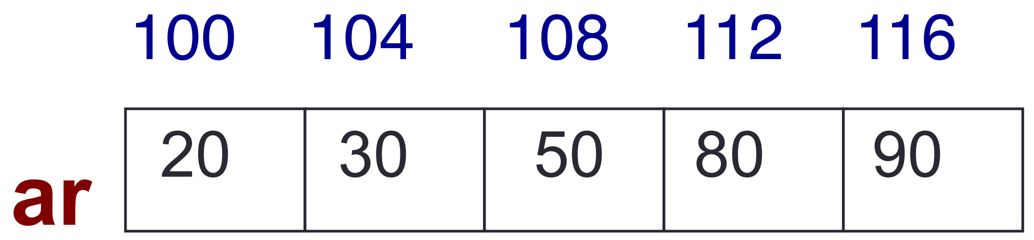
THE GOOD, BAD AND UGLY ABOUT POINTERS

Problem Solving with Computers-I

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

int main(){
    cout<<"Hola Facebook!";
    return 0;
}
```

The good: Pointers pass data around efficiently

`Pointers and arrays`



- `ar` is like a pointer to the first element
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`

- Use pointers to pass arrays in functions
- Use *pointer arithmetic* to access arrays more conveniently

Pointer Arithmetic

```
int arr[]={50, 60, 70};
int *p;
p = arr;
p = p + 1;
*p = *p + 1;
```

Pointer Arithmetic

- What if we have an array of large structs (objects)?
  - C++ takes care of it: In reality, `ptr+1` doesn't add `1` to the memory address, but rather adds the size of the array element.
  - C++ knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes: 1 byte for a char, 4 bytes for an int, etc.

The bad? Using pointers needs work!

1) A pointer can only point to one type –(basic or derived) such as `int`, `char`, a `struct`, another pointer, etc

2) After declaring a pointer: `int *ptr;`

   `ptr` doesn't actually point to anything yet.

   We can either:

   ➢ make it point to something that already exists, OR

   ➢ allocate room in memory for something new that it will point to

The ugly: memory errors!

"*The overwhelming majority of program bugs and computer crashes stem from problems of memory access... Such memory-related problems are also notoriously difficult to debug. Yet the role that memory plays in C and C++ programming is a subject often overlooked…. Most professional programmers learn about memory entirely through experience of the trouble it causes.*"

…. Frantisek Franek

(Memory as a programming concept)

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.

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

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

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

Pointer Arithmetic Question

How many of the following are invalid?

I. pointer + integer (ptr+1)
II. integer + pointer (1+ptr)
III. pointer + pointer (ptr + ptr)
IV. pointer – integer (ptr – 1)
V. integer – pointer (1 – ptr)
VI. pointer – pointer (ptr – ptr)
VII. compare pointer to pointer (ptr == ptr)
VIII. compare pointer to integer (1 == ptr)
IX. compare pointer to 0 (ptr == 0)
X. compare pointer to NULL (ptr == NULL)

