CPSC 260
Pointers and memory management

Winter 2013

Instructor: Hassan Khosravi

Variable declaration

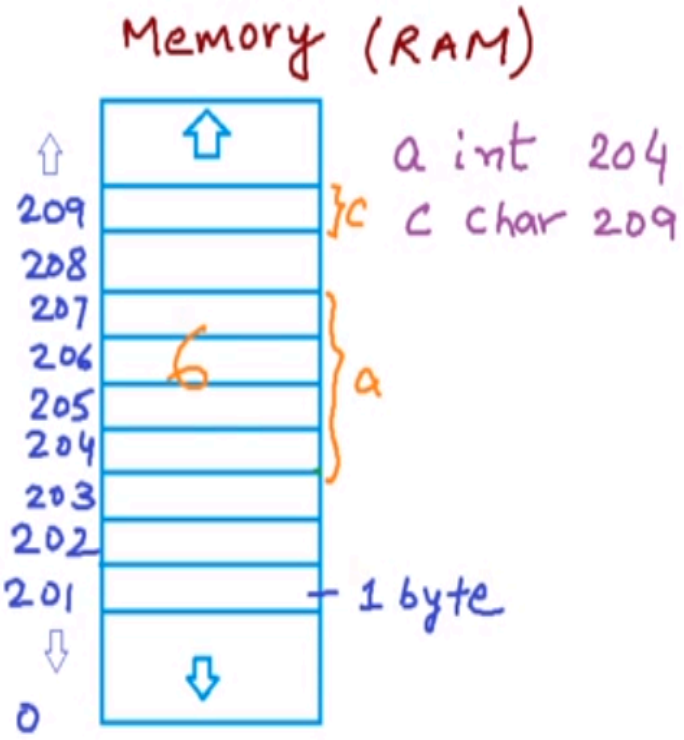
Introduction to pointers in C

int - 4 bytes

char - 1 byte

float - 4 bytes

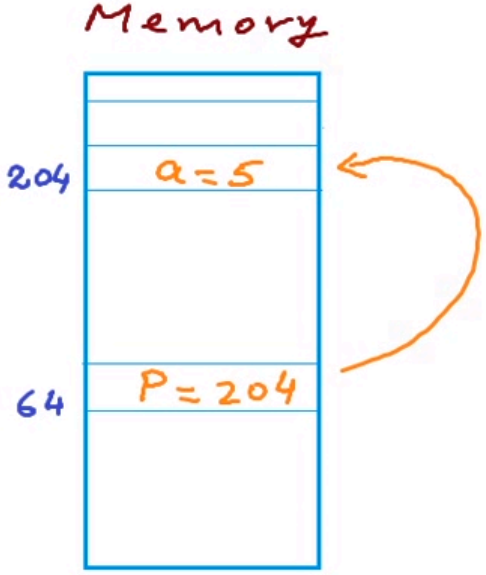
```
int a;
char c;
a = 5;
...
...
a++;
```



pointers

Pointers - variables that store address of another variable

```
int a;
int *p;
p = &a;
a = 5;
Print p    // 204
Print &a   // 204
Print &p   // 64
print *p   // 5   ⇒ derefrencing
```




```
int a=5;
int* p =&a;
cout << p<<endl;
cout <<&a << endl;
cout <<*p <<endl;
cout << &p<<endl;
```

pointers

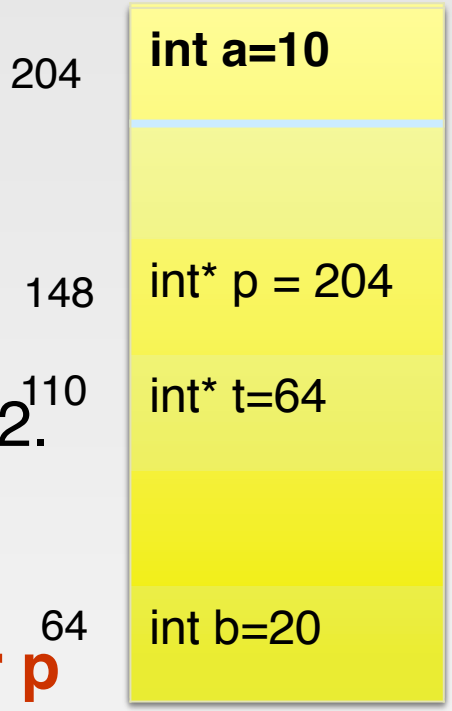
int a=10;

int b= 20;

int* p= &a;

Int* t= &b ;

*p = 12 // changes the value stored at 204 to 12.



IMPORTANT – The two different stars

Int *p or int* p→ declares an integer pointer p

***p = a → uses the * operator to dereference p**

My suggestion is to do

int* p and *p

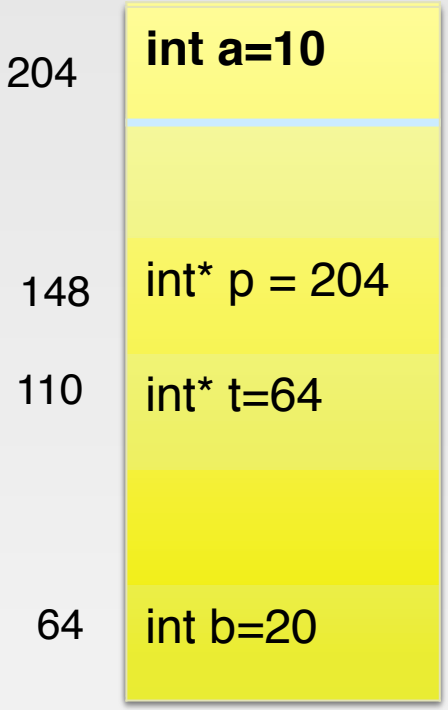
Clicker Question

int a=10;

int b= 20;

int* p= &a

Int* t= &b



After performing a=b **memory** at

A: 204 will change

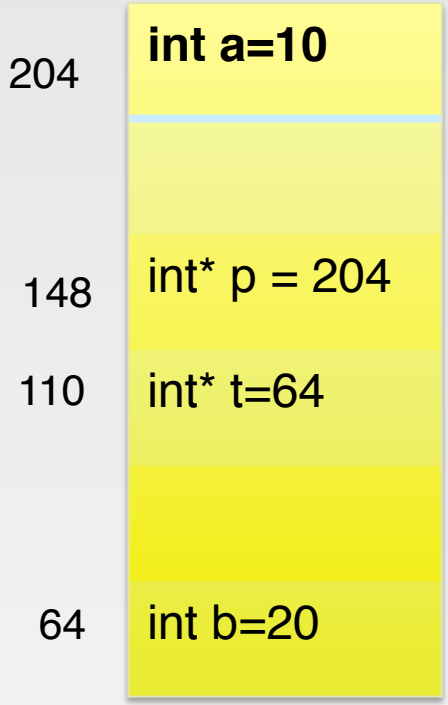
B: 148 will change

C: both A and B

D: none of the above

Clicker Question

```
int a=10;
int b= 20;
int* p= &a
Int* t= &b
```



After performing *p = *t **memory** at

A: 204 will change

B: 148 will change

C: both A and B

D: none of the above

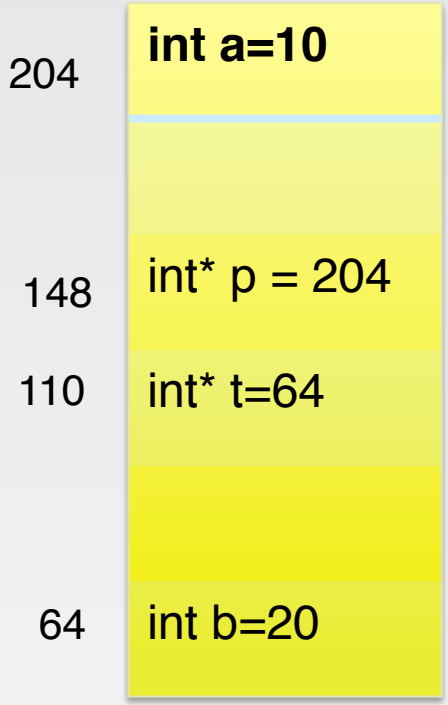
Clicker Question

int a=10;

int b= 20;

int* p= &a

Int* t= &b



After performing p = &b

A: the value of p changes

B: the value of *p

C: both A and B

D: none of the above

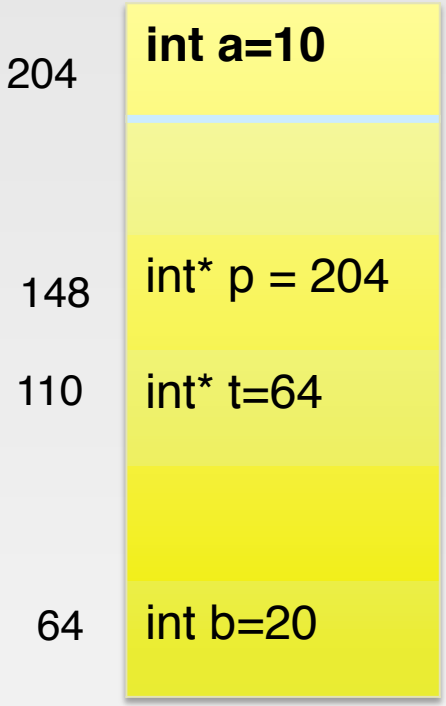
Clicker Question

int a=10;

int b= 20;

int* p= &a

Int* t= &b



After performing p = &b; *p=a

A: the value of p changes

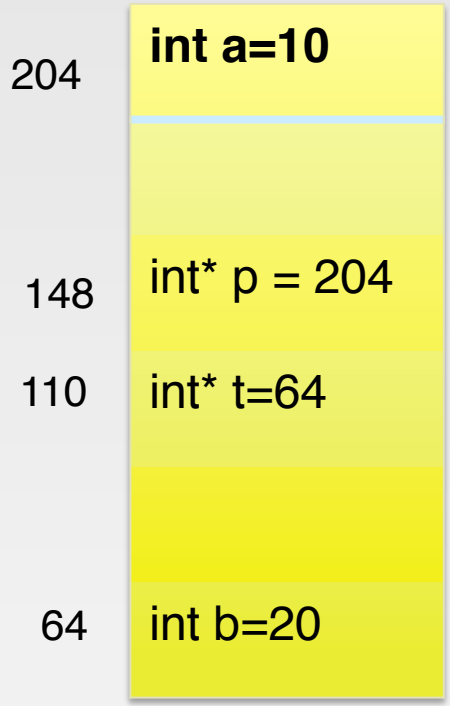
B: the value of b changes

C: both A and B

D: none of the above

Clicker Question

```
int a=10;
int b= 20;
int* p= &a
Int* t= &b
```



After performing p =t;

A: the address of a cannot be retrieved

B: changing *t will change *p

C: both A and B

D: none of the above

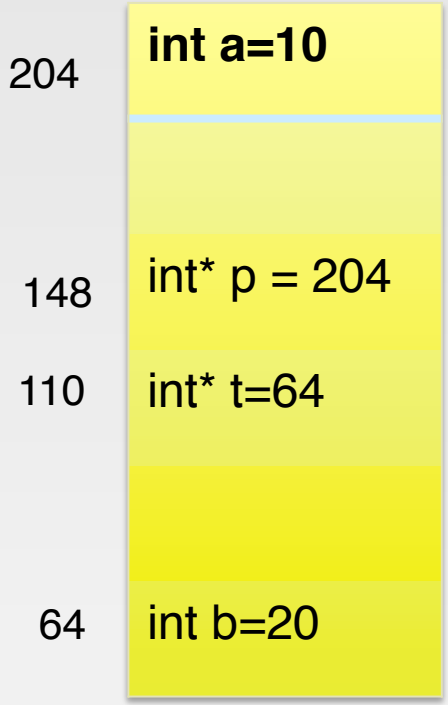
Clicker Question

int a=10;

int b= 20;

int* p= &a

Int* t= &b



*p=b and p=&b

A: Are literally the same;

B: both change the value of *p to 20

C: both end up making p be equal to t

D: none of the above

Question

```
int main()
{
    int a=10;
    int* p= &a;
    cout << p <<"    " << *p<< endl;
    p++;
    cout << p <<"    " << *p<< endl;
    return 0;
}
```

What is the value of p and *p in the second cout?

