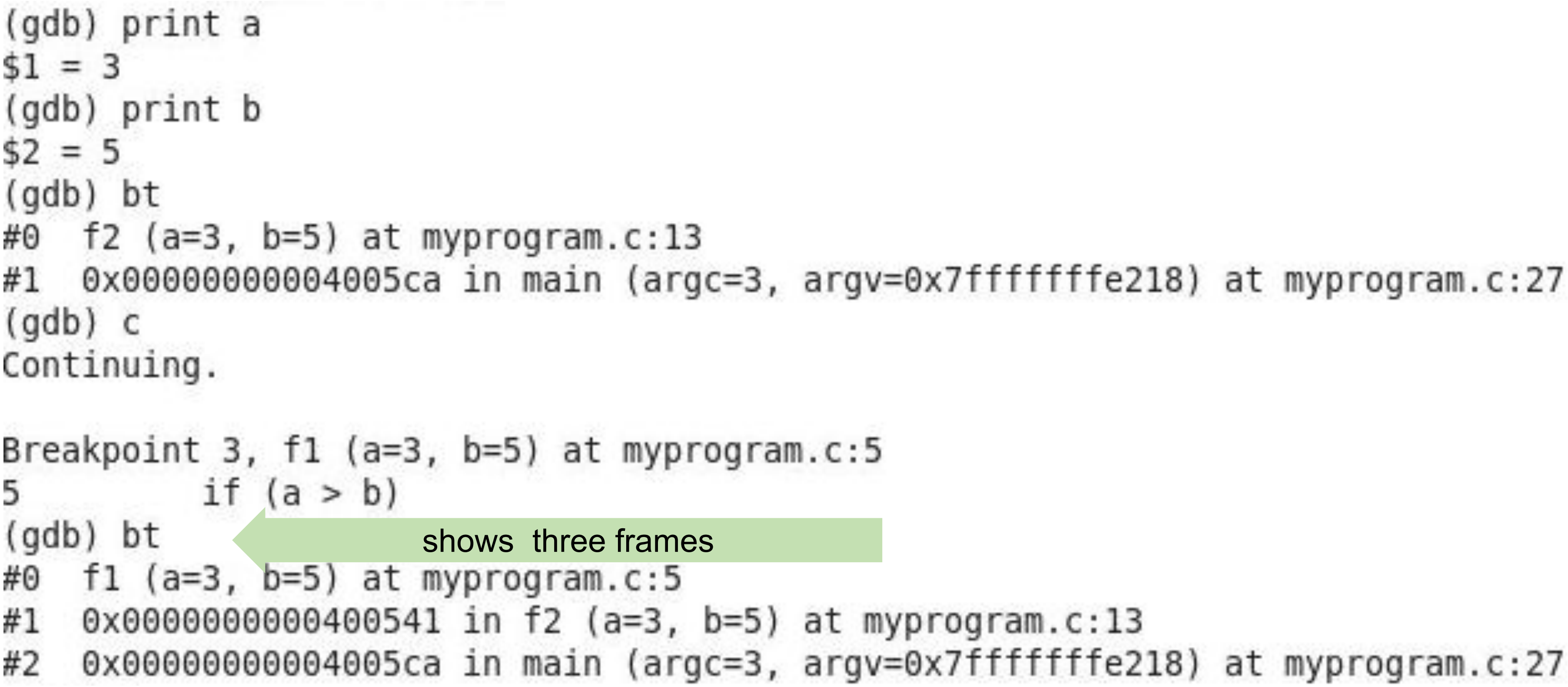


```
(gdb) print a
$1 = 3
(gdb) print b
$2 = 5
(gdb) bt
#0  f2 (a=3, b=5) at myprogram.c:13
#1  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
(gdb) c
Continuing.

Breakpoint 3, f1 (a=3, b=5) at myprogram.c:5
5         if (a > b)
(gdb) bt
#0  f1 (a=3, b=5) at myprogram.c:5
#1  0x0000000000400541 in f2 (a=3, b=5) at myprogram.c:13
#2  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
```



```
(gdb) print a
$1 = 3
(gdb) print b
$2 = 5
(gdb) bt
#0  f2 (a=3, b=5) at myprogram.c:13
#1  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
(gdb) c
Continuing.

Breakpoint 3, f1 (a=3, b=5) at myprogram.c:5
5         if (a > b)
(gdb) bt
#0  f1 (a=3, b=5) at myprogram.c:5
#1  0x0000000000400541 in f2 (a=3, b=5) at myprogram.c:13
#2  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
```

continue



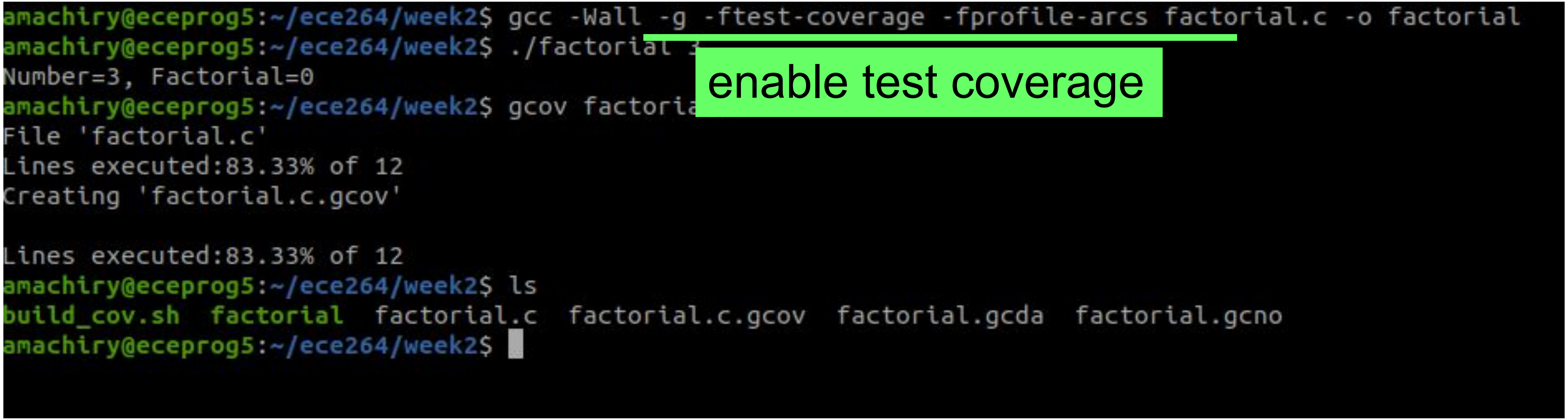
```
(gdb) print a
$1 = 3
(gdb) print b
$2 = 5
(gdb) bt
#0  f2 (a=3, b=5) at myprogram.c:13
#1  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
(gdb) c
Continuing.

Breakpoint 3, f1 (a=3, b=5) at myprogram.c:5
5         if (a > b)
(gdb) bt
#0  f1 (a=3, b=5) at myprogram.c:5
#1  0x0000000000400541 in f2 (a=3, b=5) at myprogram.c:13
#2  0x00000000004005ca in main (argc=3, argv=0x7fffffffe218) at myprogram.c:27
```

shows three frames

gdb commands

- print: print a variable
- c: continue to the next breakpoint
- bt: show call stack
- b: set a breakpoint
- r: run the program
- list: list the code