```
#invalid
  A: 1
  B: 2
  C: 3
  D: 4
  E: 5
```

III, V, VIII are the problems

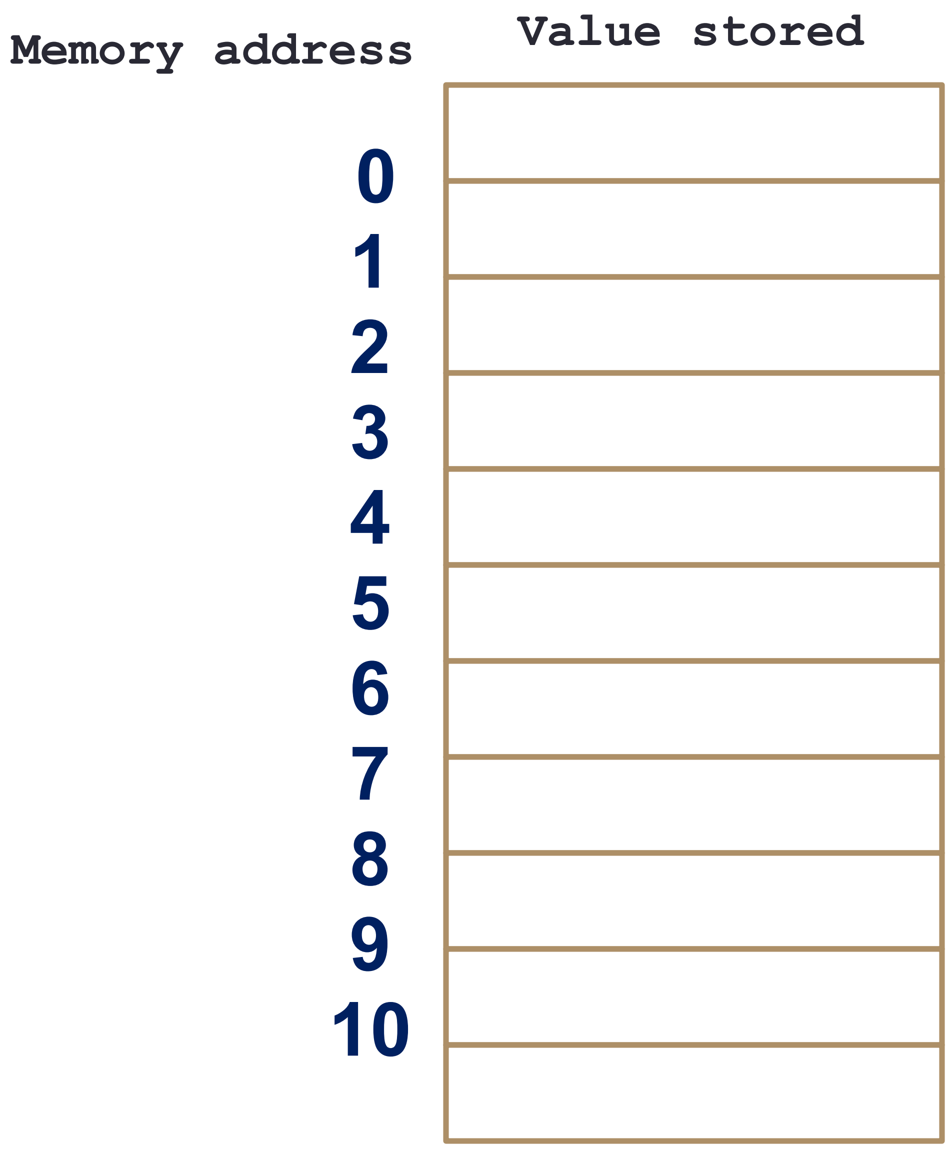
C++ MEMORY MODEL, DYNAMIC MEMORY MANAGEMENT

Problem Solving with Computers-I

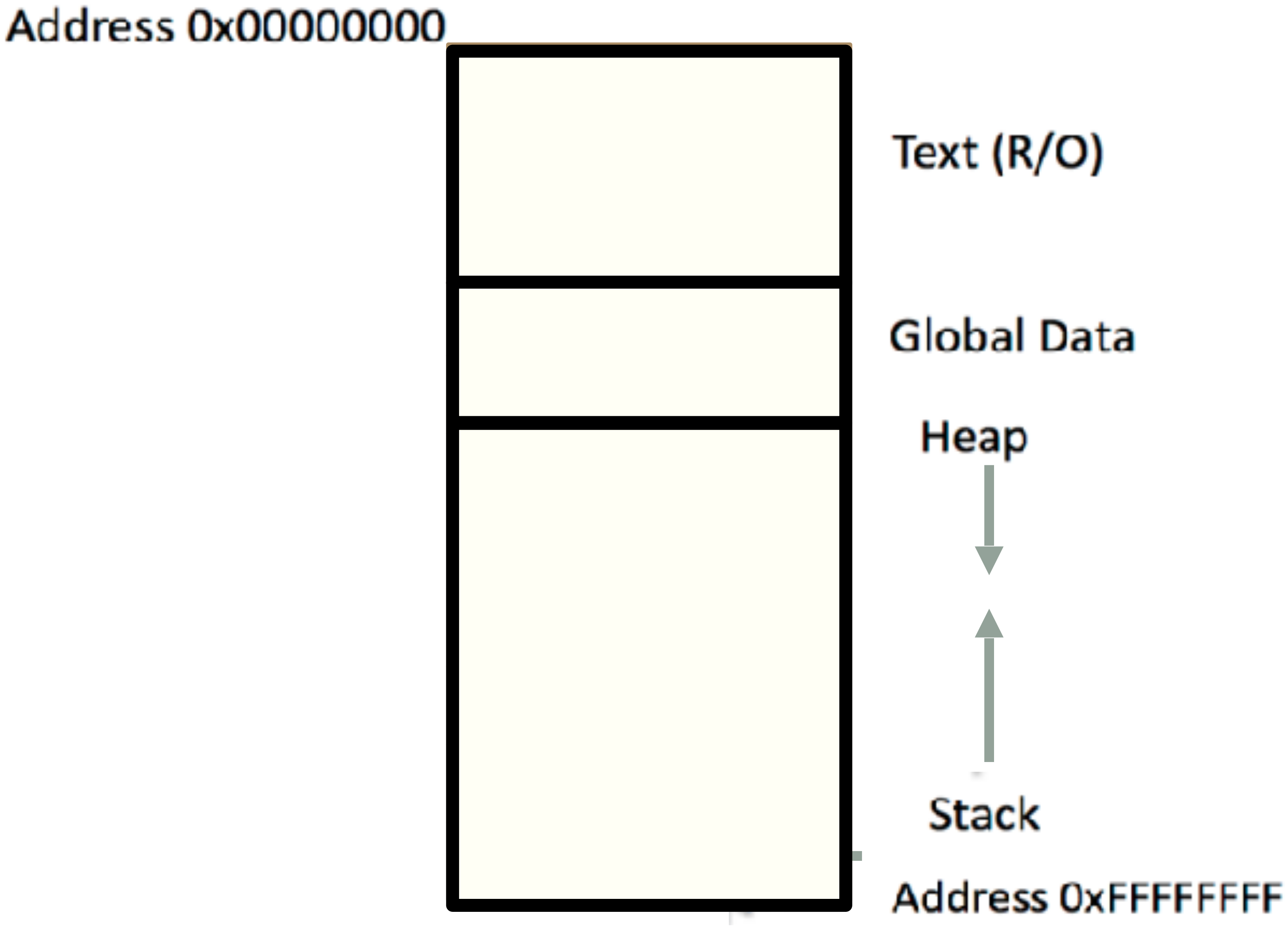
```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

General model of memory

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



C++ Memory Model



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

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

int B;

int* foo(){
    int A;
    A = 15;
    return &A;
}
int bar(){

    B = 20;
    return B;

}
```

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

A. foo()
B. bar()
C. Both
D. Neither

Dynamically managed memory: Heap

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

int* createAnInt(){




}
```

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

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

int* createAnIntArray(int len){

    int arr[len];
    return arr;

}
```

Does the code correctly create an array of integers?

A. Yes

B. No

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;
```

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

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
```