Stack example

Pointers as function arguments - Call by reference


```
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("a = %d",a);
}
```

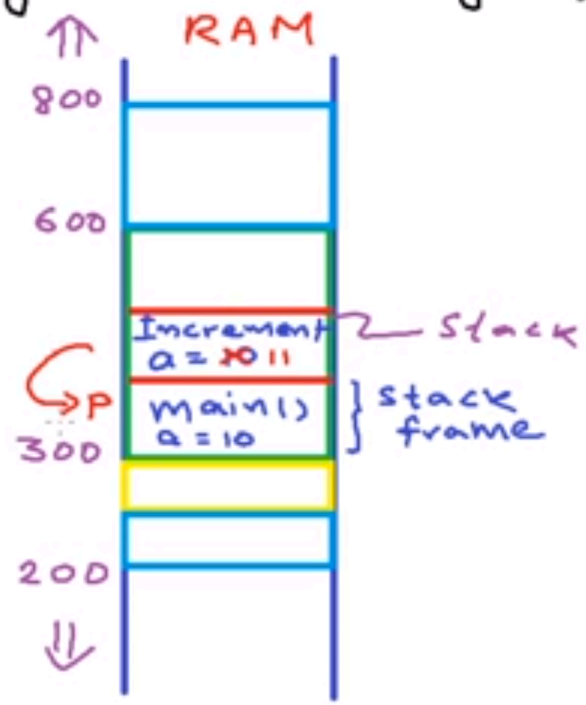
A: a=10

B: a=11

Call by Value

Pointers as function arguments - Call by reference

```
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("a = %d",a);
}
```



a from main is mapped to a in Increment: a → a

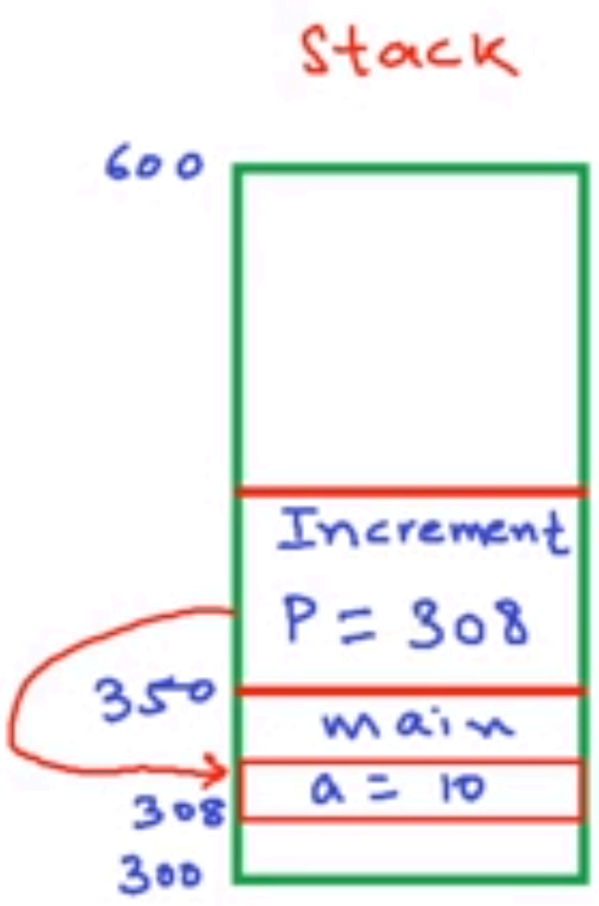
If the argument in increment was called x then a → x

mapped/copied into another variable→ **call by value**

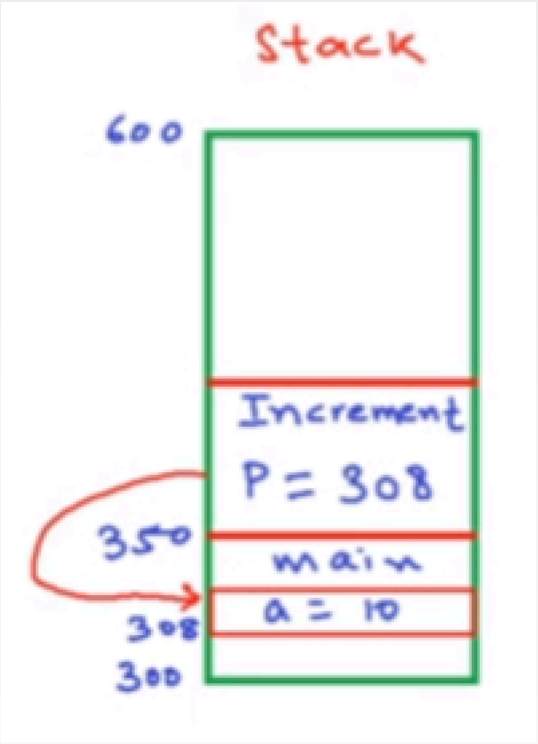
Call by reference

- Can what Albert had in mind work?

```c
#include<stdio.h>
void Increment(int *p)
{
    *p = (*p) + 1;
}
int main()
{
    int a;
    a = 10;
    Increment(&a);
    printf("a = %d",a);
}
```



```
void increment(int* p)
{
    cout << "address of pointer P in function "<< &p<<endl; // 400
    cout << "value of pointer P in function "<< p<<endl; // 308
    cout << "Value P is pointing to is " << *p << endl; //10
            *p = *p +1;
}
int main()
{
    int a=10;
    cout << "address of a in main "<< &a<<endl; // 308
    increment(&a);
}
```



Another example Adding- call by value

```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int Add(int a,int b){ // Called funtion
    printf("Address of a in Add = %d\n",&a);
    int c = a+b;
    return c;
}
int main() {  //Calling function
    int a = 2, b =4;
    printf("Address of a in main = %d\n",&a);
    //Call by value
    int c = Add(a,b); // value in a of main is copied to a of Add.
                      // value in b of main is copied to b of Add
    printf("Sum = %d\n",c);
}
```

The two addresses for a are different

Another example Adding- call by reference

```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int Add(int* a,int* b){ // Called funtion
    int c = (*a) + (*b);
    return c;
}
int main() {  //Calling function
    int a = 2, b =4;
    printf("Address of a in main = %d\n",&a);
    int c= Add(&a,&b); // a and b are integers local to Main
    printf("Sum = %d\n",c);
}
```

Returning a pointer

Next

```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int* Add(int* a,int* b){ // Called funtion
    int c = (*a) + (*b);
    return &c;
}
int main() {  //Calling function
    int a = 2, b =4;
    printf("Address of a in main = %d\n",&a);
    int* ptr = Add(&a,&b); // a and b are integers local to Main
    printf("Sum = %d\n",*ptr);
}
```

Would this work correctly?
If it works correctly does that mean it is correct?

Returning a pointer

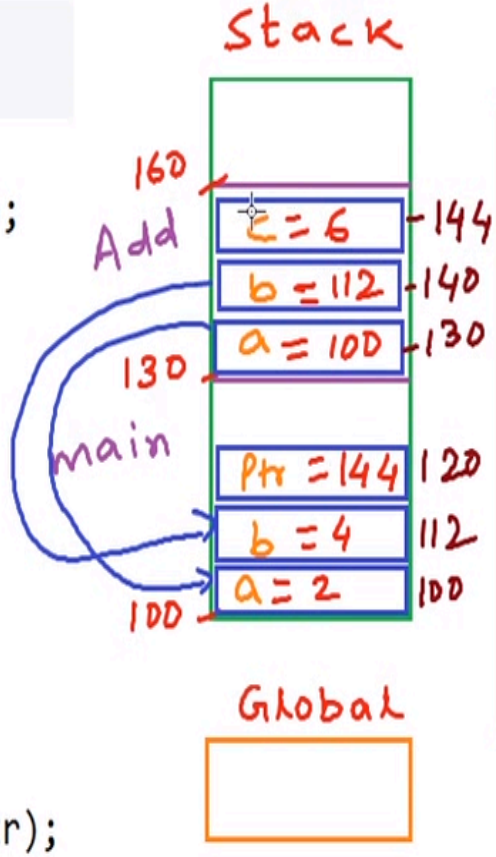
```
int* add(int* a, int* b)
{
    int c= *a + *b;
    return &c;
}
void printHelloWorld()
{
    cout<< "hello world"<<endl;
}
int main()
{
    int a=2, b=4;
    int* ptr = add(&a, &b);
    printHelloWorld();
    cout << *ptr<<endl;
}
```

Now if you check again the output is not correct. Why?

Returning a pointer

Pointers as function returns

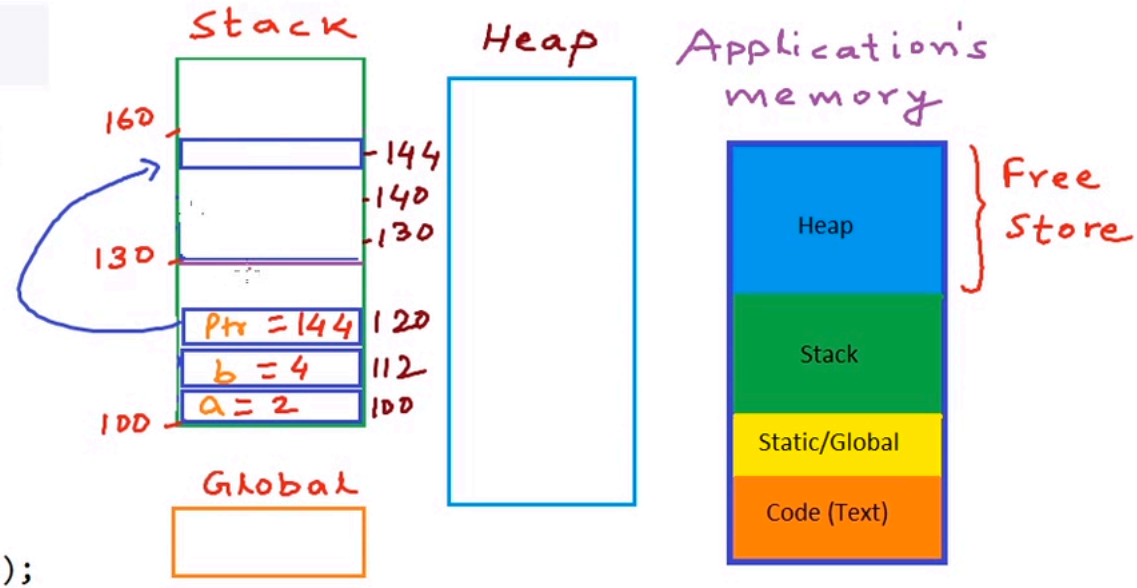
```c
#include<stdio.h>
#include<stdlib.h>
void PrintHelloWorld(){
    printf("Hello World\n");
}
int *Add(int* a,int* b){
    int c = (*a) + (*b);
    return &c;
}
int main() {
  ✓int a = 2, b =4;
  ✓int* ptr = Add(&a,&b);
    PrintHelloWorld();
    printf("Sum = %d\n",*ptr);
}
```