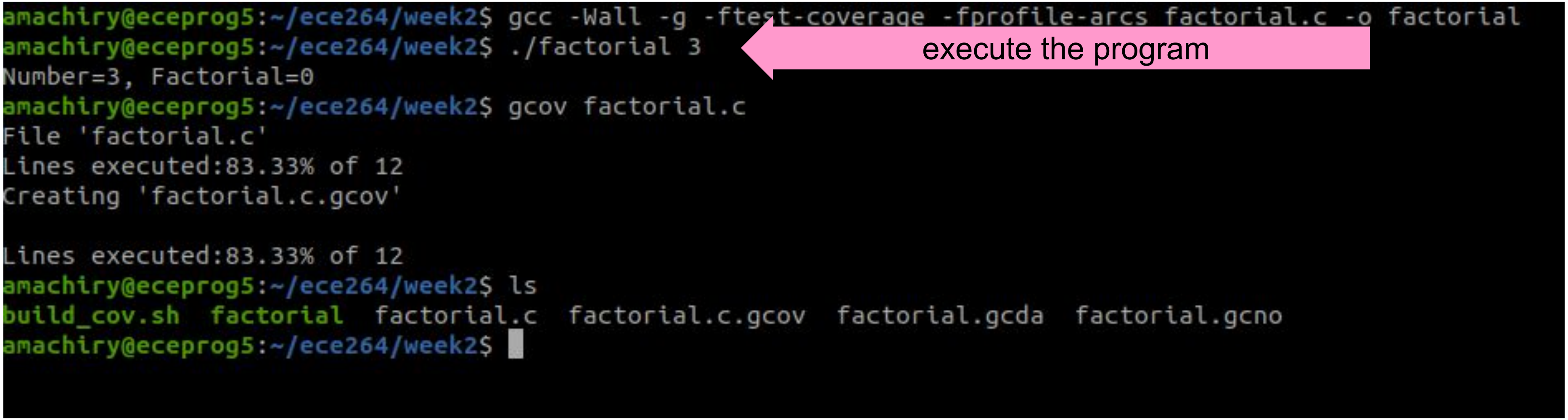
test coverage



```
amachiry@eceprog5:~/ece264/week2$ gcc -Wall -g -ftest-coverage -fprofile-arcs factorial.c -o factorial
amachiry@eceprog5:~/ece264/week2$ ./factorial 3
Number=3, Factorial=0
amachiry@eceprog5:~/ece264/week2$ gcov factoria
File 'factorial.c'
Lines executed:83.33% of 12
Creating 'factorial.c.gcov'

Lines executed:83.33% of 12
amachiry@eceprog5:~/ece264/week2$ ls
build_cov.sh  factorial  factorial.c  factorial.c.gcov  factorial.gcda  factorial.gcno
amachiry@eceprog5:~/ece264/week2$ 
```

enable test coverage

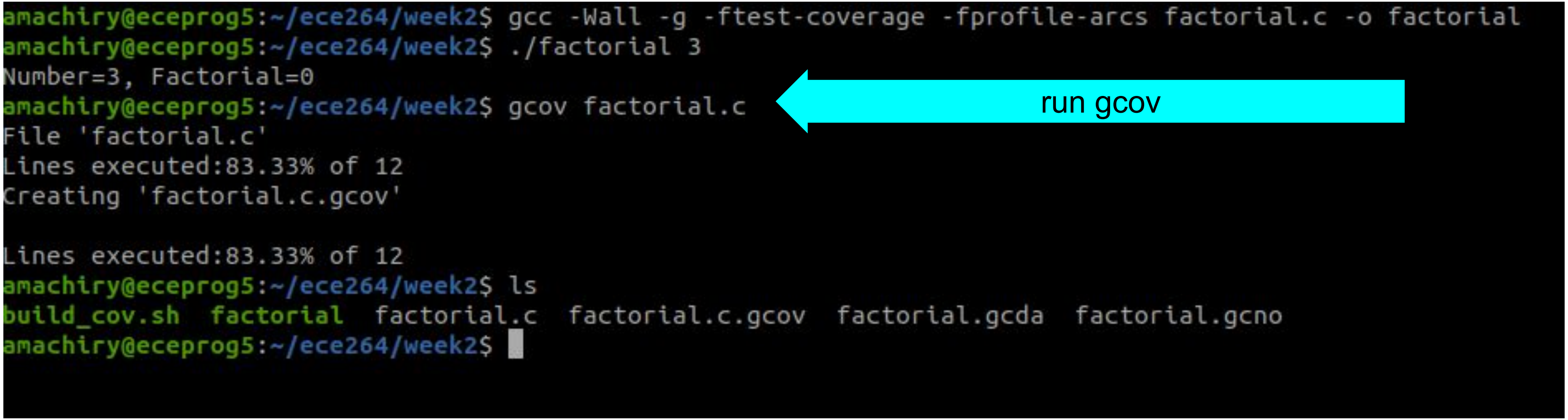
test coverage



```
amachiry@eceprog5:~/ece264/week2$ gcc -Wall -g -ftest-coverage -fprofile-arcs factorial.c -o factorial
amachiry@eceprog5:~/ece264/week2$ ./factorial 3
Number=3, Factorial=0
amachiry@eceprog5:~/ece264/week2$ gcov factorial.c
File 'factorial.c'
Lines executed:83.33% of 12
Creating 'factorial.c.gcov'

Lines executed:83.33% of 12
amachiry@eceprog5:~/ece264/week2$ ls
build_cov.sh  factorial  factorial.c  factorial.c.gcov  factorial.gcda  factorial.gcno
amachiry@eceprog5:~/ece264/week2$ 
```

execute the program

test coverage



```
amachiry@eceprog5:~/ece264/week2$ gcc -Wall -g -ftest-coverage -fprofile-arcs factorial.c -o factorial
amachiry@eceprog5:~/ece264/week2$ ./factorial 3
Number=3, Factorial=0
amachiry@eceprog5:~/ece264/week2$ gcov factorial.c
File 'factorial.c'
Lines executed:83.33% of 12
Creating 'factorial.c.gcov'

Lines executed:83.33% of 12
amachiry@eceprog5:~/ece264/week2$ ls
build_cov.sh  factorial  factorial.c  factorial.c.gcov  factorial.gcda  factorial.gcno
amachiry@eceprog5:~/ece264/week2$ 
```

