



# Level 0x06

More C / The Stack

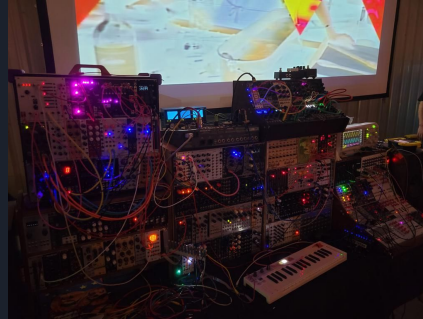


# Topics

- Events
- Hacker History
- C Stuff
  - Branching
  - Looping
  - The Stack

# Upcoming Events

- Maker Faire Orlando - “Greatest Show & Tell on Earth”
  - Saturday Nov 4th - Sunday Nov 5th, 2023
  - <https://www.makerfaireorlando.com/>
  - Orange County Fairgrounds
  - \$25 Adults, \$20 Students, \$5 off if pre-purchased





# rms

## Richard Stallman

- Founder of GNU project (1983)
  - He got angry at a printer...
- GNU General Public License (GPL)
- GNU Compiler Collection (gcc)
- GNU Debugger (gdb)
- GNU Make
- GNU Emacs
- Insufferable / Low Charisma
  - GNU/Linux
  - Refuses to use anything non-free
  - Epstein controversy
  - Controversial political views





# GPL

- Software License for Open Source Software
  - the freedom to use the software for any purpose,
  - the freedom to change the software to suit your needs,
  - the freedom to share the software with your friends and neighbors, and
  - the freedom to share the changes you make.
- If you distribute GPL software, you must offer source code for it
- Derivative works can't be more restricted than GPL
- GPL Virus (Microsoft):
  - Steve Ballmer declared that code released under GPL is useless to the commercial sector (since it can only be used if the resulting surrounding code becomes GPL)
  - "A cancer that attaches itself in an intellectual property sense to everything it touches".



# If Conditionals

```
if (condition)
{
    // code block
}
```

```
if (condition)
{
    // code block
}
else
{
    // other code block
}
```



# Switch Statements

```
switch (variable or expression)
{
case 1:
    // do stuff, fallthrough to next case
case 2:
    // so stuff, no fallthrough
    break;
case 3:
    {
        // code block variables
        // this is gross though
        // just call a function
    }
    break;
default:
    // if no other case was picked, do this
}
```



# While loop

```
while(condition)
{
    // do this block of instructions over and over

    if (diff_condition)
    {
        // restart loop right now!
        continue;
    }

    if (some_other_condition)
    {
        // let's get out of this while loop,
        // regardless of loop condition
        break;
    }
}
```





# Do While Loop

```
do
{
    // No matter what, I'm doing this code block
    // at least 1 time!

    // break and continue work in this too

} while(condition); // don't forget this semicolon...
```



# For loop

```
// My loop had a simple counting variable
for (int i = 0; i < 10; i++)
{
    printf("I've run this loop %d times\n", i);
}

// Initialization statement only runs 1 time

// Loop condition runs every loop, even before first loop

// Increment statement runs after every loop
```



# Functions

```
int my_div_func(int arg1, int arg2)
{
    if (arg2 == 0)
    {
        printf("Don't divide by zero!\n");
        return 0;
    }

    return arg1 / arg2;
}

int main(int argc, char** argv)
{
    int topNum = atoi(argv[1]);
    int bottomNum = atoi(argv[2]);
    printf("Div result = %d\n", my_div_func(topNum, bottomNum) );
    return 0;
}

// Function must be declared before you can call them
```



# Void function

```
void hexprint(int val)
{
    If (val < 0)
    {
        printf("-0x%08x", val * -1);
        return;
    }

    printf("0x%08", val);

    // Doesn't return anything, don't even need a return statement
}
```



# Arrays

```
int listOfNums[4];
```

```
int differentList[] = { 1, 2, 3, 4};
```

```
// This won't work, size of array must be known
```

```
// at time of compilation
```

```
int badList[variable_from_user];
```

```
for(int i = 0; i < 4; i++)
```

