THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

CS330

C: Pointers

Spring 2022

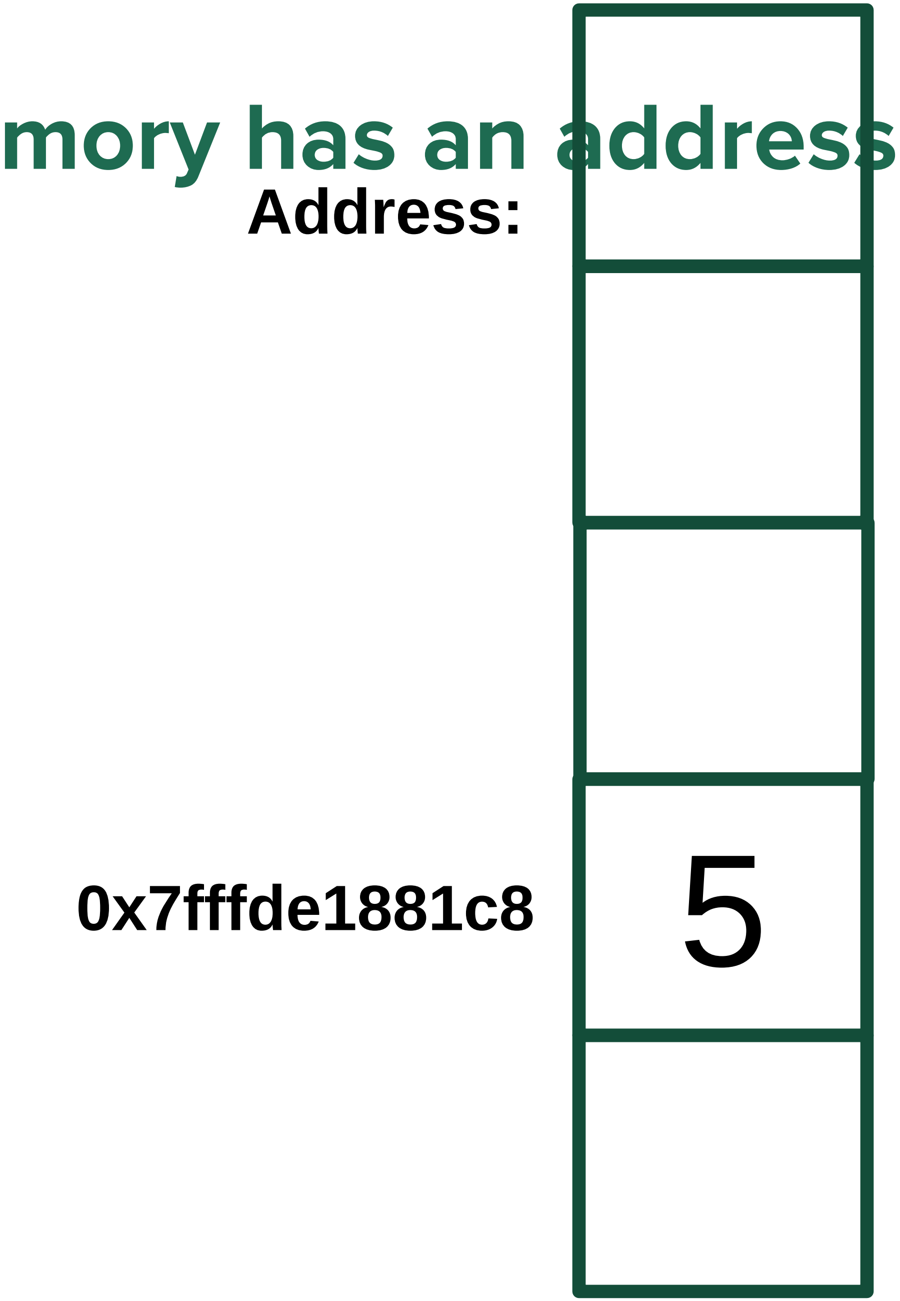
Lab 5

Big Idea #1: Every byte in memory has an address

- Perhaps helpful to think of memory as a big array

Index:	0	1	2	3	4	5	6
		5					

Big Idea #1: Every byte in memory has an address

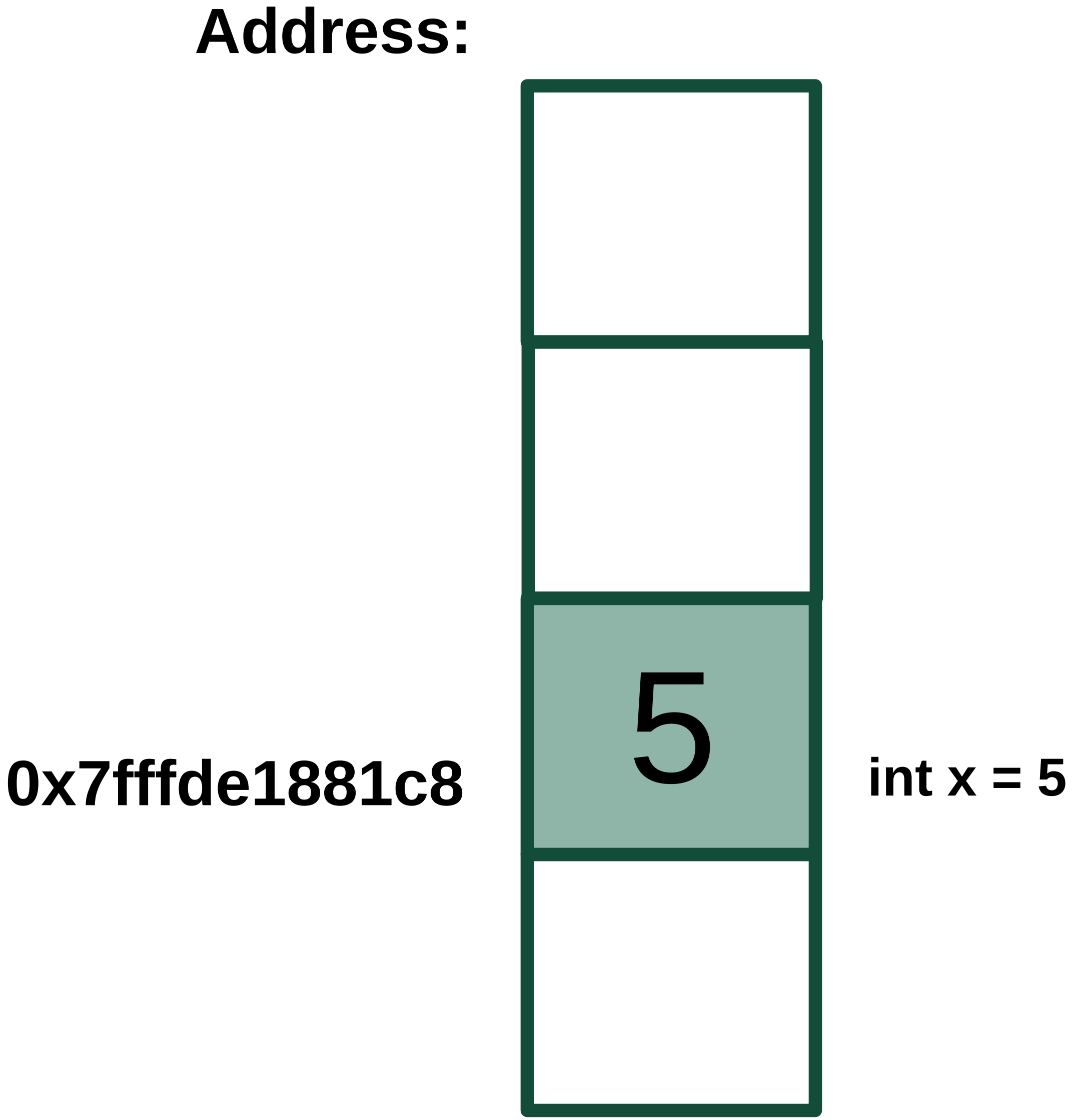


Address:

0x7fffde1881c8

5

