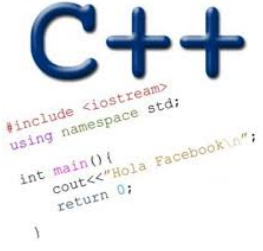
AUTOMATING COMPILATION
C++ MEMORY MODEL

Problem Solving with Computers-I



Pointer Diagrams (Review)

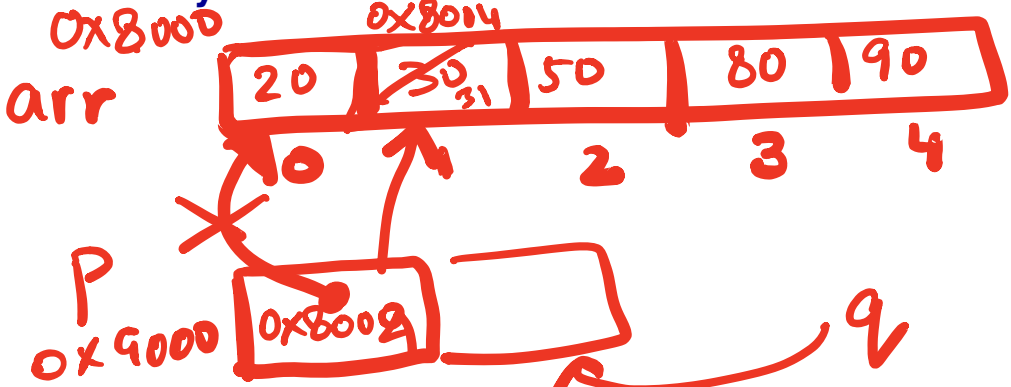
```
int arr[]={20, 30, 50, 80, 90};
int *p = arr;
p = p + 1;
*p = *p + 1;
```

p (int *)
*p (int)
&p (int**)

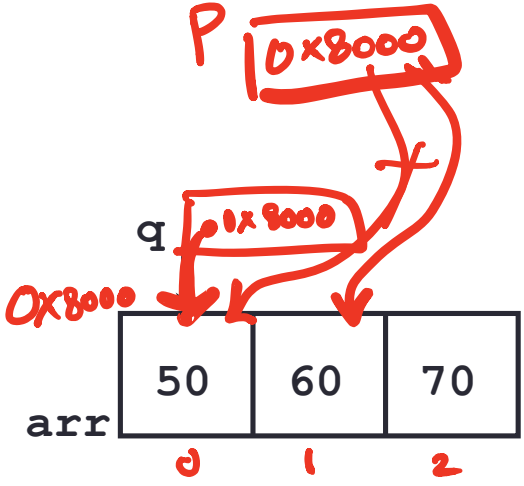
0x8000 + 1 * sizeof(int)

int **q = &p + 1; 0x9000 + 1 * sizeof(int*)

Draw the array ar after the above code is executed



```
void IncrementPtr(int *p){
    p++;
}
int main(){
    int arr[3] = {50, 60, 70};
    int *q = arr;
    IncrementPtr(q);
    return 0;
}
```



Which of the following is true after **`IncrementPtr(q)`** is called in the above code:

A. '**q**' points to the next element in the array with value 60

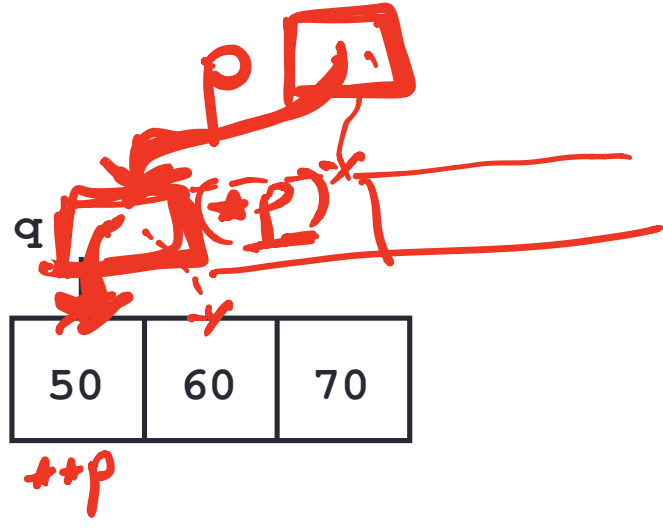
B. '**q**' points to the first element in the array with value 50

How should we implement `IncrementPtr()`, so that 'q' points to 60 when the following code executes?