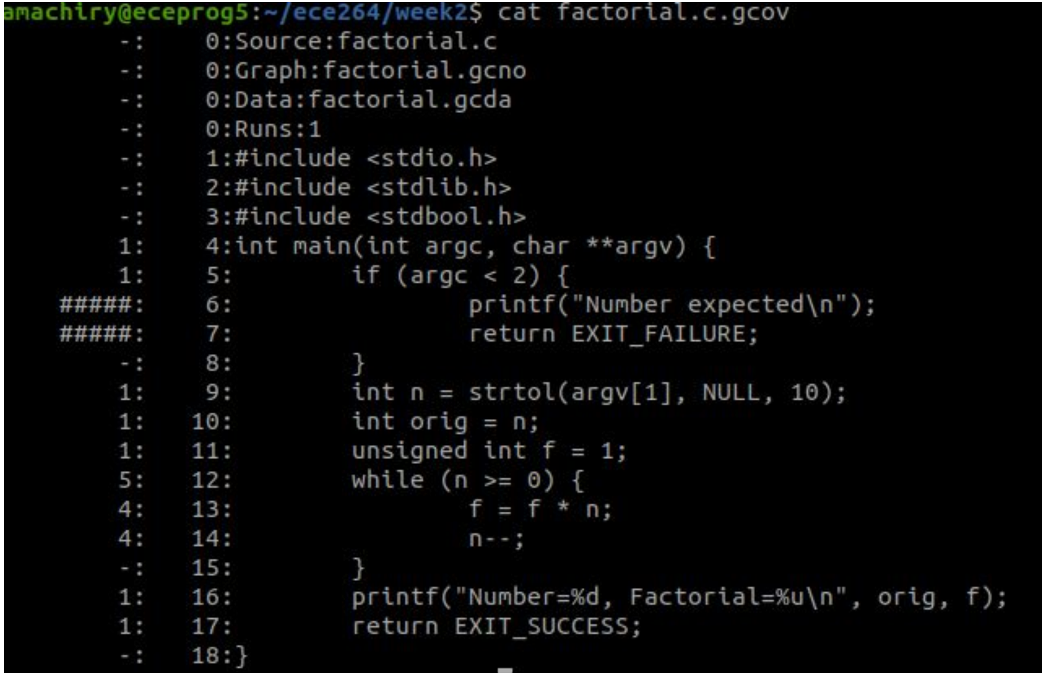
run gcov

test coverage

```
amachiry@eceprog5:~/ece264/week2$ gcc -Wall -g -ftest-coverage -fprofile-arcs factorial.c -o factorial
amachiry@eceprog5:~/ece264/week2$ ./factorial 3
Number=3, Factorial=0
amachiry@eceprog5:~/ece264/week2$ gcov factorial.c
File 'factorial.c'
Lines executed:83.33% of 12
Creating 'factorial.c.gcov'

Lines executed:83.33% of 12
amachiry@eceprog5:~/ece264/week2$ ls
build_cov.sh  factorial  factorial.c  factorial.c.gcov  factorial.gcda  factorial.gcno
amachiry@eceprog5:~/ece264/week2$ 
```

generated files

means untested

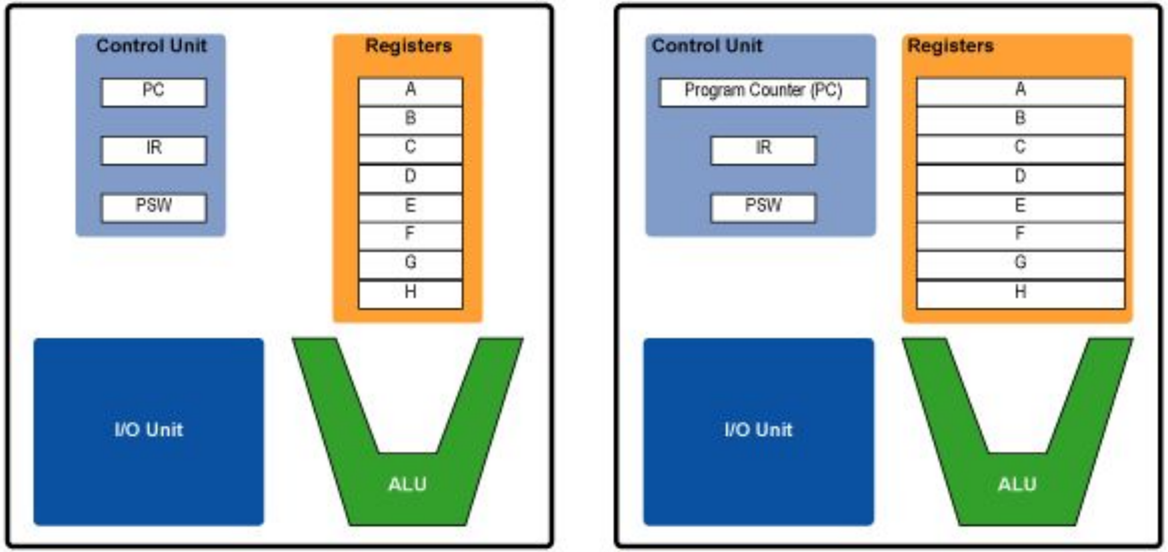


```
amachiry@eceprog5:~/ece264/week2$ cat factorial.c.gcov
        -:    0:Source:factorial.c
        -:    0:Graph:factorial.gcno
        -:    0:Data:factorial.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:#include <stdlib.h>
        -:    3:#include <stdbool.h>
        1:    4:int main(int argc, char **argv) {
        1:    5:        if (argc < 2) {
    #####:    6:                printf("Number expected\n");
    #####:    7:                return EXIT_FAILURE;
        -:    8:        }
        1:    9:        int n = strtol(argv[1], NULL, 10);
        1:   10:        int orig = n;
        1:   11:        unsigned int f = 1;
        5:   12:        while (n >= 0) {
        4:   13:                f = f * n;
        4:   14:                n--;
        -:   15:        }
        1:   16:        printf("Number=%d, Factorial=%u\n", orig, f);
        1:   17:        return EXIT_SUCCESS;
        -:   18:}
```

5: Some lines have been tested twice.

Where does a C program execute?

Processor (CPU)



32-bit **64-bit**

32-bit or 64-bit registers

The Need for Memory

```
int main() {
  char buff[4096];
  printf("Hello World\n");
  return EXIT_SUCCESS;
}
```

The Need for Memory

- CPU has limited registers!
- A program might need more data than that can be stored in registers.
- Where do we store additional data?

