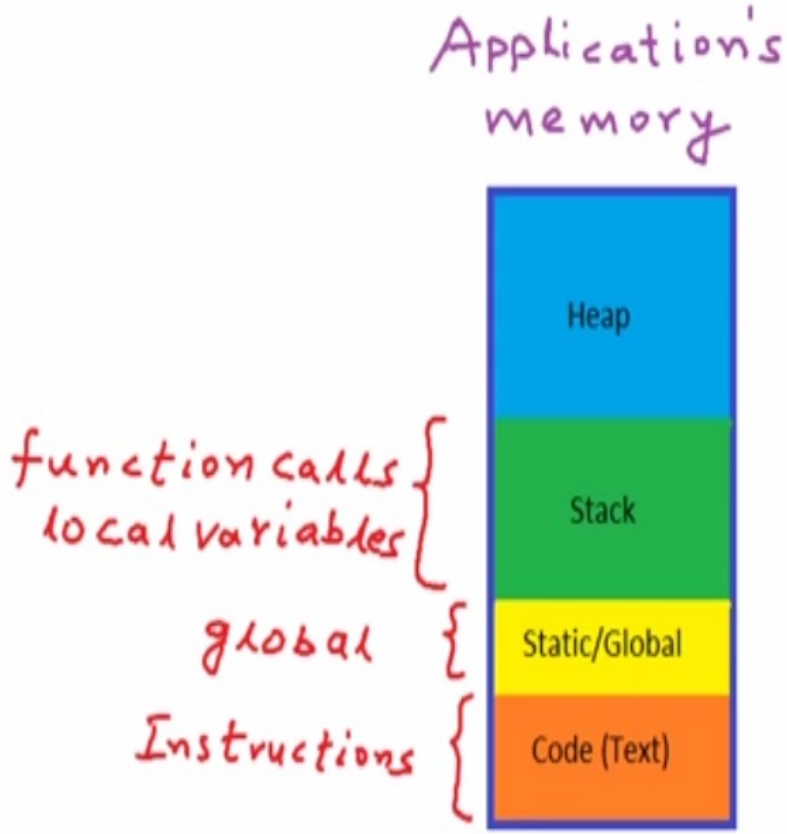
BBM 201 Data Structures
Hacettepe University

