CMSC202
Computer Science II for Majors

Lecture 04 –
Pointers

Dr. Katherine Gibson

Last Class We Covered

- C++ Functions
  - Parts of a function:
    - Prototype
    - Definition
    - Call
- Arrays
  - Declaration
  - Initialization
- Passing arrays to function

Any Questions from Last Time?

Note Taker Still Needed

A peer note taker is still needed for this class. A peer note taker is a volunteer student who provides a copy of his or her notes for each class session to another member of the class who has been deemed eligible for this service based on a disability. Peer note takers will be paid a $200 stipend for their service. Peer note taking is not a part time job but rather a volunteer service for which enrolled students can earn a stipend for sharing the notes they are already taking for themselves.

If you are interested in serving in this important role, please fill out a note taker application on the Student Support Services website or in person in the SSS office in Math/Psychology 213.

Today's Objectives

- To review functions and how they work
- To *begin* to understand pointers
  - Pointers are a complicated and complex concept
  - You may not immediately "get it" – that's fine!
- To learn how pointers can be used in functions
  - Passing in entire arrays
  - "Returning" more than one value

Functions and Arguments

Review of Functions

- Here is a simple function that adds one to an integer and returns the new value
  - Definition:

```
int AddOne (int  num) {
    return num++;
}
```

  - Call:

```
int enrolled = 99;
enrolled = AddOne(enrolled);
```

Function Arguments

- What is happening "behind the scenes"?
- When the **`AddOne()`** function is called, the value of the variable is passed in as an argument
  - The value is saved in **`AddOne`**'s local variable **`num`**
- Changes made to **`x`** do not affect anything outside of the function **`AddOne()`**
  - This is called the *scope* of the variable

Scope

- Scope is the "visibility" of variables
  - Which parts of your program can "see" a variable
- Every function has its own scope:
  - The **`main()`** function has a set of variables
  - So does the **`AddOne()`** function
- They can't "see" each other's variables
  - Which is why we must pass arguments and return values between functions

Addresses

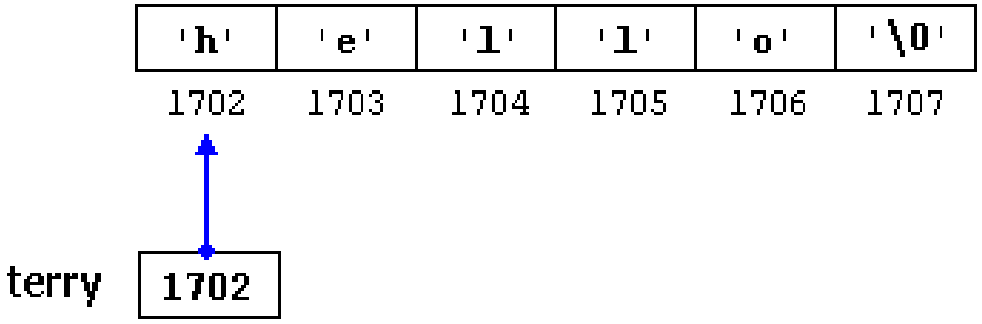
- Every variable in a program is stored somewhere in the computer's *memory*
  - This location is called the address
  - All variables have a unique address

- Addresses are normally expressed in hex:
  - `0xFF00`
  - `0x70BF`
  - `0x659B`

Array Addresses

- An array also has an address
  - The location of the first element of the array

```
char terry[6] = "hello";
```



- We'll discuss arrays more later today

Function Scope

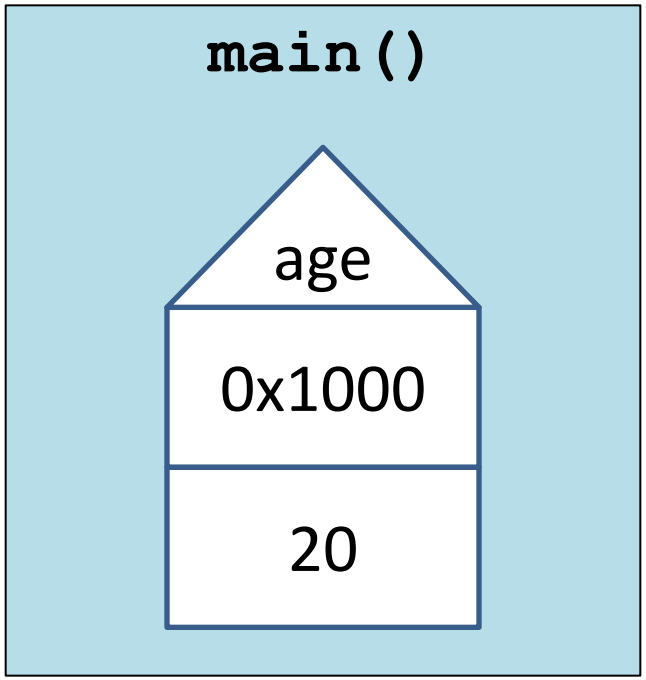
- What happens when **`AddOne()`** is called?

```
int age = 20;
age = AddOne(age);
```

