

Software Security

Module 2 - C Call Stack & Pointers

CMPE279
Software Security Technologies
San Jose State University

MEMORY LAYOUT

```
1
2 #include<stdio.h>
3 #include<malloc.h>
4
5 int glb_uninit;          /* Part of BSS Segment -- global uninitialized variable, at runtime i
6 int glb_init = 10;       /* Part of DATA Segment -- global initialized variable */
7
8 void foo(void)
9 {
10     static int num = 0;   /* stack frame count */
11     int autovar;          /* automatic variable/Local variable */
12     int *ptr_foo = (int*)malloc(sizeof(int));
13     if (++num == 4)       /* Creating four stack frames */
14         return;
15     printf("Stack frame number %d: address of autovar: %p\n", num, &autovar);
16     printf("Address of heap allocated inside foo() %p\n", ptr_foo);
17     foo();                /* function call */
18 }
19
20 int main()
21 {
22     char *p, *b, *nb;
23     int *ptr_main = (int*)malloc(sizeof(int));
24     printf("Text Segment:\n");
25     printf("Address of main: %p\n", main);
26     printf("Address of func foo: %p\n", foo);
27     printf("Stack Locations:\n");
28     foo();
29     printf("Data Segment:\n");
30     printf("Address of glb_init: %p\n", &glb_init);
31     printf("BSS Segment:\n");
32     printf("Address of glb_uninit: %p\n", &glb_uninit);
33     printf("Heap Segment:\n");
34     printf("Address of heap allocated inside main() %p\n", ptr_main);
35
36     return 0;
37 }
38
```

Highest
memory
addresses

The
Stack

ESP moves up
and down as
items are pushed
onto or popped
from the stack

ESP always
points to the last
item pushed
onto the stack

Free
Memory

.bss section
(Uninitialized data items)

.data section
(Initialized data items)

.text section
(Program code)

Lowest
memory
addresses

Group: all

eax	-1073744556
ecx	0xbffff4d0
edx	0x1
ebx	0xb7fc9ff4
esp	0xbffff490
ebp	0xbffff4b8
esi	0x8048570
edi	0x8048370
eip	0x8048497
eflags	0x200286
cs	0x73
ss	0x7b
ds	0x7b
es	0x7b
fs	0x0
gs	0x33

layout.c - Source Window

File Run View Control Preferences Help

layout.c main SOURCE

```

9 {
10     static int num = 0;          /* stack frame count */
11     int autovar;                /* automatic variable/Local variable */
12     int *ptr_foo = (int*)malloc(sizeof(int));
13     if (++num == 4)              /* Creating four stack frames */
14         return;
15     printf("Stack frame number %d: address of autovar: %p\n", num, &autovar);
16     printf("Address of heap allocated inside foo() %p\n", ptr_foo);
17     foo();                       /* function call */
18 }
19
20 int main()
21 {
22     char *p, *b, *nb;
23     int *ptr_main = (int*)malloc(sizeof(int));
24     printf("Text Segment:\n");
25     printf("Address of main: %p\n", main);
26     printf("Address of func foo: %p\n", foo);

```

Program stopped at line 23 8048497 23

```

seed@seed-desktop: ~/CMPE279/modules/module2/stack
File Edit View Terminal Help
seed@seed-desktop:~/CMPE279/modules/module2/stack$ ./layout.out
Text Segment:
Address of main: 0x8048486
Address of func foo: 0x8048424
Stack Locations:
Stack frame number 1: address of autovar: 0xbffff4d4
Address of heap allocated inside foo() 0x804b018
Stack frame number 2: address of autovar: 0xbffff4a4
Address of heap allocated inside foo() 0x804b028
Stack frame number 3: address of autovar: 0xbffff474
Address of heap allocated inside foo() 0x804b038
Data Segment:
Address of glb_init: 0x804a01c
BSS Segment:
Address of glb_uninit: 0x804a02c
Heap Segment:
Address of heap allocated inside main() 0x804b008
seed@seed-desktop:~/CMPE279/modules/module2/stack$

```

*** not to scale ***

```

1
2 #include<stdio.h>
3 #include<malloc.h>
4
5 int glb_uninit;
6 int glb_init = 10;
7
8 void foo(void)
9 {
10     static int num = 0;
11     int autovar;
12     int *ptr_foo = (int*)malloc(sizeof(int));
13     if (++num == 4)
14         return;
15     printf("Stack frame number %d: address of autovar: %p\n", num, &autovar);
16     printf("Address of heap allocated inside foo() %p\n", ptr_foo);
17     foo();
18 }
19
20 int main()
21 {
22     char *p, *b, *nb;
23     int *ptr_main = (int*)malloc(sizeof(int));
24     printf("Text Segment:\n");
25     printf("Address of main: %p\n", main);
26     printf("Address of func foo: %p\n", foo);
27     printf("Stack Locations:\n");
28     foo();
29     printf("Data Segment:\n");
30     printf("Address of glb_init: %p\n", &glb_init);
31     printf("BSS Segment:\n");
32     printf("Address of glb_uninit: %p\n", &glb_uninit);
33     printf("Heap Segment:\n");
34     printf("Address of heap allocated inside main() %p\n", ptr_main);
35
36     return 0;
37 }

```

REGISTER	CONTENTS
EIP	0x8048497
ESI	0x8048570
EDI	0x8048570
EFLAGS	
EAX	
EBX	
ECX	
EDX	
ESP	0xbffff490
EBP	0xbffff5b8

	ADDR	MEMORY
STACK		
	0xbffff4df	/** autovar #1 **/
STACK	0xbffff4a4	/** autovar #2 **/
	0xbffff474	/** autovar #3 **/
UNUSED MEMORY		
HEAP	0x804b028	/** ptr_foo #2 **/
	0x804b018	/** ptr_foo #1 **/
	0x804b008	/** ptr_main **/
BSS		
	0x804a02c	/** glb_uninit **/
DATA		
	0x804a01c	/** glb_init **/
TEXT		
	0x8048497	/** main() **/
	0x8048424	/** foo() **/