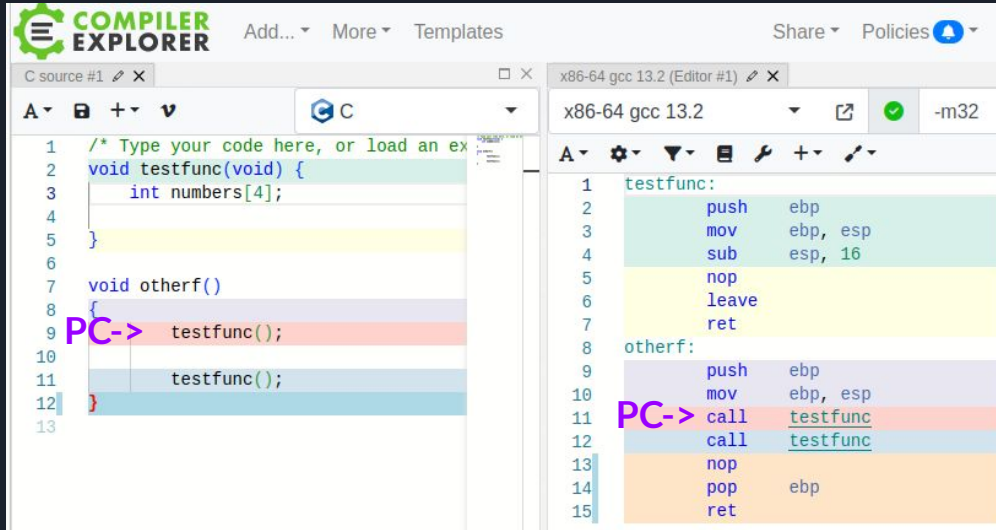
```
{
```

```
    // Uninitialized arrays are filled with garbage data
```

```
    print("listOfNums index %d = %d\n", i, listOfNums[i]);
```

```
}
```

# The Stack



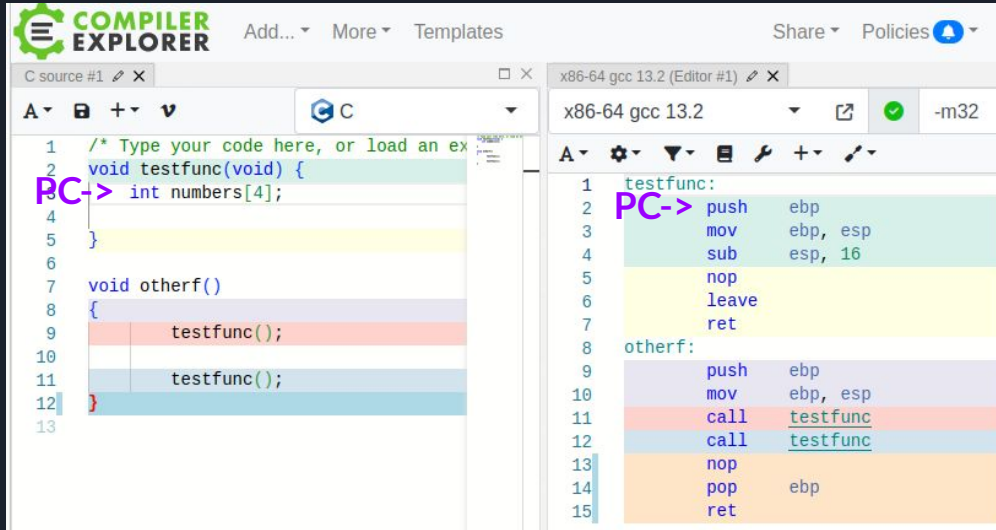
The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed with line numbers 1 through 13. The code defines two functions: `testfunc` and `otherf`. Line 9, `testfunc();`, is highlighted in red and labeled with a purple `PC->`. Line 12, `testfunc();`, is highlighted in blue. On the right, the assembly output for x86-64 gcc 13.2 is shown. It includes the assembly for `testfunc` (lines 1-7) and `otherf` (lines 8-15). Line 11, `call testfunc`, is highlighted in red and labeled with a purple `PC->`. The assembly shows stack frame setup with `push ebp`, `mov ebp, esp`, and `sub esp, 16`.

