Comp 11 Lectures

Mike Shah

Tufts University

June 26, 2017

Please do not distribute or host these slides without prior permission.

Pointers

Comp 11 - Pre-Class warm up

If I write the address to your house down on an envelope, I can conveniently send that letter to you and retrieve some response.

There are alternatives though–I could just move you and your house to where I am standing! Or just rebuild a copy of your house next to me (and also clone you).

Lecture

Radia Perlman

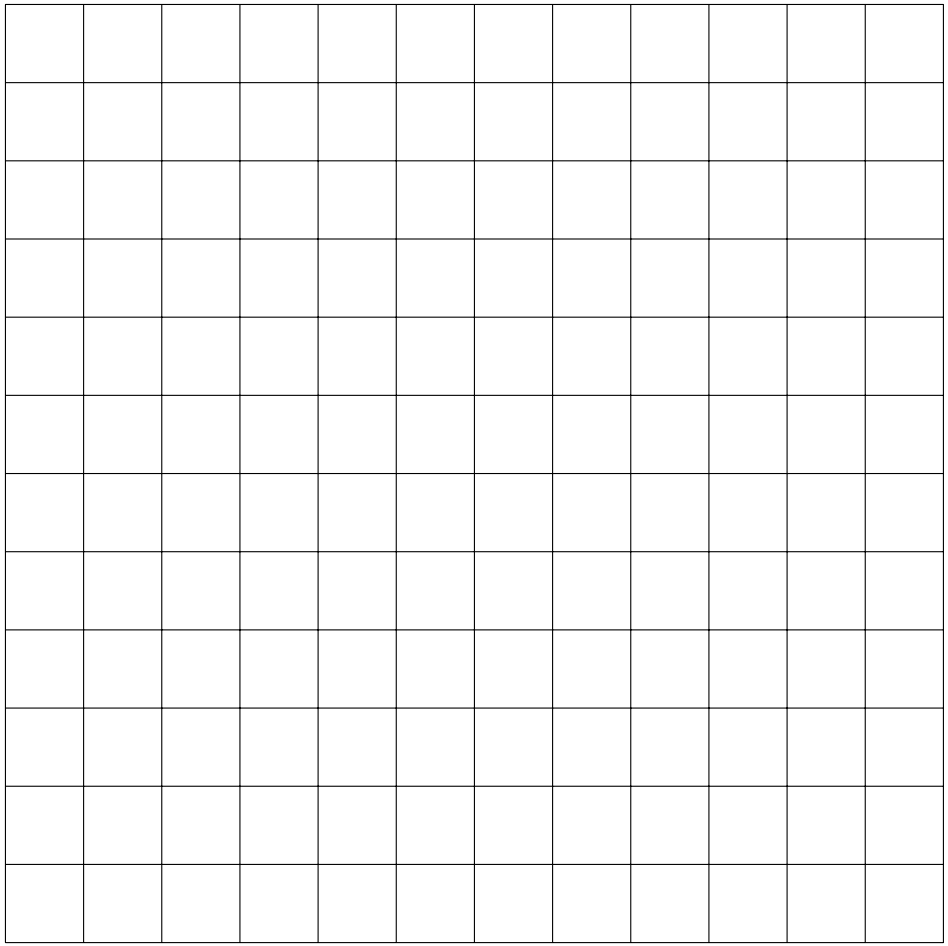
Figure 1: Radia Perlman is a MIT Ph.D. graduate best known for her work in computer networks. She holds over 100 patents and is responsible for transforming how Ethernets are able to utilized to create large networks