EIP points
to somewhere
in Code Segment

```

seed@seed-desktop: ~/CMPE279/modules/module2/stack
File Edit View Terminal Help
seed@seed-desktop:~/CMPE279/modules/module2/stack$ ./layout.out
Text Segment:
Address of main: 0x8048486
Address of func foo: 0x8048424
Stack Locations:
Stack frame number 1: address of autovar: 0xbffff4d4
Address of heap allocated inside foo() 0x804b018
Stack frame number 2: address of autovar: 0xbffff4a4
Address of heap allocated inside foo() 0x804b028
Stack frame number 3: address of autovar: 0xbffff474
Address of heap allocated inside foo() 0x804b038
Data Segment:
Address of glb_init: 0x804a01c
BSS Segment:
Address of glb_uninit: 0x804a02c
Heap Segment:
Address of heap allocated inside main() 0x804b008
seed@seed-desktop:~/CMPE279/modules/module2/stack$

```

CDECL

C Calling Convention

CDECL / GCC/VSC++

C Calling Conventions

(note: *callee* = called procedure)

Caller saves: EAX, ECX, EDX

Callee saves: EBX, ESP, EBP, ESI & EDI

Caller pass parameters on stack from right to left.

Example: Func(A, B, C). Push C, then B, and A.

Callee sets up new stack frame: push old frame pointer (EBP), adjusts EBP and pushes local variables on the stack.

Callee's return value will be in EAX (if <= 32-bits). If >32-bits, high bits in EDX.

Caller cleans up the stack. *Callee do not pop from Stack! Callers will pop or change SP offset upon return.*

cdecl [\[edit\]](#)

The **cdecl** (which stands for **C declaration**) is a calling convention that originates from the **C programming language** and is used by many C compilers for the **x86 architecture**.^[1] In cdecl, subroutine arguments are passed on the **stack**. Integer values and memory addresses are returned in the **EAX register**, floating point values in the **ST0 x87 register**. Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved. The **x87** floating point registers ST0 to ST7 must be empty (popped or freed) when calling a new function, and ST1 to ST7 must be empty on exiting a function.

In context of the C programming language, function arguments are pushed on the stack in the reverse order. In **GNU/Linux**, **GCC** sets the *de facto* standard for calling conventions. Since GCC version 4.5, the stack must be aligned to a 16-byte boundary when calling a function (previous versions only required a 4-byte alignment.)^[citation needed]

Consider the following C source code snippet:

```
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

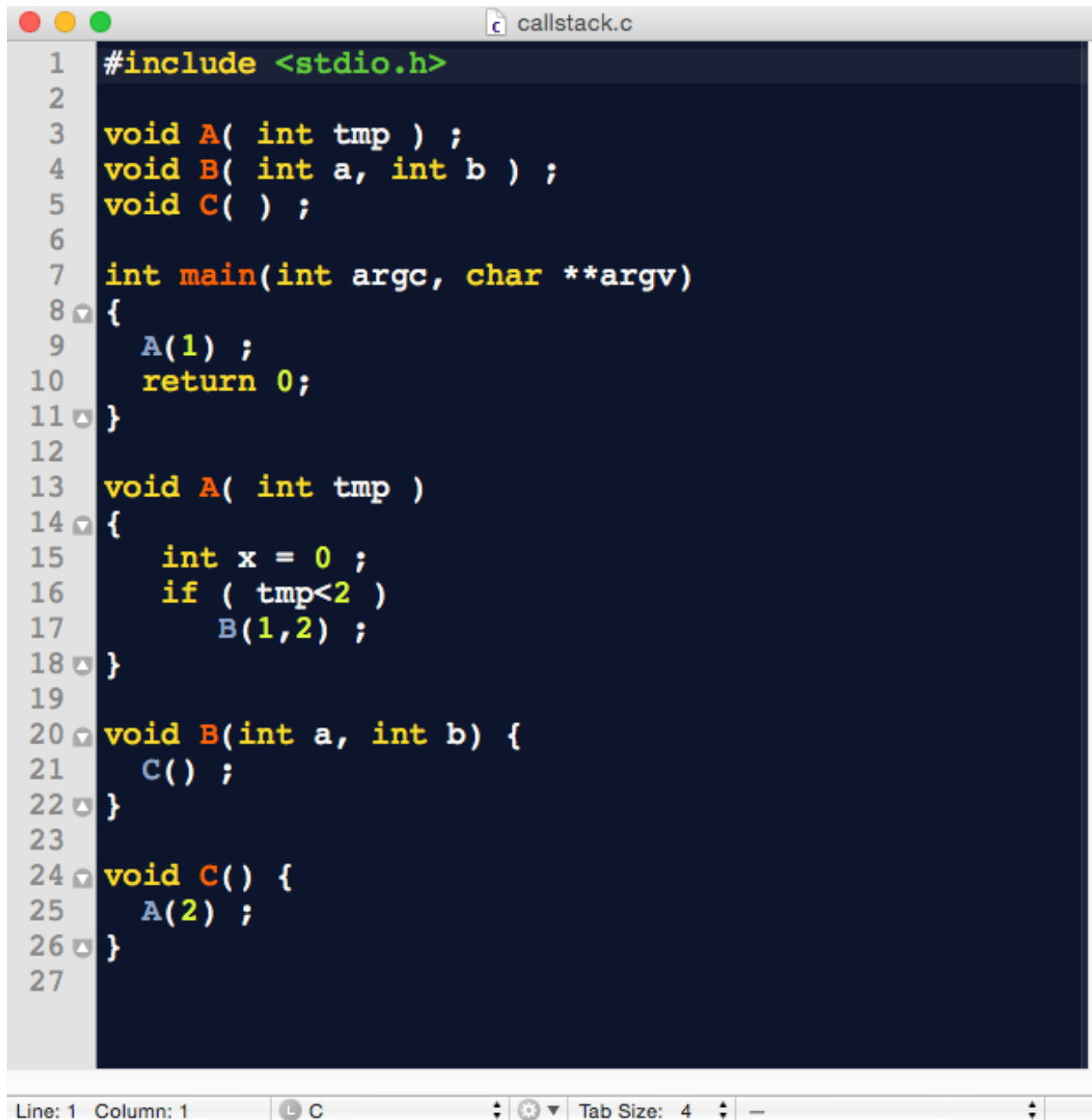
On **x86**, it will produce the following **assembly code** (Intel syntax):

