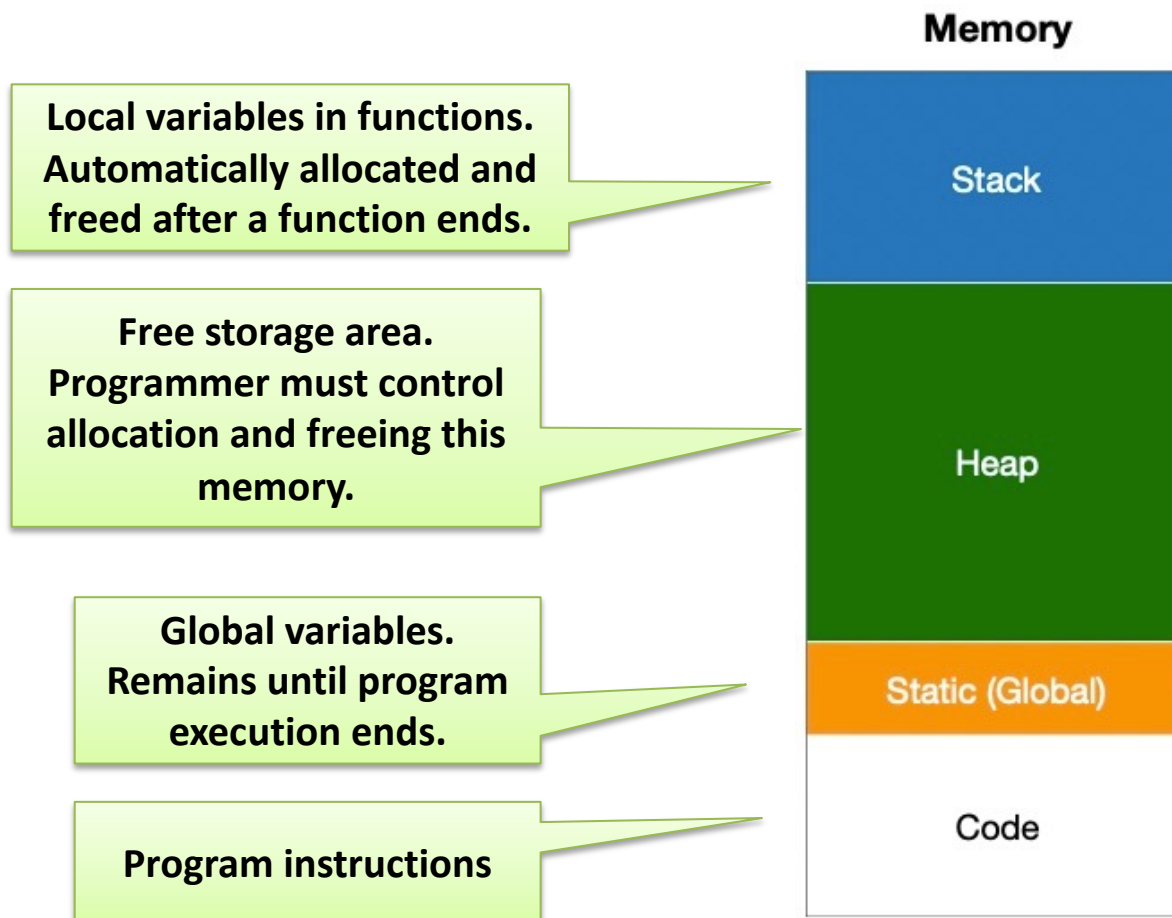


# Pointers, Dynamic Memory Allocation, & Built-in Arrays

# Memory 101

Any discussion of pointers must begin with a review of memory.



# Static vs. Dynamic Memory

Prior to this chapter, we have only been concerned with data/memory that has been allocated on the stack.

**Static Memory Allocation** occurs at compile time.

**Dynamic Memory Allocation** occurs at runtime, and uses the **Heap**.

# Memory 101

## Example:

```
#include <iostream>
using namespace std;

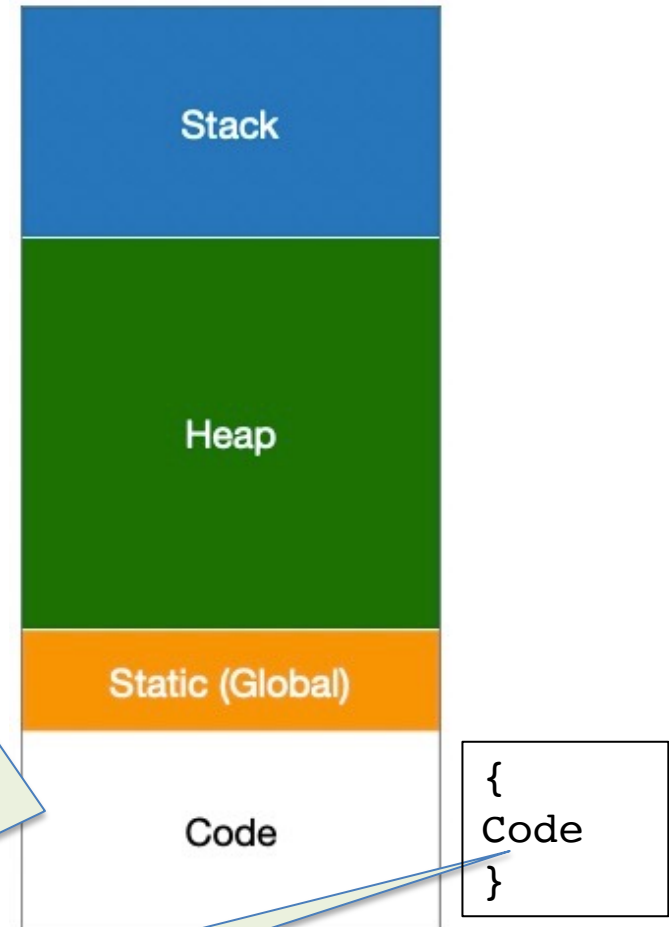
int myGlobal = 33;

void MyFunc() {
    int myLocal;
    myLocal = 999;
    cout << " " << myLocal;
}

int main() {
    MyFunc();
    int myInt;
    myInt = 555;
    int* myPtr = nullptr;
    myPtr = new int;
    *myPtr = 222;
    cout << *myPtr << " " << myInt;
    delete myPtr;
    return 0;
}
```