Different kinds of memory

- Main memory or RAM.
  - Fast.
  - Only available during a program execution.
  - Relatively expensive.

- Secondary storage or Disk.
  - Slow.
  - Available for the entire lifetime of the disk.
  - Cheap (Flash drive, External drive).

Accessing Memory

- Main memory or RAM:
  - Accessed in terms of bytes.
  - Each byte has an address.
  - Address:
    - 32 or 64-bit number depending on the size of registers.

Memory Size

- Maximum number of bytes in Main memory?
  - 32-bit Addresses?
  - 64-bit Addresses?

Memory Size

- Maximum number of bytes in Main memory?
  - 32-bit Addresses? 2^32
  - 64-bit Addresses? 2^64

Memory Sizes

- Secondary storage or Disk.
  - Unlimited.

Main Memory

- Every program has access to the entire main memory.
  - 2^64 bytes (mostly less because some memory will be used for operating system).
  - Virtual Memory:
    - Address in one program is different from address in another program.

Main Memory

- What do we need memory for?

Main Memory

- What do we need memory for?
  - To store instructions of the program.
  - To store local variables.
  - To store global variables.
  - To store heap (allocated through malloc).

Main Memory

- What do we need memory for?
  - To store instructions of the program:
    - Available for the entire lifetime of program.
    - Read-only (We do not modify instructions)

Main Memory

- What do we need memory for?
  - To store instructions of the program:
    - Available for the entire lifetime of program (readonly).
  - To store local variables:
    - Available only during the function execution.

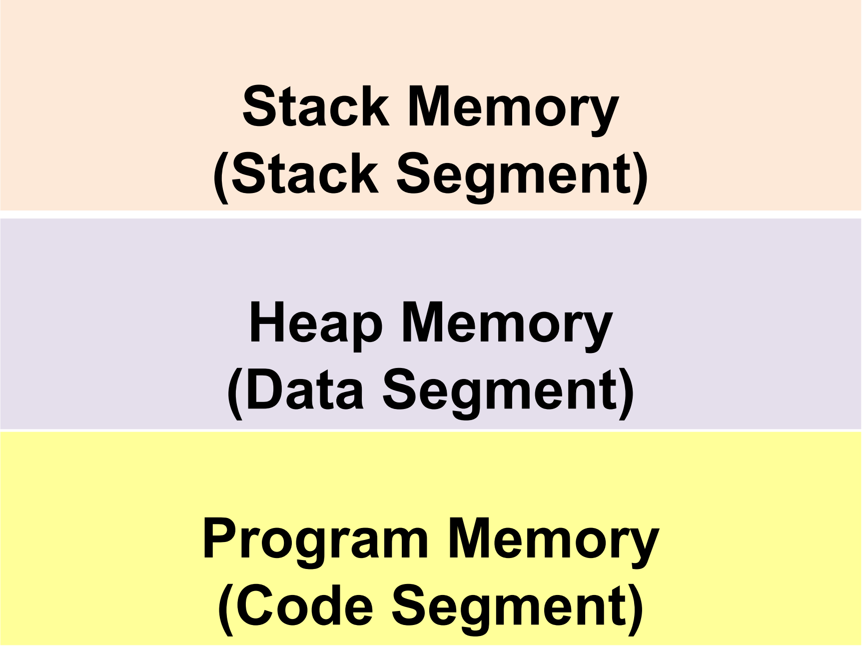
Main Memory

- What do we need memory for?
  - To store instructions of the program:
    - Available for the entire lifetime of program (readonly).
  - To store local variables:
    - Available only during the function execution.
  - To store global variables:
    - Available for the entire lifetime of program.

Main Memory

- What do we need memory for?
  - To store instructions of the program:
    - Available for the entire lifetime of program (readonly).
  - To store local variables:
    - Available only during the function execution.
  - To store global variables:
    - Available for the entire lifetime of program.
  - To store heap (allocated through malloc).
    - Available for the entire lifetime of program.

Types of Program Memory



Stack Memory
(Stack Segment)

Heap Memory
(Data Segment)

Program Memory
(Code Segment)

Types of Program Memory

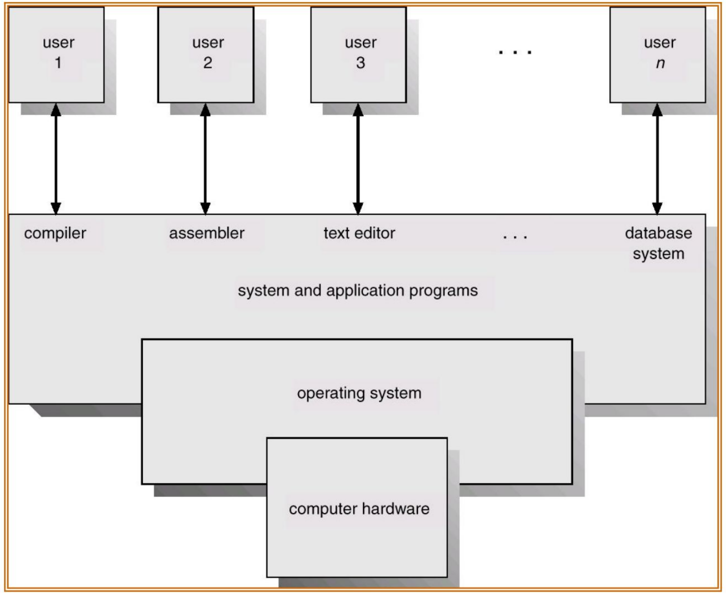
Stack Memory (Stack Segment)	Used to store local variables and return addresses.
Heap Memory (Data Segment)	Used to store global variables and malloced buffers.
Program Memory (Code Segment)	To Store instructions.

Memory Management

- Every program has access to entire memory (2^64 bytes) = 16 Million GB
  - I have only 16 GB RAM!! How can run a program?
  - Can I run multiple programs?

Operating System

- Operating System mediates all access to hardware (e.g., memory) and gives an illusion that every program has 2^64 bytes.



How often we interact with OS?

Memory Allocation

- Every program has access to entire memory (2^64 bytes) = 16 Million GB.

- Can we access it freely?
  - NO! Why?

Memory Allocation

- OS allocates memory on request and also on demand.

- We need to ask OS to allocate our memory!