```
caller:
    push    ebp
    mov     ebp, esp
    push    3
    push    2
    push    1
    call    callee
    add     esp, 12
    add     eax, 5
    pop     ebp
    ret
```

The caller cleans the stack after the function call returns.

REF: http://en.wikipedia.org/wiki/X86_calling_conventions

C CALL STACK



```
1 #include <stdio.h>
2
3 void A( int tmp ) ;
4 void B( int a, int b ) ;
5 void C( ) ;
6
7 int main(int argc, char **argv)
8 {
9     A(1) ;
10    return 0;
11 }
12
13 void A( int tmp )
14 {
15     int x = 0 ;
16     if ( tmp<2 )
17         B(1,2) ;
18 }
19
20 void B(int a, int b) {
21     C() ;
22 }
23
24 void C() {
25     A(2) ;
26 }
27
```

Line: 1 Column: 1 C Tab Size: 4

The screenshot shows a debugger window with the following components:

- Register Window (Top Left):** Displays the state of various registers. The `eip` register is highlighted in green, showing the address `0x80483cc`.
- Source Window (Top Right):** Shows the source code of `callstack.c`. Line 16, `if (tmp<2)`, is highlighted in green, corresponding to the current instruction address.
- Assembly Window (Bottom Right):** Displays the assembly code for the current instruction address. The instruction `cmp DWORD PTR [ebp+0x8],0x1` at address `0x80483cc` is highlighted in green.
- Command Window (Bottom Left):** Shows the user's input `(gdb) x/24x $esp` and the debugger's output, which lists memory addresses and their contents in hexadecimal.
- Program Status (Bottom):** A status bar at the bottom indicates "Program stopped at line 16, 0x80483cc".

[illegible]

```
(gdb) x/i 0x080483b1
0x080483b1 <main+29>:      mov     eax,0x0

(gdb) x/24x $esp
0xbffff490:      0xb7f8c329      0x08049ff4      0xbffff4a8      0x080482a0
0xbffff4a0:      0xb7fc9ff4      0x00000000      0xbffff4b8      0x080483b1
0xbffff4b0:      0x00000001      0xbffff4d0      0xbffff528      0xb7e81775
0xbffff4c0:      0x08048420      0x080482e0      0xbffff528      0xb7e81775
0xbffff4d0:      0x00000001      0xbffff554      0xbffff55c      0xb7fe0b40
0xbffff4e0:      0x00000001      0x00000001      0x00000000      0x08048215

(gdb) |
```


x86 Stack Frame Instructions

Enter and Leave [\[edit\]](#)

enter arg

Creates a stack frame with the specified amount of space allocated on the stack.

leave

destroys the current stack frame, and restores the previous frame. Using Intel syntax this is equivalent to:

```
mov esp, ebp  
pop ebp
```

This will set `EBP` and `ESP` to their respective value before the function prologue began therefore reversing any modification to the stack that took place during the prologue.

http://en.wikibooks.org/wiki/X86_Assembly/Control_Flow#Function_Calls

A() Calling B(1,2)

```

1 #include <stdio.h>
2
3 void A( int tmp ) ;
4 void B( int a, int b ) ;
5 void C( ) ;
6
7 int main(int argc, char **argv)
8 {
9     A(1) ;
10    return 0;
11 }
12
13 void A( int tmp )
14 {
15     int x = 0 ;
16     if ( tmp < 2 )
17         B(1,2) ;
18 }
19
20 void B(int a, int b) {
21     C() ;
22 }
23
24 void C() {
25     A(2) ;
26 }
27

```

	ADDR	MEMORY
main: ebp	0xbffff4b8	Main's Stack Frame
	0xbffff4b4	
main: esp	0xbffff4b0	
ret: main	0xbffff4ac	Return Address to Main
A: ebp	0xbffff4a8	Main's (Old) EBP
	0xbffff4a4	
	0xbffff4a0	
	0xbffff49c	
	0xbffff498	
	0xbffff494	2nd Argument to B()
A: esp	0xbffff490	1st Argument to B()
ret: A	0xbffff48c	Return Address to A
B: ebp	0xbffff488	A's (Old) EBP
	0xbffff484	0x00000000
B: esp	0xbffff480	0x00000000

	ADDR	MEMORY
main: ebp	0xbffff4b8	0xbffff528
	0xbffff4b4	0xbffff4d0
main: esp	0xbffff4b0	0x00000001
ret: main	0xbffff4ac	0x080483b1
A: ebp	0xbffff4a8	0xbffff4b8
ebp - 4	0xbffff4a4	0x00000000
ebp - 8	0xbffff4a0	
ebp - 12	0xbffff49c	
ebp - 16	0xbffff498	
ebp - 20	0xbffff494	0x00000002
A: esp	0xbffff490	0x00000001
ret: A	0xbffff48c	0x080483e6
B: ebp	0xbffff488	0xbffff4a8
	0xbffff484	0x00000000
B: esp	0xbffff480	0x00000000

Prev EBP		
	2nd argument	EBP + 12
Prev ESP	1st argument	EBP + 8
	Return Address	EBP + 4
EBP	Previous EBP	EBP + 0
	Local Var #1	EBP - 4
ESP	Local Var #2	EBP + 8

STRINGS

Array Concepts

Size

Number of bytes allocated to the array (same as `sizeof(array)`).

Bound

The number of elements in the array.

Count

Number of elements in the array (same as the Visual Studio 2010 `_countof(array)`).

Lo

The address of the first element of the array.

Length

Number of characters before null terminator.

Hi

The address of the last element of the array.

Target size (Tsize)

Same as `sizeof(array)`.

TooFar

The address of the one-too-far element of the array, the element just past the Hi element.

String Concepts

Null-terminated

At or before Hi, the null terminator is present.

Length

Number of characters prior to the null terminator.

Strings are also “Array of Chars”

Improperly Bounded String Copies

```
gets_1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 void get_y_or_n(void) {
6     char response[8];
7     puts("Continue? [y] n: ");
8     gets(response);
9     if (response[0] == 'n')
10         exit(0);
11     return;
12 }
13
14 int main() {
15     get_y_or_n();
16     return 0;
17 }
18
```