- The value of **`age`** is passed in, and stored in another variable called **`num`**
  - What is the *scope* of each of these variables?
  - **`age`** is in the scope of **`main()`**
  - **`num`** is in the scope of **`AddOne()`**

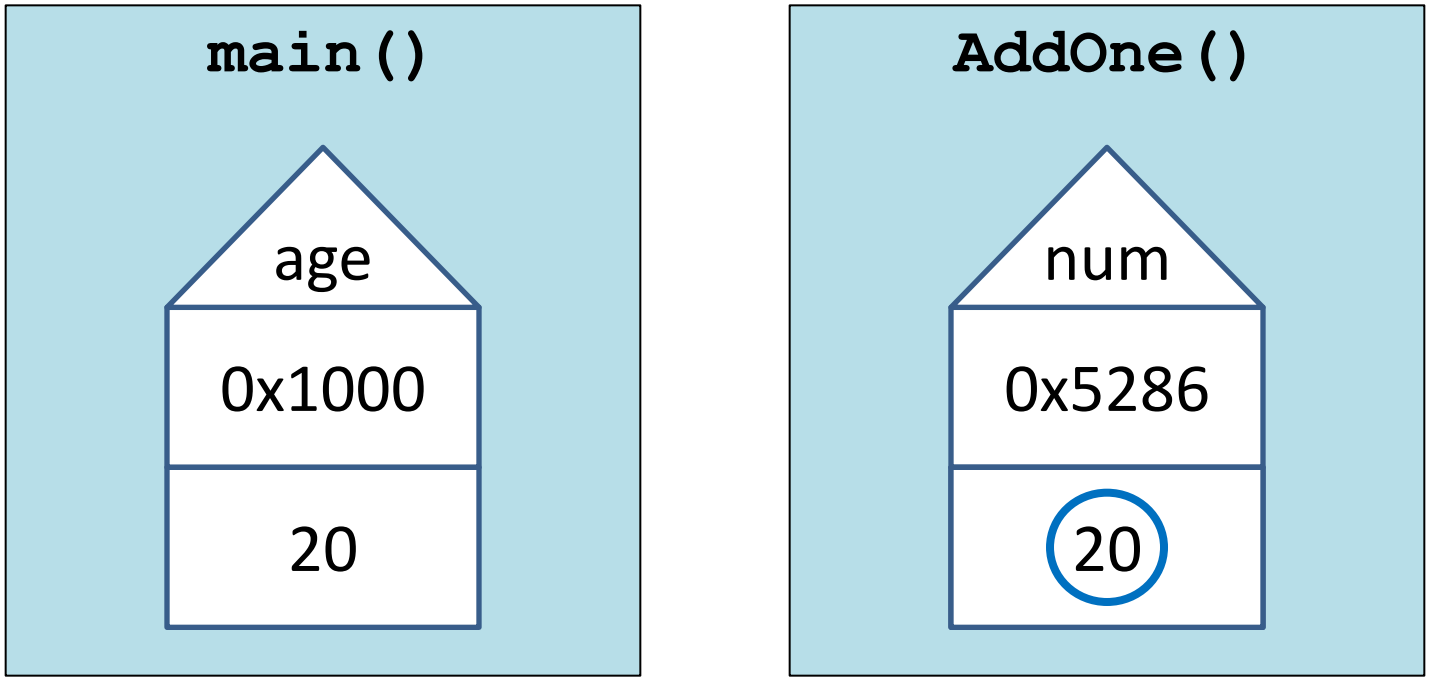
Function Calls

- The blue box represents scope
- The "house" shape is a variable's name, address, and value