Returning a pointer

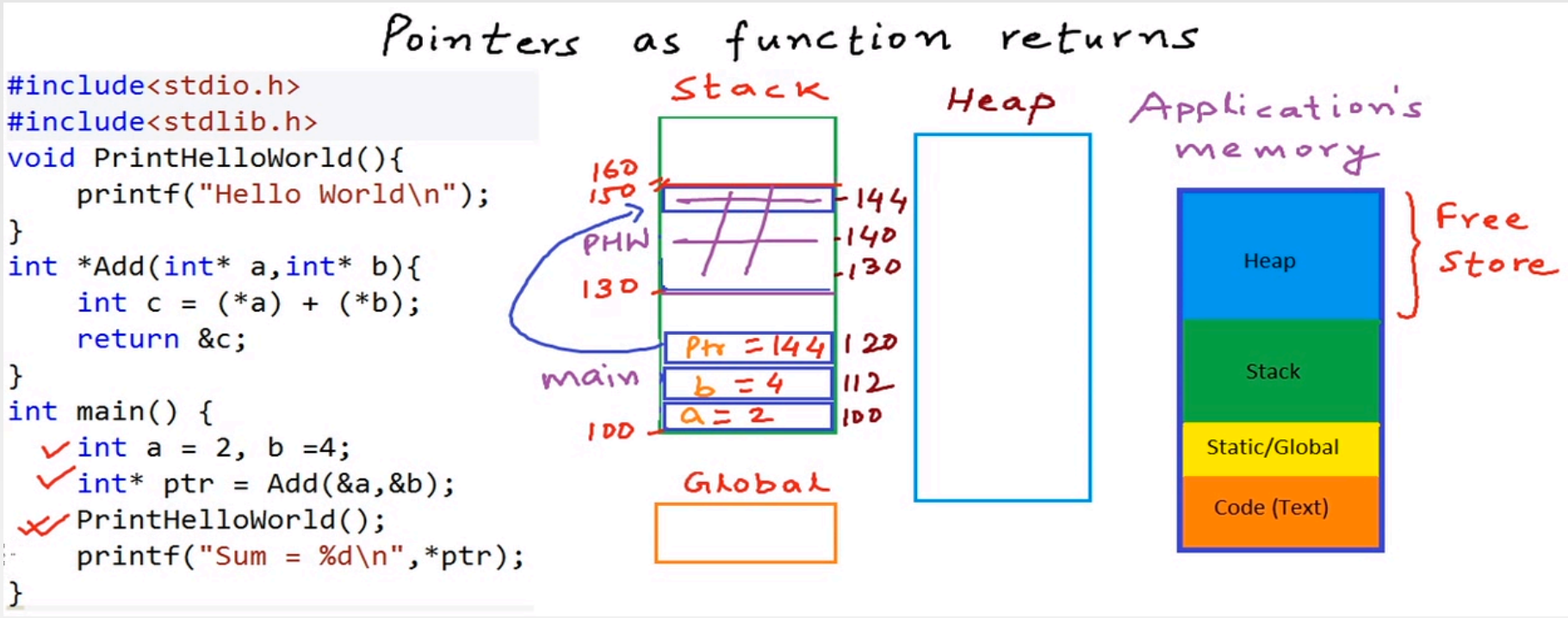
Pointers as function returns

```
#include<stdio.h>
#include<stdlib.h>
void PrintHelloWorld(){
    printf("Hello World\n");
}
int *Add(int* a,int* b){
    int c = (*a) + (*b);
    return &c;
}
int main() {
  int a = 2, b =4;
  int* ptr = Add(&a,&b);
  PrintHelloWorld();
    printf("Sum = %d\n",*ptr);
}
```



if we access address 144 it MIGHT still contains the value 6

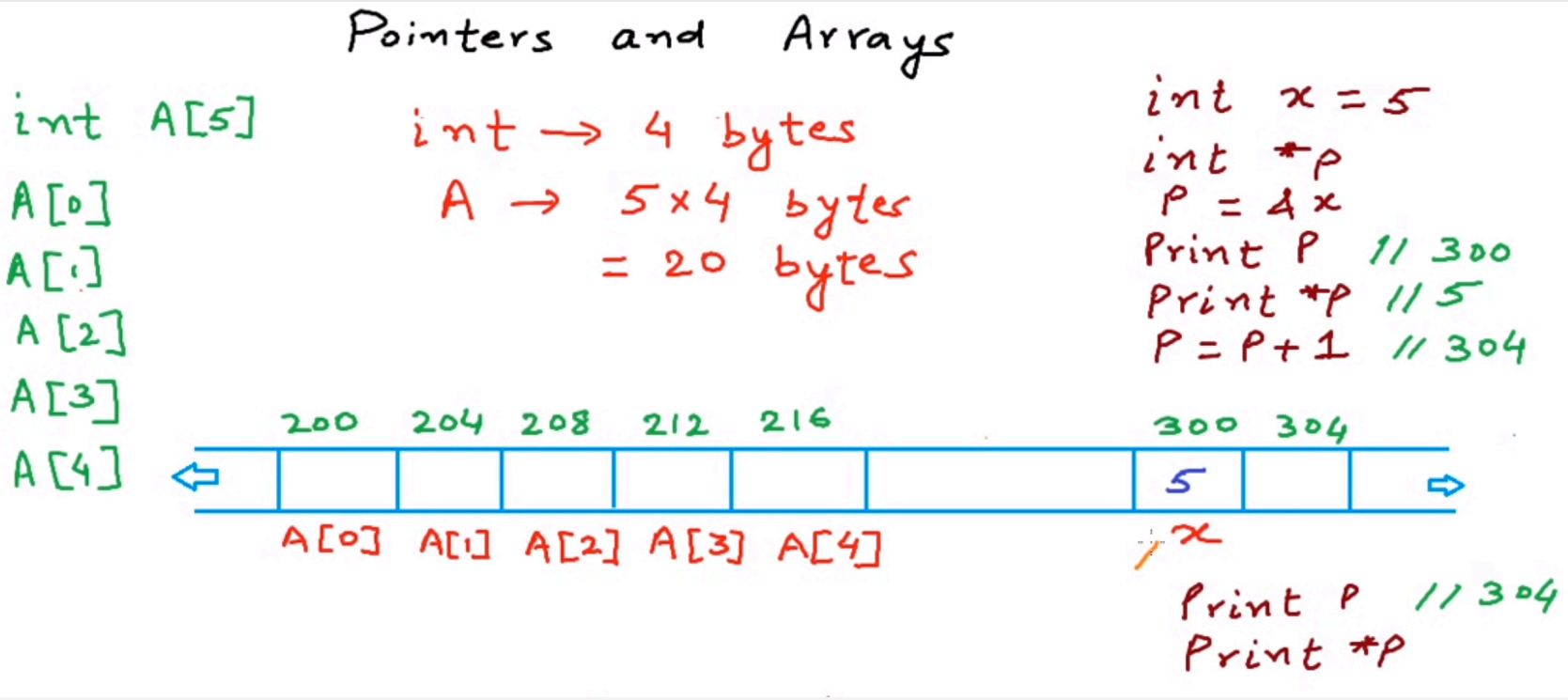
However this is still the wrong way of doing it



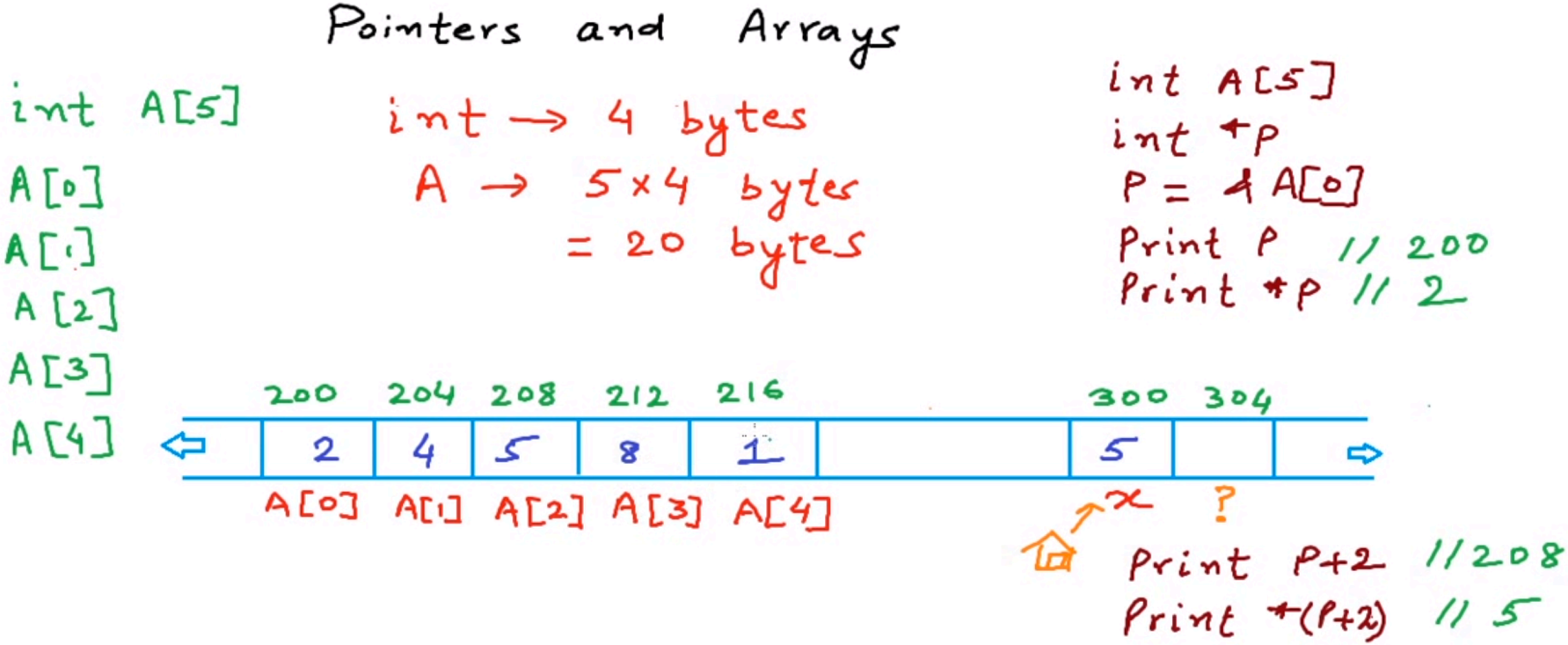
- The printHelloWorld() now writes over address 144

- Does it ever make sense for a function to return a pointer? YES
  - More on this example soon.