```
gets_2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char *gets(char *dest) {
5     int c = getchar();
6     char *p = dest;
7     while (c != EOF && c != '\n') {
8         *p++ = c;
9         c = getchar();
10    }
11    *p = '\0';
12    return dest;
13 }
14
15 int main() {
16     char buffer[8];
17     gets(buffer);
18     return 0;
19 }
20
21
```

gets() doesn't limit number of chars to read from input

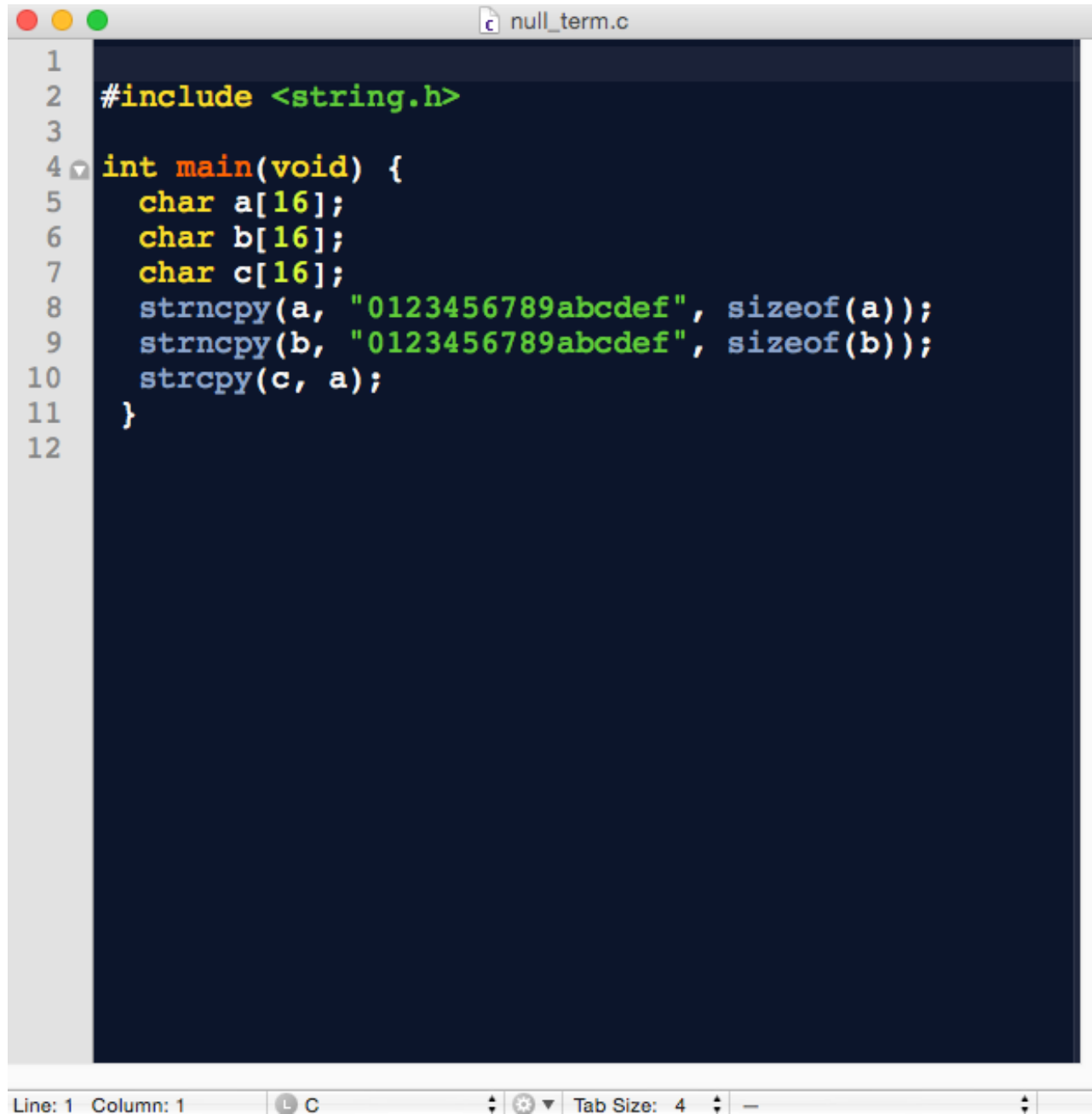
Un-Bounded Copies



```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[])
5 {
6     char prog_name[128];
7     strcpy(prog_name, argv[0]);
8
9     char arg1[80];
10    strcpy(arg1, argv[1]) ;
11    strcat( arg1, prog_name ) ;
12
13    char buf[80] ;
14    sprintf(buf, "%s", argv[2]) ;
15
16 }
17
```

The screenshot shows a code editor window with the title 'unbounded_copy.c'. The code is written in C and contains several unbounded string operations. Line 6 declares a character array 'prog_name' of size 128. Line 7 uses 'strcpy' to copy the first command-line argument into 'prog_name'. Line 9 declares a character array 'arg1' of size 80. Line 10 uses 'strcpy' to copy the second command-line argument into 'arg1'. Line 11 uses 'strcat' to append the contents of 'prog_name' to 'arg1'. Line 13 declares a character array 'buf' of size 80. Line 14 uses 'sprintf' to format the third command-line argument into 'buf'. The code is enclosed in a 'main' function with 'argc' and 'argv' parameters. The editor has a dark blue background and a light gray sidebar on the left showing line numbers. The status bar at the bottom indicates 'Line: 1 Column: 1' and 'Tab Size: 4'.

Null-Termination Errors



The image shows a code editor window titled "null_term.c". The code is as follows:

```
1
2 #include <string.h>
3
4 int main(void) {
5     char a[16];
6     char b[16];
7     char c[16];
8     strncpy(a, "0123456789abcdef", sizeof(a));
9     strncpy(b, "0123456789abcdef", sizeof(b));
10    strcpy(c, a);
11 }
12
```

The code defines three character arrays of size 16. It uses `strncpy` to copy the string "0123456789abcdef" into both `a` and `b`. The length of the string is 16, which is equal to the size of the arrays. However, `strncpy` does not automatically append a null terminator. The final line, `strcpy(c, a);`, attempts to copy the contents of `a` into `c`. Since `a` does not contain a null terminator, this operation will copy 16 bytes, including the memory immediately following the intended string, leading to undefined behavior or a crash.

Line: 1 Column: 1 C Tab Size: 4

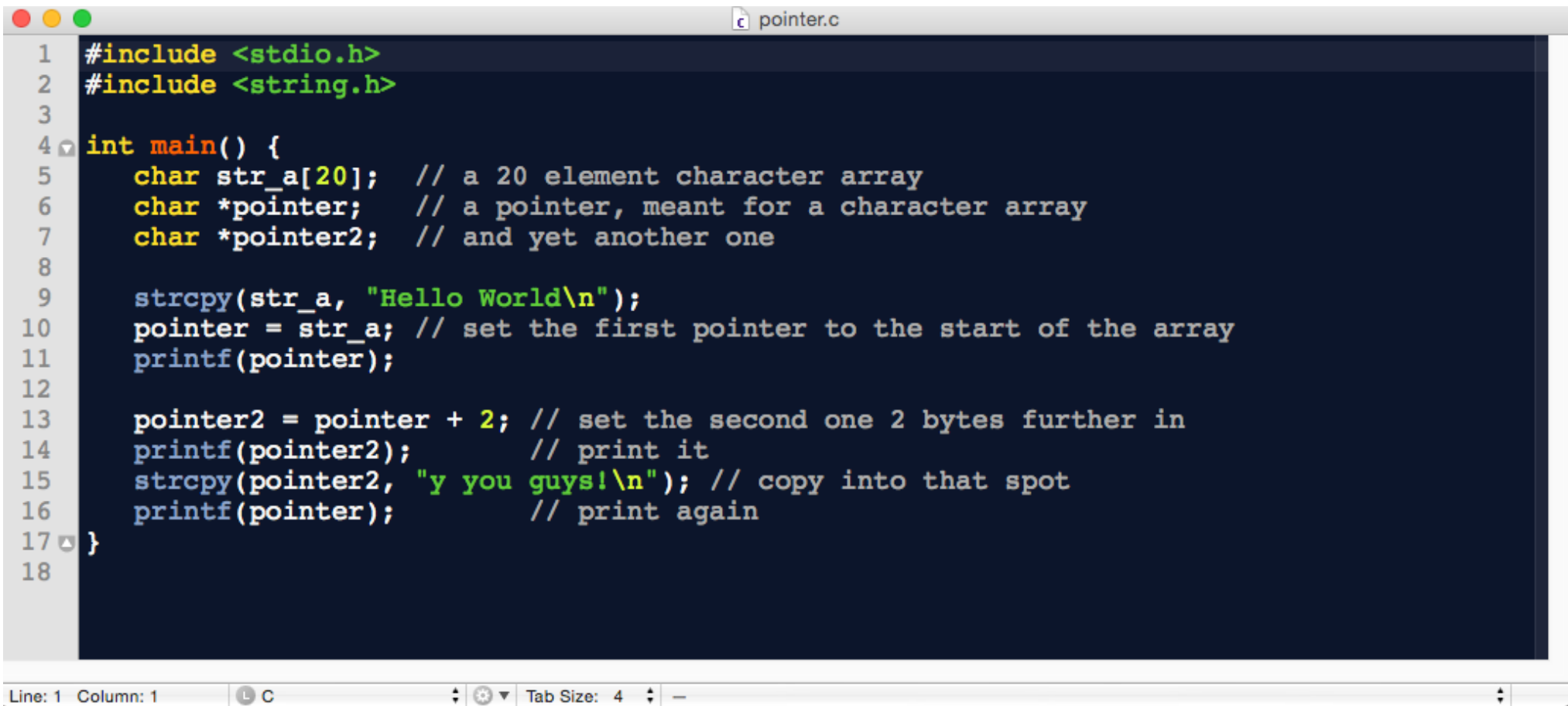
POINTERS


```
datatype_sizes.c
1 #include <stdio.h>
2
3 int main() {
4     printf("The 'int' data type is\t\t %d bytes\n", sizeof(int));
5     printf("The 'unsigned int' data type is\t %d bytes\n", sizeof(unsigned int));
6     printf("The 'short int' data type is\t %d bytes\n", sizeof(short int));
7     printf("The 'long int' data type is\t %d bytes\n", sizeof(long int));
8     printf("The 'long long int' data type is %d bytes\n", sizeof(long long int));
9     printf("The 'float' data type is\t %d bytes\n", sizeof(float));
10    printf("The 'char' data type is\t\t %d bytes\n", sizeof(char));
11
12    printf("sizeof(int*) = %lu\n", sizeof(int*)) ;
13    printf("sizeof(int) = %lu\n", sizeof(int)) ;
14    printf("sizeof(int*)==sizeof(int) = %d\n", sizeof(int*)==sizeof(int)) ;
15 }
16
```

Line: 14 Column: 5 C Tab Size: 4 main

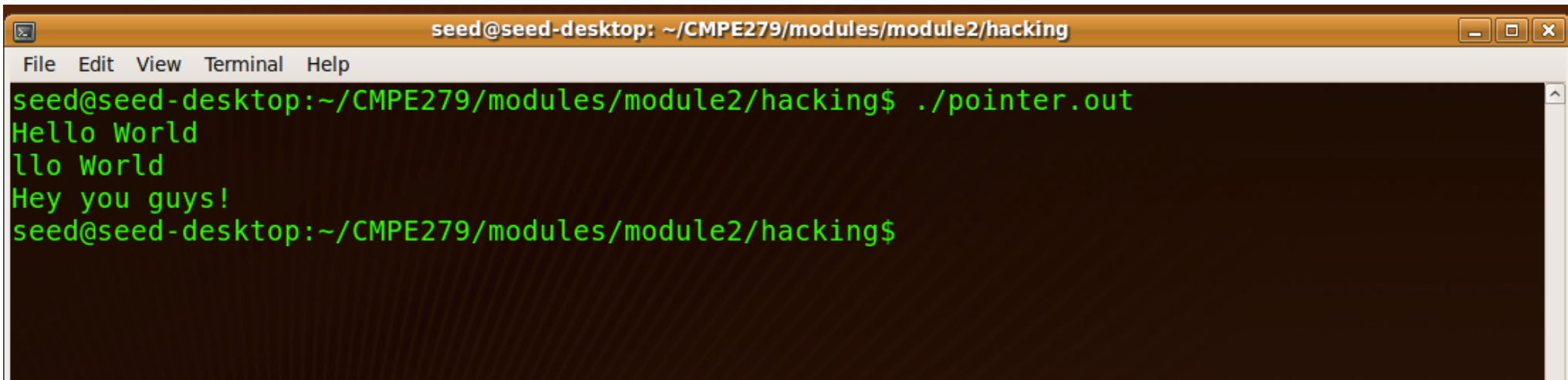
```
seed@seed-desktop: ~/CMPE279/modules/module
File Edit View Terminal Help
seed@seed-desktop:~/CMPE279/modules/module
The 'int' data type is 4 bytes
The 'unsigned int' data type is 4 bytes
The 'short int' data type is 2 bytes
The 'long int' data type is 4 bytes
The 'long long int' data type is 8 bytes
The 'float' data type is 4 bytes
The 'char' data type is 1 bytes
sizeof(int*) = 4
sizeof(int) = 4
sizeof(int*)==sizeof(int) = 1
seed@seed-desktop:~/CMPE279/modules/module
```