Memory Allocation

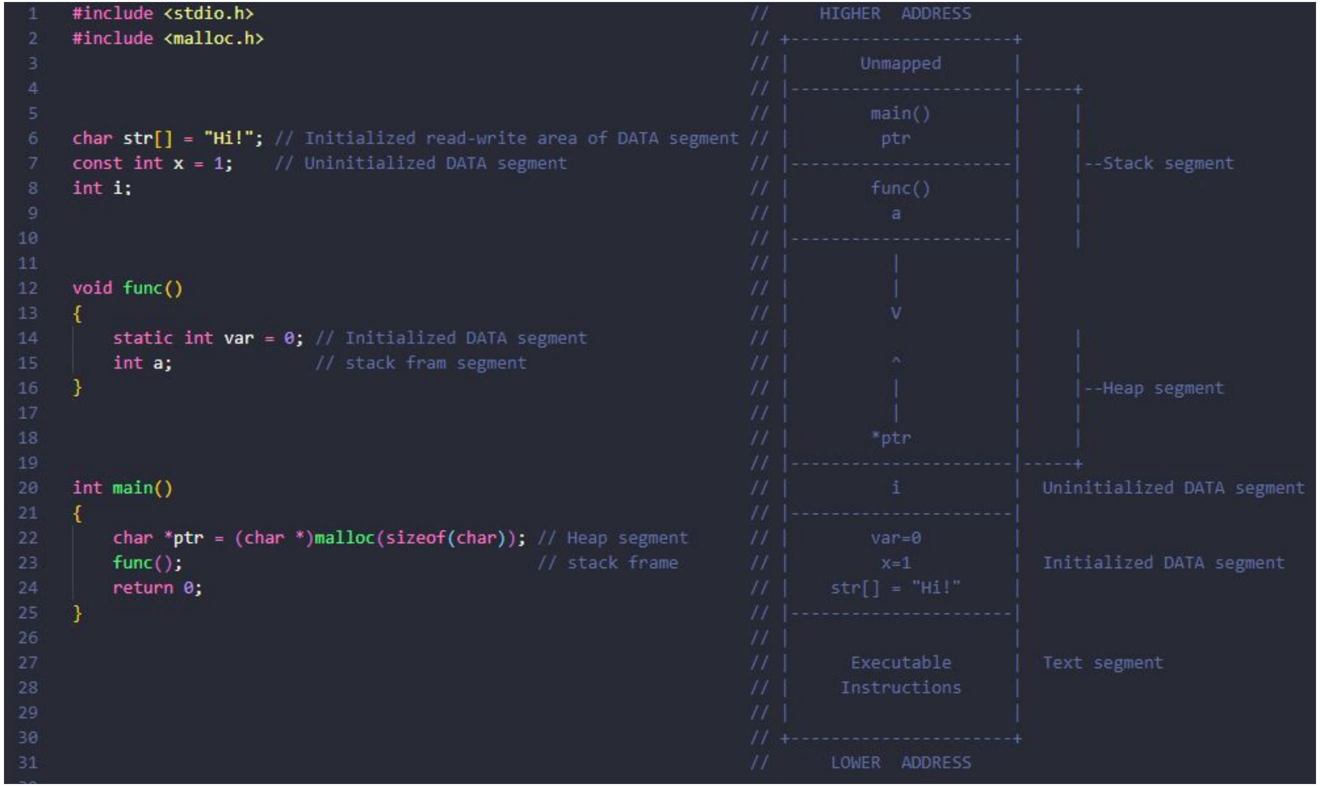
- What happens if OS always allocates entire 2^64 bytes to all programs?

- Small programs v/s large programs?

Types of Program Memory

Stack Memory (Stack Segment)	**Allocated on Demand (When a function starts).**
Heap Memory (Data Segment)	**Allocated on Request.**
Program Memory (Code Segment)	**Allocated at the Beginning.**

Types of Program Memory

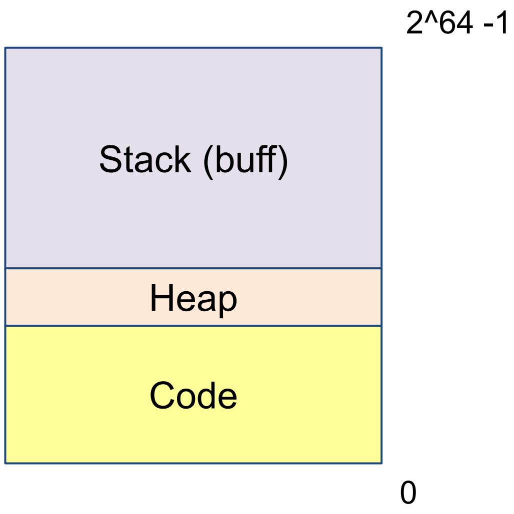


```
#include <stdio.h>                                                 //     HIGHER  ADDRESS
#include <malloc.h>                                                // +----------------------+
                                                                   // |       Unmapped       |
                                                                   // |----------------------|-----+
                                                                   // |        main()        |     |
char str[] = "Hi!"; // Initialized read-write area of DATA segment // |         ptr          |     |
const int x = 1;    // Uninitialized DATA segment                  // |----------------------|     |--Stack segment
int i;                                                             // |        func()        |     |
                                                                   // |          a           |     |
                                                                   // |----------------------|     |
                                                                   // |          |           |
void func()                                                        // |          |           |
{                                                                  // |          V           |
    static int var = 0; // Initialized DATA segment                // |                      |     |
    int a;              // stack fram segment                      // |          ^           |     |
}                                                                  // |          |           |     |--Heap segment
                                                                   // |          |           |     |
                                                                   // |        *ptr          |     |
                                                                   // |----------------------|-----+
int main()                                                         // |          i           |  Uninitialized DATA segment
{                                                                  // |----------------------|
    char *ptr = (char *)malloc(sizeof(char)); // Heap segment      // |        var=0         |
    func();                                   // stack frame       // |         x=1          |  Initialized DATA segment
    return 0;                                                      // |    str[] = "Hi!"     |
}                                                                  // |----------------------|
                                                                   // |                      |
                                                                   // |      Executable      |  Text segment
                                                                   // |     Instructions     |
                                                                   // |                      |
                                                                   // +----------------------+
                                                                   //      LOWER  ADDRESS
```

How should we allocate memory?

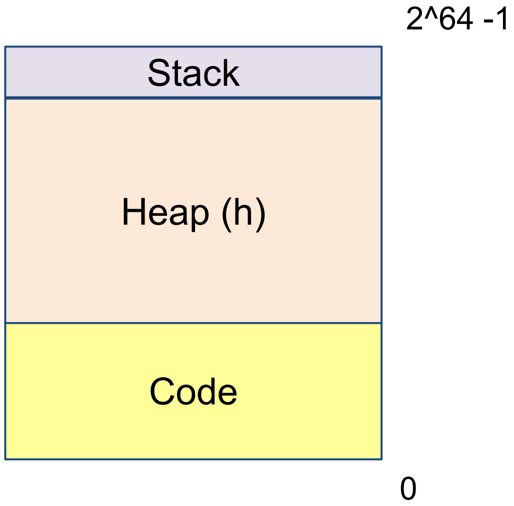
- Program 1:
  - Require **No heap** memory.
  - **Large stack** memory.
  - Code segment.

```
int main() {
  char buff[4096];
  printf("Hello World\n");
  return EXIT_SUCCESS;
}
```



How should we allocate memory?