## Memory

## Contents



Deleted when  
*main* terminates

# Memory 101

## Example:

```
#include <iostream>
using namespace std;
```

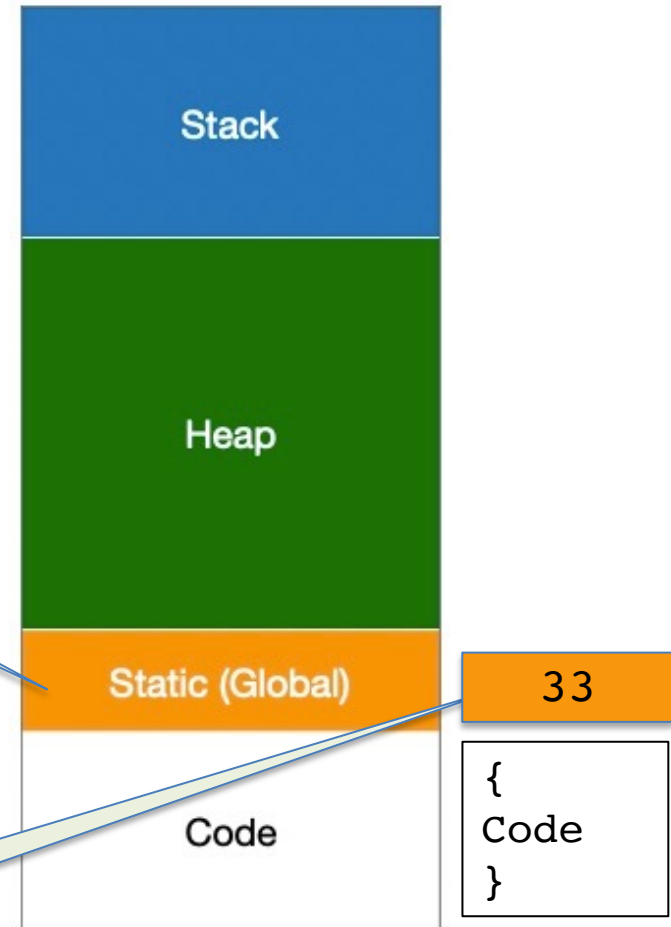
```
int myGlobal = 33;
```

```
void MyFunc() {
    int myLocal;
    myLocal = 999;
    cout << " " << myLocal;
}
```

```
int main() {
    MyFunc();
    int myInt;
    myInt = 555;
    int* myPtr = nullptr;
    myPtr = new int;
    *myPtr = 222;
    cout << *myPtr << " " << myInt;
    delete myPtr; return 0;
}
```

Memory

Contents



# Memory 101

## Example:

```
#include <iostream>
using namespace std;
```

```
int myGlobal = 33;
```

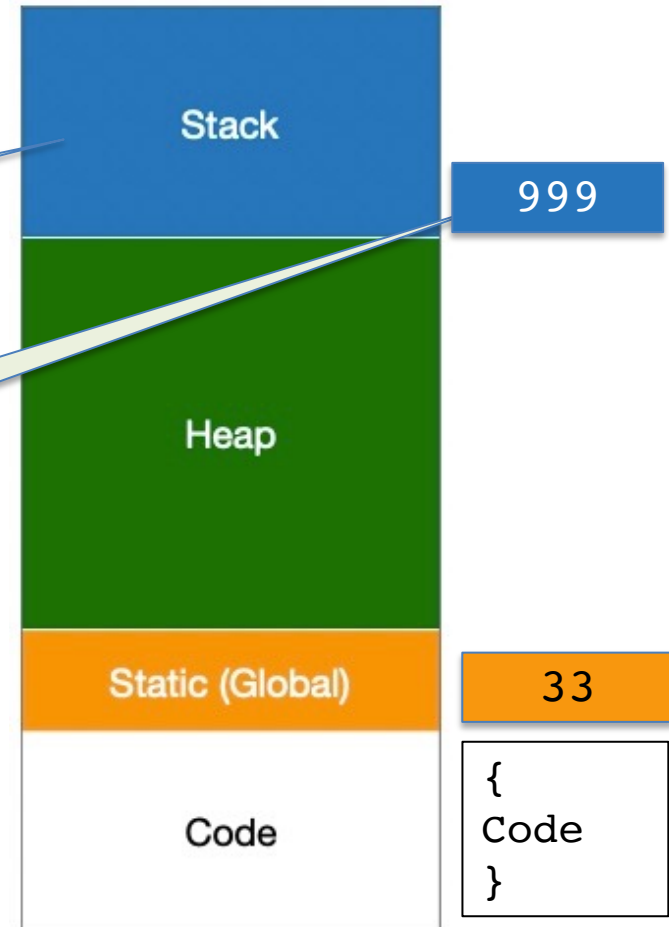
```
void MyFunc() {
    int myLocal;
    myLocal = 999;
    cout << " " << myLocal;
}
```

```
int main() {
    MyFunc();
    int myInt;
    myInt = 555;
    int* myPtr = nullptr;
    myPtr = new int;
    *myPtr = 222;
    cout << *myPtr << " " << myInt;
    delete myPtr;
    return 0;
}
```