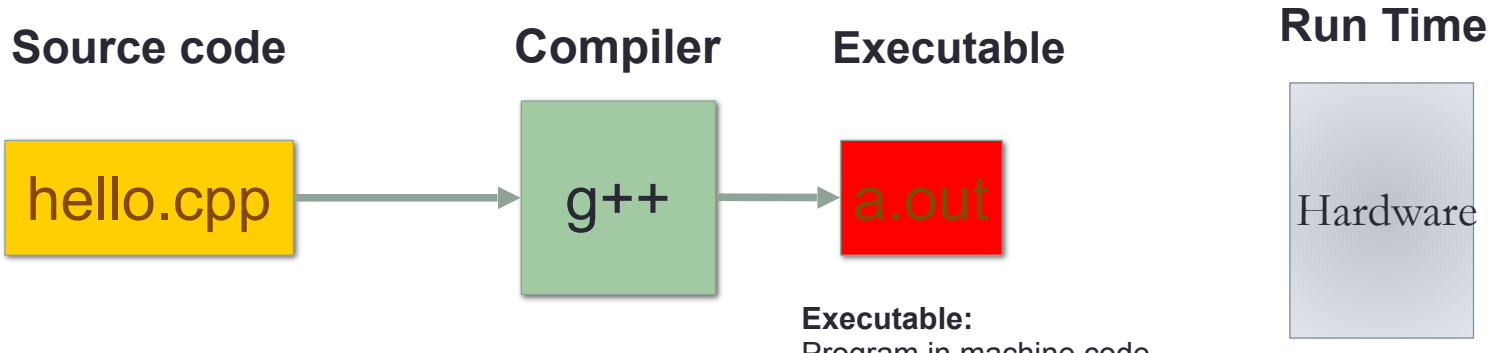
```
void IncrementPtr(int **p){
    p++;
}

int arr[3] = {50, 60, 70};
int *q = arr;
IncrementPtr(&q);
```

A. `p = p + 1;`
B. `&p = &p + 1;`
C. `*p= *p + 1;`
D. `p= &p+1;`



The compilation process



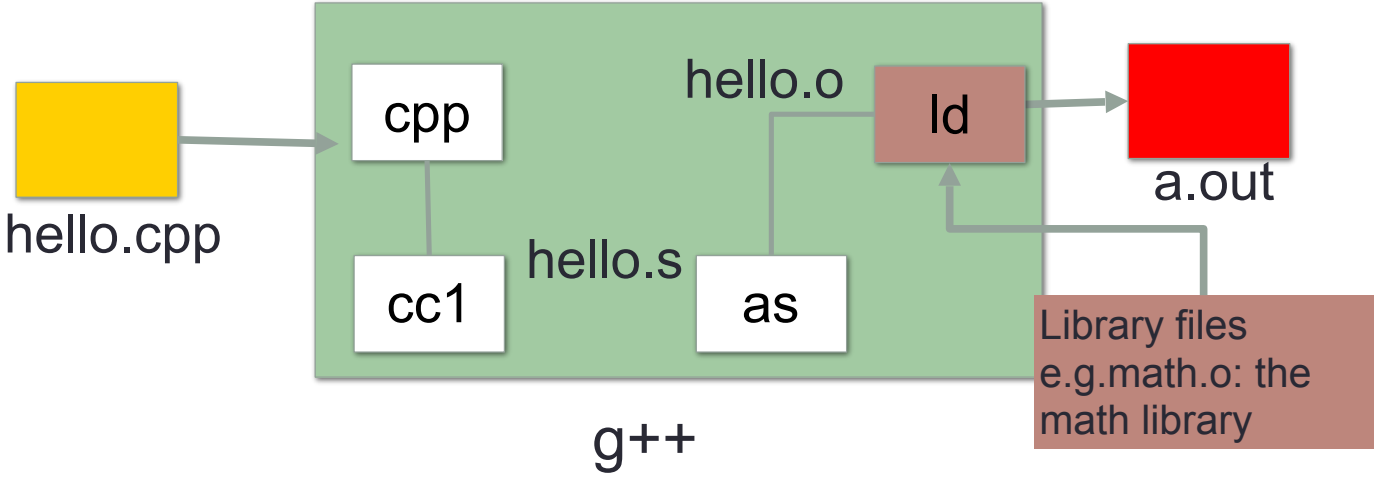
Executable:
Program in machine code
+Data in binary