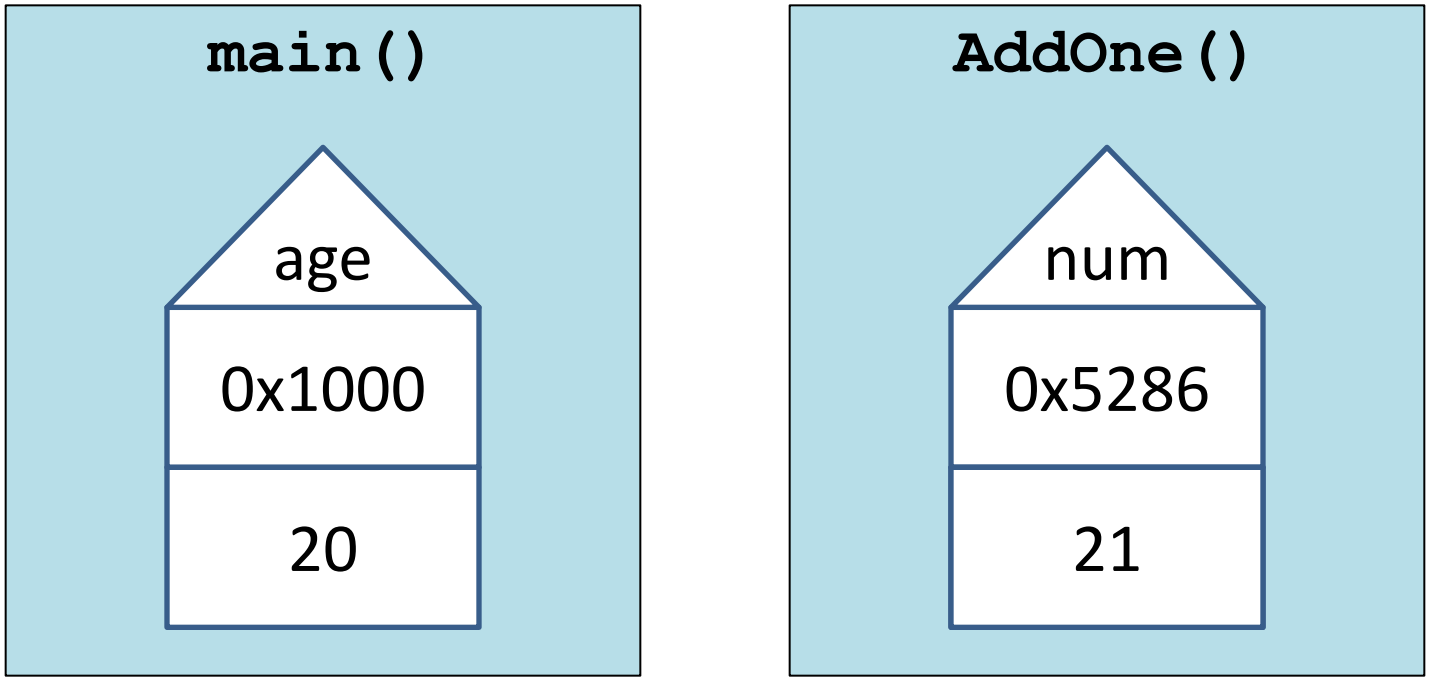
Function Calls

- When **`main()`** calls **`AddOne()`**
  - The value is passed in, and stored in **`num`**