Memory and Pass by Reference

Memory

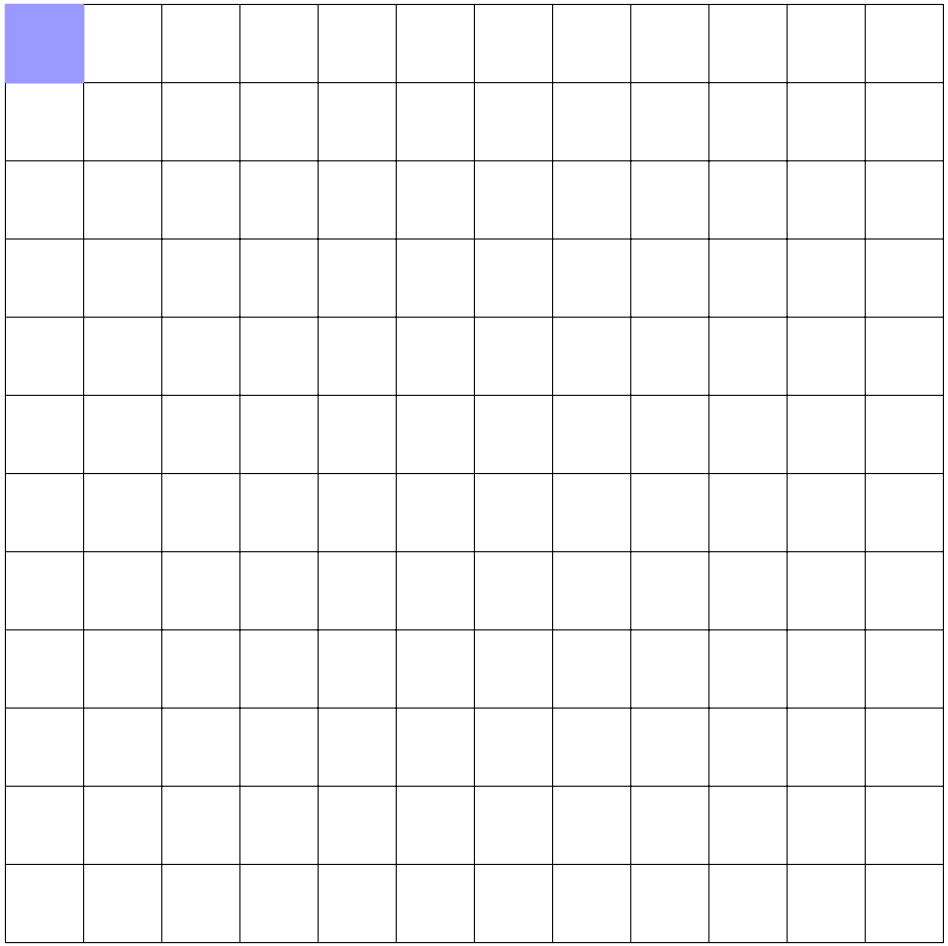
- We have talked about memory briefly when we talked about arrays.
- A modern computer these days, has quite a bit of memory available.
- You have memory in your hard disk, memory in RAM, and storage in the Cache.
- These are all different levels of memory, and our operating system helps us manage where it goes.
- As an example, your operating system hands you memory, when you run your program