Source code:
Text file stored on computers hard disk or some secondary storage

```
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
```

g++ is composed of a number of smaller programs

- Code written by others (libraries) can be included
- ld (linkage editor) merges one or more object files with the relevant libraries to produce a single executable

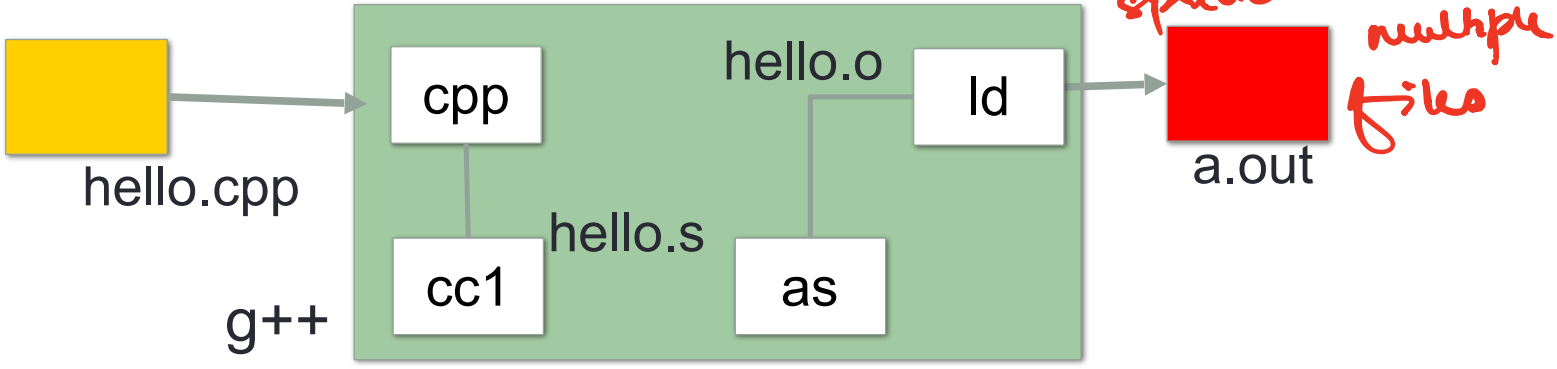


Steps in gcc

- Ask compiler to show temporary files:

```
$ g++ –S hello.cpp
$ g++ –c hello.o
$ g++ –o hello hello.cpp
$ g++ functions.o main.o –o myhello
```

We'll discuss these later when talking about compiling programs that are spread across multiple files



Make and makefiles

- The unix make program automates the compilation process as specified in a Makefile
- Specifies how the different pieces of a program in different files fit together to make a complete program
- In the makefile you provide a recipe for compilation
- When you run make it will use that recipe to compile the program

```
$ make
g++ testShapes.o shapes.o tdd.o –o testShapes
```

Specifying a recipe in the makefile

- **Comments** start with a #
- **Definitions** typically are a variable in all caps followed by an equals sign and a string, such as:

```
CXX=g++
CXXFLAGS=-Wall

BINARIES=proj1
```

```
# testShapes is the target - it is what we want to produce
# To produce the executable testShapes we need all the .o files
# Everything to the right of ":" is a dependency for testShapes

testShapes: testShapes.o shapes.o tdd.o
     #The recipe for producing the target (testshapes) is below
     g++ testShapes.o shapes.o tdd.o -o testShapes
```

Demo

- Basics of code compilation in C++ (review)
- Makefiles (used to automate compilation of medium to large projects) consisting of many files
- We will start by using a makefile to compile just a single program
- Extend to the case where your program is split between multiple files
- Understand what each of the following are and how they are used in program compilation
  - Header file (.h)
  - Source file (.cpp)
  - Object file (.o)
  - Executable
  - Makefile
  - Compile-time errors
  - Link-time errors