Pointer arithmatics and arrays



Pointers and arrays



Arrays as Function arguments

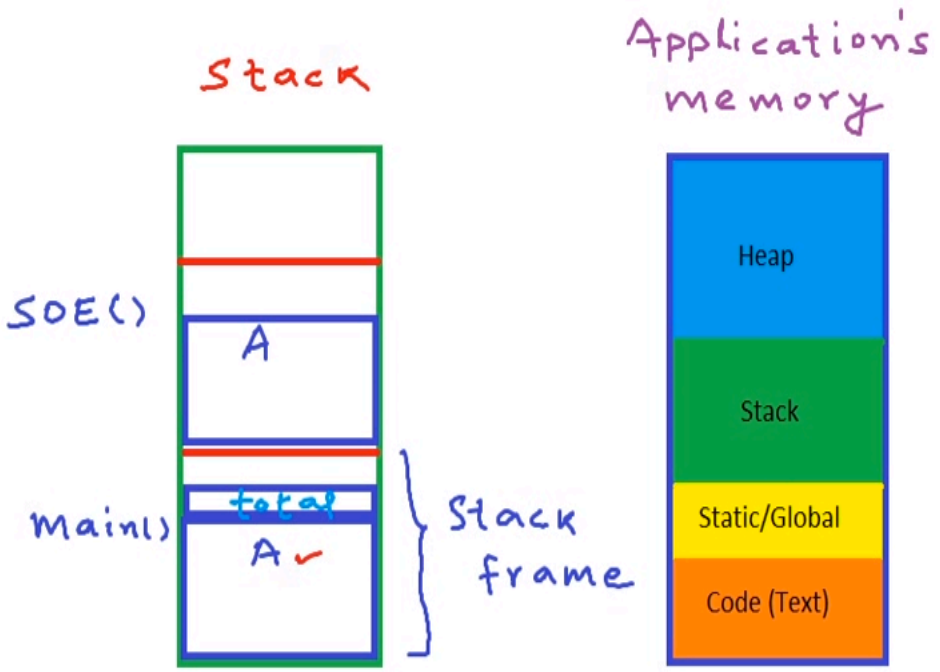
```
int sumOfElements(int a[], int size)
{
    int i,sum=0;
    for(i; i<size;i++)
        sum+= a[i];
    return sum;
}
int main()
{
    int array[]= {1,2,3,4,5};
    int size = sizeof(array)/sizeof(array[0]);
    cout << "size is " << size<<endl; // 5 is returned
    int sum = sumOfElements(array, size);
    cout << "sum is " << sum; // 15 is returned
}
```

Arrays as Function arguments

```
int sumOfElements(int a[])
{
    int i,sum=0;
    int size = sizeof(a)/sizeof(a[0]);
    cout << "size is " << size<<endl;  what is returned?
    for(i=0; i<size;i++)
        sum+= a[i];
    return sum;
}
int main()
{
    int array[]= {1,2,3,4,5};
    int sum = sumOfElements(array);
    cout << "sum is " << sum; // what is returned?
}
```

Arrays as Function arguments

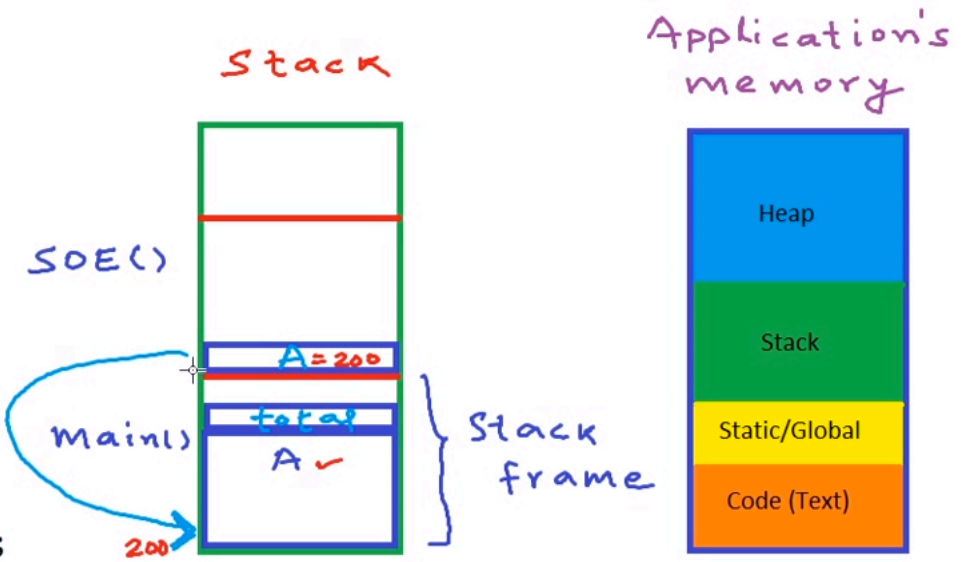
```c
#include<stdio.h>
int SumOfElements(int A[])
{
    int i, sum = 0;
    int size = sizeof(A)/sizeof(A[0]);
    for(i = 0;i< size;i++)
    {
        sum+= A[i];
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};
    int total = SumOfElements(A);
    printf("Sum of elements = %d\n",total);
}
```



This does not happen

Arrays as Function arguments

```c
#include<stdio.h>
int SumOfElements(int A[])
{
    int i, sum = 0;
    int size = sizeof(A)/sizeof(A[0]);
    for(i = 0;i< size;i++)
    {
        sum+= A[i];
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};
    int total = SumOfElements(A);
    printf("Sum of elements = %d\n",total);
}
```



- Only the pointer is sent to the function.
  - This is referred to as **call by reference**
  - What happends if you modify the array in the function?

Pointer types

Pointer types, void pointer, pointer arithmetic

int* → int

char* → char

Why strong types?

Why not some generic type?

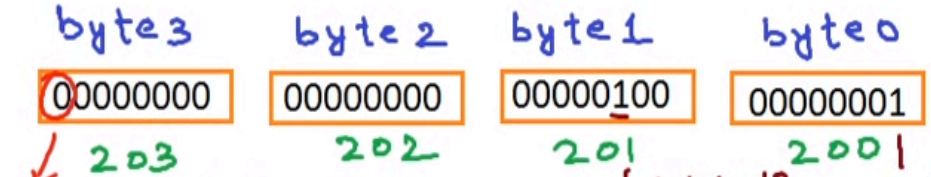
Dereference

└ Access/modify value

int - 4 bytes

char - 1 byte

float - 4 bytes



int a = 1025

int *p

p = &a

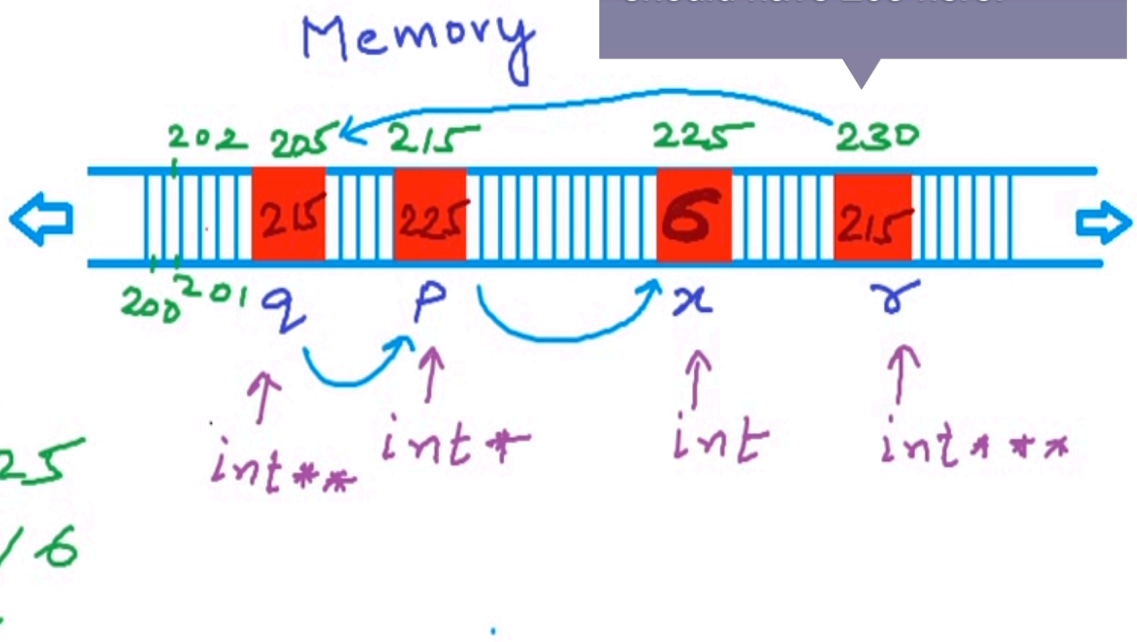
Print p // 200

Print *p // Look at 4 bytes starting 200

Pointer to pointer

```c
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p); // 6
    printf("%d\n",*q); // 225
    printf("%d\n",*(*q)); // 6
```

Pointer to pointer



Correction: Address of q is 205 and not 215. So, we should have 205 here.

Example

```
int main()
{
    int a=5;
    int* p= &a;
    int** p1= &p;
    cout << "value a= " << a<< endl;
    cout << "value p= " << p << endl;
    cout << "value p1= "<< p1<< "   value *p1= "<< *p1 << "   value **p1= "<<
**p1<<endl;
}
```

MEMORY MANAGEMENT

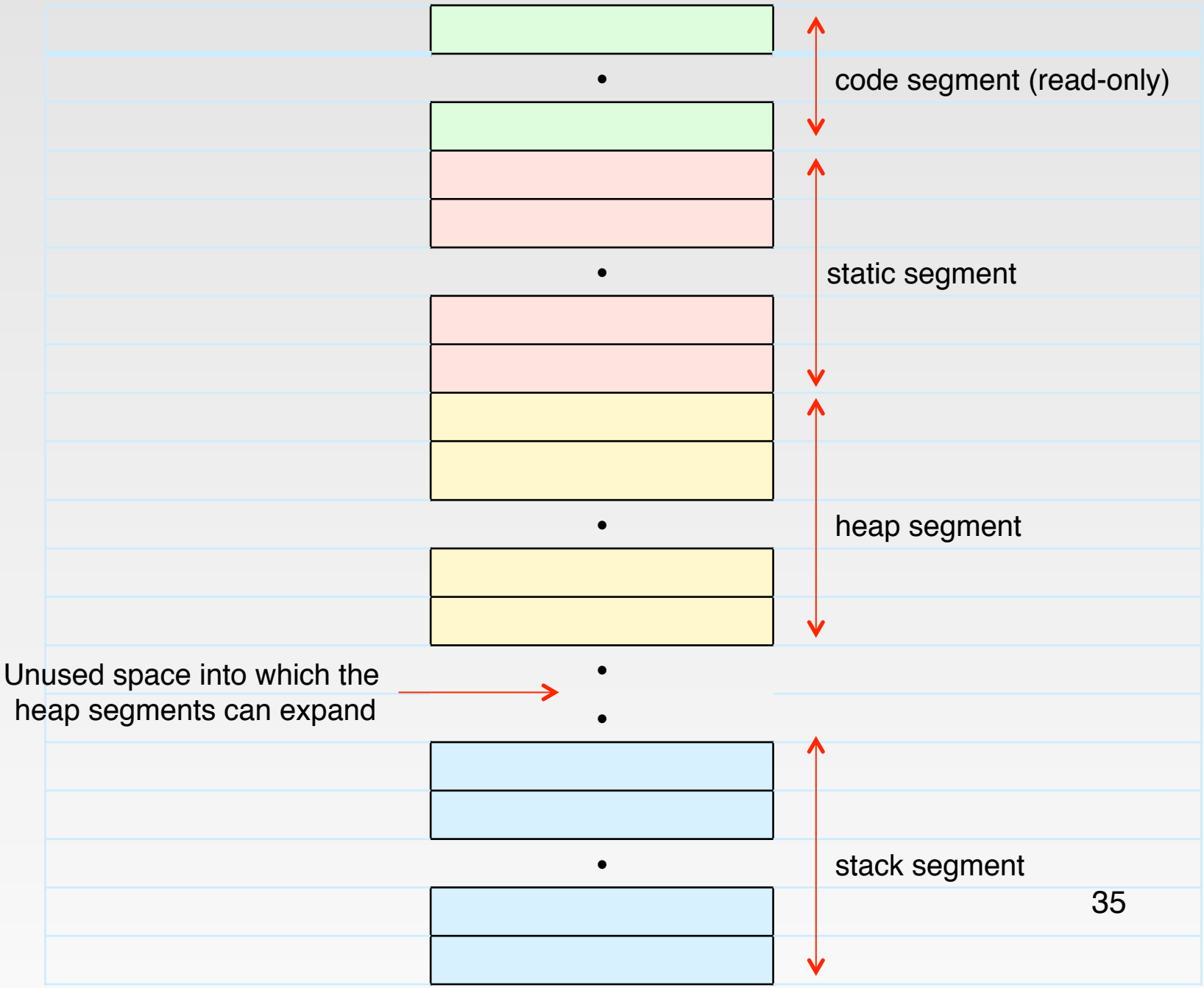
Main memory segments in C++

- Code segment: The compiled program
  - Memory allocated to Code segment is fixed and read only
- Static/global segment
  - Mostly static variables that are available throughout the life of the program.