Big Idea #2: The address of any variable is knowable



Address:

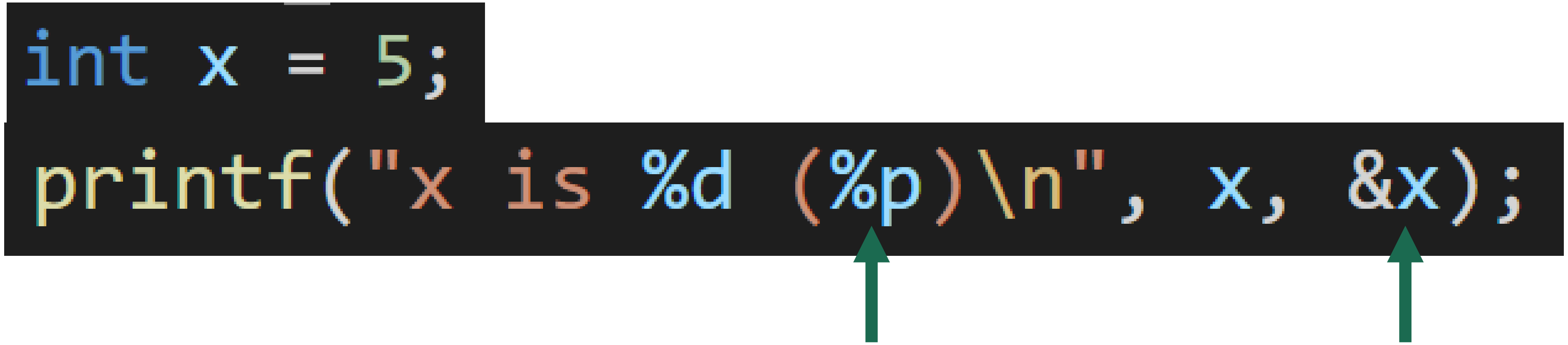
0x7fffde1881c8

5

int x = 5

Big Idea #2: The address of any variable is knowable

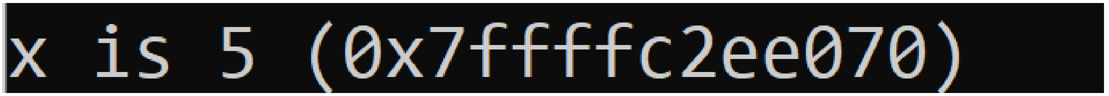
- We already know how to declare/initialize a variable, and how to print its address (see vars.c example):
- The ‘&’ operator



```
int x = 5;
printf("x is %d (%p)\n", x, &x);
```