Lecture 4:
Review of Dynamic Memory

Resource:
Lecture Videos
www.mycodeschool.com/videos/pointers-and-arrays

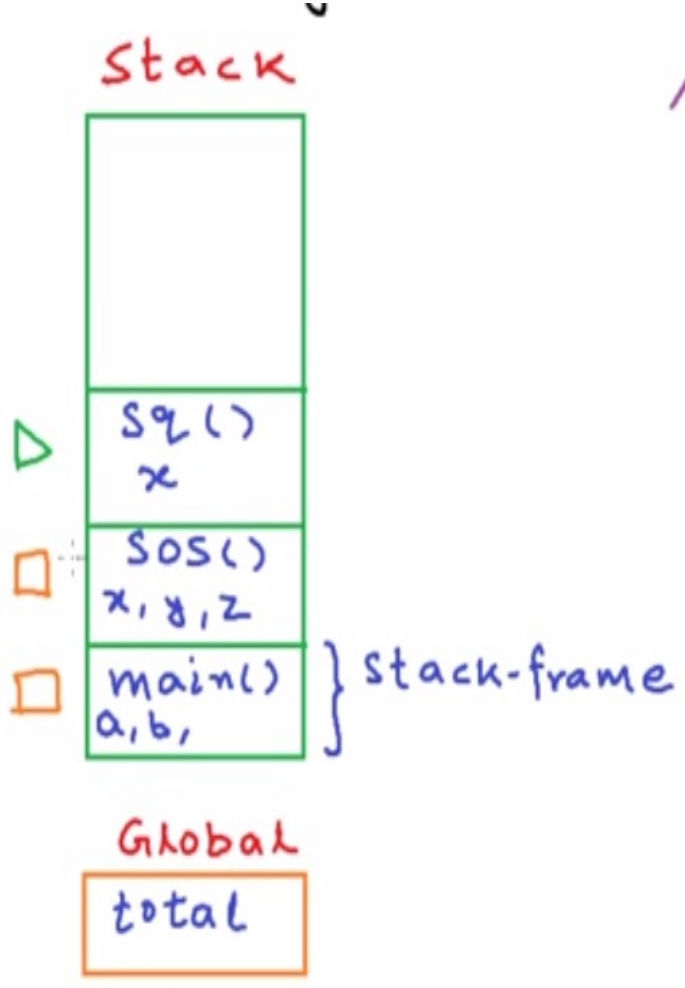
Pointers and dynamic memory

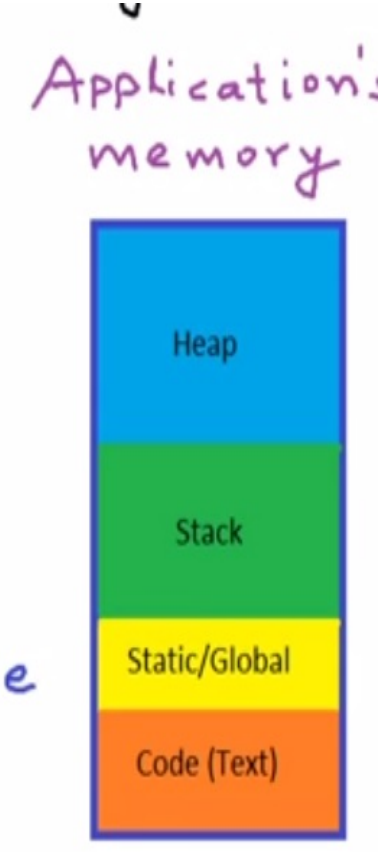
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z; // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



- *The local variables live in the function and they are stored until the function is executed.*
- *Global variables can be accessed anytime.*
- *Next: we see how the memory is allocated during this program's execution.*

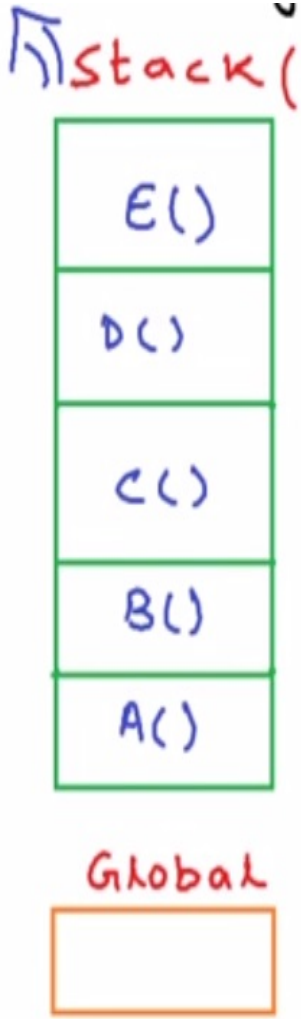
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



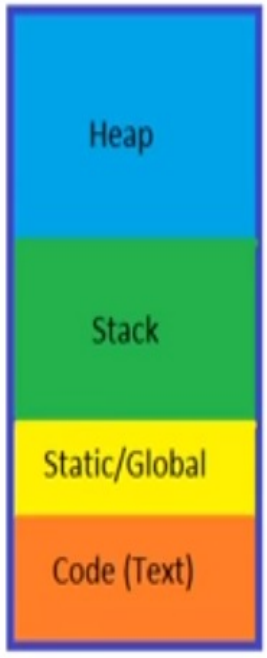


- *At any time of the program, the function at the top of the stack is executing and the rest are paused.*
- *As soon as Square function returns, it is removed from stack.*
- *Then main will call printf and printf will come to the top of the stack.*

