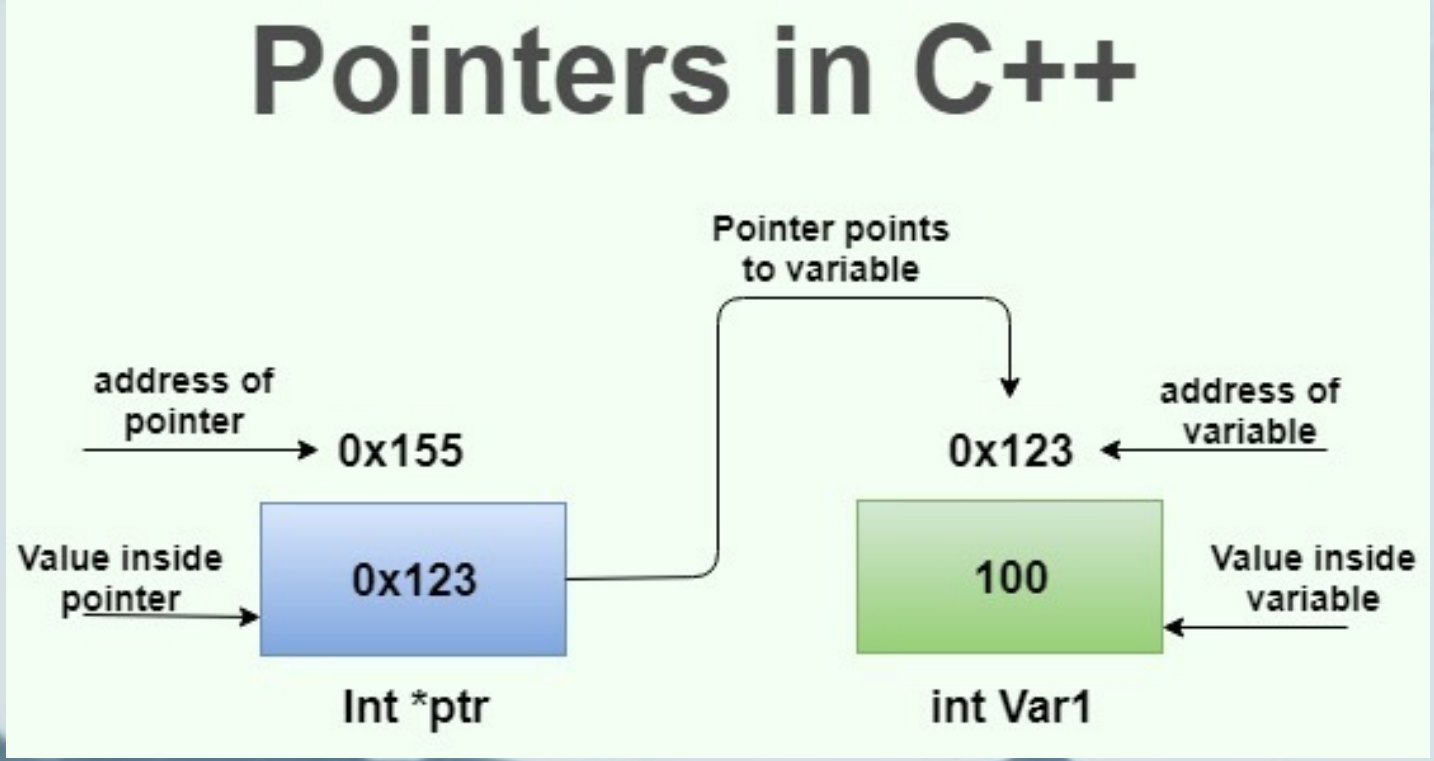
DISCUSSION SESSION
WEEK 6

C++ POINTERS

A pointer is a special type of variable representing a memory address.

A pointer should always be set equal to the memory address of another variable.



If there is nothing to set your pointer equal to, default it to NULL.

Pointers and References..

```
int a = 5;
int& b = a;    // b is a reference variable to a
```

If the ampersand is not included in the declaration of a variable, it is considered a get-address operator.

```
std::cout << &a;  // prints out the memory address of a
int* ptr; // declaration of a pointer expected to hold  the
          // memory address of an integer variable
int* ptr2 = &a; // pointer holding the address of a
```

Pointers and References..

```
int a = 5;
int* b = &a;  // b is holding the memory address of a
```

If the asterisk is not included in the declaration of a variable, it is considered a dereference operator.

Dereferencing a pointer means accessing the data stored at the memory address that the pointer is pointing to.

```
std::cout << *b;  // prints 5
std::cout << *a;  // Invalid because a is not a pointer and
                  // therefore cannot be dereferenced!
```

Both pointers and references use the ampersand operator (**&**), therefore you must be careful so you don’t mix them up!