Dynamic Memory
& Heap vs Stack

The case of the disappearing data!

```
int getInt(){
      int x=5;
      return x;
}
int* getAddressOfInt(){
      int x=10;
      return &x;
}
int main(){
      int y=0, *p=nullptr, z=0;
      y = getInt();
      p = getAddressOfInt();
      z = *p;
     cout<<y<<", "<<z<<", "<<*p<<endl;
}
```

int* means function returns the address of an int

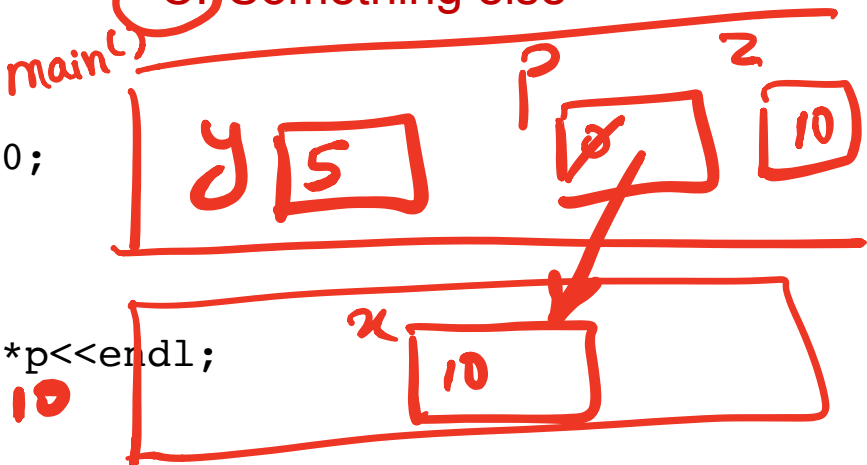
// x is a local variable

What is the output?

A. 5, 0, 10

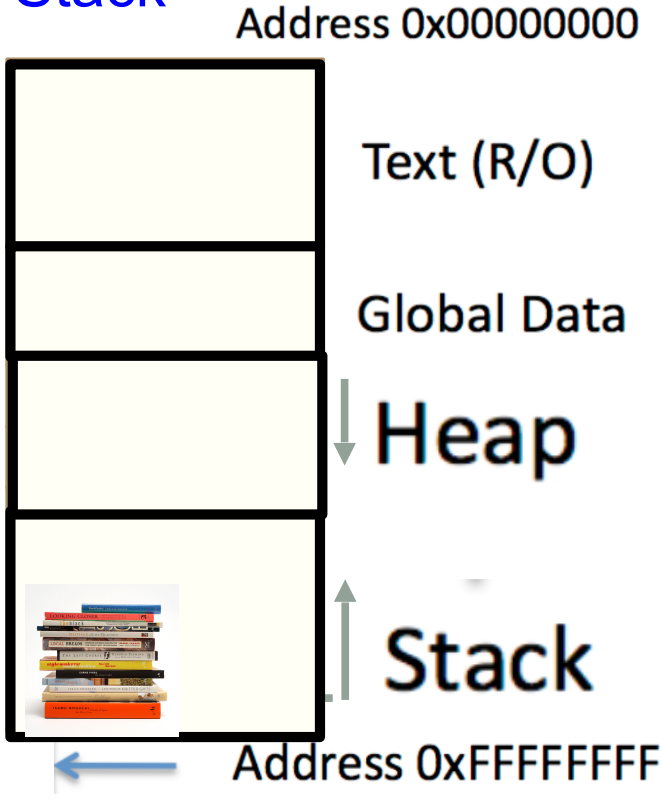
B. 5, 10, 10

C. Something else



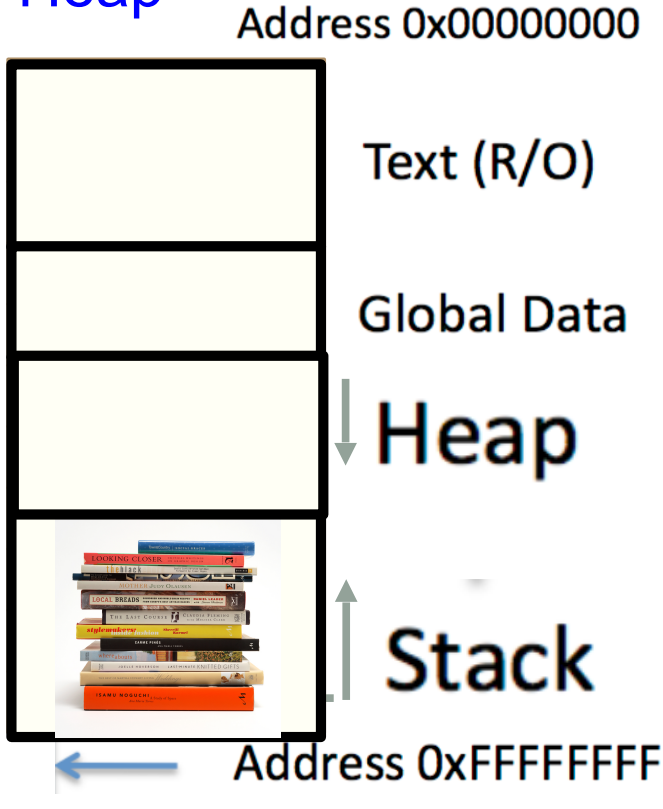
C++ Memory Model: Stack

- Stack: Segment of memory managed automatically using a Last in First Out (LIFO) principle
- Think of it like a stack of books!