SP->

BP->

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	
0xf000 0040	
0xf000 0044	
0xf000 0048	
0xf000 004c	
0xf000 0050	

# The Stack



The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed: 

```
1 /* Type your code here, or load an example */
2 void testfunc(void) {
3     int numbers[4];
4 }
5
6
7 void otherf()
8 {
9     testfunc();
10
11     testfunc();
12 }
13
```

 On the right, the assembly code for x86-64 gcc 13.2 is shown: 

```
1 testfunc:
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 16
5     nop
6     leave
7     ret
8
9 otherf:
10    push    ebp
11    mov     ebp, esp
12    call    testfunc
13    call    testfunc
14    nop
15    pop     ebp
16    ret
```

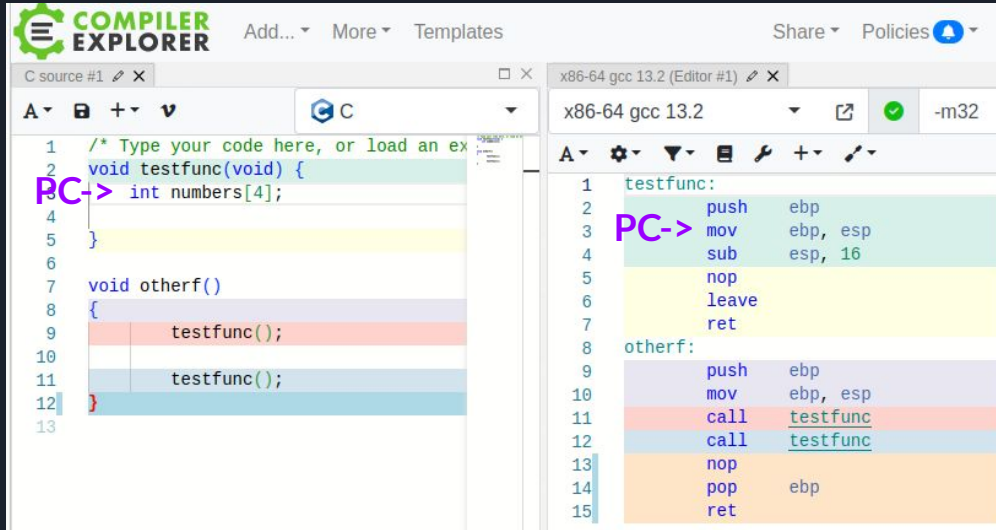
Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	
0xf000 0040	
0xf000 0044	
0xf000 0048	
0xf000 004c	
0xf000 0050	12

SP->

BP->

Return address get pushed onto stack by call

# The Stack



The screenshot shows the Compiler Explorer interface. On the left, the C source code for `testfunc` and `otherf` is displayed. `testfunc` is a void function that declares an array `numbers[4]`. `otherf` calls `testfunc` twice. On the right, the assembly output for `x86-64 gcc 13.2` is shown. The assembly for `testfunc` (lines 1-4) shows `push ebp`, `mov ebp, esp`, `sub esp, 16`, and `leave`. The assembly for `otherf` (lines 9-14) shows `push ebp`, `mov ebp, esp`, `call testfunc`, `call testfunc`, `pop ebp`, and `ret`. A purple label `PC->` points to the first line of the `testfunc` assembly block.

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	
0xf000 0040	
0xf000 0044	
0xf000 0048	
0xf000 004c	0xf000 0050
0xf000 0050	12