```
dragon:hacking pnguyen$ ./a.out
The 'int' data type is 4 bytes
The 'unsigned int' data type is 4 bytes
The 'short int' data type is 2 bytes
The 'long int' data type is 8 bytes
The 'long long int' data type is 8 bytes
The 'float' data type is 4 bytes
The 'char' data type is 1 bytes
sizeof(int*) = 8
sizeof(int) = 4
sizeof(int*)==sizeof(int) = 0
dragon:hacking pnguyen$
```

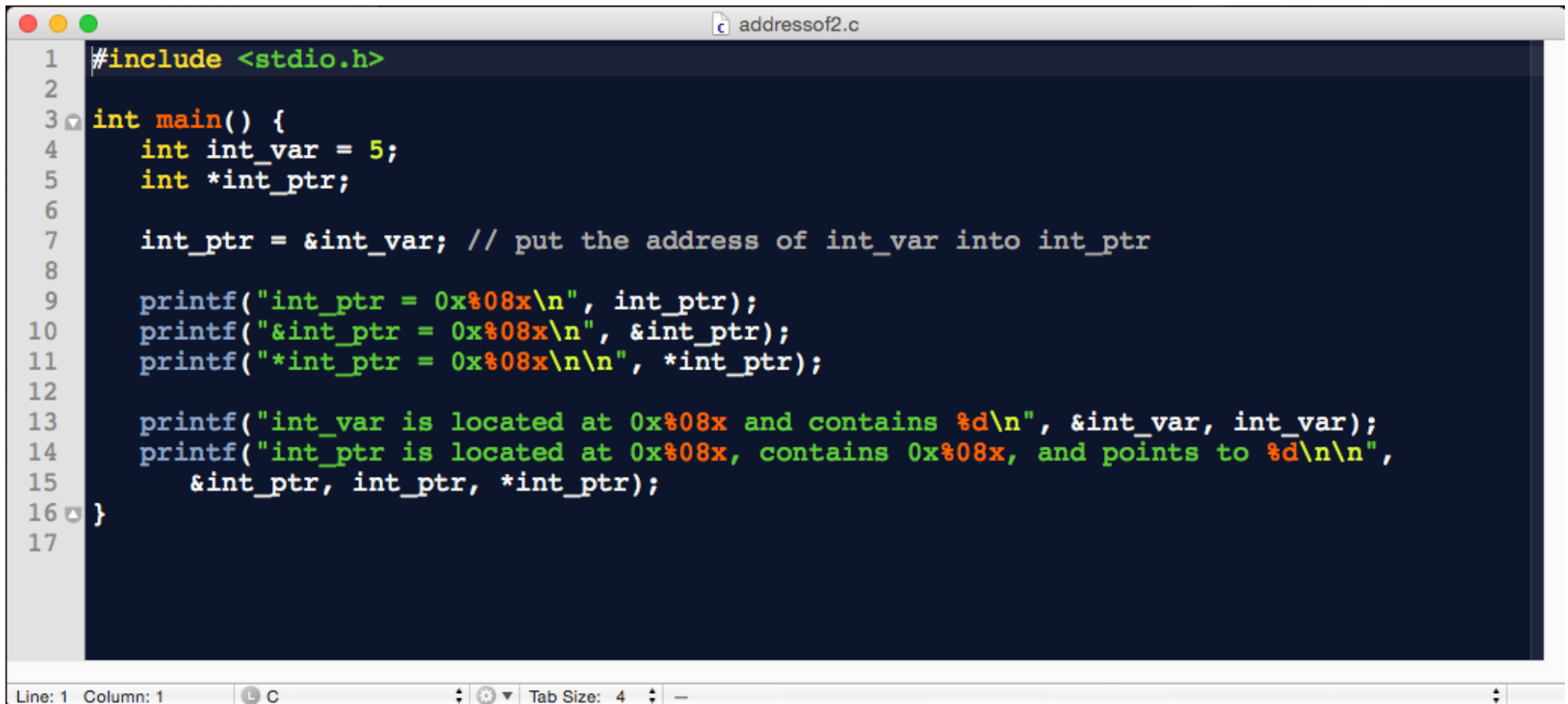


