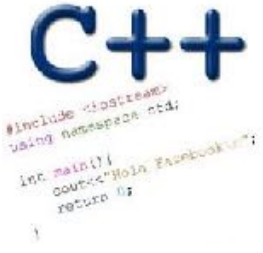
C++ MEMORY MODEL, DYNAMIC MEMORY MANAGEMENT

Problem Solving with Computers-I



Review: Pointers (good bad and ugly)

The good: Versatile (can be used to modify data that is out of scope)

Efficient (efficient way to pass large arrays to function)

The bad: Needs more work (declare, points dereference)

The ugly: Memory-related errors! (see next slide)

Pointer pitfalls and memory errors

- **Segmentation faults**: Program crashes because it attempted to access a memory location that either doesn't exist or doesn't have permission to access
- Examples
  - Out of bound array access
  - Dereferencing a pointer that does not point to anything results in undefined behavior.

Out of Bound access

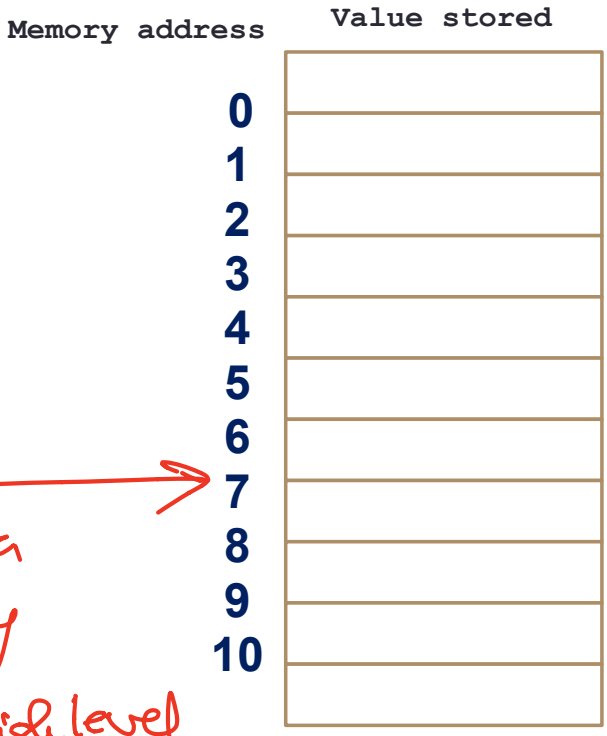
```
int arr[] = {50, 60, 70};

for(int i=0; i<=3; i++){
   cout<<arr[i]<<endl;
}
```

```
int x = 10;
int* p;
cout<<*p<<endl;
```

General model of memory

- Sequence of adjacent cells
- Each cell has 1-byte stored in it
- Each cell has an address (memory location)



This is a very low level abstraction of how data is stored in memory

See the next slide for a more high level abstraction

C++ Memory Model

Low Address Random Access Memory

Read only

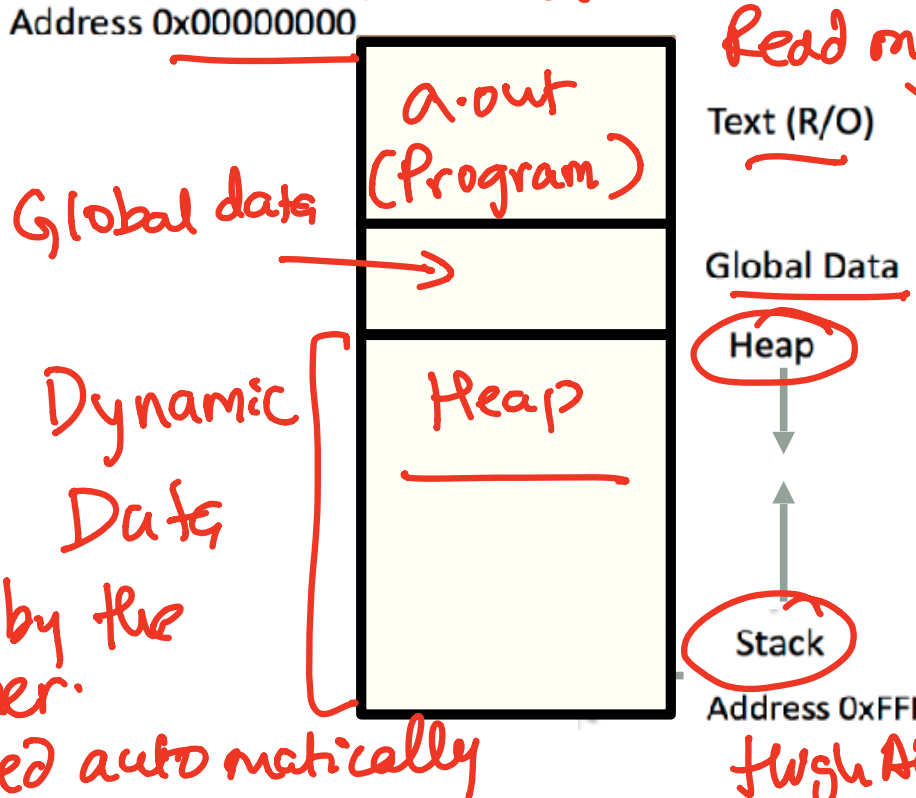
hello.cpp

↓

g++

↓

executable (a.out)



Heap: Managed by the programmer.