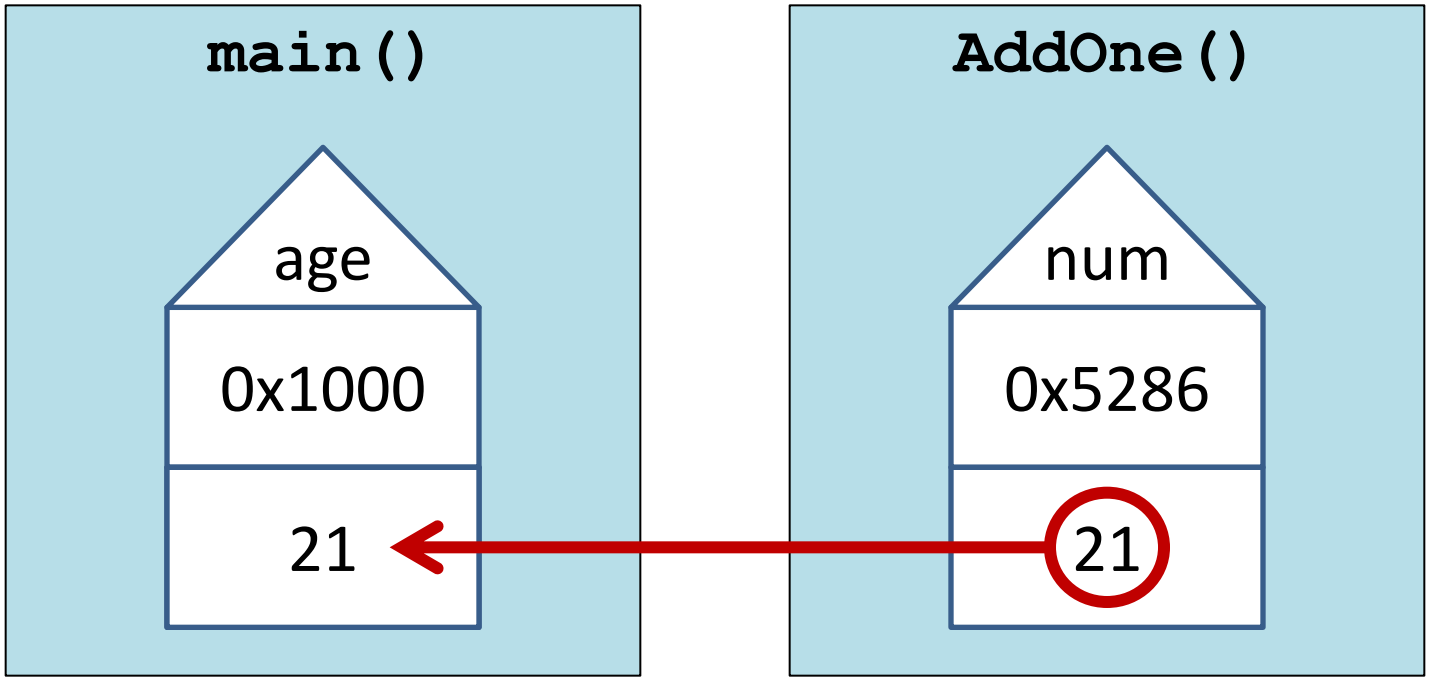
Function Calls

- When the **`AddOne()`** function changes **`num`**, what happens to the **`age`** variable?
  - Nothing!



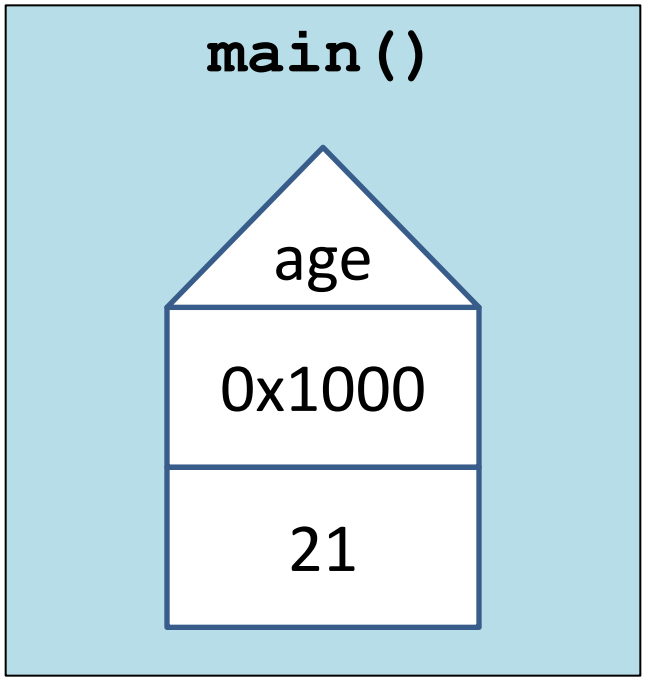
Function Calls

- How do we update the value of **`age`**?
  - By *returning* the new value and assigning it to **`age`**



Function Calls

- What happens when the function returns?
  - The function is over
  - **`AddOne()`** and **`num`** are "out of scope"



And are no longer available to us!

Pointer Introduction

Pointers

- A pointer is a variable whose value is an address to somewhere in memory

```
cout << "x   is " << x << endl;
cout << "ptr is " << ptr << endl;
```

- This will print out something like:

```
x   is 37
ptr is 0x7ffedcaba5c4
```

Pointers

- Pointers are incredibly useful to programmers!