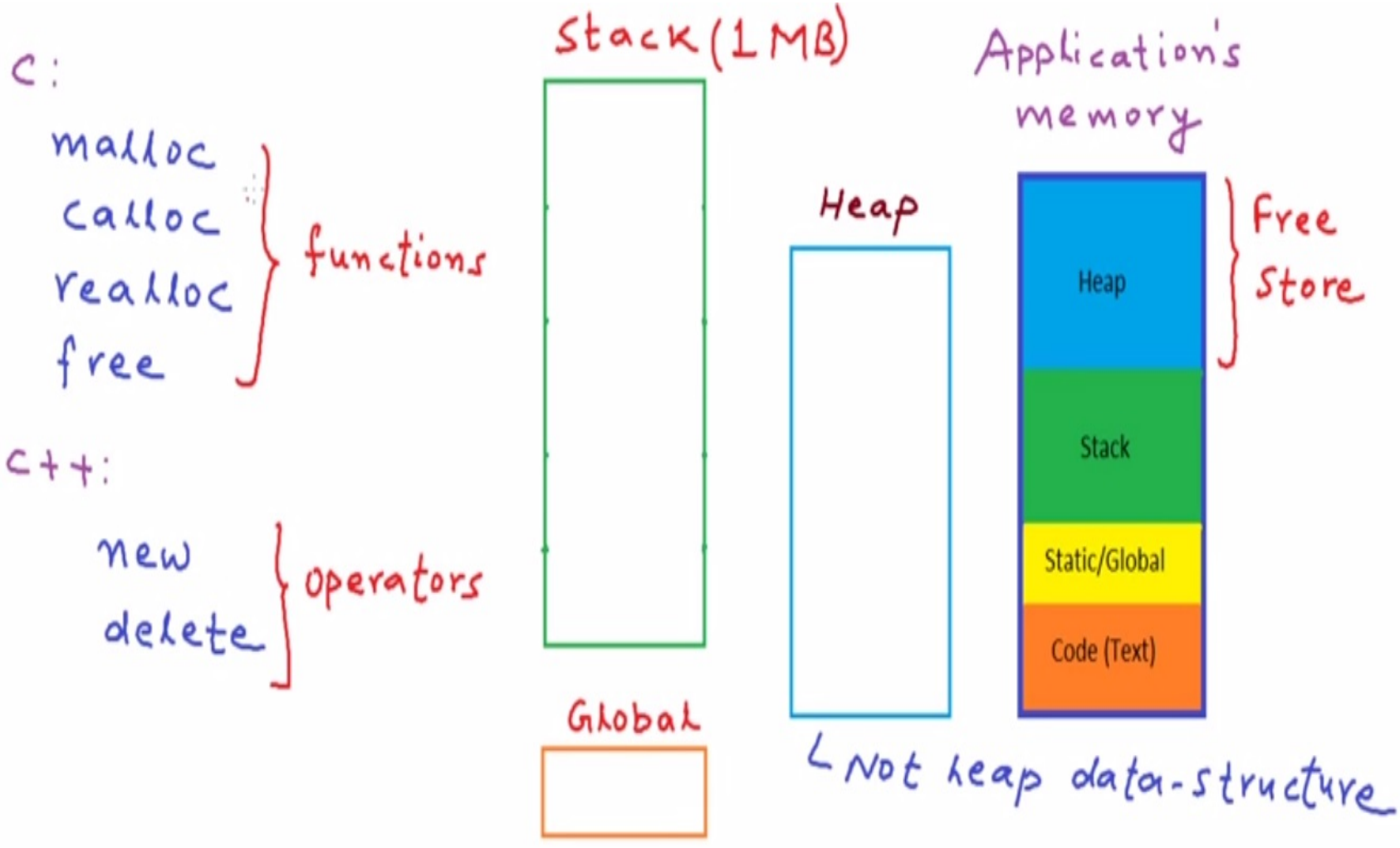
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x; // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z; // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