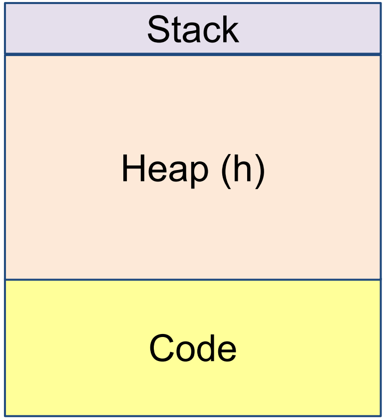
- Program 2:
  - **Large heap** memory.
  - **Small stack** memory.
  - Code segment.



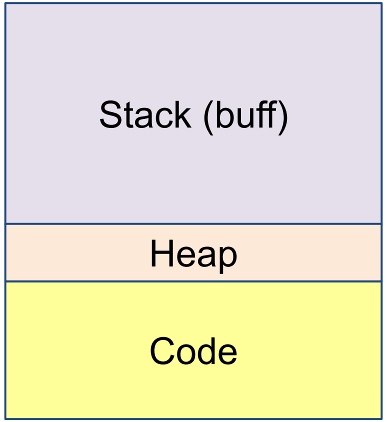
```
int main() {
   char *h = malloc(sizeof(char)*4096);
   printf("Hello World\n");
   return EXIT_SUCCESS;
}
```

How should we allocate memory?

```
int foo() {
  char buff[4096];
  ...
}
int main() {
  char *h = malloc(4096*sizeof(char));
  ...
  free(h);
  foo();
  return EXIT_SUCCESS;
}
```



main

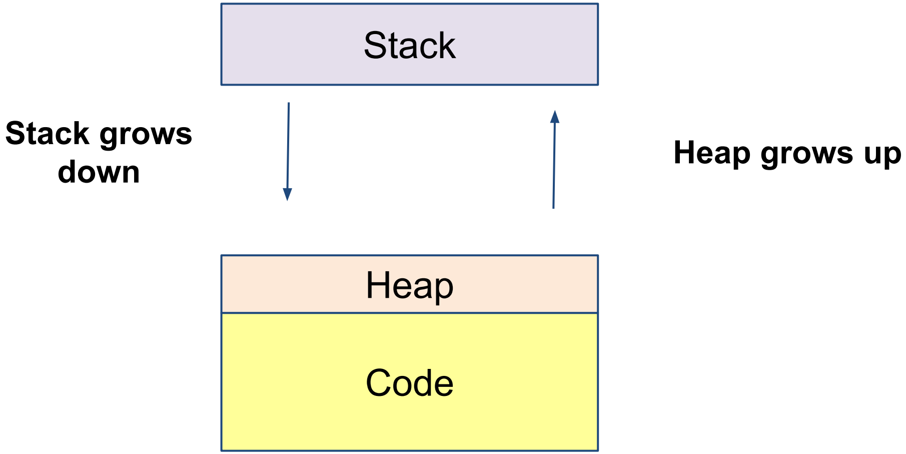


foo

- Program 3 (dynamic requirements):
  - **Large stack when in function foo.**
  - **Large heap when in function main.**
  - Code segment.

Dynamic memory allocation

- Memory allocated dynamically based on program usage.
- Why don’t these segments grow in the same direction?

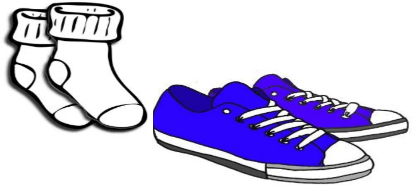


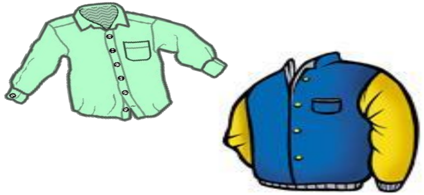
Stack Memory or Stack Segment

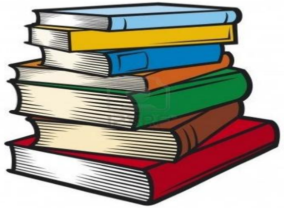
- Follows the "first-in last-out" (or last-in first-out) rule.
- is indirectly controlled by your programs.
- is directly controlled by compilers and operating systems.

Stack

- "Stack" means what comes first leaves last.
- You are using this concept everyday.
- You put on socks before putting on shoes. You take off the shoes before taking off the socks.
- You put on a shirt before wearing a jacket. You take off the jacket before taking off the shirt.
- When you put a book on the top of a pile, the last added book is removed first.







Stack Memory or Stack Segment

- Will have once record for every “Active” function.
  - Active function: Function whose execution is not finished.

- This record is also called “Stack Frame”.

Contents of a Stack Frame

- How many active functions?
  - Whose execution is not finished?

```
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

the program is here

Contents of a Stack Frame

- How many active functions?
  - Whose execution is not finished?

- f3
- f2
- f1
- main

```c
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

the program is here

Contents of a Stack Frame

- How many active functions?
  - – Whose execution is not finished?
  - f3
  - f2
  - f1
  - main

```
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

the program is here

- How many stack frames?
  - – Number of active functions = 4

Contents of a Stack Frame

- What do we need to store for each active function?

  - f1
  - main

- What do we need to continue execution in main?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
   int i = 1, j;
   f1();
   j = i + 1  + argc;
   printf("j= %d\n", j);
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
```

the program is here

Contents of a Stack Frame

- What do we need to store for each active function?
  - Arguments.
  - Local Variables.
  - Return Address.

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int i = 1, j;
   f1();
   j = i + 1  + argc;
   printf("j= %d\n", j);
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
```

the program is here

The need for return address

- What happens next?

```
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

The need for return address

- What happens next?

```c
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

The need for return address

- What happens next?

```
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

The need for return address

- What happens next?

```
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

The need for return address

- What happens next?

1. Return to f2
2. Return to f1
3. Return to main
4. Exit.

```c
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

The need for return address

- Call order v/s return order!