- ✓ References are declared using the ampersand
- ✓ Pointers are initialized with some memory address using the ampersand

```
int a = 5;
int* ptr = &a;
```

Suppose we want to use the **ptr** variable to increment the value of **a**:

```
(*ptr)++;  // makes a = 6
```

Note: Use parentheses because the postfix operator **++** has a higher precedence than the dereference operator. Here, we want to dereference first, then increment the value stored at the address, so we need to use parentheses.

Exercise (5 minutes)

1. Declare an integer variable **x** and initialize it to 10
2. Declare a pointer **ptr** and point it to **x**
3. Increment the value of **x** by 2 using **ptr**
   (*Hint: must dereference pointer!)*
4. Print the modified value using **ptr**
5. Print the memory address stored by **ptr**
6. Print the memory address of **ptr**

```cpp
int main() {
    int x = 10;
    int* ptr = &x;
    (*ptr) += 2; // Warning: Without the dereference
                 operator here, you would be
                 incrementing the actual memory
                 address of ptr. Dereferencing gets the
                 value at the memory address ptr is
                 holding, in this case, 10.
    std::cout << *ptr << "\n";
    std::cout << ptr << "\n";
    std::cout << &ptr << "\n";
}
```

```
(base) yemifasina@83 Desktop % g++ main.cpp && ./a.out
12
0x7ff7b2a6c25c
0x7ff7b2a6c250
```

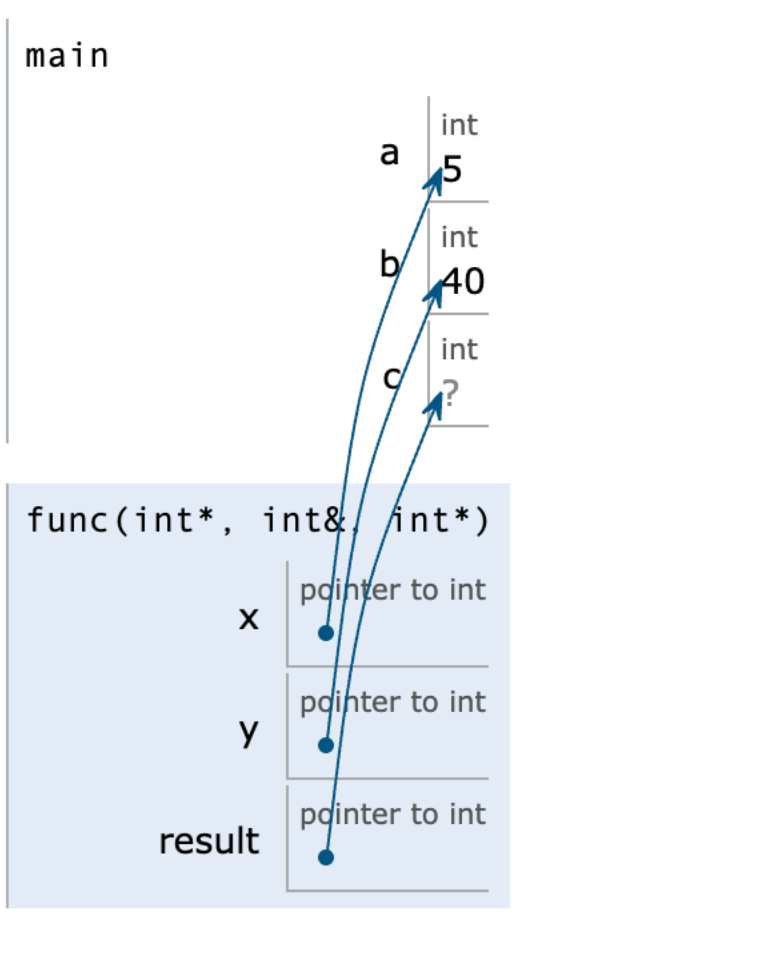
Passing Pointer into Function

✓ Function parameter is a pointer variable; argument is a memory address

```
double func(double a, double* b) {
        return a + *b;
}
int main( ) {
        double a = 3.5, b = 8.5;
        double sum = func(a, &b); // sets sum to 12
}
```