- Allow functions to
  - Modify multiple arguments
  - Use and modify arrays as arguments
- Programs can be made more efficient
- Dynamic objects can be used
  - We'll discuss this later in the semester

Creating Pointers

- A pointer is just like any regular variable
  - It must have a type
  - It must have a name
  - It must contain a value

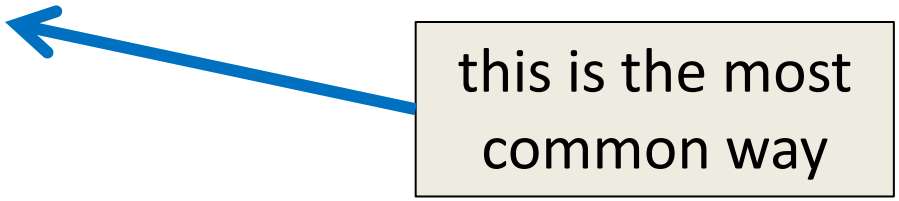
- To tell the compiler we're creating a pointer, we need to use * in the declaration

```
int *myPtr;
```

Pointer Declarations

- All of the following are valid declarations:

```
int *myPtr;
int* myPtr;
int * myPtr;
```



  - Even this is valid (but don't do this):

```
int*myPtr;
```

- The spacing and location of the star ("*") don't matter to the compiler

Pointer Declarations

- Since position doesn't matter, why use this?

```
int *myPtr;
```

- What does this code do?

```
int *myPtr,  yourPtr,  ourPtr;
```
✘

  – It creates one pointer and two integers!

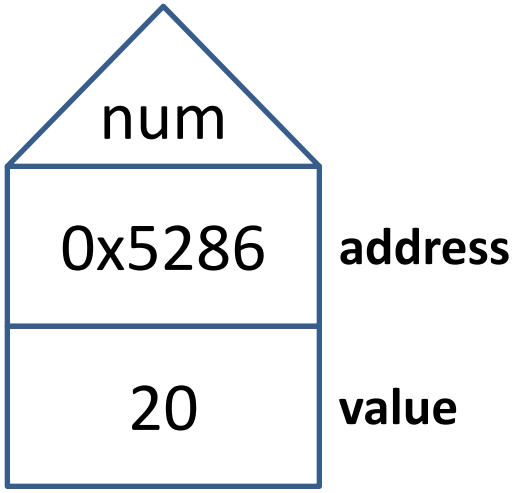
- What does this code do?

```
int *myPtr, *yourPtr, *ourPtr;
```
✓

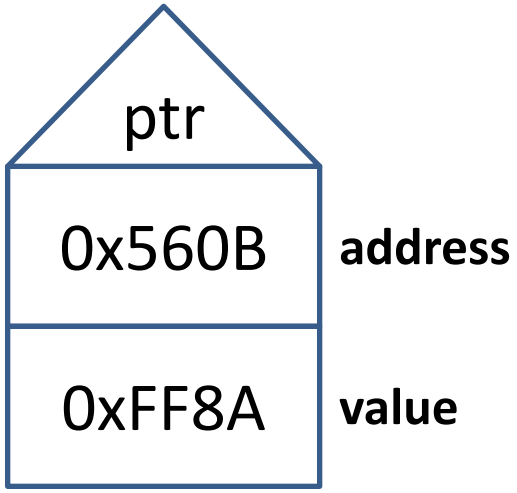
  – It creates three integers!

Pointers and "Regular" Variables

- As we said earlier, pointers are just variables
  - Instead of storing an int or a float or a char, they store an address in memory



"regular" variable



(where it lives in memory)

(where it points in memory)

pointer variable

Assigning Value to a Pointer

- The value of a pointer is always an address
- To get the address of any variable, we use an ampersand ("**&**")

```
int  x = 5;
int *xPtr;
// xPtr "points to" x
xPtr = &x;
```

Assigning to a Pointer

- All of these are valid assignments:

```
int  x = 5;
int *ptr1 = &x;
int *ptr2;
ptr2 = &x;
int *ptr3 = ptr1;
```