Main memory segments in C++

- Frame/Stack segment
  - Area of memory that temporary holds arguments and variables
  - The stack grows and shrinks as functions push and pop local variables
  - there is no need to manage the memory yourself, variables are allocated and freed automatically
  - the stack has size limits
  - stack variables only exist while the function that created them, is running
    - void MyFunction( )

```
{
        int i; //    local variable created on stack
} // variable goes out of scope and is now deleted.  The scope may be less than a function.
```

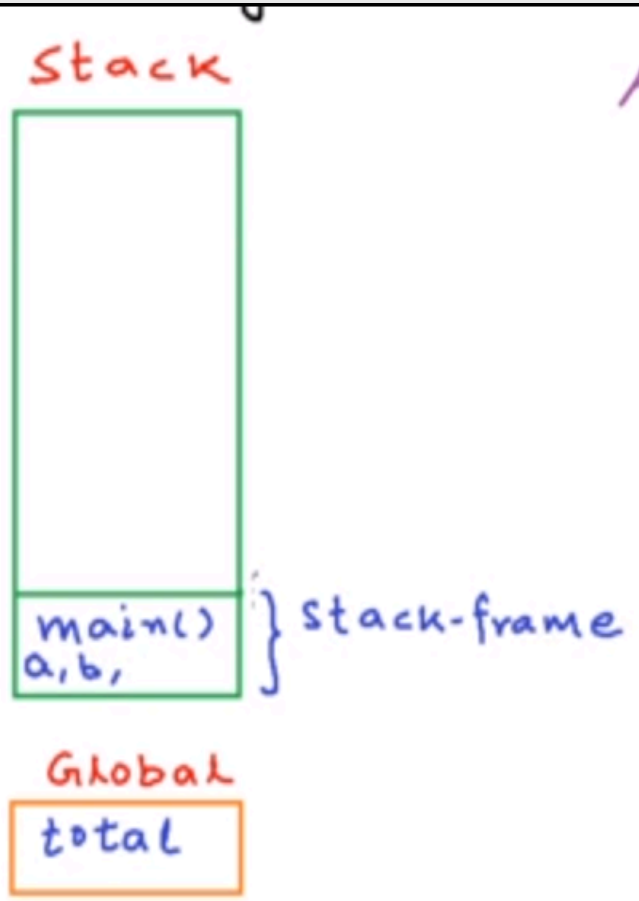

code segment (read-only)
static segment
heap segment
Unused space into which the heap segments can expand
stack segment

Stack example

```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```

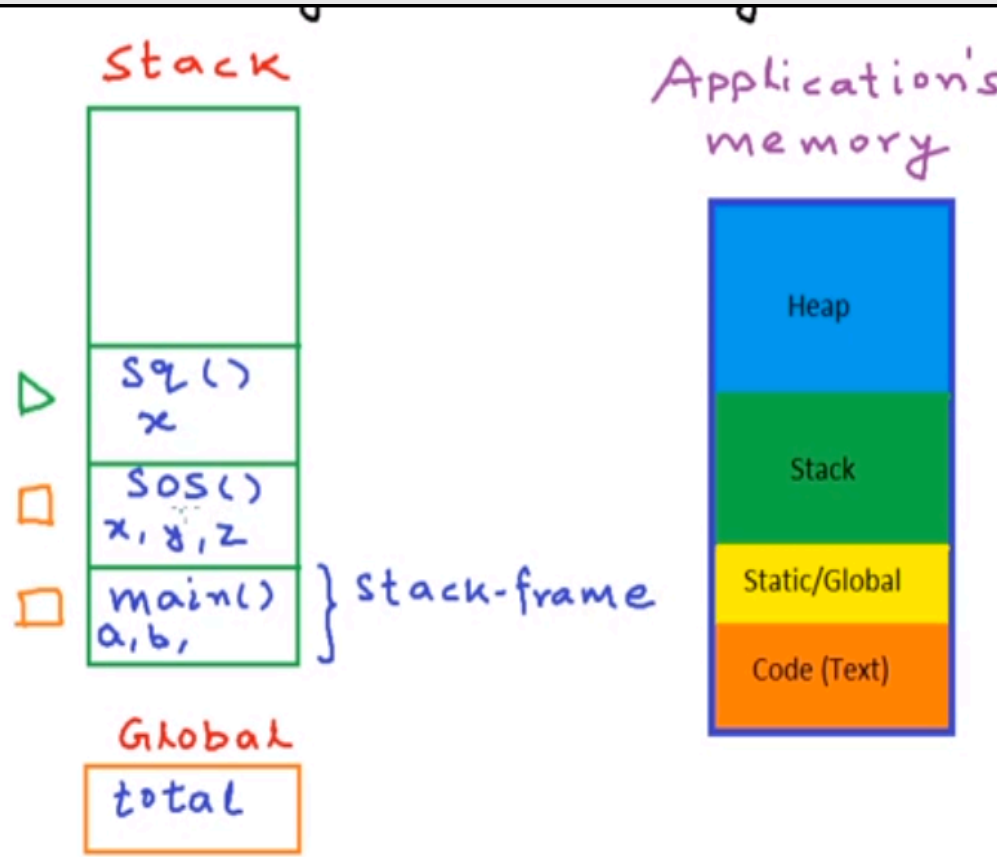
Stack example

```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



Stack example

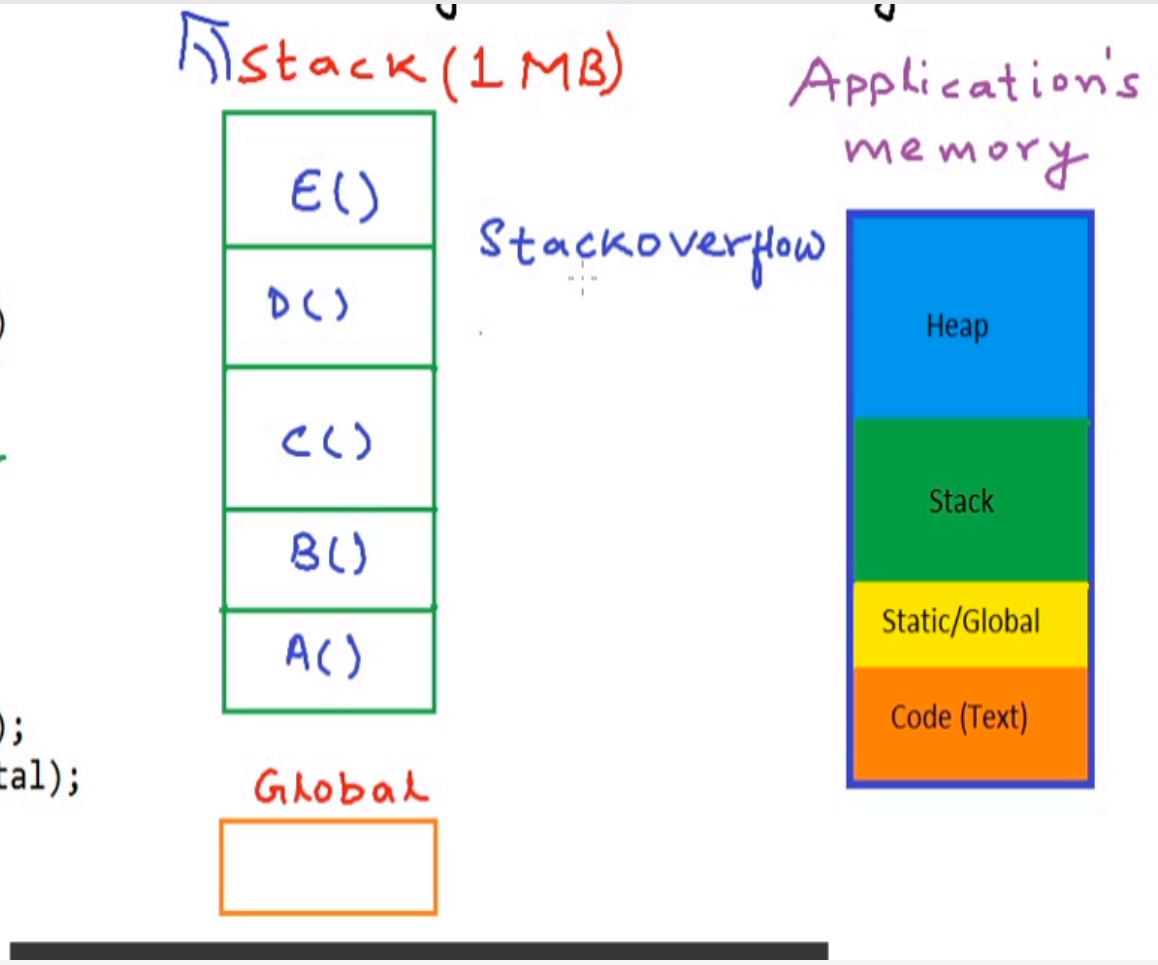
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