SP->

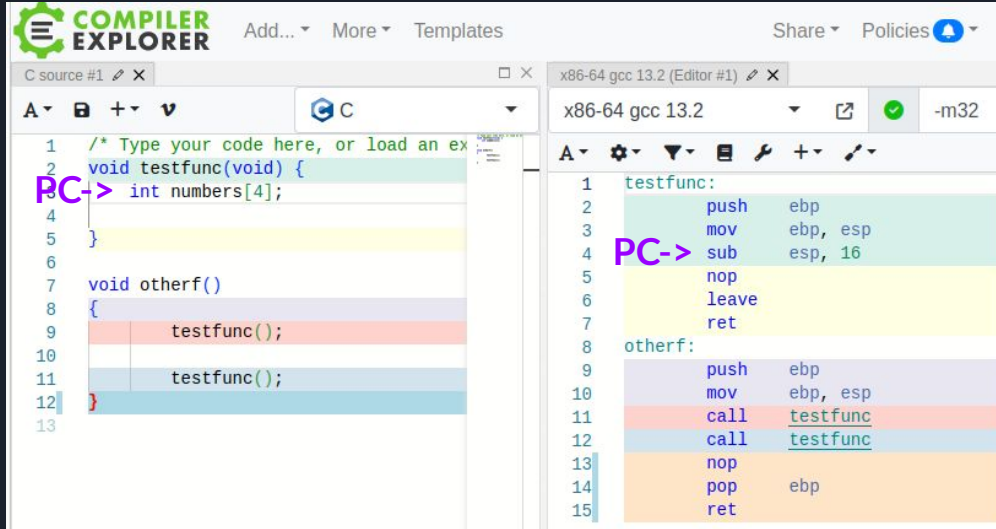
BP->

Pushed BP location onto stack





# The Stack



The screenshot shows the Compiler Explorer interface. The left pane displays C code for two functions: `testfunc` and `otherf`. The right pane shows the corresponding assembly code for `x86-64 gcc 13.2`. In the assembly, the `testfunc` function starts with `push ebp`, `mov ebp, esp`, and `sub esp, 16`. The `otherf` function starts with `push ebp`, `mov ebp, esp`, and `call testfunc`. A purple arrow labeled "PC->" points to the `int numbers[4];` line in the C code and the `sub esp, 16` instruction in the assembly.

```
1  /* Type your code here, or load an example */
2  void testfunc(void) {
3      int numbers[4];
4  }
5
6
7  void otherf()
8  {
9      testfunc();
10
11     testfunc();
12 }
13
```

```
1  testfunc:
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 16
5
6      nop
7      leave
8      ret
9
10 otherf:
11     push    ebp
12     mov     ebp, esp
13     call    testfunc
14     call    testfunc
15     nop
16     pop     ebp
17     ret
```

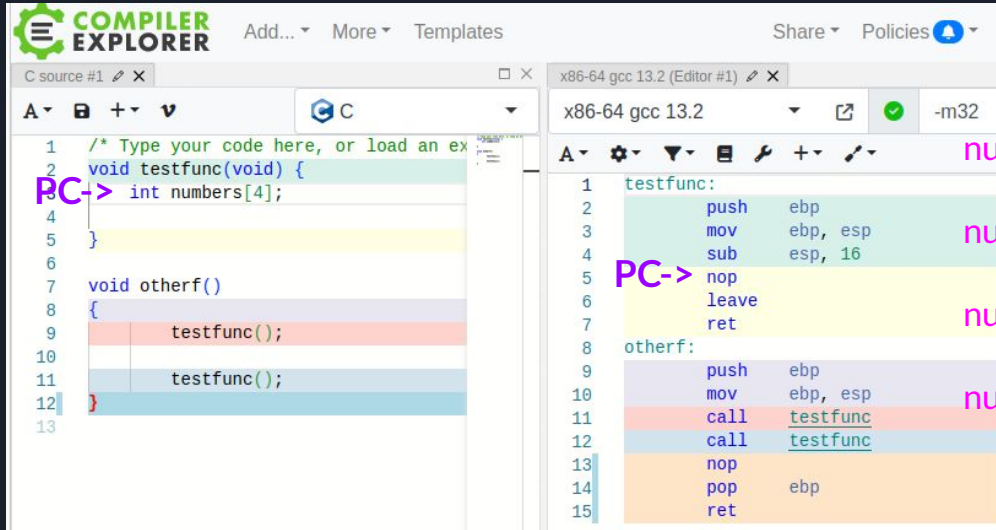
SP->  
BP->

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	
0xf000 0040	
0xf000 0044	
0xf000 0048	
0xf000 004c	0xf000 0050
0xf000 0050	12

Moved BP to same spot as SP

# The Stack

SP moved up 0x10 or 16 bytes



The screenshot shows the Compiler Explorer interface with two tabs. The left tab, 'C source #1', contains the following C code:

```
1  /* Type your code here, or load an example */
2  void testfunc(void) {
3      int numbers[4];
4  }
5
6
7  void otherf()
8  {
9      testfunc();
10
11     testfunc();
12 }
13
```

The right tab, 'x86-64 gcc 13.2 (Editor #1)', shows the generated assembly:

```
1  testfunc:
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 16
5
6      nop
7      leave
8      ret
9
10 otherf:
11     push    ebp
12     mov     ebp, esp
13     call    testfunc
14     call    testfunc
15     nop
16     pop     ebp
17     ret
```

SP->

numbers[0]

numbers[1]

numbers[2]

numbers[3]