Stackoverflow

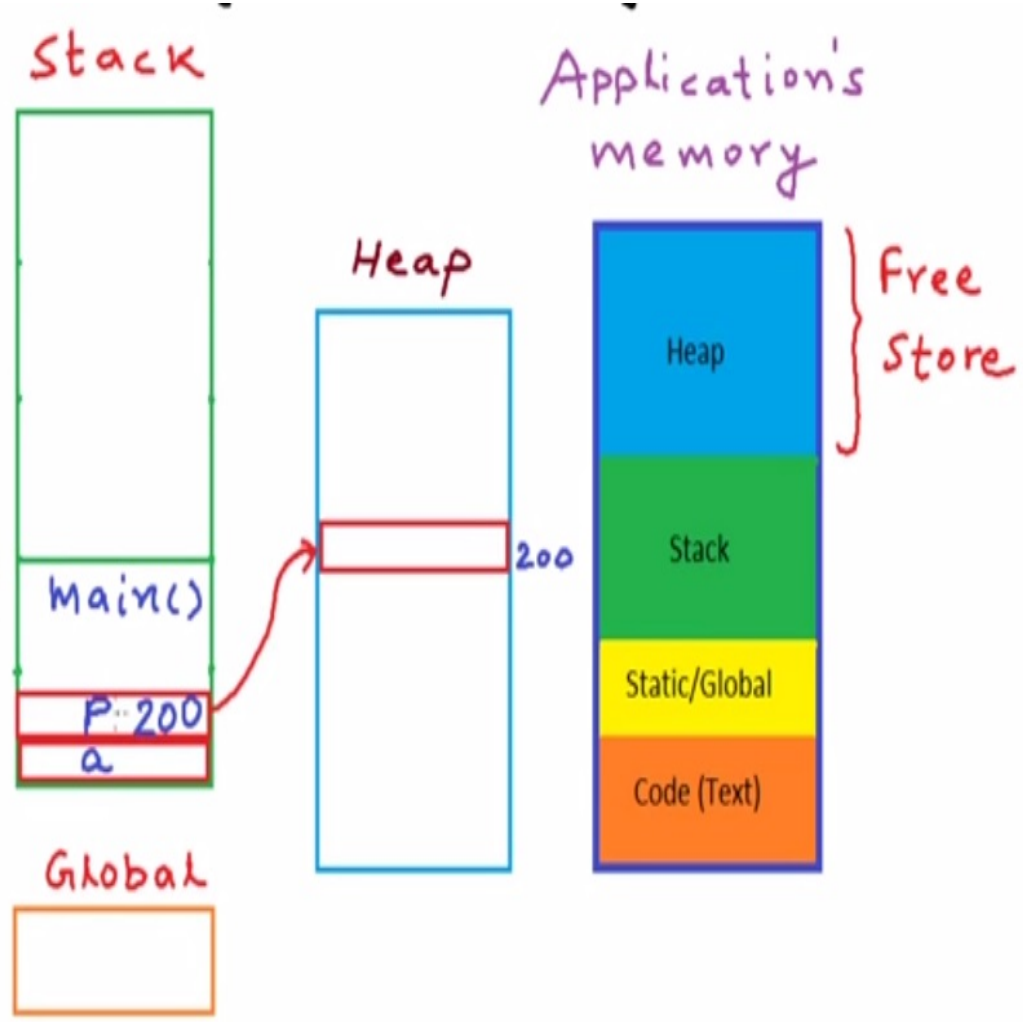


- *If, as here, A calls B, B calls C, C calls D, and so on and we exhaust all the space in stack, we see stack overflow and our program will crash. (Ex: infinite recursion).Since application cannot request more memory for stack, we use heap.*
- *It is possible to control how much to use from heap. Heap is also called dynamic memory. Using the heap is called dynamic memory allocation.*



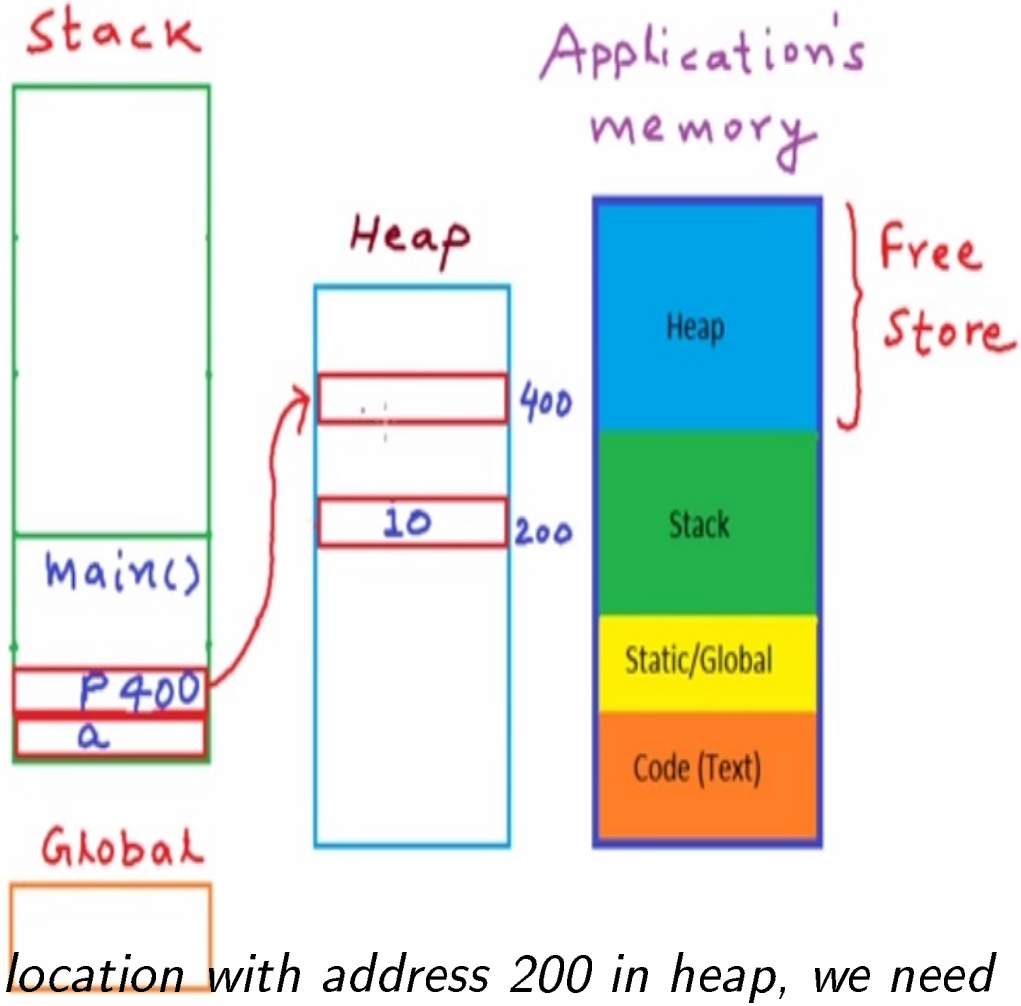
- *We will see examples applying these C functions and C++ operators.*