Function on top of stack is running and the rest are paused

Stack example

```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```

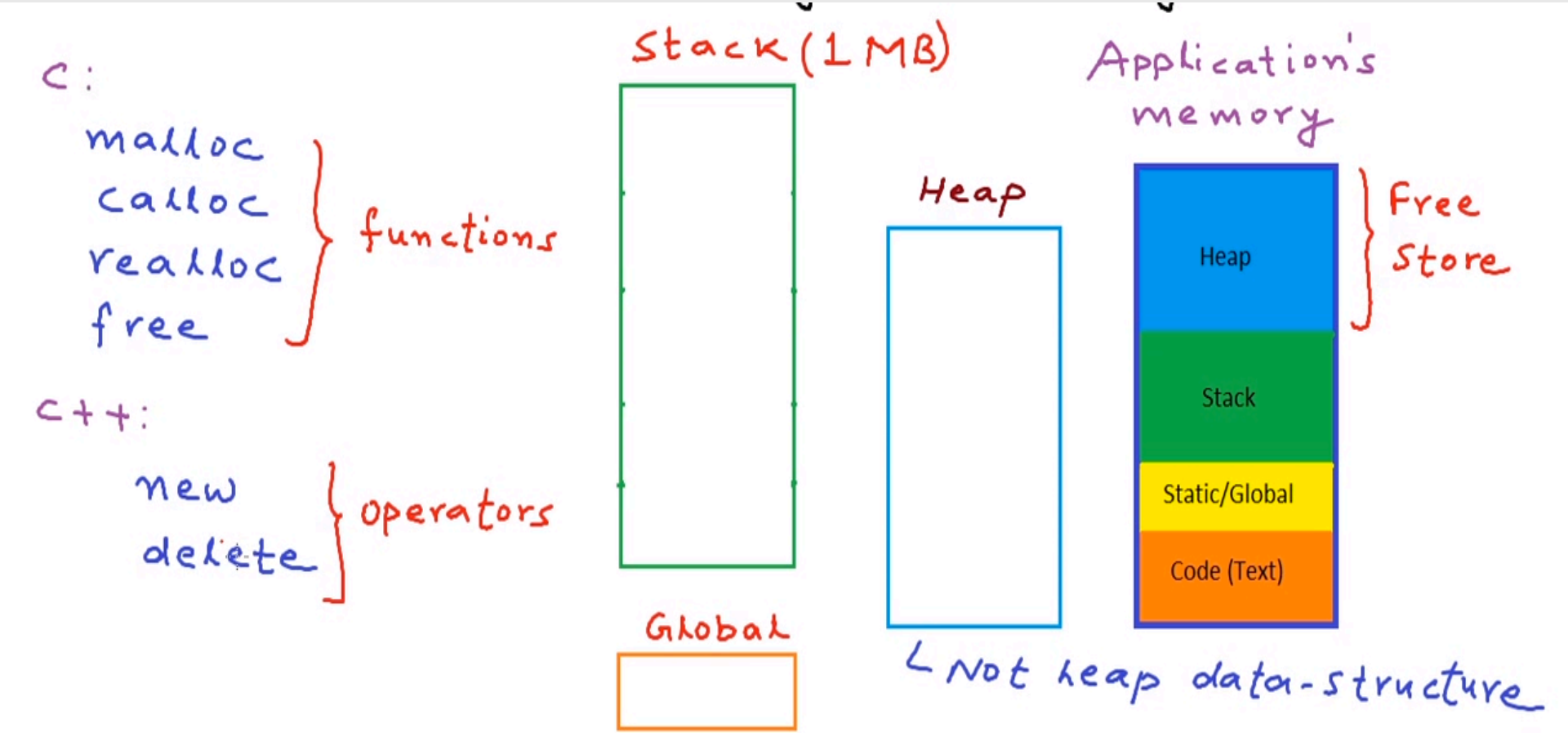


Allocating large data structures in stack is a problem

Main memory segments in C++

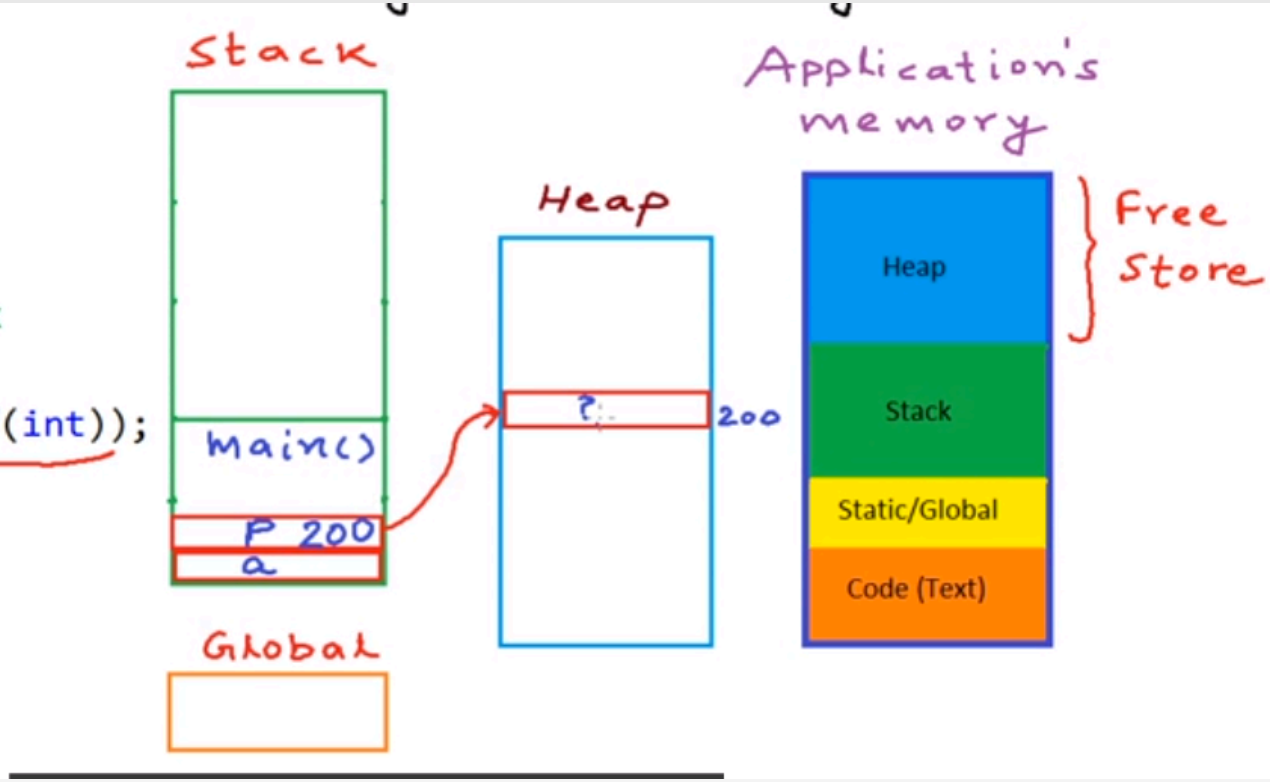
- Heap segment
  - Reserved for dynamic memory allocations of the program
  - Use new and delete keywords to allocate and deallocate memory.
  - Heap objects must **explicitly be deleted** by the programmer.

Heap example



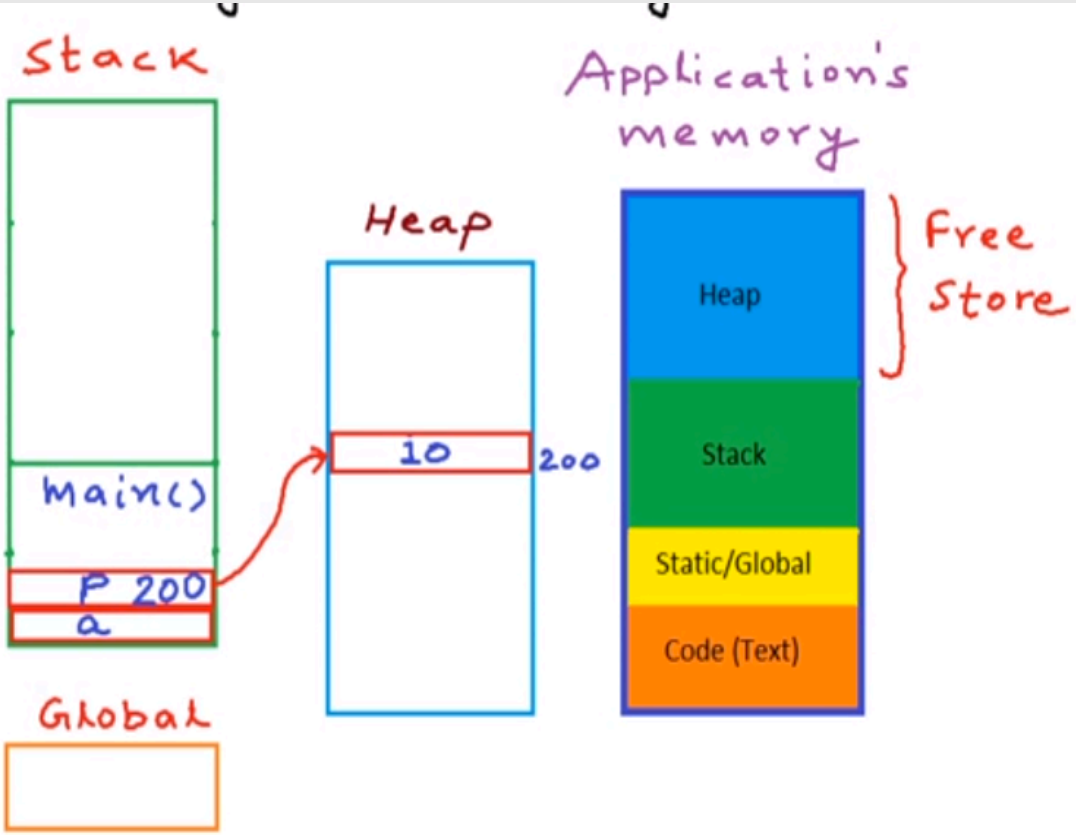
Heap example in C

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int a; // goes on stack
   int *p;
   p = (int*)malloc(sizeof(int));
```