Here is some Memory when you start your program



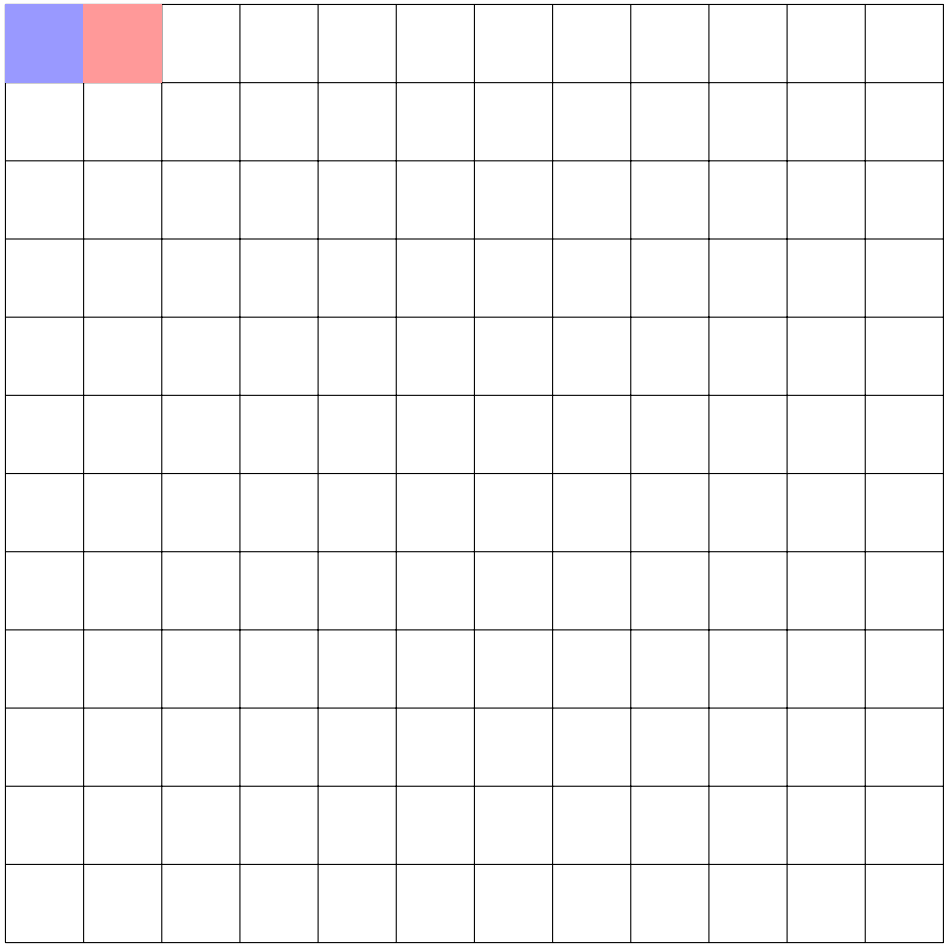
(Note that each box represents 4 bytes of memory in the following slides.)