```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char str_a[20]; // a 20 element character array
6     char *pointer; // a pointer, meant for a character array
7     char *pointer2; // and yet another one
8
9     strcpy(str_a, "Hello World\n");
10    pointer = str_a; // set the first pointer to the start of the array
11    printf(pointer);
12
13    pointer2 = pointer + 2; // set the second one 2 bytes further in
14    printf(pointer2); // print it
15    strcpy(pointer2, "y you guys!\n"); // copy into that spot
16    printf(pointer); // print again
17 }
18
```

Line: 1 Column: 1 C Tab Size: 4

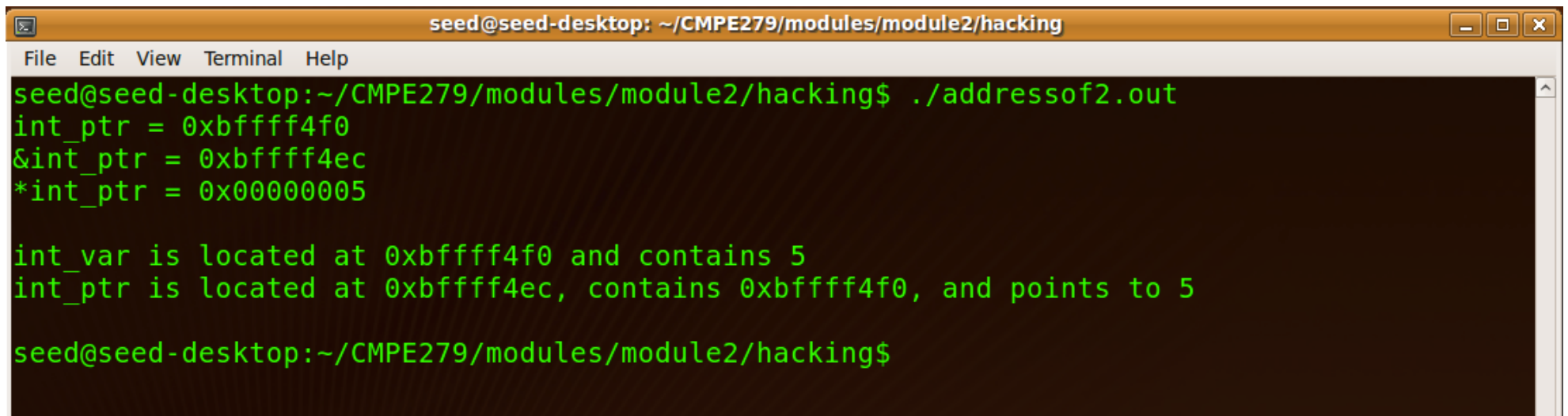


```
seed@seed-desktop: ~/CMPE279/modules/module2/hacking
File Edit View Terminal Help
seed@seed-desktop:~/CMPE279/modules/module2/hacking$ ./pointer.out
Hello World
llo World
Hey you guys!
seed@seed-desktop:~/CMPE279/modules/module2/hacking$
```



```
1 #include <stdio.h>
2
3 int main() {
4     int int_var = 5;
5     int *int_ptr;
6
7     int_ptr = &int_var; // put the address of int_var into int_ptr
8
9     printf("int_ptr = 0x%08x\n", int_ptr);
10    printf("&int_ptr = 0x%08x\n", &int_ptr);
11    printf("*int_ptr = 0x%08x\n\n", *int_ptr);
12
13    printf("int_var is located at 0x%08x and contains %d\n", &int_var, int_var);
14    printf("int_ptr is located at 0x%08x, contains 0x%08x, and points to %d\n\n",
15          &int_ptr, int_ptr, *int_ptr);
16 }
17
```

Line: 1 Column: 1 L C Tab Size: 4

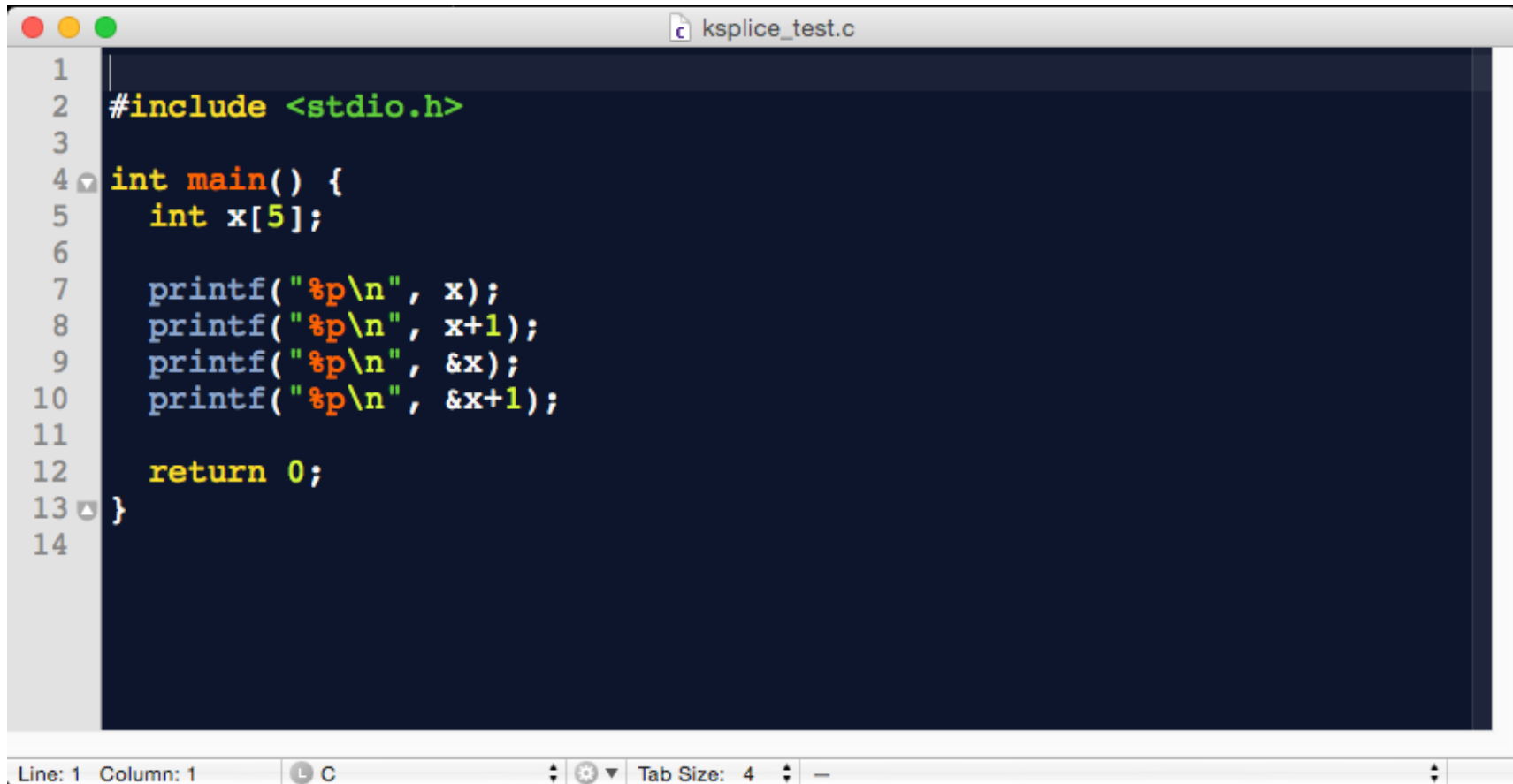