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	???
0xf000 0040	???
0xf000 0044	???
0xf000 0048	???
0xf000 004c	0xf000 0050
0xf000 0050	12

# The Stack

NOP = no operations, does nothing

```
1  /* Type your code here, or load an example */
2  void testfunc(void) {
3      int numbers[4];
4  }
5
6
7  void otherf()
8  {
9      testfunc();
10
11     testfunc();
12 }
13
```

```
1  testfunc:
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 16
5
6      PC->   leave
7      ret
8
9  otherf:
10     push    ebp
11     mov     ebp, esp
12     call    testfunc
13     call    testfunc
14     nop
15     pop     ebp
16     ret
```

SP->

numbers[0]

numbers[1]

numbers[2]

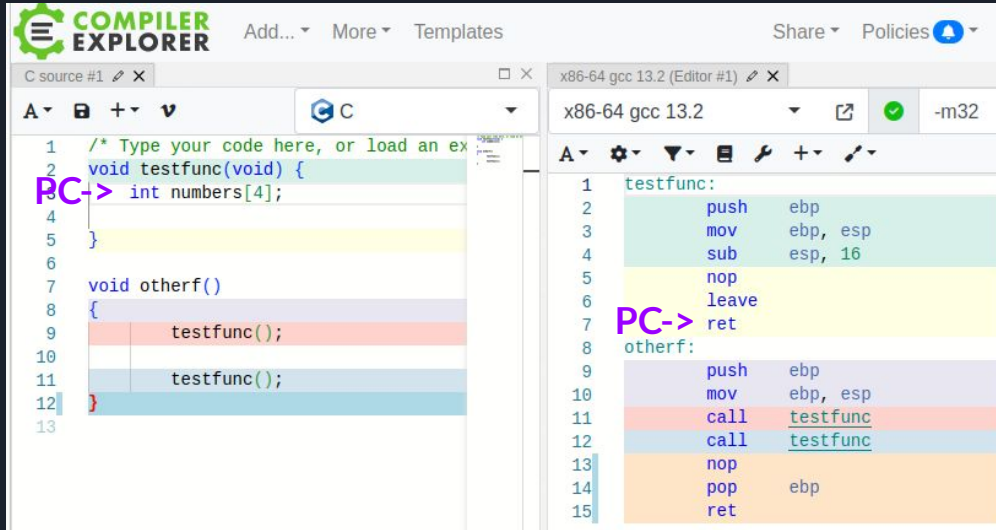
numbers[3]

BP->

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	???
0xf000 0040	???
0xf000 0044	???
0xf000 0048	???
0xf000 004c	0xf000 0050
0xf000 0050	12

# The Stack

`leave` is equiv to `mov esp, ebp`  
`pop ebp`



The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed: `void testfunc(void) { int numbers[4]; }` and `void otherf() { testfunc(); testfunc(); }`. The assembly output on the right for `testfunc` shows: `push ebp, mov ebp, esp, sub esp, 16, nop, leave, ret`. The assembly for `otherf` shows: `push ebp, mov ebp, esp, call testfunc, call testfunc, nop, pop ebp, ret`. A green arrow points from the `leave` instruction in the `testfunc` assembly to the memory table on the right.

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	???
0xf000 0040	???
0xf000 0044	???
0xf000 0048	???
0xf000 004c	0xf000 0050
0xf000 0050	12

We moved SP to where BP is

# The Stack

**leave** is equiv to `mov esp, ebp`  
**pop ebp**

```
1  /* Type your code here, or load an example */
2  void testfunc(void) {
3      int numbers[4];
4  }
5
6
7  void otherf()
8  {
9      testfunc();
10
11     testfunc();
12 }
13
```