Deleted when  
*MyFunc* terminates

## Memory

## Contents



# Memory 101

## Example:

```
#include <iostream>
using namespace std;

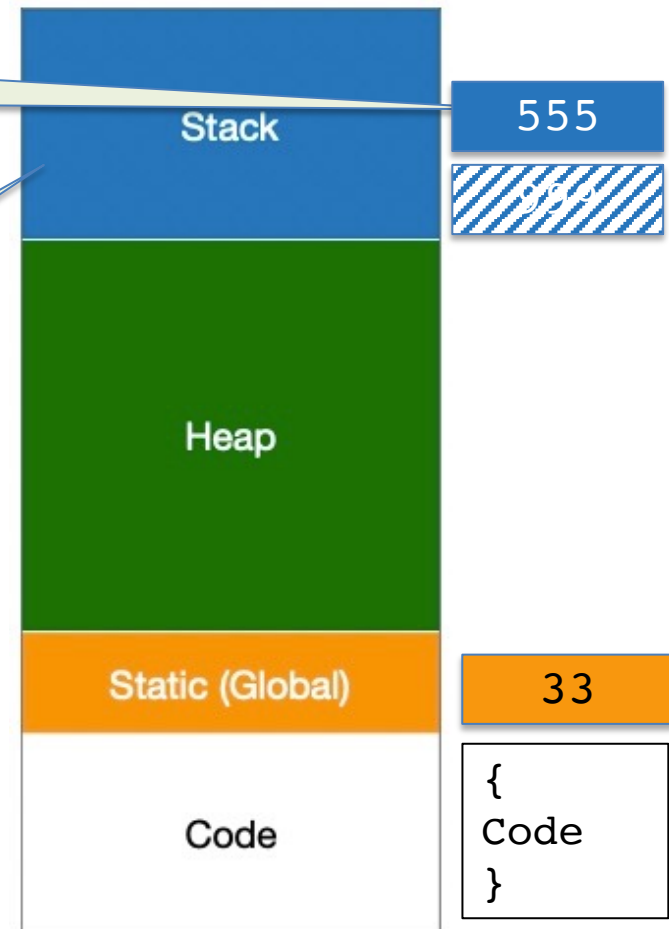
int myGlobal = 33;

void MyFunc() {
    int myLocal;
    myLocal = 999;
    cout << " " << myLocal;
}

int main() {
    MyFunc();
    int myInt;
    myInt = 555;
    int* myPtr = nullptr;
    myPtr = new int;
    *myPtr = 222;
    cout << *myPtr << " " << myInt;
    delete myPtr;
    return 0;
}
```