Stack: managed automatically

High Address

C++ data/variables: the not so obvious facts

The not so obvious facts about data/variables in C++ are that there are:

- two scopes: local and global
- three different regions of memory: global data, heap, stack
- four variable types: local variable, global variables, dynamically allocated variables, and function parameters

Variable: scope: Local vs global

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int B;
 5
 6 int* foo(){
 7     int A;
 8     A = 15;
 9     return &A;
10 }
11 int bar(){
12
13     B = 20;
14     return B;
15
16 }
```

global variable

local variable

Which of the functions on the left has a memory related bug?

A. foo()

B. bar()

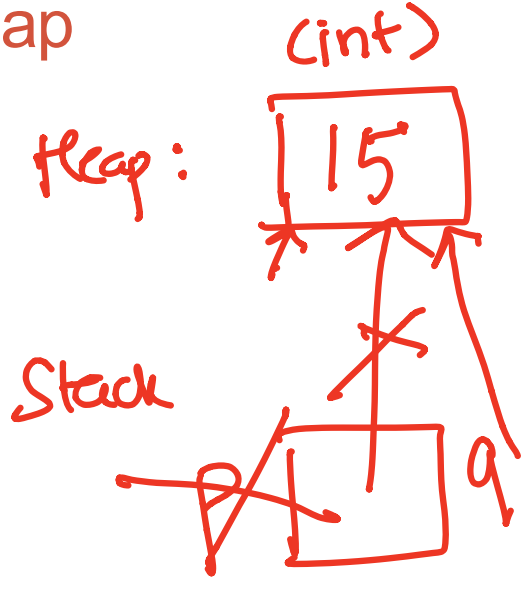
C. Both

D. Neither

Dynamically managed memory: Heap

```
1  #include <iostream>
2  using namespace std;
3
4  int* createAnInt(){
5
6
8
9
10 }
```

int *p;
p = new int;
return p;



Write a function to create an integer in memory

- Need to create the object on heap memory
- To create an object on the heap use the new keyword

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 }
```

// array is created on the stack

Does the code correctly create an array of integers?

A. Yes

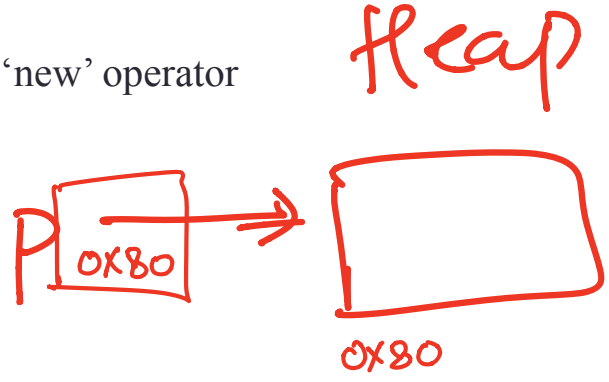
B. No

Because array was created on the stack it is deleted when function returns

Dynamic memory management

- To allocate memory on the heap use the 'new' operator
- To free the memory use delete

```
int *p= new int;
delete p;// This statement
```



does not delete the pointer p.
Instead it deletes the int object located in memory at the address stored in p.

In general write

delete address of heap object to be deleted;

Dangling pointers and memory leaks

- Dangling pointer: Pointer points to a memory location that no longer exists
- Memory leaks (tardy free):
  - Heap memory not deallocated before the end of program
  - Heap memory that can no longer be accessed

Dynamic memory pitfalls

- Does calling foo() result in a memory leak? A. Yes B. No

```
void foo(){
     int * p = new int;

}
```

Q: Which of the following functions returns a dangling pointer?

```
int* f1(int num){
     int *mem1 =new int[num];
     return(mem1);
}
```

```
int* f2(int num){
      int mem2[num];
      return(mem2);
}
```

A. f1

B. f2

C. Both

Review of homework 7, problem 4

```
void printRecords(UndergradStudents records [], int numRecords);
int main(){
    UndergradStudents ug[3];
    ug[0] = {"Joe", "Shmoe", "EE", {3.8, 3.3, 3.4, 3.9} };
    ug[1] = {"Macy", "Chen", "CS", {3.9, 3.9, 4.0, 4.0} };
    ug[2] = {"Peter", "Patrick", "ME", {3.8, 3.0, 2.4, 1.9} };
    printRecords(ug, 3);
}
```

Expected output

These are the student records:
ID# 1, Shmoe, Joe, Major: EE, Average GPA: 3.60
ID# 2, Chen, Macy, Major: CS, Average GPA: 3.95
ID# 3, Peter, Patrick, Major: ME, Average GPA: 2.77

Next time

- C++ Memory Model
- Dynamic memory allocation