```
seed@seed-desktop: ~/CMPE279/modules/module2/hacking
File Edit View Terminal Help
seed@seed-desktop:~/CMPE279/modules/module2/hacking$ ./addressof2.out
int_ptr = 0xbffff4f0
&int_ptr = 0xbffff4ec
*int_ptr = 0x00000005

int_var is located at 0xbffff4f0 and contains 5
int_ptr is located at 0xbffff4ec, contains 0xbffff4f0, and points to 5

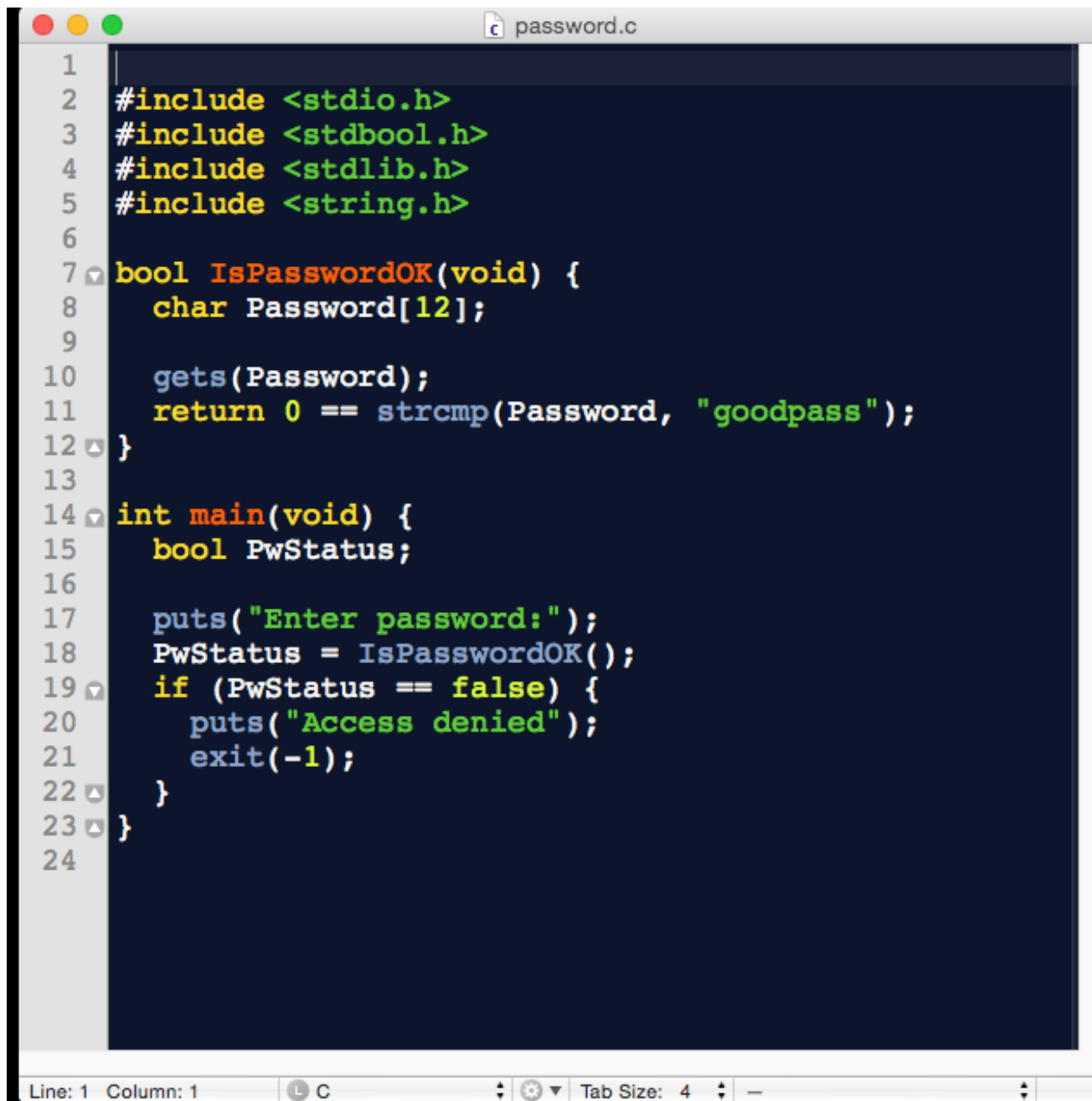
seed@seed-desktop:~/CMPE279/modules/module2/hacking$
```

Explain the Output of the Following...



```
1
2 #include <stdio.h>
3
4 int main() {
5     int x[5];
6
7     printf("%p\n", x);
8     printf("%p\n", x+1);
9     printf("%p\n", &x);
10    printf("%p\n", &x+1);
11
12    return 0;
13 }
14
```

Line: 1 Column: 1 L C Tab Size: 4



```
1
2 #include <stdio.h>
3 #include <stdbool.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 bool IsPasswordOK(void) {
8     char Password[12];
9
10    gets(Password);
11    return 0 == strcmp(Password, "goodpass");
12 }
13
14 int main(void) {
15     bool PwStatus;
16
17     puts("Enter password:");
18     PwStatus = IsPasswordOK();
19     if (PwStatus == false) {
20         puts("Access denied");
21         exit(-1);
22     }
23 }
24
```

Line: 1 Column: 1 C Tab Size: 4