Deleted when *main*  
terminates

## Memory



# Memory 101

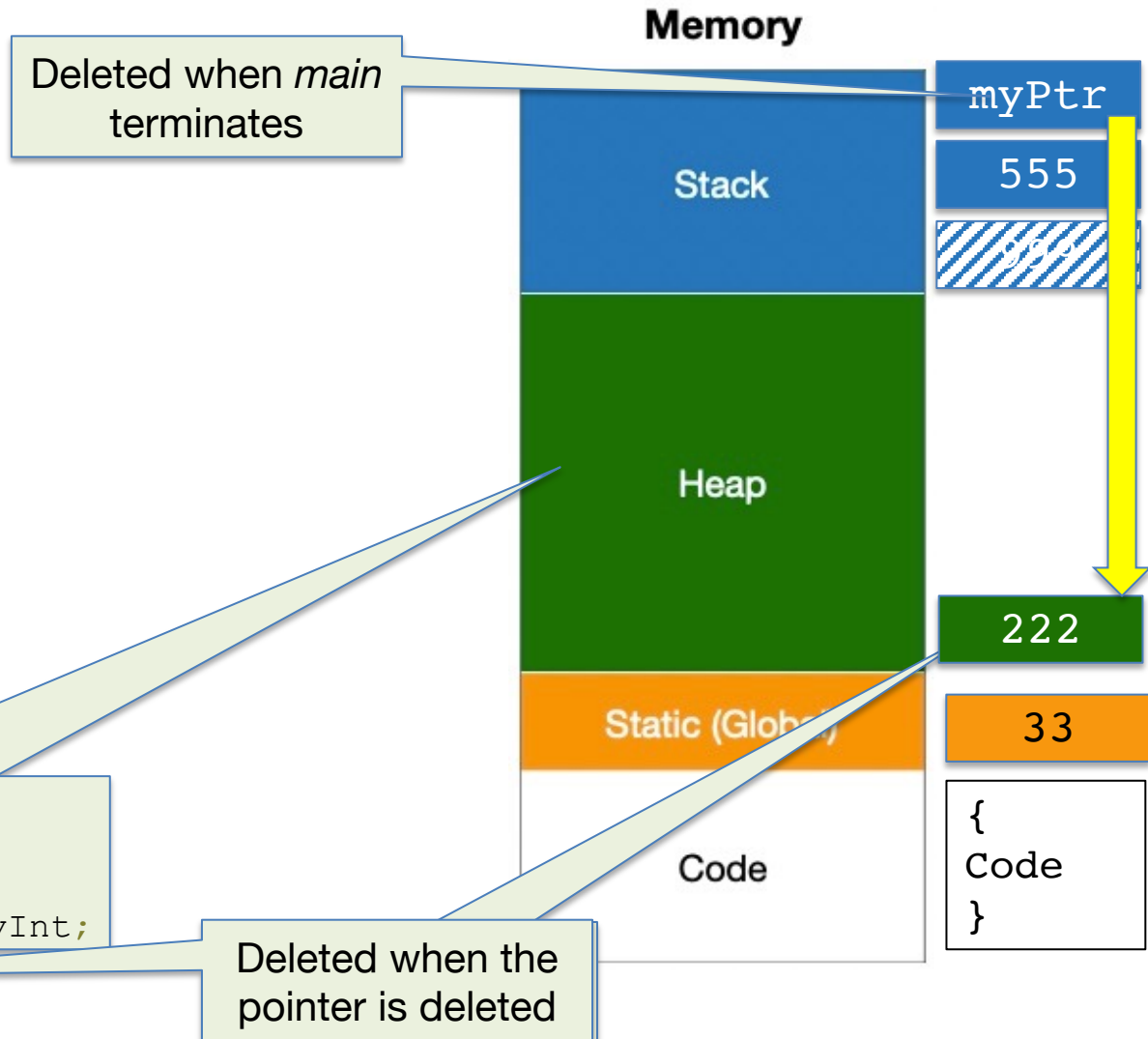
## Example:

```
#include <iostream>
using namespace std;

int myGlobal = 33;

void MyFunc() {
    int myLocal;
    myLocal = 999;
    cout << " " << myLocal;
}

int main() {
    MyFunc();
    int myInt;
    myInt = 555;
    int* myPtr = nullptr;
    myPtr = new int;
    *myPtr = 222;
    cout << *myPtr << " " << myInt;
    delete myPtr;
    return 0;
}
```





# Memory 101

## Why use the Heap?

- There is limited space on the Stack.
  - *Big objects shouldn't go on the Stack*
- You can't expand a memory slot on the stack if you need more space.
- You can't selectively remove something in the middle of the Stack to free up Stack memory.
- To grab a value in the middle of the stack, the computer needs to dig through everything on top first.

+ STL containers like *vector* manage dynamic memory allocation for you automatically.

- The Heap is slower and less efficient than the stack, but provides much more flexibility.



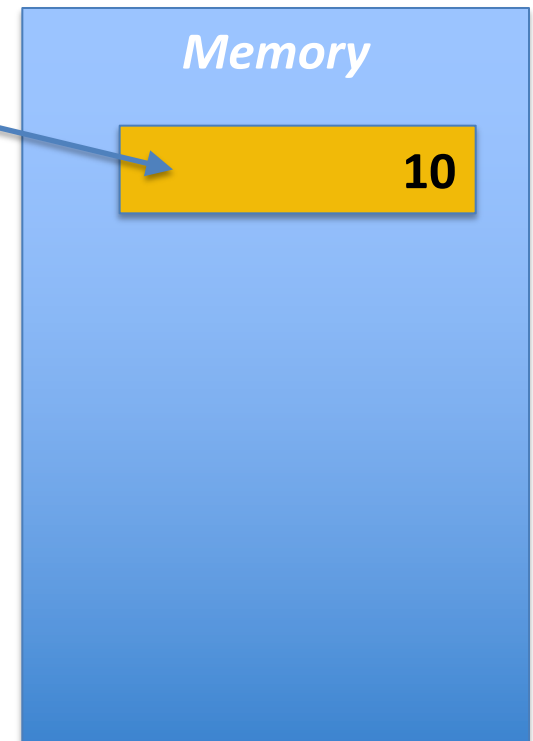
# "Normal" Variable Declarations

Normal variable instantiation automatically assigned a free memory address to a variable and stores that value in that memory address.

```
int x = 10;
```

*x* represents an alias for the memory address that holds the value 10.

When compiled, the variable name gets translated to its appropriate memory address.

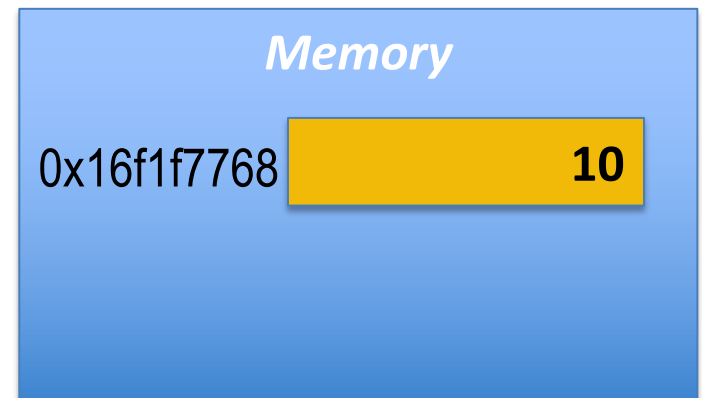


# Reference Operator (&)

The **reference** operator (&) is used to retrieve the memory address assigned to a variable.

```
int x = 10;  
cout << x; //prints the value 10  
cout << &x; //prints memory address of x
```