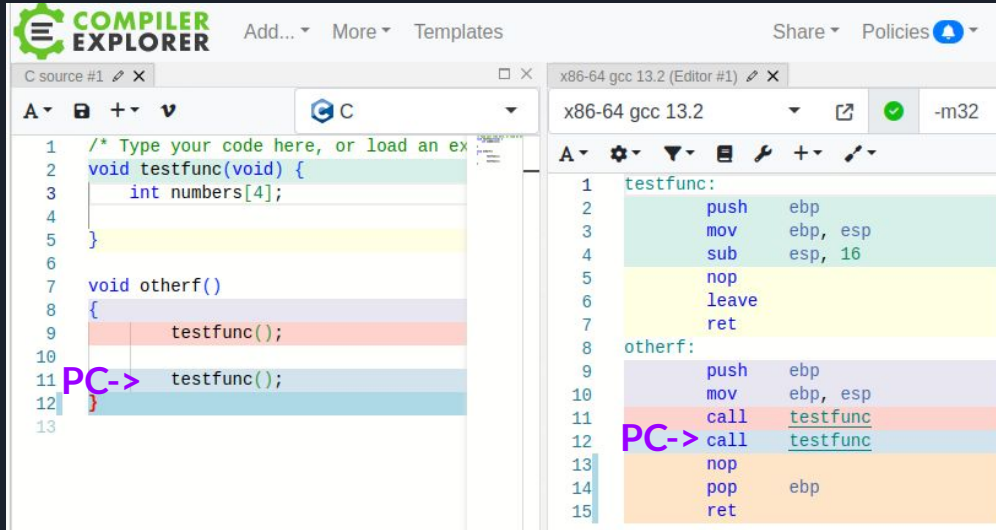
```
1  testfunc:
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 16
5      nop
6      leave
7  PC-> ret
8  otherf:
9      push    ebp
10     mov     ebp, esp
11     call    testfunc
12     call    testfunc
13     nop
14     pop     ebp
15     ret
```

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	???
0xf000 0040	???
0xf000 0044	???
0xf000 0048	???
0xf000 004c	0xf000 0050
0xf000 0050	12

SP moves down cause it was a pop.  
BP moves to the value popped

# The Stack

ret is equiv to pop PC



The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed with line numbers 1 through 13. Line 11, `testfunc();`, is highlighted in blue, and a purple arrow labeled "PC->" points to it. On the right, the assembly output for x86-64 gcc 13.2 is shown. Line 12, `call testfunc`, is highlighted in blue, and a purple arrow labeled "PC->" points to it. The assembly code shows the `testfunc` function being called, with instructions like `push ebp`, `mov ebp, esp`, `sub esp, 16`, `nop`, `leave`, and `ret`.

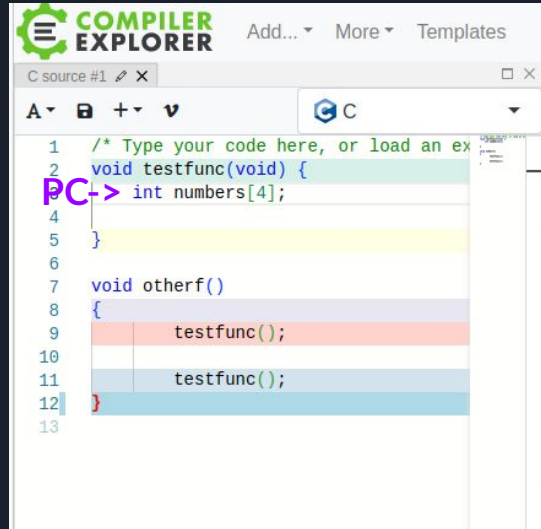
We are now back in original function, about to call next instruction. SP and BP haven't moved.

SP->  
BP->

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	???
0xf000 0040	???
0xf000 0044	???
0xf000 0048	???
0xf000 004c	0xf000 0050
0xf000 0050	12

# Stack Variable Problems

- What data is stored right now in numbers[]?
- What happens if I wrote to numbers[5]?



```
1  /* Type your code here, or load an example */
2  void testfunc(void) {
3      PC-> int numbers[4];
4
5  }
6
7  void otherf()
8  {
9      testfunc();
10
11     testfunc();
12 }
13
```

SP->

numbers[0]

numbers[1]

numbers[2]

numbers[3]

Addresses	Memory Vals
0xf000 0030	
0xf000 0034	
0xf000 0038	
0xf000 003c	???
0xf000 0040	???
0xf000 0044	???
0xf000 0048	???
0xf000 004c	0xf000 0050
0xf000 0050	12



# Links

- Many graphics from Wikipedia / wikimedia.org (Creative Commons License)
  - [https://en.wikipedia.org/wiki/Richard\\_Stallman](https://en.wikipedia.org/wiki/Richard_Stallman)
- <https://www.gnu.org/licenses/quick-guide-gplv3.html>
- 
-