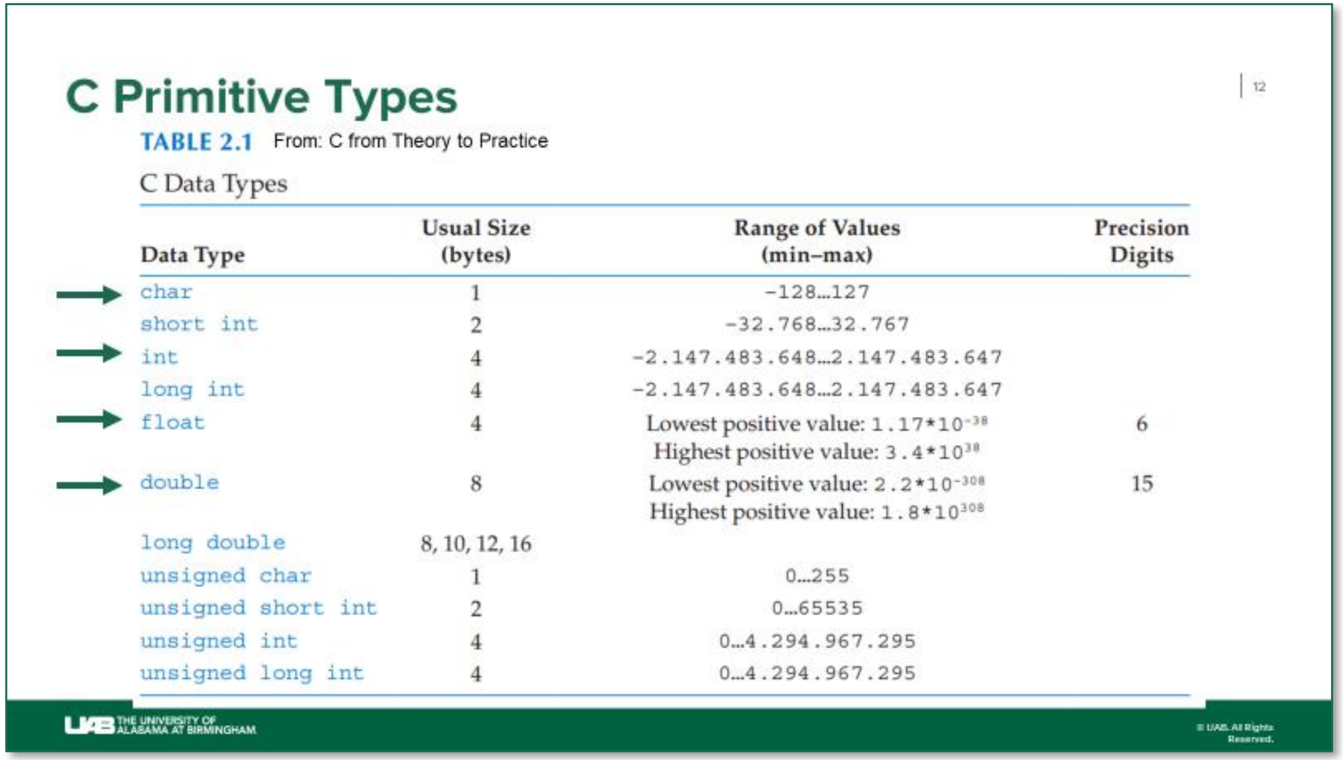
Print a pointer

Get the base address of x



```
x is 5 (0x7ffffc2ee070)
```

Big Idea #2: The address of any variable is knowable (base address)

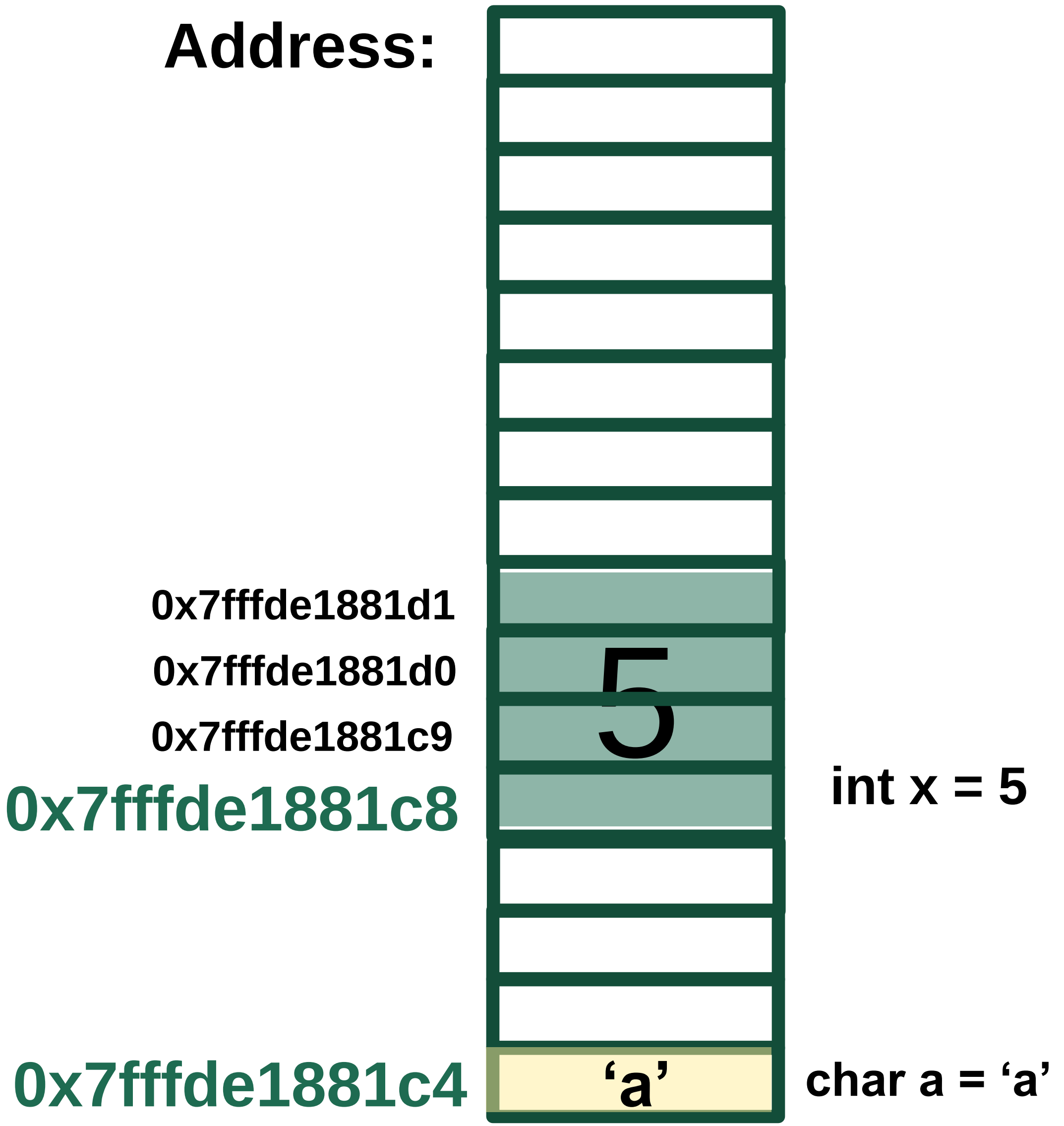


C Primitive Types

TABLE 2.1 From: C from Theory to Practice

C Data Types

Data Type	Usual Size (bytes)	Range of Values (min–max)	Precision Digits
→ char	1	-128…127	
short int	2	-32.768…32.767	
→ int	4	-2.147.483.648…2.147.483.647	
long int	4	-2.147.483.648…2.147.483.647	
→ float	4	Lowest positive value: $1.17*10^{-38}$ Highest positive value: $3.4*10^{38}$	6
→ double	8	Lowest positive value: $2.2*10^{-308}$ Highest positive value: $1.8*10^{308}$	15
long double	8, 10, 12, 16		
unsigned char	1	0…255	
unsigned short int	2	0…65535	
unsigned int	4	0…4.294.967.295	
unsigned long int	4	0…4.294.967.295	



Address:

0x7fffde1881d1

0x7fffde1881d0

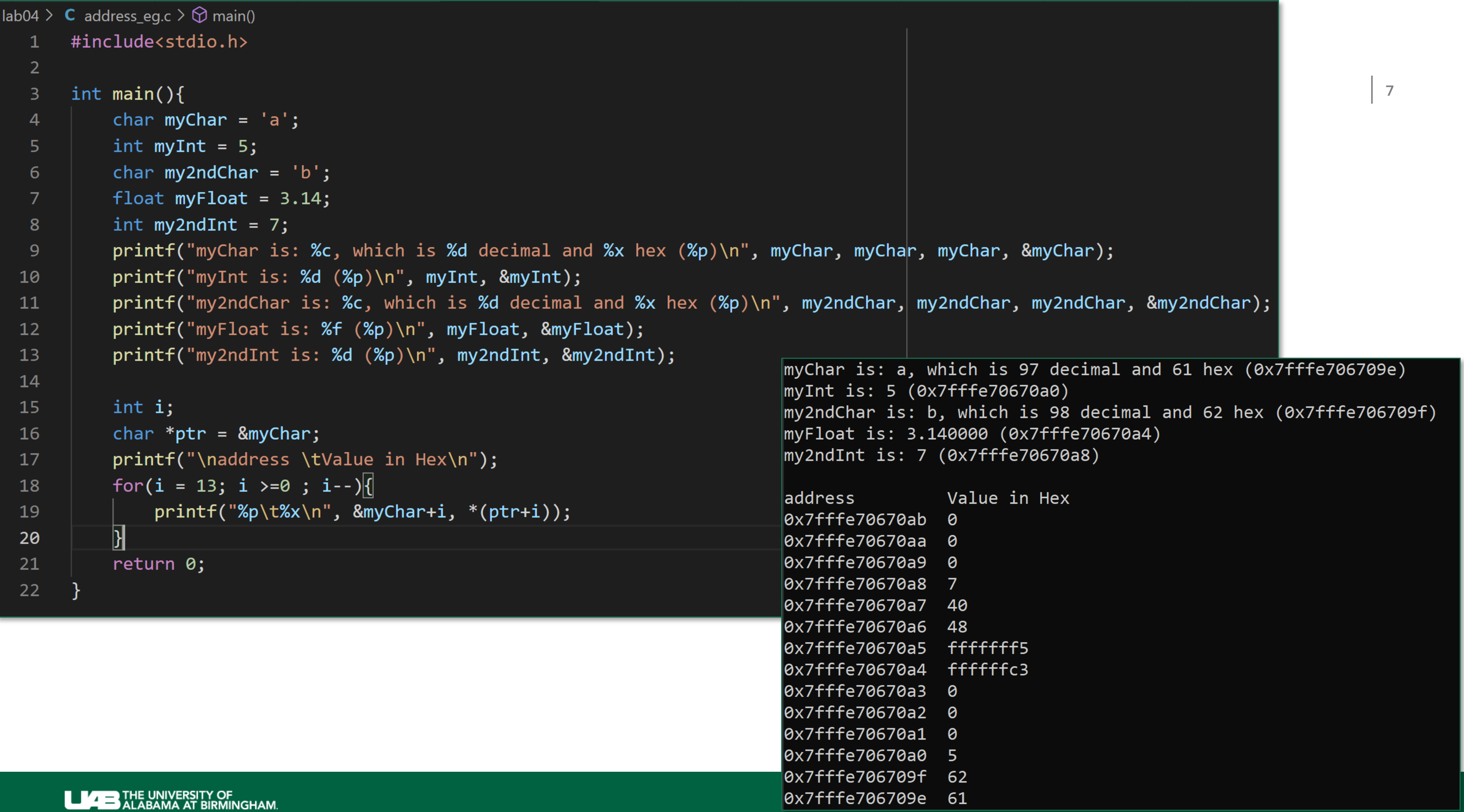
0x7fffde1881c9

0x7fffde1881c8

int x = 5

0x7fffde1881c4

char a = ‘a’

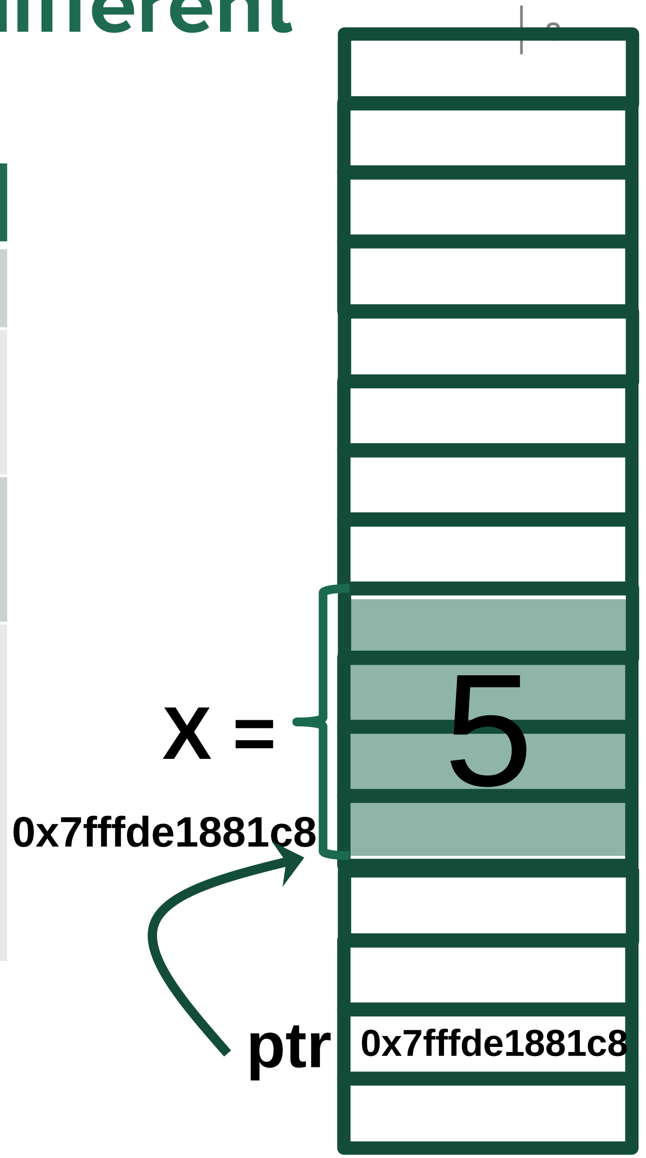


```c
#include<stdio.h>

int main(){
    char myChar = 'a';
    int myInt = 5;
    char my2ndChar = 'b';
    float myFloat = 3.14;
    int my2ndInt = 7;
    printf("myChar is: %c, which is %d decimal and %x hex (%p)\n", myChar, myChar, myChar, &myChar);
    printf("myInt is: %d (%p)\n", myInt, &myInt);
    printf("my2ndChar is: %c, which is %d decimal and %x hex (%p)\n", my2ndChar, my2ndChar, my2ndChar, &my2ndChar);
    printf("myFloat is: %f (%p)\n", myFloat, &myFloat);
    printf("my2ndInt is: %d (%p)\n", my2ndInt, &my2ndInt);

    int i;
    char *ptr = &myChar;
    printf("\naddress \tValue in Hex\n");
    for(i = 13; i >=0 ; i--){
        printf("%p\t%x\n", &myChar+i, *(ptr+i));
    }
    return 0;
}
```

```
myChar is: a, which is 97 decimal and 61 hex (0x7fffe706709e)
myInt is: 5 (0x7fffe70670a0)
my2ndChar is: b, which is 98 decimal and 62 hex (0x7fffe706709f)
myFloat is: 3.140000 (0x7fffe70670a4)
my2ndInt is: 7 (0x7fffe70670a8)

address         Value in Hex
0x7fffe70670ab  0
0x7fffe70670aa  0
0x7fffe70670a9  0
0x7fffe70670a8  7
0x7fffe70670a7  40
0x7fffe70670a6  48
0x7fffe70670a5  fffffff5
0x7fffe70670a4  ffffffc3
0x7fffe70670a3  0
0x7fffe70670a2  0
0x7fffe70670a1  0
0x7fffe70670a0  5
0x7fffe706709f  62
0x7fffe706709e  61
```