Then you start storing things



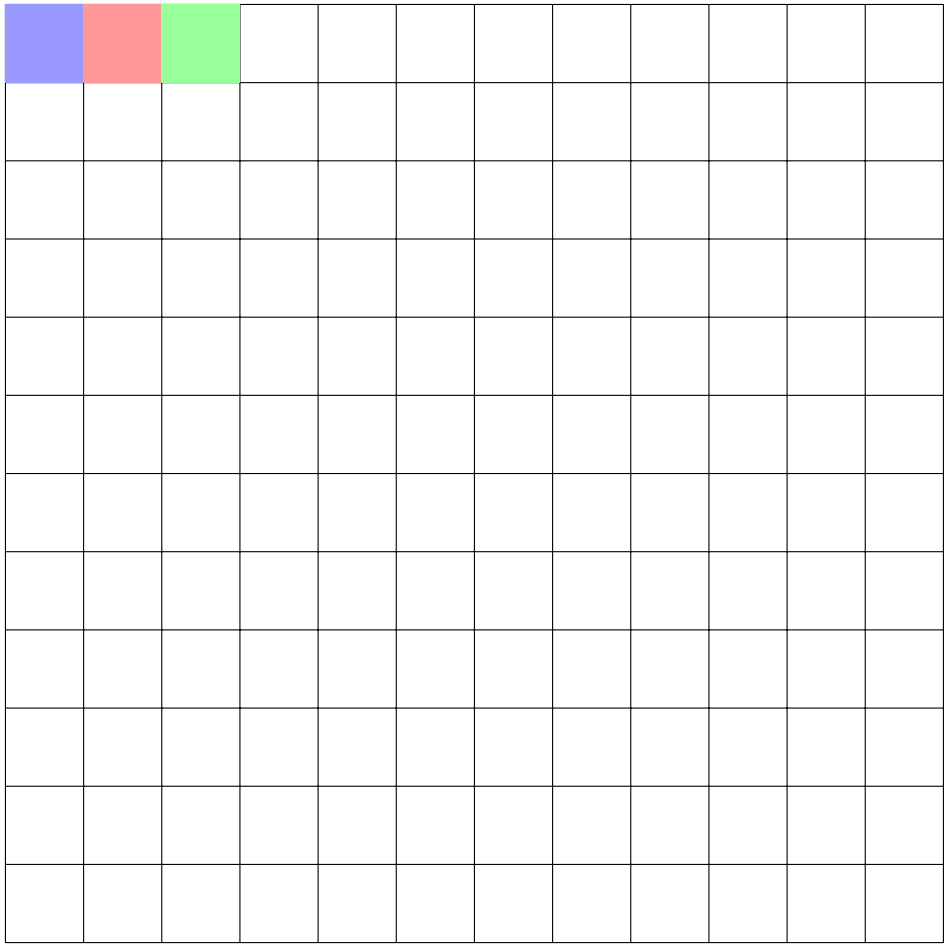
```
int a = 2;
```

Then you start storing things



```
int a = 2;
int b = 7;
```

And some more



```
int a = 2;
int b = 7;
int c = 9;
```

Maybe an entire array

```
int a = 2;
int b = 7;
int c = 9;
int d[9]; // Notice, how the array is contiguous in our memory!
```

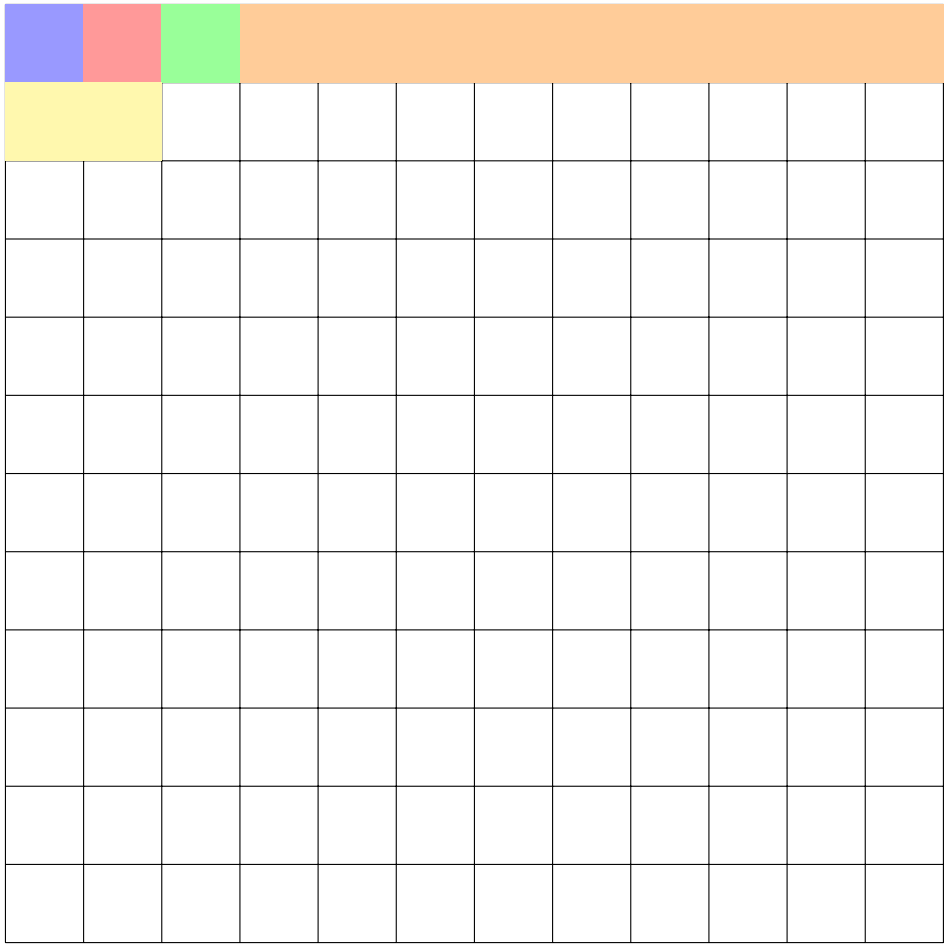
Space in C++

- So remember what these little boxes are.
- They are bytes of memory.
- Depending on the variable or data structure we use, they take up a different amount of memory.