Assigning to a Pointer

- This is not a valid assignment – why?

```
int  x = 5;
char *ptr4 = &x;
```

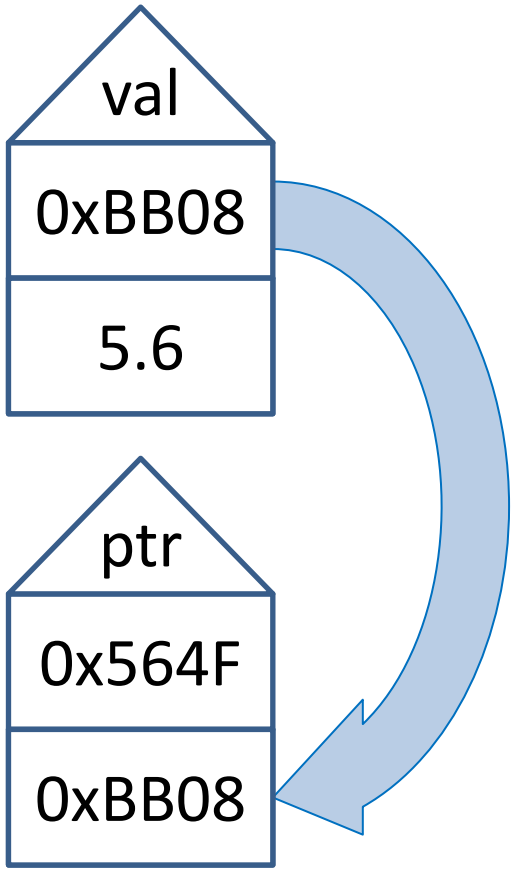
- Pointer type must match the type of the variable whose address it stores
- Compiler will give you an error:

```
cannot convert 'int*' to 'char*' in initialization
```

Making Pointers "Point"

- When we assign a value to a pointer, we are telling it where in memory to point to

```
// create both variables
double val;
double *ptr;
// assign values
val = 5.6;
ptr = &val;
```



The Asterisk and the Ampersand

Review: The Ampersand

- The ampersand
  - Returns the address of a variable
  - Must be placed in front of the variable name

```
int  x = 5;
int *varPtr = &x;
int  y = 7;
varPtr = &y;
```

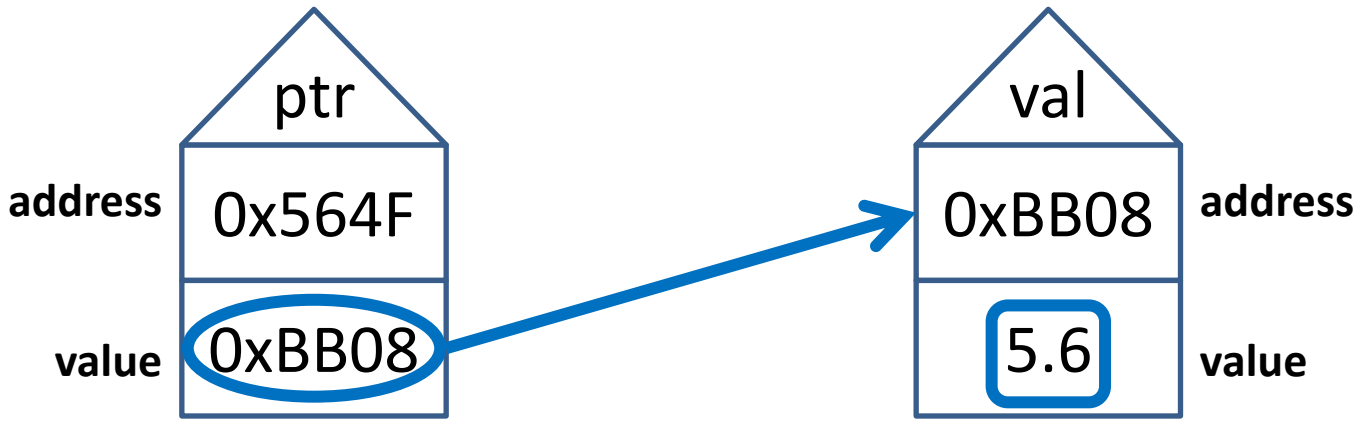
The Asterisk (or "Star")

- The star symbol ("*") has two purposes when working with pointers
- The first purpose is to tell the compiler that the variable will store an address
  - In other words, "declaring a pointer"

```
int *varPtr = &x;
void fxnName (float *fltPtr);
```

The Asterisk (or "Star")

- The second purpose is to *dereference* a pointer
- Dereferencing a pointer means the compiler
  - Looks at the address stored in the pointer
  - Goes to that address in memory
  - Looks at the value stored at that address