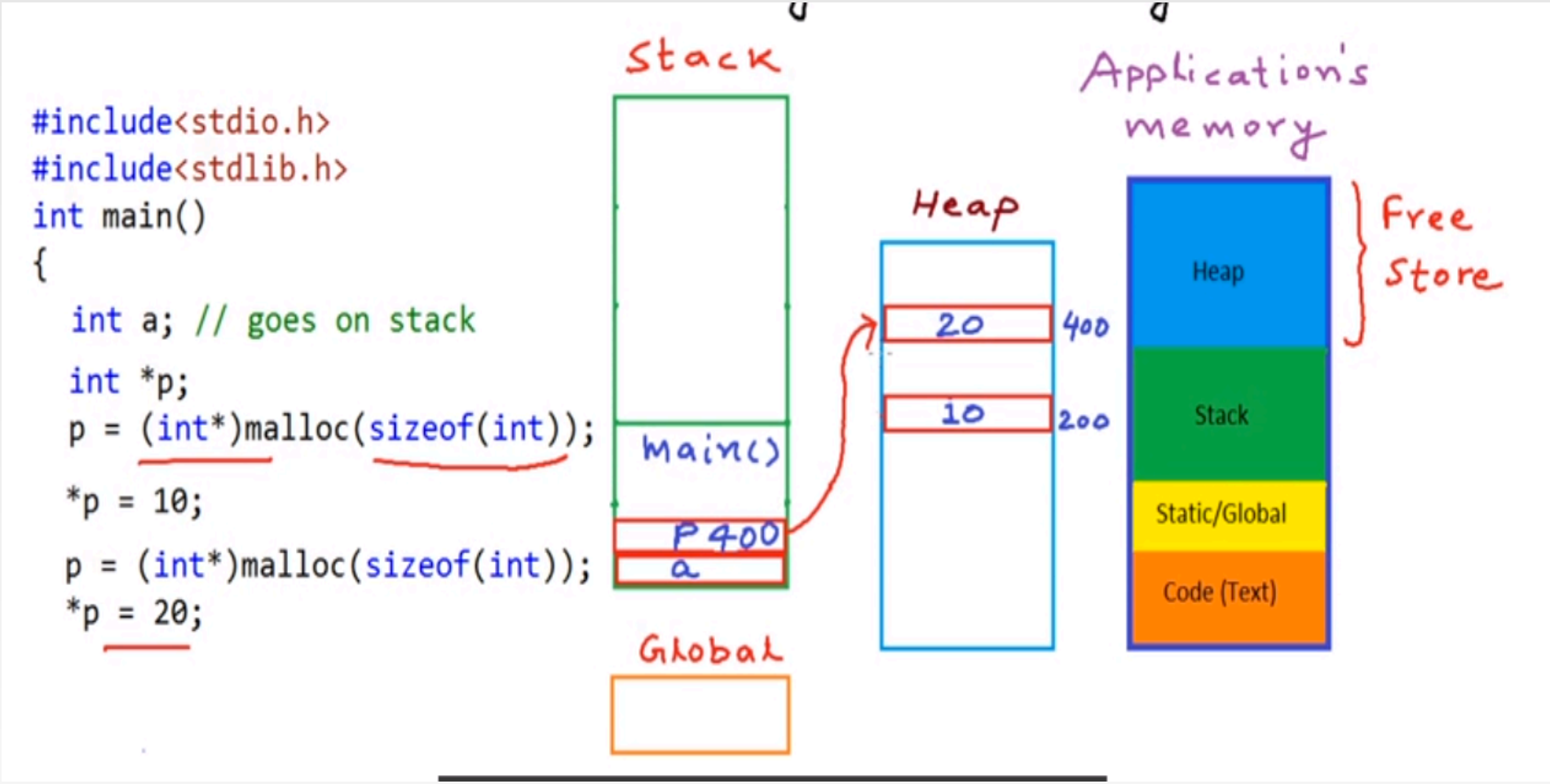
Heap example in C

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int a; // goes on stack
   int *p;
   p = (int*)malloc(sizeof(int));
   *p = 10;
```



Dereference operator (*)

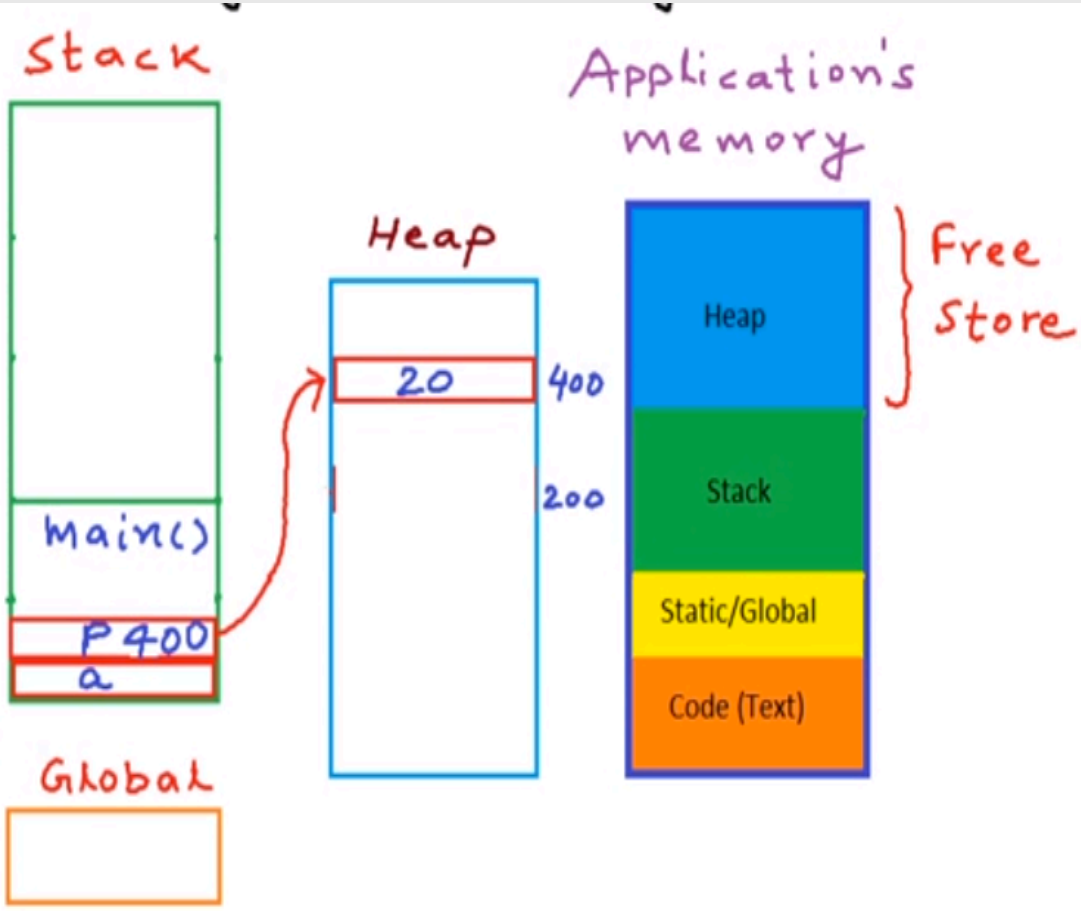
Heap example in C



Memory leakage

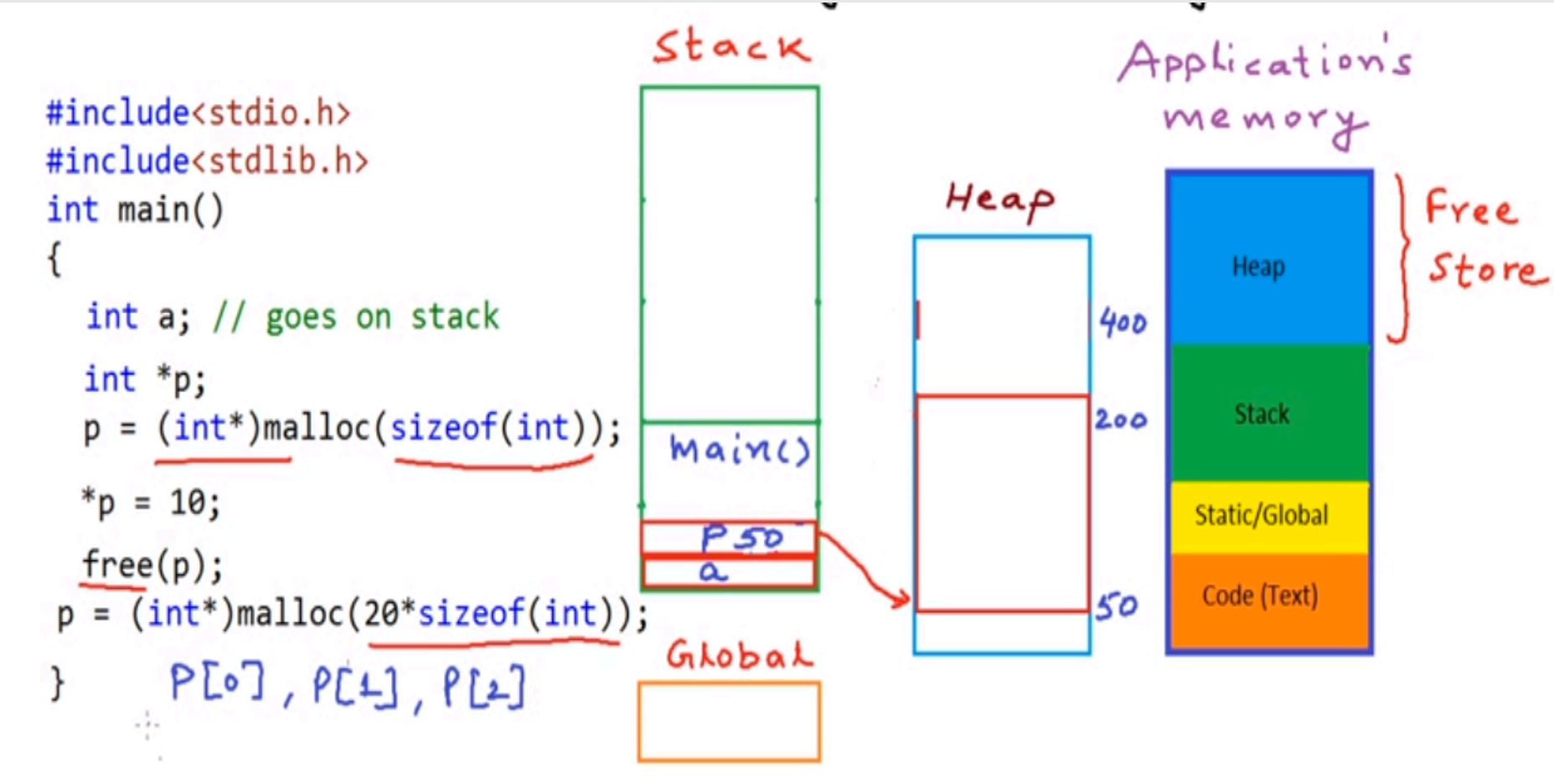
Heap example in C

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int a; // goes on stack
   int *p;
   p = (int*)malloc(sizeof(int));
   *p = 10;
   free(p);
   p = (int*)malloc(sizeof(int));
   *p = 20;
```