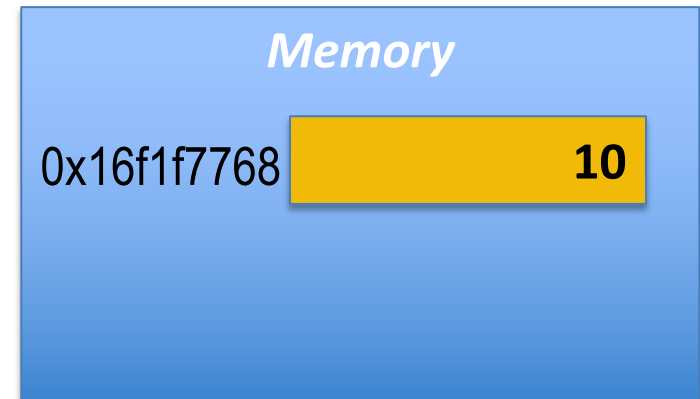
We use the reference operator when using function/method parameter passing by reference.



# Dereference Operator (\*)

The **dereference** operator (\*) allows you to access the value at a specific memory address.

```
int x = 10;  
cout << x; //prints the value 10  
cout << &x; //prints memory address of x  
cout << *&x; //prints the value 10
```



# So...

## **Simple int variable:**

```
int a = 10;
```

## **Reference to a simple int variable:**

```
int &b = a;
```

## **Pointer to a simple int variable's memory address:**

```
int* c = &a;
```

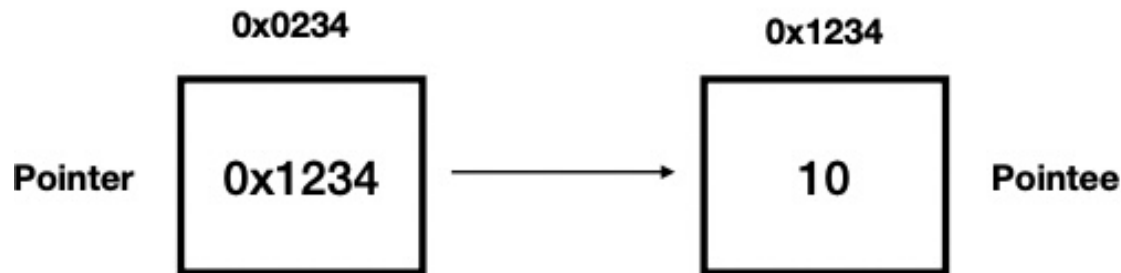
## **De-reference a reference variable:**

```
int d = *&x;
```



# Pointers

A pointer is a variable that contains the memory address of another location in memory that contains information.



Remember that pointers *only* hold a memory address. When we assign a value to a pointer, the value must be a memory address.

# Pointers

To declare a pointer, you need to specify the variable type of the pointee and provide the asterisk between the data type and the name.

It is a best practice to declare pointers on their own lines.

```
int *iPtr;  
double *dPtr;      OR      int* iPtr;  
                     double* dPtr;
```

**Declare multiple pointers in one statement:**

```
int *ptr1, *ptr2, *ptr3;
```

It's also a best practice to define the function return type as a pointer with the asterisk next to the data type, not the function name.

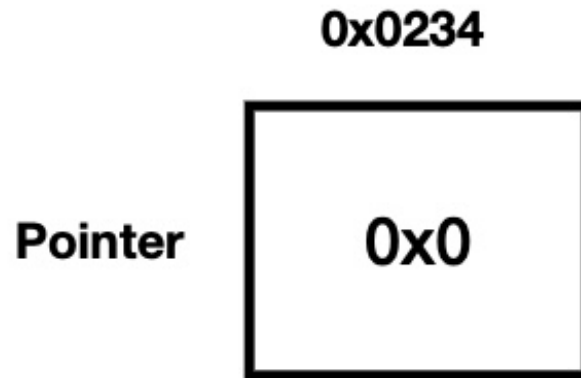
```
double* GetSalary() {  
    double salary = 26.48;  
    double *HourlySalary = &salary;  
    return HourlySalary;  
}  
  
Double *GetSalary() {  
    double salary = 26.48;  
    return &salary;  
}
```



# nullptr

**nullptr** – The null pointer keyword is used to indicate that a pointer points to nothing.

A pointer assigned to *nullptr* is said to be null.





# The *new* keyword

**new** – The *new* keyword is used to indicate that the memory used for holding the data is being allocated on the **heap**.

The *new* operator allocates memory during runtime and is persisted independently of any particular function.

Construct 11.3.1: The new operator.

```
pointerVariable = new type;
```

[Feedback?](#)



# Memory Diagrams

Memory diagrams help understand what's going on in the Stack and the Heap when running a program:

The Stack – used for statically allocated memory

The Heap – used for dynamically allocated memory (e.g. with *new*)

```
int foo(int a)
{
    int b = 10;
    a = b + 3;
    return b;
}
```

```
int main() {
    int x = 5;
    int y = 10;
    y = foo(x);
    return 0;
}
```

## Stack

### main

x

5

y

10

### foo

a

5

13

b

10

## Heap

# Memory Diagrams

Let's build a memory diagram for the following:

**Stack**

**Heap**

```
int main() {  
  
    int *ptr1 = nullptr;  
    int value = 10;  
  
    int *ptr2 = &value;  
  
    int *ptr3 = new int;  
  
    int ptr4 = new int(10);  
  
}
```

# Implementing Swap with Pointers

Here's a reminder of a swap function using regular variables:

```
void swap(int &x, int &y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}
```

Let's rebuild this function with pointer support.