Dereferencing

- What we do at that point depends on why the pointer is being dereferenced

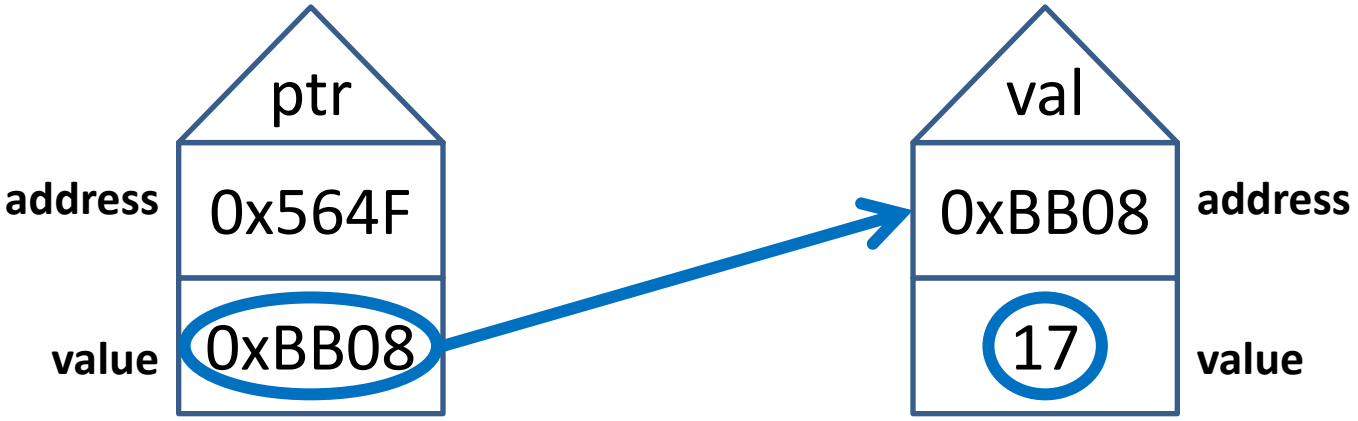
- A dereference can be in three "places"
  - On the left hand side of the assignment operator
  - On the right hand side of the assignment operator
  - In an expression without an assignment operator
    - For example, a print statement