Sample Data Sizes

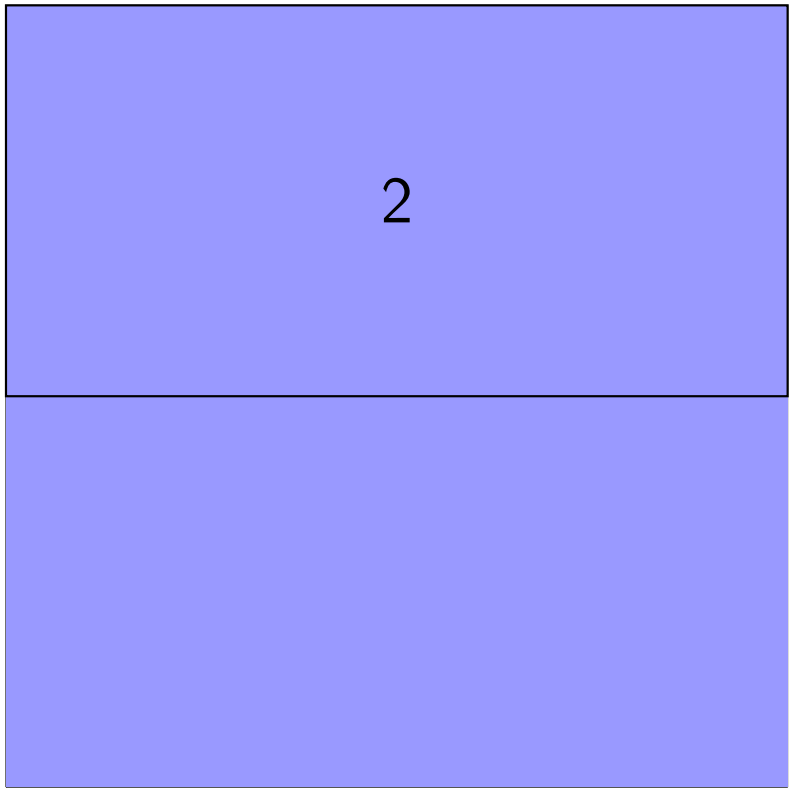
Type	Number of Bytes
bool	1
char	1
short	2
float	4
int	4
long	4
double	8
long long	8

Lets allocate some more memory, of different types



```
int a = 2;
int b = 7;
int c = 9;
int d[9];
double e = 1.01; // Ah, notice that a double takes twice as much space as our int because it is twice as many bytes
```