# Arrays

Why did this chapter include a discussion of arrays???

Because...built-in arrays in C++ are built using pointers!

```
int myArray[] = {1, 2, 3, 4};
```

The above actually creates a pointer called *myArray* that we can use to access elements inside the array.

The compiler knows that the `[]` syntax indicates an array of (in this case) `int` values.

# Arrays

One of the only differences in arrays declared with `[]` or using pointers is that we can reassign pointers and not arrays.

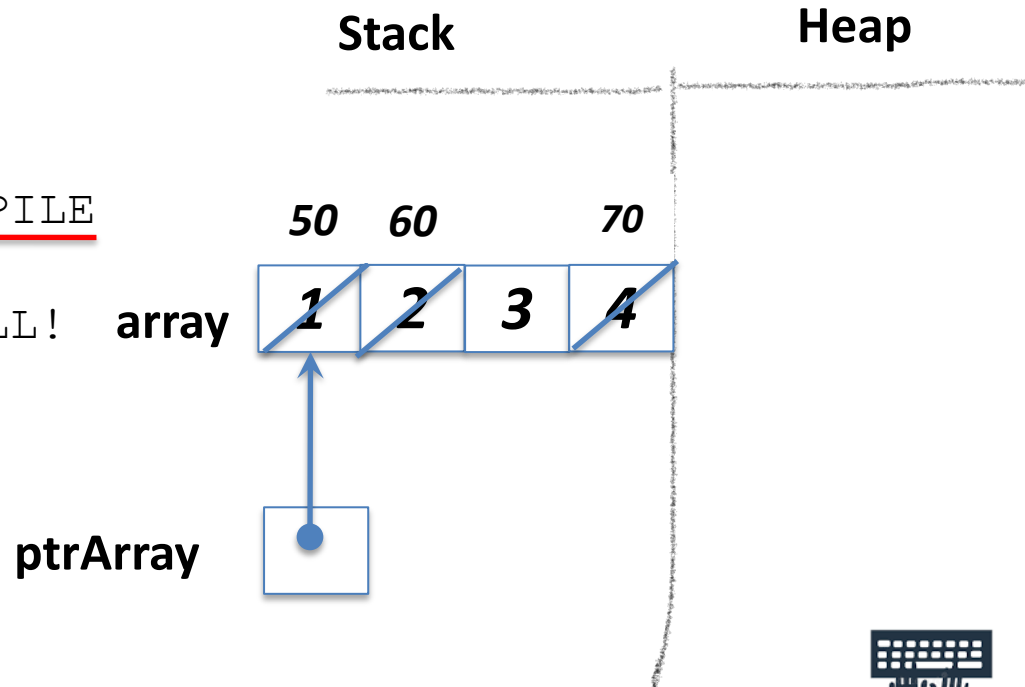
An arrays is always declared **const** when using `[]` notation.

```
int array1[] = {1, 2, 3, 4};
```

```
int array2[4];  
array2 = array; //WON'T COMPILE
```

```
int *ptrArray = array; //WILL!
```

```
*ptrArray = 50;  
*(ptrArray + 1) = 60;  
*(ptrArray + 3) = 70;
```



# Dynamically Allocated Arrays

We can also dynamically allocate arrays:

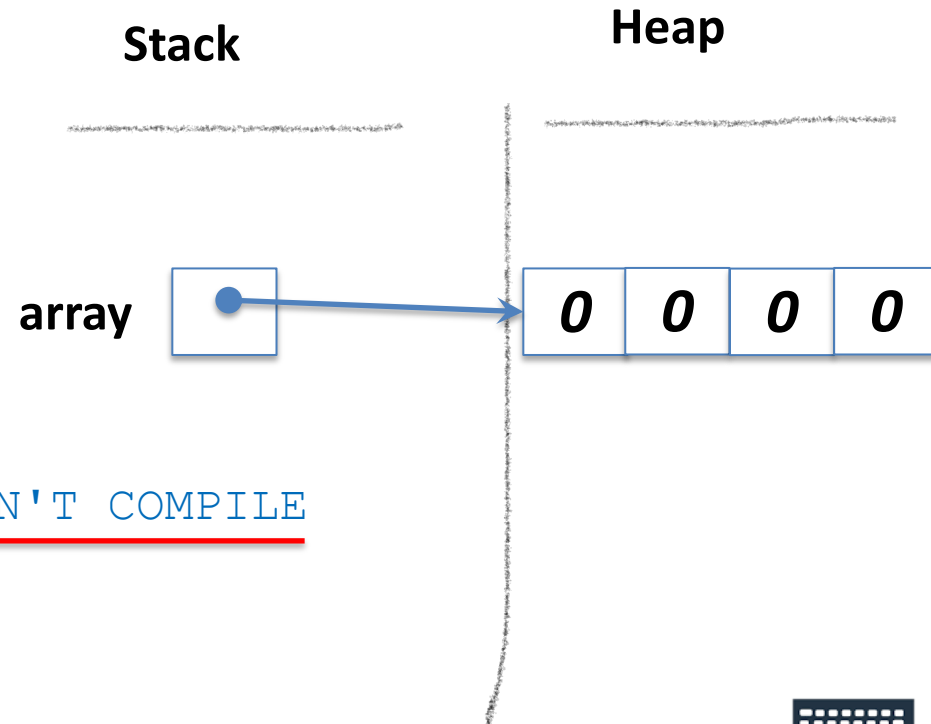
```
int *array = new int[4];
```

We can access arrays on the heap, the same way as before:

```
array[index];
```

However, initialization is different:

```
int *array = {1,2,3,4}; //WON'T COMPILE
```