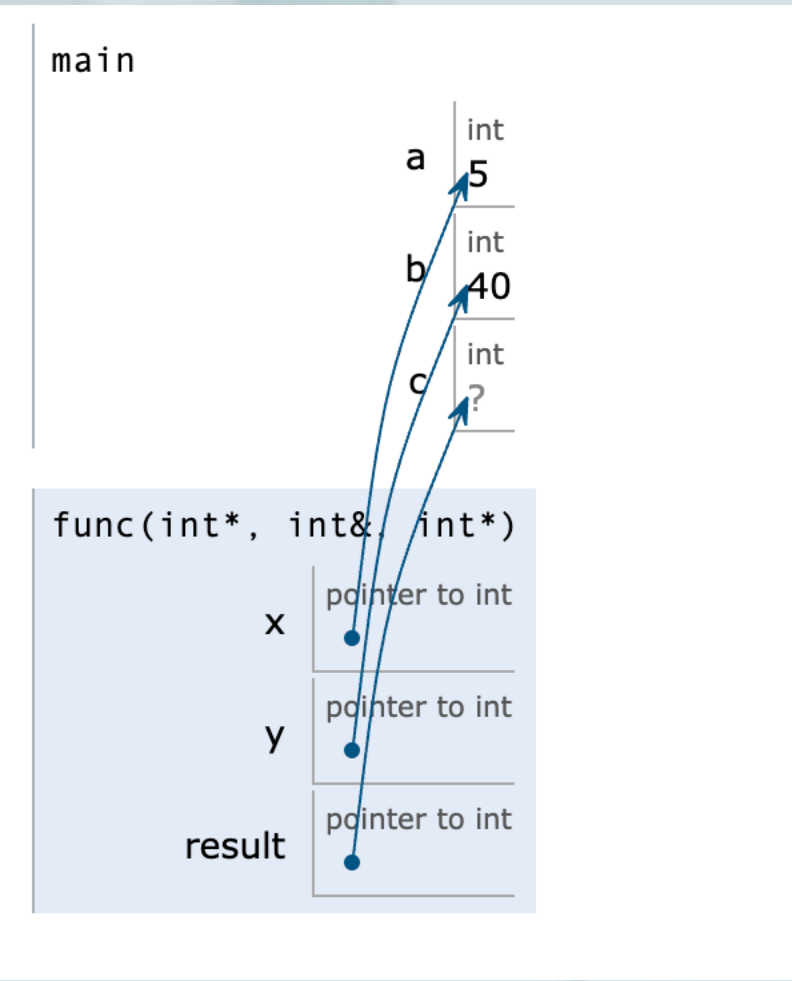
Exercise (1 minute): Use image on right to fill in the blank arguments in main function.

```
void func(int*x, int& y, int* result) {
        y /= 10;
        *result = *x * y;
}
int main( ) {
        int a = 5, b = 40, c;
        func(____, ____, ____);
        std::cout << c << std::endl;
}
```



Now what's the output?

```
void func(int*x, int& y, int* result) {
        y /= 10;
        *result = *x * y;
}
int main( ) {
        int a = 5, b = 40, c;
        func(&a, b, &c);
        std::cout << c << std::endl;
}
```



Arrays and Pointers

Arrays are pointers. Array names are constant pointers that point to the base address of the array. The base address of an array is the memory address of the first element of the array.

```
int arr[ ] = {3, 5, 2, 7};
std::cout << arr << std::endl;
std::cout << &arr[0] << std::endl;
```

These two print statements will output the same memory address.

Pointer Arithmetic

Reminder: Elements in arrays are laid out in memory sequentially/contiguously, one right after the other. So, given the base address of an array, we can iterate through the elements of the array.

```
char arr[ ] = {'f', 'y', 'p', 'm', 't'};
std::cout << arr; // prints out base address of array
std::cout << *arr; // dereferences address & prints out f
```

If we add 1 to the base address and dereference it, we can get the next element, and so on:

```
std::cout << *(arr + 1); // prints out y
```

Pointer Arithmetic

```
char arr[ ] = {'f', 'y', 'p', 'm', 't'};
std::cout << *(arr + 1); // move to the next address and
                         // dereference it. prints out y
```

Warning: Parentheses are super important here because the dereference operator has a higher precedence than the addition operator. So, without the parentheses, we would be doing:

```
std::cout << *arr + 1;
```

which dereferences the base address and adds 1 to it. This would cause our code to perform 'f' + 1 which would output the ascii value of **g**. Lack of parentheses here is a **logic error**.

sizeof operator

The sizeof( ) operator returns the size of a variable or data type, **in bytes.**

Refresher: An int has 4 bytes. So therefore,

```
int arr[ ] = {6, 3, 3, 7, 8}; // an array of 20 bytes
std::cout << sizeof(arr); // prints 20
```

How do we get the number of elements in the array? By dividing total array bytes by the bytes of one element!

```
std::cout << sizeof(arr) / sizeof(arr[0]); // prints 5
```

Warning: The sizeof operator works differently when used in a function where the array is a parameter (click to see why!) You must continue to pass in the size as a separate parameter.