Big Idea #3: The address can be stored in a different variable: the Pointer

syntax	example	description
&<variable name>	&x	Provides the **address** of the variable x
<type> *<name>	int *ptr	**Declares** ptr is a Pointer to an int This also means: *ptr is an int
	ptr = &x	This **initializes** the pointer ptr points to x
*<pointer name>	int y = *ptr	**Dereferences** the pointer, And provides the value of the variable at that address y = the value of the variable ptr points to, or y = value of x, or y = x



A **Pointer** is a variable that contains the address of a variable – K&R, pg 93

Pointers

- * is an overloaded operator
  - Usually means multiplication
  - Used to declare a pointer: int *myPtr
  - Used to dereference a pointer: int y = *myPtr;

- When declaring a pointer, whitespace doesn’t matter. All of these are valid:

```
int* ptr, x, y;
int * ptr, x, y;
int *ptr, x, y;
```

← recommend this one (in all cases, x and y are ints)

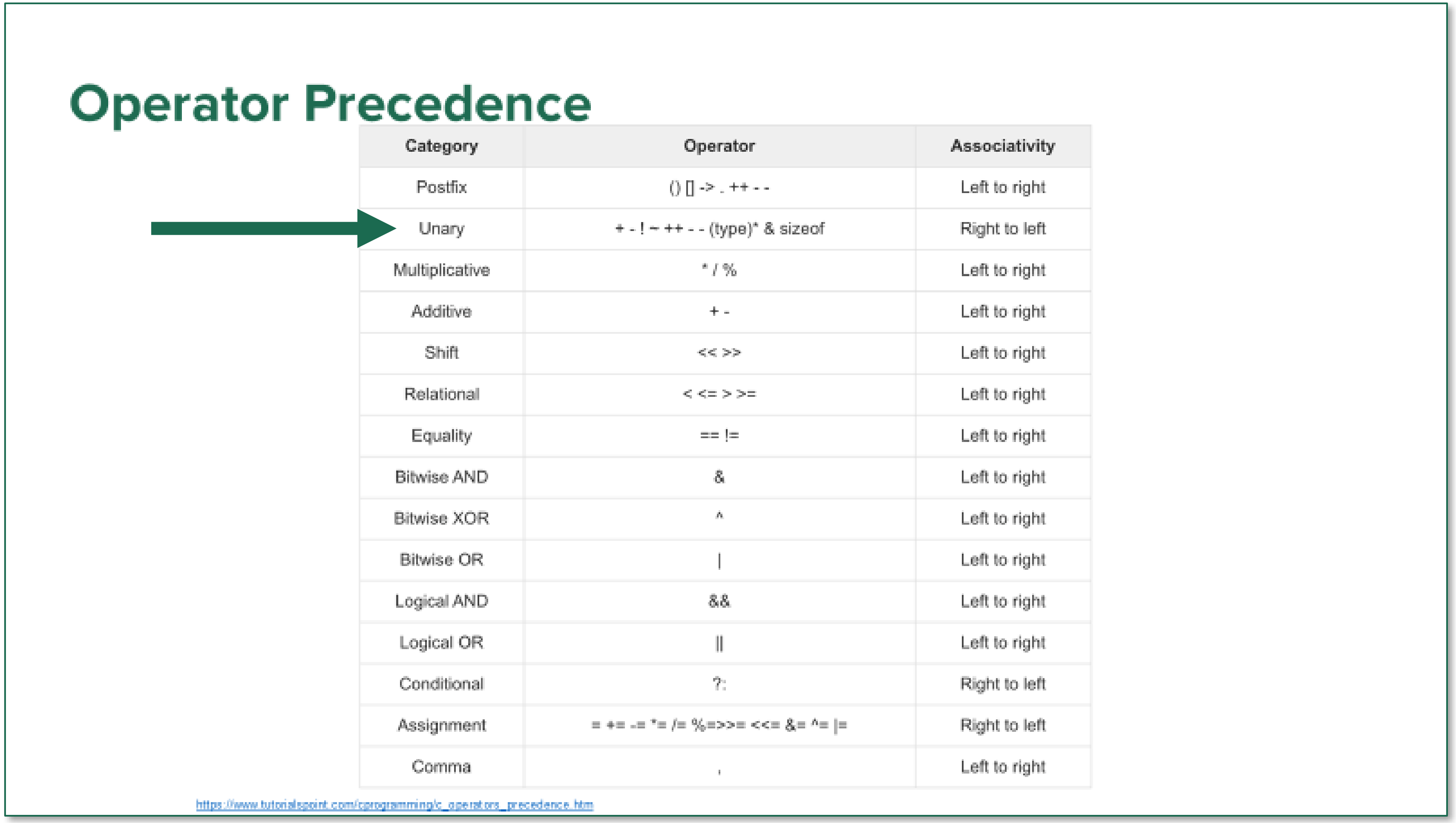
Pointers

- The type of pointer must be specified in the declaration, otherwise the pointer doesn’t know what it is pointing to
  - And how many bytes after the address are needed. The Pointer points to the base address.
    - char *charPtr; // just need one byte
    - int *intPtr; // need four bytes
    - double *doublePtr; // eight bytes
  - And how to dereference it later
  - Can have void type: void *myPtr;