# The *delete* Operator

The **delete** operator does the opposite of the new operator.

This tells the OS to de-allocate or free up the memory block that is pointed to.

```
delete ptr;
```

Generally, for every *new* operator, there must be a *delete* operation.

```
int *ptr = new int(12);  
delete ptr;  
ptr = new int(2);
```

*Without this, the int  
12 is still occupying  
heap memory*

**Memory Leak** - a memory leak occurs when a programmer forgets to de-allocate memory that is pointed to by a pointer, e.g.,:

- Delete a pointer or re-assign the pointer to a new value.
- Allocate memory in a loop then call delete outside the loop.
- Not deleting data inside a dynamically allocated object.



# *delete[]* operator

When dynamically allocating **array's**, we use the `delete[]` operator to indicate that we are freeing a block of memory.

```
int *array = new int[5];
```

```
delete[] array;
```



# Pointers with Objects/Classes

The conventional approach to creating objects is with pointers and the *new* keyword.

```
Student *a = new Student;
```

When the program executes this line, we first allocate the space for *Student* on the **heap**, then we call the *Student* default constructor.

We can also use other overloaded constructors for a class during initialization:

```
Student *a = new Student;  
Student *a = new Student("Erik");
```

# Member Access Operator

Accessing data members of a class can be done using pointers:

```
Student *a = new Student;  
(*a).data_member;  
(*a).function();
```

However, with C++ and classes, we can use the **member access operator** `->` as an alternative (and easier) way to access data members or functions of a class:

```
a->data_member;  
a->function();
```

# Destructors

A **destructor** is a member function that is called which destructs or deletes an object. Same format as a constructor with **~** as prefix character:

```
Employee::~Employee () {  
  
    /* Clean up data */  
  
}
```

Destructors are called when:

1. A function ends.
2. The program ends.
3. A block containing local variables end.
4. A delete operator is called.

A class only has one destructor.

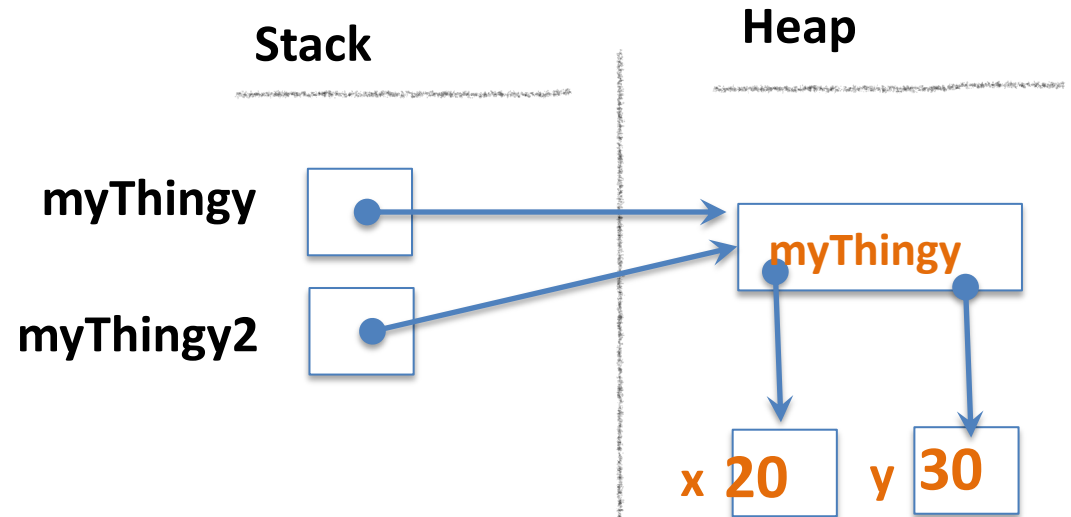
***Anything in the  
heap must be  
deleted***



# Copying Objects

When we copy an object using the assignment operator, our new pointer is pointing to the same object the original pointer is pointing to:

```
Thingy *myThingy = new Thingy(20, 30); //initialize *x & *y  
Thingy *myThingy2 = myThingy;
```



***This is a Shallow copy***

# Copy Constructor

A **copy constructor** is to make a **deep copy** of an existing object/instance.

We use constants when passing the object to be copied to ensure no changes are made to the original object.

\*Remember: *Pass by value* creates a copy of the data and performs any operations on that copy in the function.

Without a copy constructor, the default result is a shallow copy (previous slide.)

```
MyClass::MyClass(const MyClass &copyClass) {  
    //copy member values here  
}
```

The most common use for the copy constructor is when the class has raw pointers as member variables and we need to make a deep copy of the data.