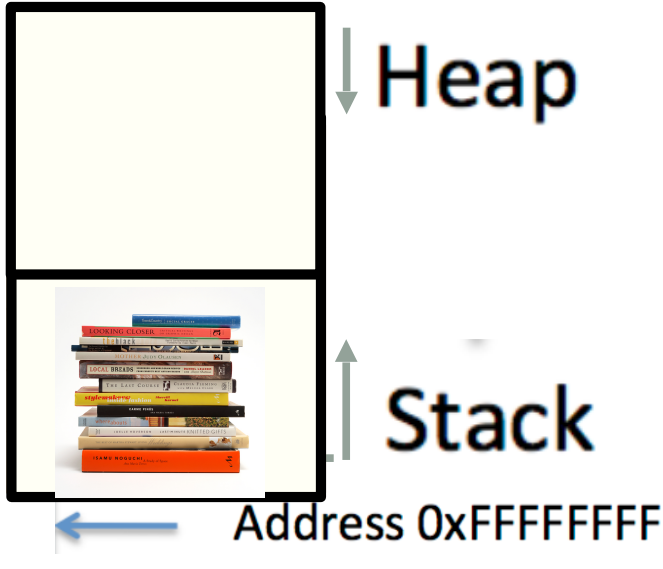
C++ Memory Model: Heap

- Heap: Segment of memory managed by the programmer
- Data created on the heap stays there
  - FOREVER or
  - until the programmer explicitly deletes it



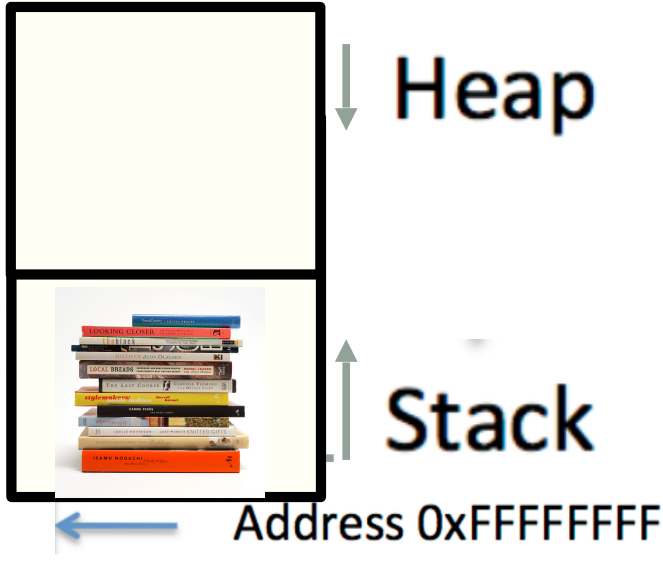
Creating data on the Heap: new

To **allocate** memory on the heap use the **new** operator



Deleting data on the Heap: delete

To **free** memory on the heap use the **delete** operator



Dynamic memory management = Managing data on the heap

```
int* p= new int; //creates a new integer on the
heap

SuperHero* n = new SuperHero;

                  //creates a new Student on the
heap

delete p; //Frees the integer

delete n; //Frees the Student
```

Solve the case of the disappearing data!

```
int getInt(){
     int x=5;
     return x;
}
int* getAddressOfInt(){
     int x=10;
     return &x;
}
int main(){
    int y=0, *p=nullptr, z=0;
    y = getInt();
    p = getAddressOfInt();
    z = *p;
    cout<<y<<", "<<z<<", "<<*p<<endl;
}
```

Change the code so that *p does not disappear

Desired output:
5, 10, 10

Heap vs. stack

```
1 #include <iostream>
2 using namespace std;
3
4 int* createAnIntArray(int len){
5
6     int arr[len];
7     return arr;
8
9 }
```

Does the above function correctly return an array of integers?

A. Yes

B. No

Next time

- Dynamic Memory Pitfalls
- Linked Lists