Call Order	**Return Order**
1. main	1. f3
2. f1	2. f2
3. f2	3. f1
4. f3	4. main

```c
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

The need for return address

- How to return correctly?

We need to store the return location (or return address)

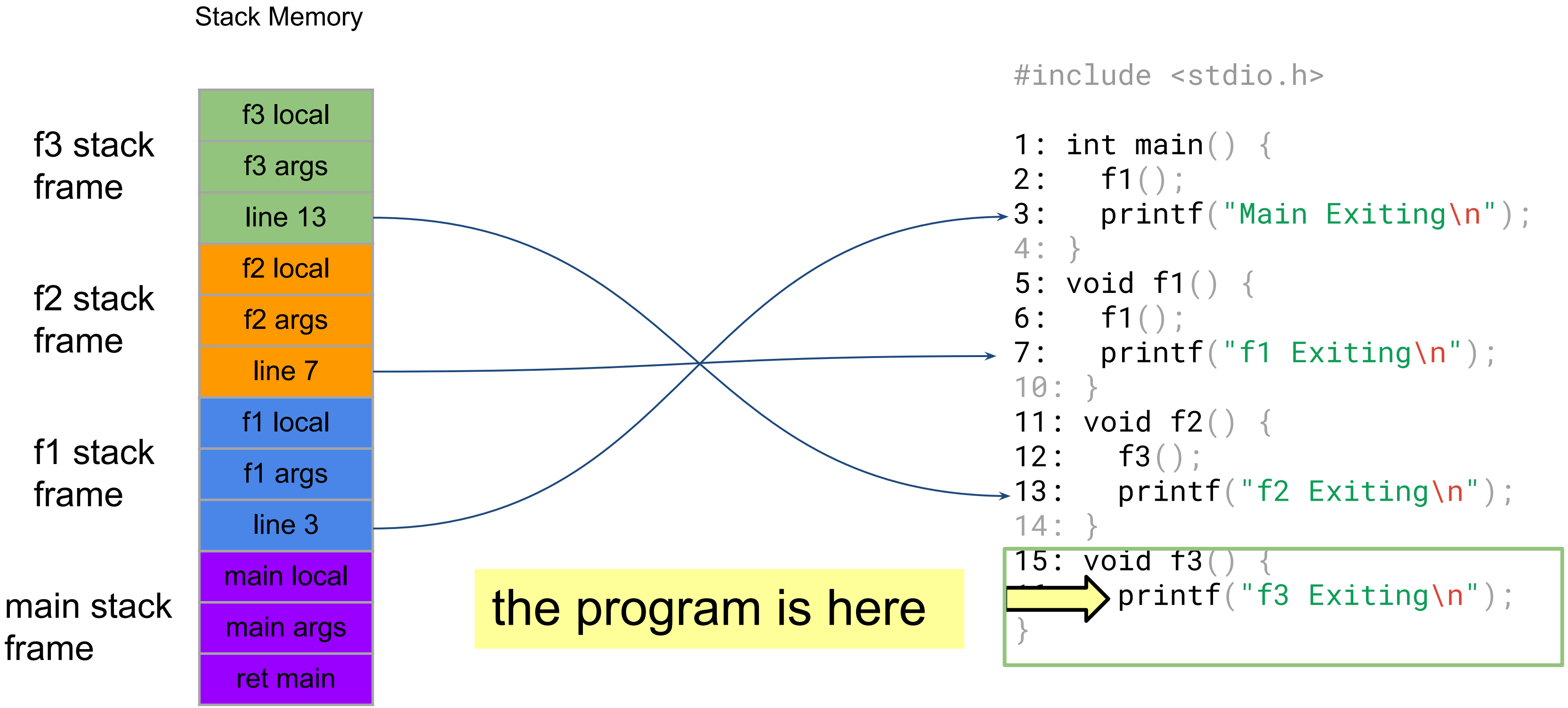
the program is here

```
#include <stdio.h>

int main() {
   f1();
   printf("Main Exiting\n");
}
void f1() {
   f1();
   printf("f1 Exiting\n");
}
void f2() {
   f3();
   printf("f2 Exiting\n");
}
void f3() {
   printf("f3 Exiting\n");
}
```

Contents of a Stack Frame

- What do we need to store for each active function?
  - Arguments.
  - Local Variables.
  - Return Address.

Stack frames



Stack Memory

f3 stack frame

f2 stack frame

f1 stack frame

main stack frame

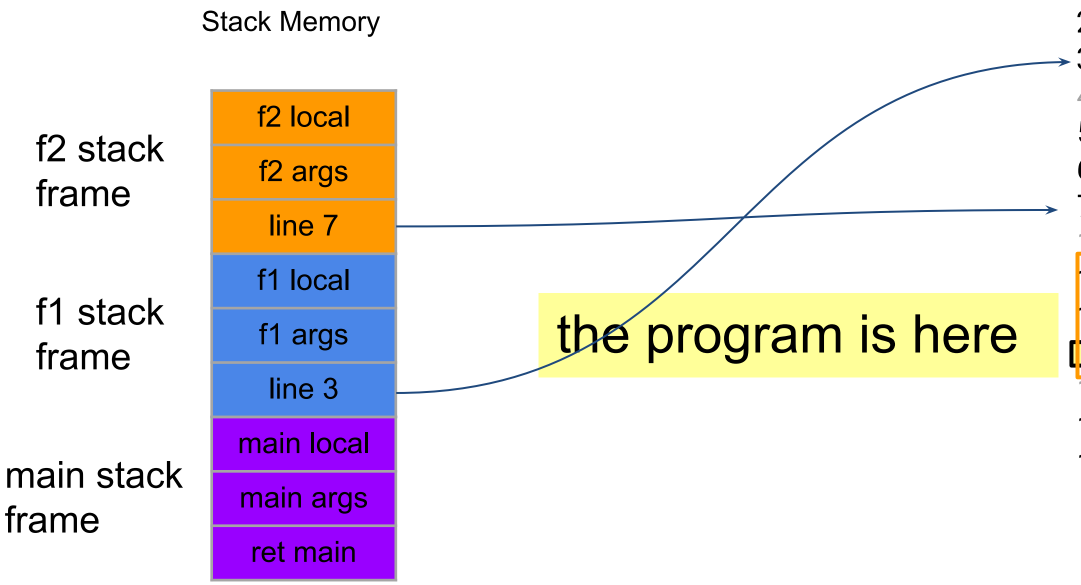
f3 local
f3 args
line 13
f2 local
f2 args
line 7
f1 local
f1 args
line 3
main local
main args
ret main

```
#include <stdio.h>

1: int main() {
2:   f1();
3:   printf("Main Exiting\n");
4: }
5: void f1() {
6:   f1();
7:   printf("f1 Exiting\n");
10: }
11: void f2() {
12:   f3();
13:   printf("f2 Exiting\n");
14: }
15: void f3() {
      printf("f3 Exiting\n");
}
```

the program is here

Stack frames

Pop stack frame and continue using return address

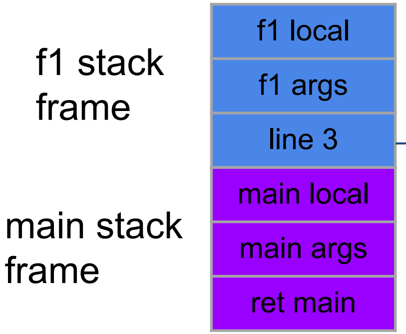