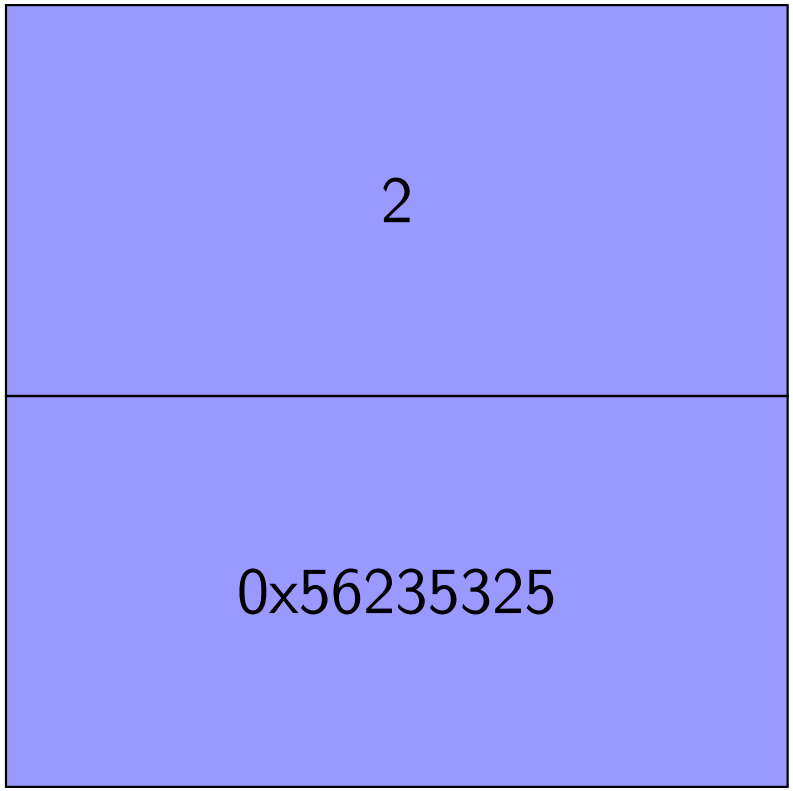
A closer look at just one of the boxes



int a = 2;

- So let's just look at the first integer.
- It contains our integer value
- But what about the second part of the box?

An address!



int a = 2;

- The address (in hexadecimal) tells us where we are!
- So just like addresses we have in real life, memory has a unique address.

Okay, but who cares?

- Well, having access to the memory address is actually quite powerful!
- In fact, this is what makes a language like C++ very powerful. We are close to the metal, and have the opportunity to work with the hardware.
- And at the very least, our C++ compilers and operating systems care about memory (and our end users!)

Address Of

If I want to see the memory address, I can use the ampersand operator. Here is an example.

```
#include <iostream>

int main(){

  int x = 567;

    std::cout << "Address of x is: " << &x << "\n";

  return 0;
}
```

Listing 1: Where in our memory is a variable

Reference

- Neat, so we can get the address of some variable in memory. We now know where it lives.
- But we do not get the actual *value*.
- Instead we are getting a **reference** to where some value resides.
- So when you think &, you think reference from now on.
- What does this allow us to do? (where is the power? See next slide)

Pass by Value

- In this class, we have always passed by value.
- That is, we have not used the & sign, which is pass by reference.
- When I say **pass by**, I am referring to functions, lets take a look at an example.

Pass by Value - Example

```
#include <iostream>

void setValue(int x){
  x = 9999;
}

int main(){

  int a = 6;
  setValue(a);
    // What is 'a' here?
    std::cout << a << "\n";

  return 0;
}
```

Listing 2: Pass by Value

Why is a not 9999?

```
#include <iostream>

void setValue(int x){
  x = 9999;
}

int main(){

  int a = 6;
  setValue(a);
    // What is 'a' here?
    std::cout << a << "\n";

  return 0;
}
```

Listing 3: Pass by Value

- Well, when we pass by value, we are actually passing a copy of 'a' on the stack.
- So pass by value, is 'copying' a value into the function to be used, but it does not modify the value.
- Where does that value live again? On the stack (see 2 bullet points up)

Why is 'a' not 9999 - The Stack

Table 1: This is a stack with two items

parameter 1 - int a	4 bytes of memory here
return value	In this case, void

- Stack Memory is allocated when we call a function.
- The amount of memory is the number of parameters we pass (in our previous example, 1 int), and the number of items we need to return.
- When we are done with the stack, C++ (and the OS) wants to reclaim that memory for other functions.

Pass by Reference - Example

```
#include <iostream>

// A one character change!
void setValue(int &x){
  x = 9999;
}

int main(){

  int a = 6;
  setValue(a);
    // What is 'a' here?
    std::cout << a << "\n";

  return 0;
}
```

Listing 4: Pass by Reference

Pass by Reference

- So why does it work?
- Remember what we are passing into the function this time
- We are saying, pass the address of an 'int' type.
- The address lives in our huge grid of memory that we saw earlier.
- This means that if we modify an address that is not on the stack, it does not go away with the stack after the function call.