Dereferencing Examples

```
int val = *ptr;
```

on the right hand side of the assignment operator

- Look at the value, but don't change it

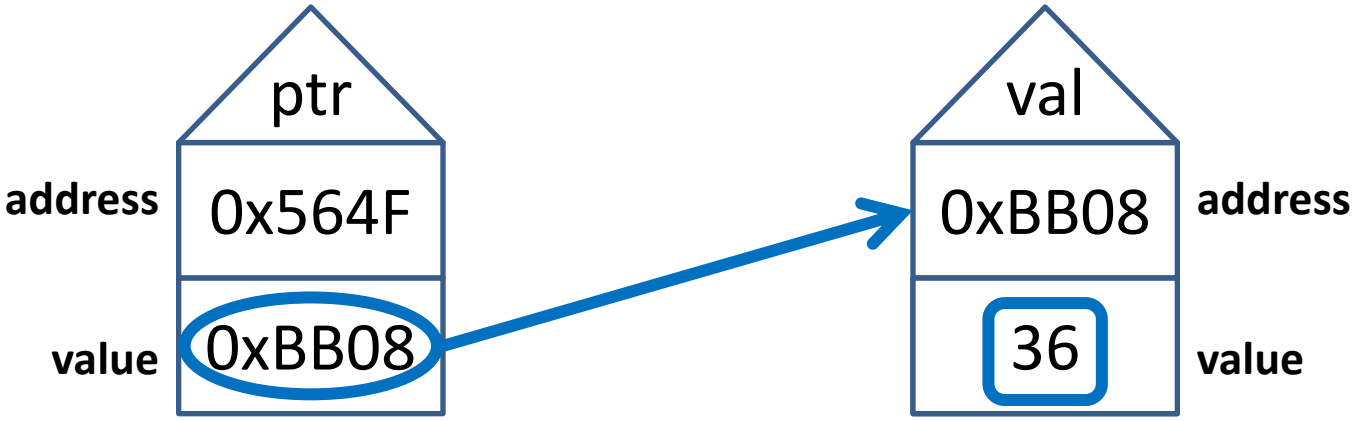


Dereferencing Examples

```
*ptr = 36;
```

on the left hand side of the assignment operator

- Access the variable and change its value

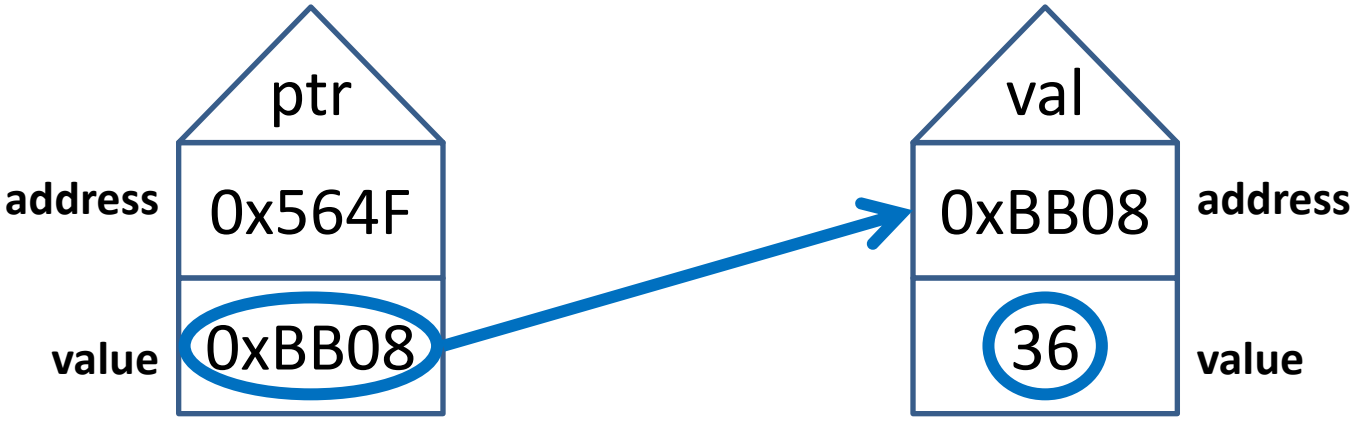


Dereferencing Examples

```
cout << "Value stored is " << *ptr;
```

in an expression without an assignment operator

- Look at the value, but don't change it



AddTwo()

The `AddTwo()` Function

- Let's create a new function that adds 2 to two integers
  - So 22 and 98 will become 24 and 100
- Can we do this with a "regular" function?
  - (That is, without using pointers?)
  - No! Functions can only *return* one value!
- We must use pointers to change more than one value in a single function

The `AddTwo()` Function

- We want our function to look something like this pseudocode:

```
// take in two ints, return nothing
void AddTwo( <two integers> ) {
    // add two to the first int
    // add two to the second int
    // keep the values -- but how?
}
```

Passing Pointers to a Function

- To tell the compiler we are passing an address to a function, we will use `int *varPtr`

```
void AddTwo (int *ptr1, int *ptr2)
```

- Just like `int num` tells the compiler that we are passing in an integer value

```
int AddOne (int num)
```

Writing AddTwo()

- Given that **AddOne()** looks like this:

```
int AddOne (int num) {
   return num++;
}
```

- How do we write the AddTwo function?

```
void AddTwo (int *ptr1, int *ptr2) {



}
```

AddTwo()

```
void AddTwo (int *ptr1, int *ptr2) {
    /* add two to the value of the
       integer ptr1 points to */
    *ptr1 = *ptr1 + 2;
    /* add two to the value of the
       integer ptr2 points to */
    *ptr2 = *ptr2 + 2;
    /* return nothing */
}
```

Calling `AddTwo()`

- Now that the function is defined, let's call it

- It takes in the address of two integers
  - Pass it two int pointers:

```
AddTwo(numPtr1, numPtr2);
```

  - Pass it the addresses of two ints:

```
AddTwo(&num1,    &num2);
```

  - Pass it a combination:

```
AddTwo(numPtr1, &num2);
```

Literals and Pointers

- What about the following – does it work?

```
AddTwo(&15, &3);
```

- No! **15** and **3** are literals, not variables
  - They are not stored in memory
  - They have no address
  - (They're homeless!)

Announcements

- The course policy agreement is due today
- Project 1 has been released
  - Found on Professor's Marron website
  - Due by 9:00 PM on February 23rd
- Get started on it now!
- Next time: References
  - And a review of pointers