The programmer is responsible for deallocation

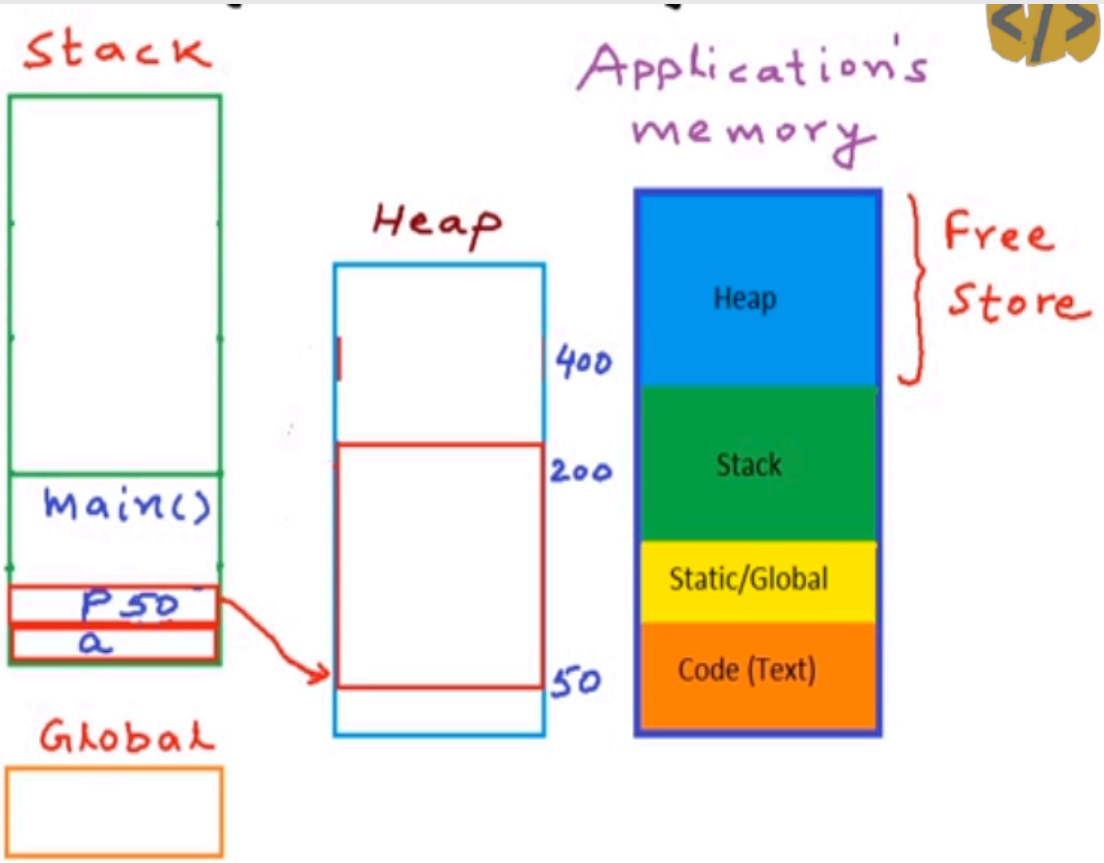
Heap example in C



The programmer is responsible for deallocation

Heap example in C++

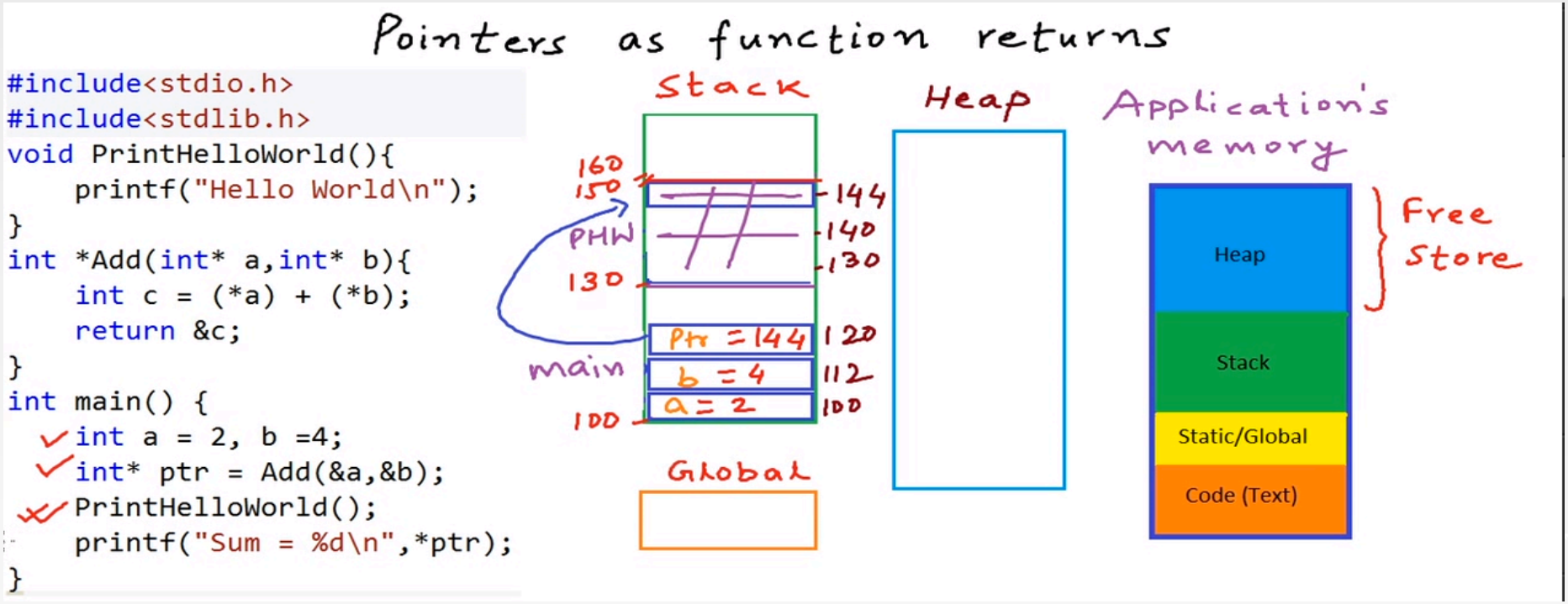
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a; // goes on stack
    int *p;
    p = new int;
    *p = 10;
    delete p;
    p = new int[20];
    delete[] p;
}
```



How can we use p in a function?

Use heap for storing variables

- Let's revisit this problem again
  - How can we use functions that return a pointer.

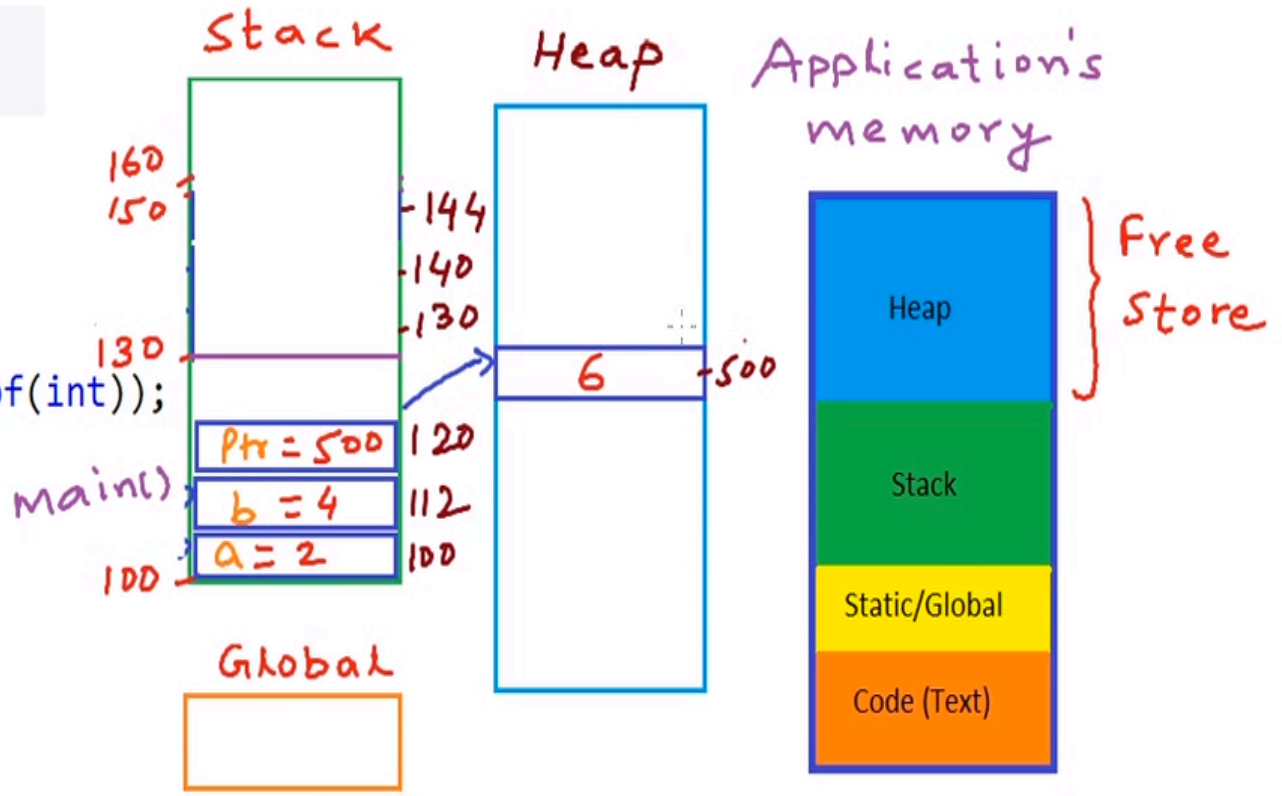


Use heap for storing variables

```
int* add(int* a, int* b)
{
    int* c=new int;
    *c= *a + *b;
    return c;
}
void printHelloWorld()
{
    cout<< "hello world"<<endl;
}
int main()
{
    int a=2, b=4;
    int* ptr = add(&a, &b);
    printHelloWorld();
    cout << *ptr<<endl;
}
```

Use heap for storing variables

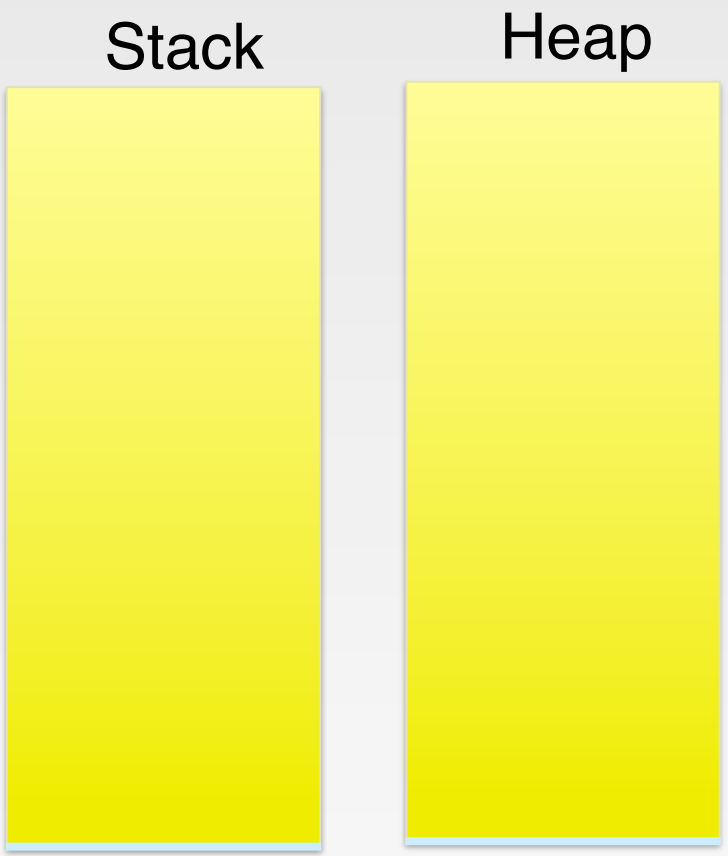
```c
#include<stdio.h>
#include<stdlib.h>
void PrintHelloWorld(){
    printf("Hello World\n");
}
int *Add(int* a,int* b){
  int* c = (int*)malloc(sizeof(int));
  *c = (*a) + (*b);
  return c;
}
int main() {
  int a = 2, b =4;
  int* ptr = Add(&a,&b);
  PrintHelloWorld();
  printf("Sum = %d\n",*ptr);
}
```



Stack and Heap example

- Draw the stack and heap diagram for the following code

```
int main()
{
    Set* first = new (Set);
    first->insert(2);
    cout  << first<<endl;
    cout <<(*first).size();
    cout << first->size();
    Set second;
    second.insert(3);
    Set* third = &second;
    third->print();
    return 0;
}
```



Stack vs Heap Pros and Cons

- Stack
  - very fast access
  - don't have to explicitly de-allocate variables
  - space is managed efficiently by CPU, memory will not become fragmented
  - local variables only
  - limit on stack size (OS-dependent)
  - variables cannot be resized
- Heap
  - variables can be accessed globally
  - no limit on memory size
  - (relatively) slower access
  - no guaranteed efficient use of space
  - you must manage memory (you're in charge of allocating and freeing variables)
  - variables can be resized using realloc()

Stack array vs. Heap array

- Stack

```
const int size = 20;
int anotherSize=30;
int array[size];
int anotherArray[anotherSize];
//error: expression must have a constant value
```

- Heap

```
int anotherSize=30;
int* secondArray = new int[anotherSize];
delete [] secondArray;
return 0;
```

Memory leakage

```
int main()
{
    int a,b;
    int* p= &a;
    p=&b;
    int* p1= new int;
    p1 = new int;
}
```

In what ways are p and p1 different?

Memory leakage

```
void leakMemory()
{
        int* mem = new int;
        //delete mem;
}

int main()
{
        while(true)
        {
                leakMemory();
                cout << “leaking"<< endl;
        }
}
```

Pointers

- An object variable contains an object
- A pointer specifies where an object is located
- A pointer variable stores an object's location
- Pointers can refer to objects allocated on demand
- Pointers provide access to shared objects
- Necessary for polymorphism
- Pointers are closely related to the implementation of arrays
- Read chapter 7 of the textbook for more examples on how to use pointers.