A closer look

```
// Pass x 'by reference'
void setValue(int &x){
  // Whatever is in the memory at x, set
    // that integer value to 9999
  x = 9999;
}
```

Listing 5: Annotated function example

A quick review

- Pass by reference means we are only passing an address(0x56235325) to a function, not that actual value.
- Nothing more, nothing less. Just use the ampersand &.
- Remember, ampersand is 'address of', where something is stored in memory.

Stack Memory is different! Pass by value means copying

```
#include <iostream>

void memoryOnStack(int a){
    std::cout << &a << "\n";
}

int main(){
    int i = 5;
    // i's address should be the same
    // if it is in the same place in memory.
    // But no! You wil notice a copy is made
    // (and stored where?--the stack!)
    memoryOnStack(i);
    std::cout << &i << "\n";

    return 0;
}
```

Listing 6: Annotated function example

Pointers

We have another tool

- I have exciting news, we have another tool to work with!
- The asterisk operator *, which is a pointer.
- A pointer is a qualifier to a type that says "point to memory at this location".
- We might want to still pass by value, but not by reference if we do not want a mutation to take place.
- Again, a pointer is only a variable that holds a memory address–no values are stored!

We might want to still pass by value, but not by reference.

- A pointer when passed as a parameter, is still copied, but it is only an address, thus cheap. (imagine if you have a struct that has 100s of member variables–those all have to be copied!)
- Let me show you–then we'll dive into pointers.

Nasty vector copy

```
#include <vector>
#include <iostream>
// Doing one simple operation requires a copy of the entire
    vector.
void printVector(std::vector<int> vectorCopy){
  std::cout << vectorCopy[0] << "\n";
}

int main(){
  std::vector<int> vecCounter;

  for(int i =0; i < 200000000; i++){
      vecCounter.push_back(i);
    }
    // Pass a copy of our vector to the function
    printVector(vecCounter);
  return 0;
}
```

Listing 7: A vector with 200000000 items

Pointers

- A pointer is a type.
- There are integer pointers, floating pointers, double pointers, even pointers to your custom defined types (using structs)
- We often say, "a pointer to int", which has the same meaning as above.
- We use the asterisk (*) to signfify that we want to point to a specific data type.

Pointer syntax 1

```
// A regular integer
int x;
// A pointer to an integer
int* px;
// Sometimes we prefix our variable with a 'p' or 'p_' to
    remember it is a pointer.
```

Listing 8: Integer pointer

Pointer syntax 2

```
// A regular float
float x;
// A pointer to a float
float* p_x;
// Sometimes we prefix our variable with a 'p' or 'p_' to
    remember it is a pointer.
```

Listing 9: float pointer

Pointer syntax 3

```
// A struct we created
struct student{
  std::string name;
    int age;
};
// Create our struct object
Student mike;
// A pointer to our object
Student* p_student;
```

Listing 10: struct pointer

Pointer syntax 4

- So remember all a pointer does is point to an address.
- How do we get an address? Ampersand &!
- Okay, let us try it out.

Creating and assigning a pointer

```
#include <iostream>

int main(){
  int x = 5;
  // Integer point's types match if it points to
  // a primitive(int) that is an address
  // We get the address with &
  int* i = &x;



  return 0;
}
```

Listing 11: Creating a pointer

Dereferencing a pointer

- Great, so we can point to an integer.
- But what if we want the value for which we point at?
- We dereference, i.e. "Whatever I am referring to, don't give me the address (because remember & means reference), but instead give me the value–hence dereference."
- We dereference by putting an asterisk(*) before our variable. Note that this is a different context, then when we created a pointer type (a pointer to an integer in the previous example).
- Okay, let's give it a try.

Dereferencing a pointer - Example

```
#include <iostream>

int main(){
  int x = 5;
  // Integer point's types match if it points to
  // a primitive(int) that is an address
  // We get the address with &
  int* i = &x;
  // Dereferencing i (*i) gives us the value)
  std::cout << "value that i points to is: " << *i << "\n";

  return 0;
}
```

Listing 12: Dereference a pointer

Question

- If the pointer 'i' points to integer x, does that mean i can modify x's value?