```
#include <stdio.h>

1: int main() {
2:   f1();
3:   printf("Main Exiting\n");
4: }
5: void f1() {
6:   f1();
7:   printf("f1 Exiting\n");
10: }
11: void f2() {
12:   f3();
      printf("f2 Exiting\n");
14: }
15: void f3() {
16:   printf("f3 Exiting\n");
}
```

Stack frames

Pop stack frame and continue using return address

Stack Memory



f1 stack frame

main stack frame

f1 local

f1 args

line 3

main local

main args

ret main

the program is here

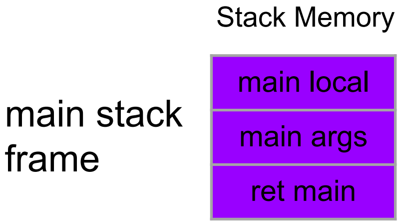
```
#include <stdio.h>

1: int main() {
2:   f1();
3:   printf("Main Exiting\n");
4: }
5: void f1() {
6:   f1();
7:   printf("f1 Exiting\n");
10: }
11: void f2() {
12:   f3();
13:   printf("f2 Exiting\n");
14: }
15: void f3() {
16:   printf("f3 Exiting\n");
}
```

Stack frames

Pop stack frame and continue using return address

the program is here

```
#include <stdio.h>

int main() {
  f1();
  printf("Main Exiting\n");
}
void f1() {
  f1();
  printf("f1 Exiting\n");
}
void f2() {
  f3();
  printf("f2 Exiting\n");
}
void f3() {
  printf("f3 Exiting\n");
}
```

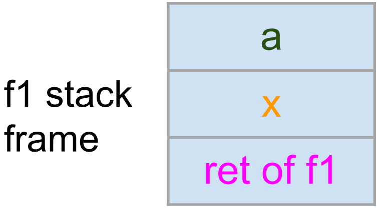


Stack Memory

main stack frame

main local
main args
ret main

How stack frames are created?


f1 stack frame

a

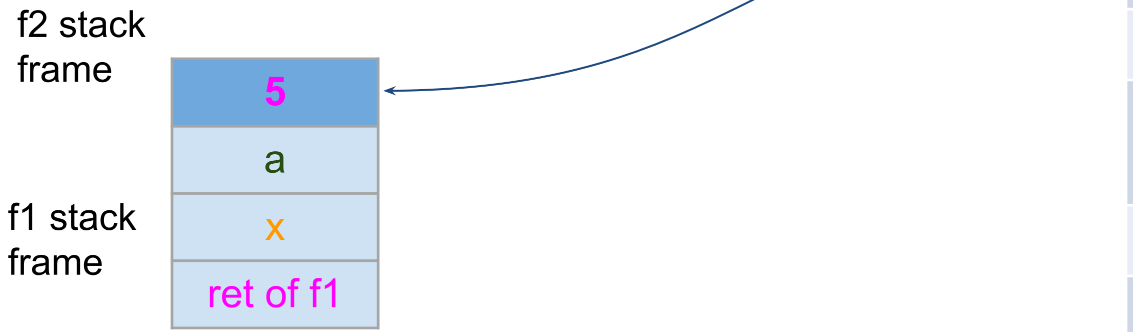
x

ret of f1

```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
          double z)
{
     int m=4;
```

How stack frames are created?

Push return address



f2 stack frame

5

a

x

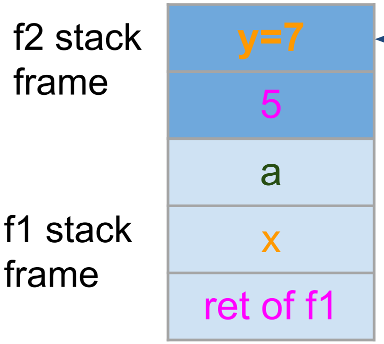
ret of f1

f1 stack frame

```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
           double z)
{
     int m=4;
```

How stack frames are created?

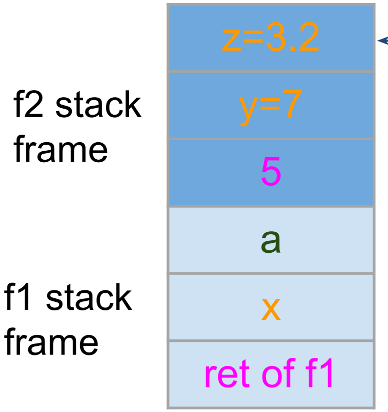
Push argument y



```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
        double z)
{
     int m=4;
```

How stack frames are created?

Push argument z



z=3.2

y=7

5

a

x

ret of f1

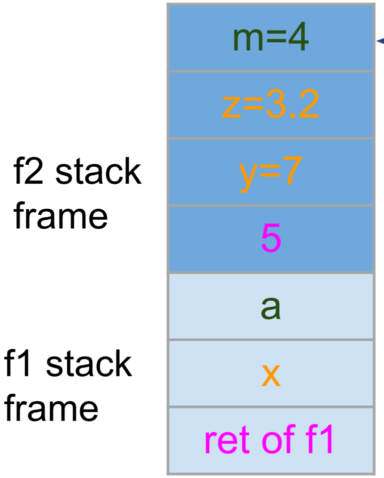
f2 stack frame

f1 stack frame

```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
         double z)
{
     int m=4;
```

How stack frames are created?

Transfer control to f3 and push local variables



```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
           double z)
{
    int m=4;
```

Stack frame memory

- Computer access memory using its address.
- Memory has address : n-bit value

  - Stack frame has address
    - All elements in stack frame also has addresses

Stack frame details

Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	y	104	7
	RL	103	line 5
Frame of f1	a	102	a =
	x	101	x =
	RL	100	line ?

```
 1  void f1(int x)
 2  {
 3      int a;
 4      a = f3(7, 3.2);
 5      x = a + 5;
 6      ...
 7  }
 8  int f3(int y,
           double z)
    {
10      int m=4;
```

Stack frame details

For Humans

Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	y	104	7
	RL	103	line 5
Frame of f1	a	102	a =
	x	101	x =
	RL	100	line ?

```
 1  void f1(int x)
 2  {
 3      int a;
 4      a = f3(7, 3.2);
 5      x = a + 5;
 6      ...
 7  }
 8  int f3(int y,
            double z)
    {
10       int m=4;
```