Arrays in Functions

We previously learned to have our array parameter as follows:

```
void func(int arr[ ], int arraySize);
```

But provided that arrays are pointers, we can explicitly declare our array parameter as a pointer to the base address of the array, and still iterate through the array by indexing or by pointer arithmetic:

```
void func(int* arr, int size) {
    for(int i = 0; i < size; i++) {
      std::cout << arr[i] << std::endl;
    }
}
```

```
void func(int* arr, int size) {
    for(int i = 0; i < size; i++) {
        std::cout << *(arr + i) << std::endl;
    }
}
```

Another Loop Iteration Method Using Pointer Arithmetic

```
void func(int* arr, int size) {
    for(int* ptr  = arr; ptr < arr + size; ptr++) {
            std::cout << *ptr << std::endl;
    }
}
```

This loop starts at the base address (arr), deferences it and prints the value, moves to the next address and repeats the process till it reaches the address of the last element in the array.

Dynamic Memory

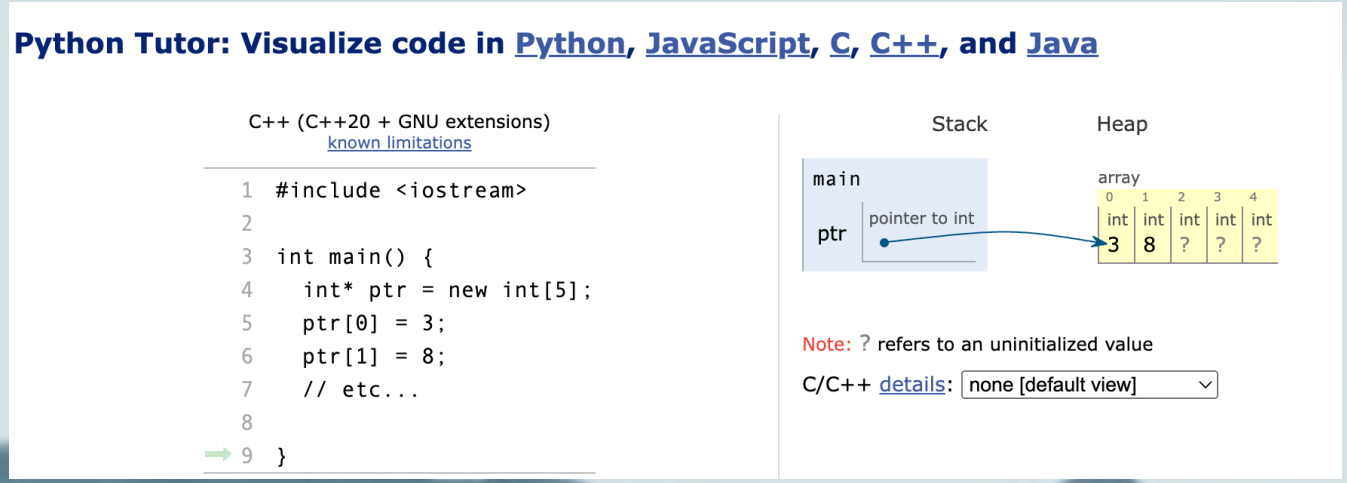
Until now, we have been declaring variables using stack memory. But we can dynamically allocate memory on the heap for data structures that require variable size or longer lifetimes.

- Unlike stack memory, heap memory can be flexible and can be used for data structures (like dynamic arrays) where the size is not known at compile-time. (A vector is a dynamic array—that is why it can grow and shrink as needed, without a size restriction).
- Objects allocated on the heap can persist beyond the scope of the function where they were allocated, until explicitly deallocated.
- Heap memory allocation is slower than stack memory. But this is almost never the right thing to worry about.

Dynamic Memory Allocation

We use the **new** keyword (typically with pointers) to allocate heap memory and the **delete** keyword to deallocate memory. It is your responsibility as a programmer to delete any dynamic memory allocation after use, so you don't have a memory leak and undefined behavior!

```
int* ptr;
ptr = new int[5];    // creates heap memory for array of 5 elements
```




```
int* num = new int; // creates heap memory for an integer
```

Dynamic Memory Allocation

```
int* ptr;
ptr = new int[5];
int* num = new int;
```

After doing stuff with *ptr* & *num* and they are no longer needed, you must deallocate heap memory:

```
delete[ ] ptr; // square brackets because ptr is an array!
delete num; // num has dynamic memory for a single integer so no [ ]
```

Some side advice..

- Pointers are a *very* important concept, especially when implementing data structures (CSC 212).
- You must fully grasp the concept to succeed in future CS courses and potentially in job/internship technical interviews.
- Expectation is that it will be on Exam-02 and will be heavily weighted.
- Put effort into the take-home lab to gain more practice with pointers.

“You get out what you put in.”