- Pointers must be initialized before they are used
  - Otherwise the program will SegFault
  - int *myPtr = &x;
  - Or myPtr = &x;

Pointers

- Order of Operations
  - Only Postfix is higher



Operator Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm

Pointers

- We can create pointers to pointers: int **myPtrToPtr
  - char **argv
  - And a pointer to a pointer to a pointer: int ***   …etc

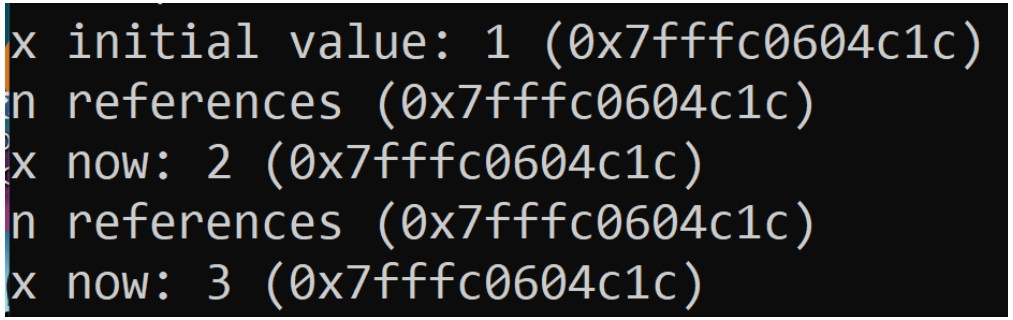
- We can create pointers to functions (a little advanced)

- Typically a Pointer is sized at 8 bytes on our 64 bit computers, regardless of which variable type the Pointer is pointing to
  - Can confirm with sizeof()
  - printf("The size of my pointer is %lu\n", sizeof(myPtr));

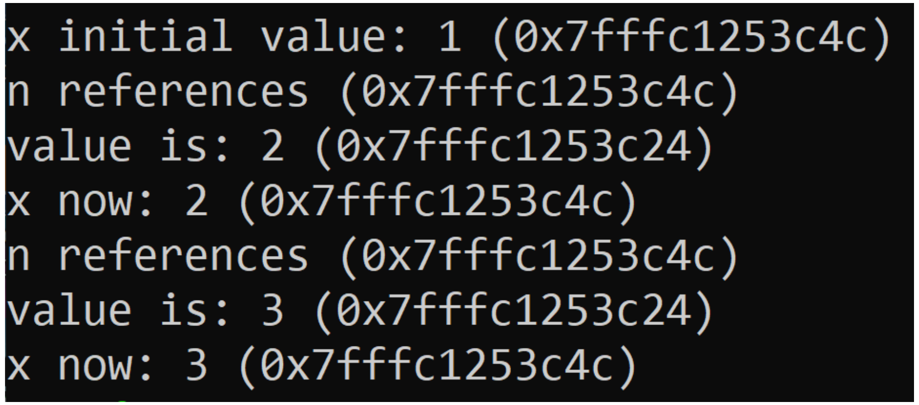
Why?

- Lots of reasons
- Let's start with Functions
  - We can pass a pointer into a function as an argument. Or return a pointer from a function.
- Recall:
  - Pass By Value: creates a copy of the variable. The function modifies the copy, and original is unchanged
  - Pass By Reference: passes in a reference to the variable. The function modifies the original
- Useful when we have a large amount of data (e.g. array) and we don't want to waste time/space to make a copy