Question Answered

```
#include <iostream>

int main(){
  int x = 5;
  int* i = &x;
  std::cout << "value that i points to is: " << *i << "\n";
  // Lets set i to something else
  // We are talking about values, so we de-reference
  // (i = 72 would try to set the memory address to 72!)
  *i = 72;
  std::cout << "what is i now: " << *i << "\n";
  std::cout << "what is x now: " << x << "\n";

  return 0;
}
```

Listing 13: Dereference a pointer

Pointers and functions - parameters are pass by value

```
#include <iostream>

// Pass by value (Only pass by reference with &)
void setToOne(int* p){
  // p points to something in memory, so we can
    // still modify its contents however!
  *p = 1;
}

int main(){
    int x = 500000;
    int* i = &x;
    setToOne(i);

    std::cout << "x is: " << x << "\n";

  return 0;
}
```

Listing 14: Parameters

Still pass by value

```
#include <iostream>
int global = 10; // Create some global we can point to.

void cannotChange(int* p){
  p = &global;
}

int main(){
    int x = 500000;
    int* i = &x;
    // Pass by value
    cannotChange(i);
    // So even if we point to something new
    // Remember, only a copy of a pointer is passed in.
    std::cout << "i should still be what x is: "<< *i <<"\n";

  return 0;
}
```

Listing 15: Still only passing a copy of the variable as a parameter

Whew, quite a bit

- First things first, pointers are fun and they give us power
- We can now modify variables in functions using either pointers or references.
- We want to start thinking of functions as having their own memory on the stack.
- We want to think about where memory is otherwise, and remember the concepts of scope.
- Speaking of memory–there's yet another curveball with arrays.

Passing arrays into functions

The first three functions are exactly the same!

```
int incrementOne(int* array){
  // ...
}

int incrementTwo(int array[]){
  // ...
}

int incrementThree(int array[5]){
  // ...
}

int incrementFour(int &array){
  // ...
}
```

Listing 16: Many variations

Passing arrays into functions - Part 2

- The reason is because an array is really just a pointer to memory. That is, a pointer to the first element in our array.
- incrementOne actually gives this away (the other two following examples decay to the same thing).
- However, we do not know how big the array is, so we will want to pass that as a parameter for our function.

wait, so you skipped incrementFour?

- This is passing an array(which is a pointer) by reference
- This is illegal. The C++ Standard forbids it, and if we really understand references, this sort of makes sense.
- An array syntax (arr[5]) actually is *(arr+5)

Pointer Arithmetic

```
#include <iostream>
int main(){
    int a[10];
    // Brackets are a convenient syntax for us
    // to avoid doing pointer arithemetic.
    // What this says is,
    // (1) Assign the value 5 to the left-hand side(lhs)
    // (2) On the lhs, increment 5 memory addresses from the start
    // (3) And de-reference this value and set it to 5.
    // (4) Now a[5] is equal to 5.
    *(a+5) = 5;

    std::cout << a[5] << "\n";
    return 0;
}
```

Listing 17: Incrementing through memory

NULL

- It is best practice when we create a pointer, to set it to NULL.
- At any point, we can also point our pointer to NULL, which essentially means it is pointing to nothing.
- We (you, me, and every C++ programmer) will run into a segfault error at some point.
- This means you dereferenced a pointer at an illegal memory location.
- Which really means, you do not have anything at that memory location, and you are trying to access something that does not exist! (e.g. Array out of bounds error).
- This is good behavior in a way, it gives us a place to start looking.

In-Class Activity

http://www.mshah.io/comp/11/activities/activity8/activity.pdf

Activity Discussion

Review of what we learned

- (At least) Two students
- Tell me each 1 thing you learned or found interesting in lecture.

5-10 minute break

To the lab!

Lab: `http://www.mshah.io/comp/11/labs/lab7/lab.pdf`
[1]

[1] You should have gotten an e-mail and hopefully setup an account at `https://www.eecs.tufts.edu/~accounts` prior to today. If not–no worries, we'll take care of it during lab!