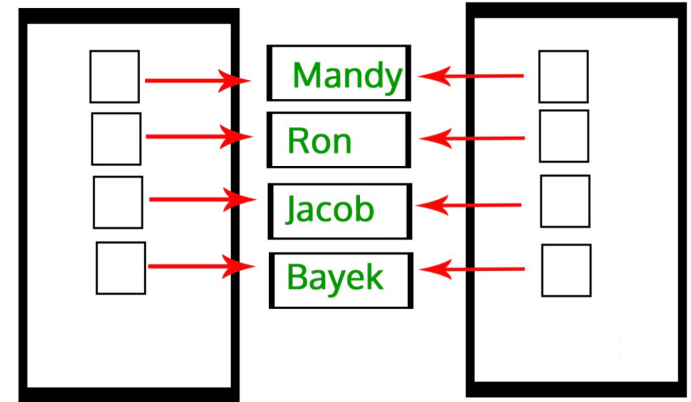


# Deep vs. Shallow Copy Summary

**Shallow Copy** - a member-wise copy where we only copy the data members and NOT any dynamically allocated memory.

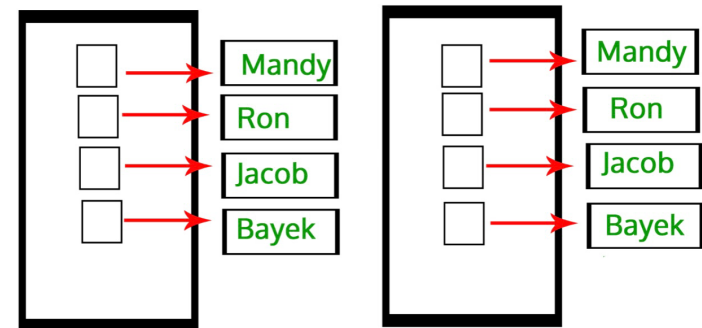
Your computer performs a shallow copy by default if no copy constructor is provided.

Shallow Copy



**Deep Copy** - a complete full copy of all data members of a class, including dynamically allocated memory.

Deep Copy



# Copy Assignment Operator

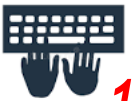
We conventionally copy one object to another using the assignment operator.

The program will implicitly create a copy assignment operator for you and do a member-wise copy.

If we want a **deep copy** with dynamically allocated objects, the default assignment operator won't support this .

However, we can overload the assignment operator to support a deep copy.

```
MyClass& Myclass::operator=(const MyClass& copy) { }
```





# Rule of Three

If you define one of the following, then it is a good practice to define all of them:

**Destructor**

**Copy Constructor**

**Copy Assignment Operator**

They are referred to as the "**Big Three**"



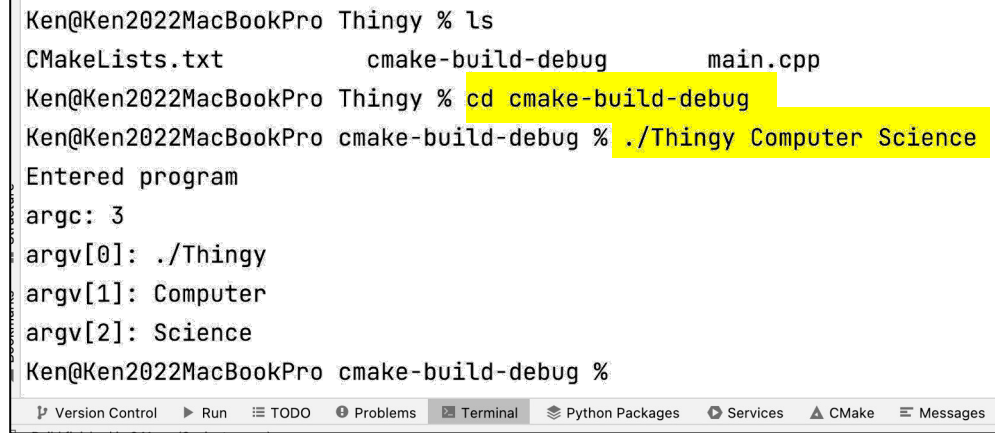
# Command Line Arguments

Command line arguments are values entered by the user when running the executable program from the console's command line.

To support this, we provide these additional parameters to the *main* function:

In C++ the only way to read in command line arguments is by using c-strings!

```
int main(int argc, char* argv[]) {  
    int i;  
    // Prints argc and argv values  
    cout << "Entered Program" << endl;  
    cout << "argc: " << argc << endl;  
  
    for (i = 0; i < argc; ++i) {  
        cout << "argv[" << i << "]: " ;  
        cout << argv[i] << endl;  
  
        return 0; }  
}
```



The screenshot shows a terminal window with the following commands and output:

```
Ken@Ken2022MacBookPro Thingy % ls  
CMakeLists.txt      cmake-build-debug    main.cpp  
Ken@Ken2022MacBookPro Thingy % cd cmake-build-debug  
Ken@Ken2022MacBookPro cmake-build-debug % ./Thingy Computer Science  
Entered program  
argc: 3  
argv[0]: ./Thingy  
argv[1]: Computer  
argv[2]: Science  
Ken@Ken2022MacBookPro cmake-build-debug %
```

The terminal window has a status bar at the bottom with icons for Version Control, Run, TODO, Problems, Terminal, Python Packages, Services, CMake, and Messages.