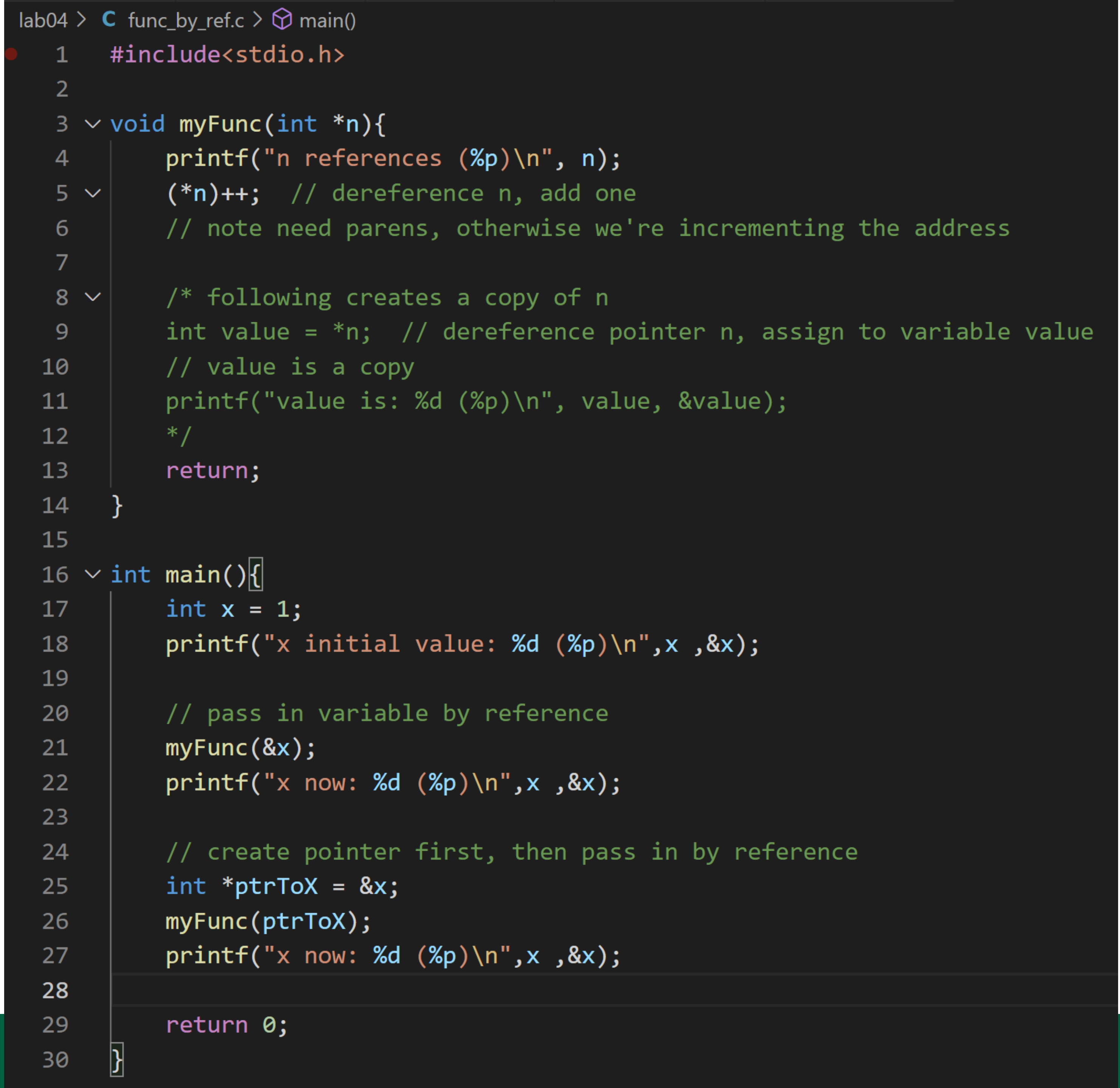
Function Example



```
x initial value: 1 (0x7fffc0604c1c)
n references (0x7fffc0604c1c)
x now: 2 (0x7fffc0604c1c)
n references (0x7fffc0604c1c)
x now: 3 (0x7fffc0604c1c)
```



```
x initial value: 1 (0x7fffc1253c4c)
n references (0x7fffc1253c4c)
value is: 2 (0x7fffc1253c24)
x now: 2 (0x7fffc1253c4c)
n references (0x7fffc1253c4c)
value is: 3 (0x7fffc1253c24)
x now: 3 (0x7fffc1253c4c)
```

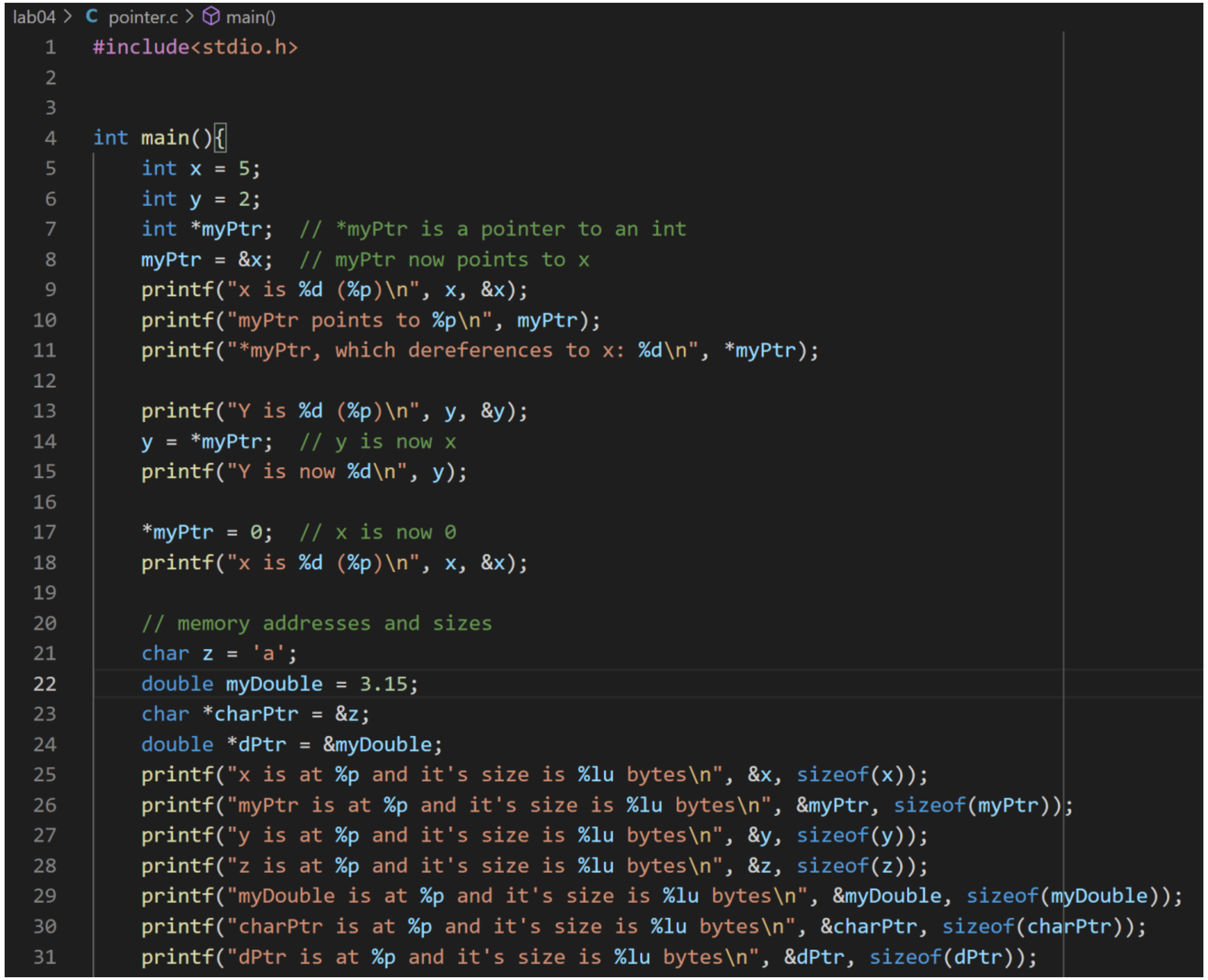


```c
#include<stdio.h>

void myFunc(int *n){
    printf("n references (%p)\n", n);
    (*n)++;  // dereference n, add one
    // note need parens, otherwise we're incrementing the address

    /* following creates a copy of n
    int value = *n;  // dereference pointer n, assign to variable value
    // value is a copy
    printf("value is: %d (%p)\n", value, &value);
    */
    return;
}

int main(){
    int x = 1;
    printf("x initial value: %d (%p)\n",x ,&x);

    // pass in variable by reference
    myFunc(&x);
    printf("x now: %d (%p)\n",x ,&x);

    // create pointer first, then pass in by reference
    int *ptrToX = &x;
    myFunc(ptrToX);
    printf("x now: %d (%p)\n",x ,&x);

    return 0;
}
```

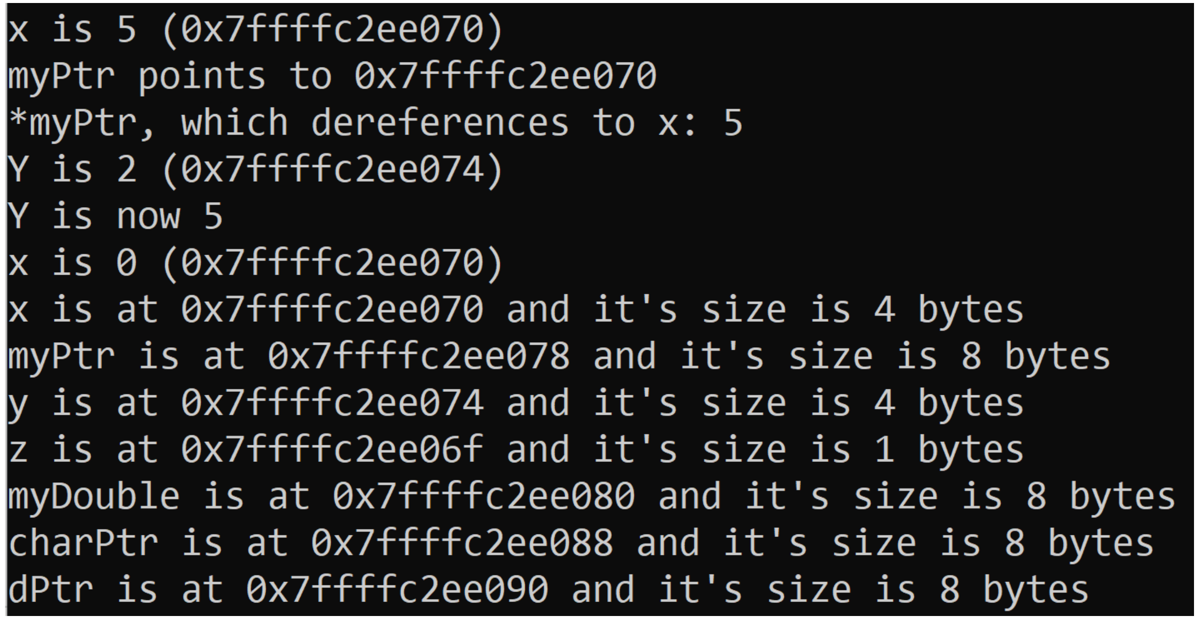
Exercise 1

- Create a variable, any type
- Create a pointer to that variable
- Print out the variable, the variable address (&x), and the pointer
  - Variable address and pointer should match
- Dereference the pointer, and print that value
  - Value should match original variable value
- Use the pointer to change the variable (e.g. increase by one)
- Print the variable

Example – Just the Basics



```c
#include<stdio.h>


int main(){
    int x = 5;
    int y = 2;
    int *myPtr;  // *myPtr is a pointer to an int
    myPtr = &x;  // myPtr now points to x
    printf("x is %d (%p)\n", x, &x);
    printf("myPtr points to %p\n", myPtr);
    printf("*myPtr, which dereferences to x: %d\n", *myPtr);

    printf("Y is %d (%p)\n", y, &y);
    y = *myPtr;  // y is now x
    printf("Y is now %d\n", y);

    *myPtr = 0;  // x is now 0
    printf("x is %d (%p)\n", x, &x);

    // memory addresses and sizes
    char z = 'a';
    double myDouble = 3.15;
    char *charPtr = &z;
    double *dPtr = &myDouble;
    printf("x is at %p and it's size is %lu bytes\n", &x, sizeof(x));
    printf("myPtr is at %p and it's size is %lu bytes\n", &myPtr, sizeof(myPtr));
    printf("y is at %p and it's size is %lu bytes\n", &y, sizeof(y));
    printf("z is at %p and it's size is %lu bytes\n", &z, sizeof(z));
    printf("myDouble is at %p and it's size is %lu bytes\n", &myDouble, sizeof(myDouble));
    printf("charPtr is at %p and it's size is %lu bytes\n", &charPtr, sizeof(charPtr));
    printf("dPtr is at %p and it's size is %lu bytes\n", &dPtr, sizeof(dPtr));
```



```
x is 5 (0x7ffffc2ee070)
myPtr points to 0x7ffffc2ee070
*myPtr, which dereferences to x: 5
Y is 2 (0x7ffffc2ee074)
Y is now 5
x is 0 (0x7ffffc2ee070)
x is at 0x7ffffc2ee070 and it's size is 4 bytes
myPtr is at 0x7ffffc2ee078 and it's size is 8 bytes
y is at 0x7ffffc2ee074 and it's size is 4 bytes
z is at 0x7ffffc2ee06f and it's size is 1 bytes
myDouble is at 0x7ffffc2ee080 and it's size is 8 bytes
charPtr is at 0x7ffffc2ee088 and it's size is 8 bytes
dPtr is at 0x7ffffc2ee090 and it's size is 8 bytes
```

Summary

- Big Idea #1: Every byte in memory has an address
- Big Idea #2: The address of any variable is knowable
  - We mostly don't care about what the number is, or endianness. Unless debugging, creating new chips, sharing info with another computer (network endianness).
- Big Idea #3: The address can be stored in a different variable: the Pointer
- Declare pointer, use pointer, dereference pointer
- Ptr must be initialized before first use, otherwise error
- Ptrs are 8 bytes (in our machines)
- Ptrs point to base address, or first byte, the type tells the compiler how many bytes to consider after the first byte
- Pass by reference
- Sizeof()