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

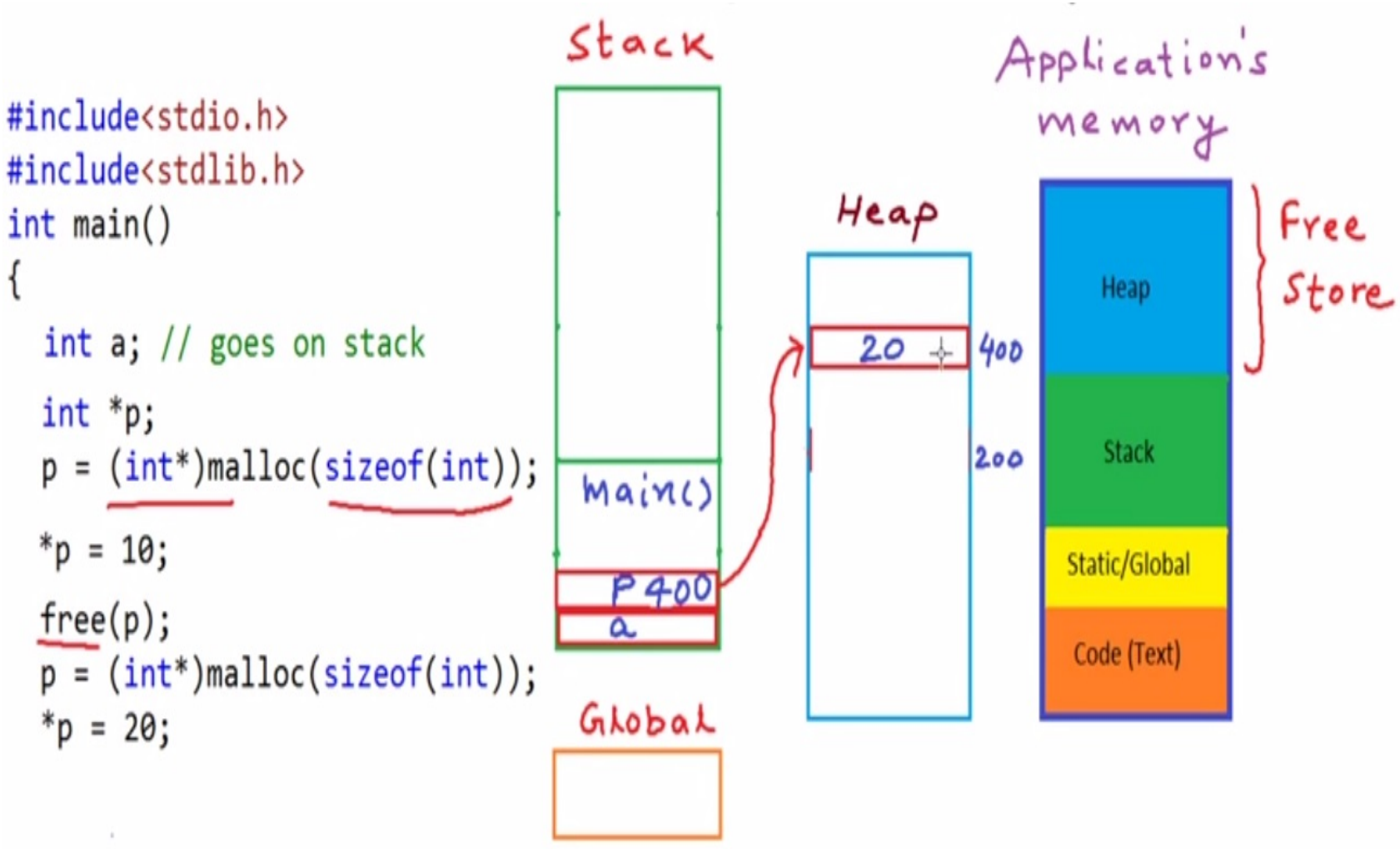


- *Malloc function asks for how much memory to allocate in the heap. Above one block of 4 bytes is reserved for p.*
- *Malloc will return a void pointer to the starting address of this block. Since starting address of this block is 200, malloc will return us 200. p stores the address 200.*

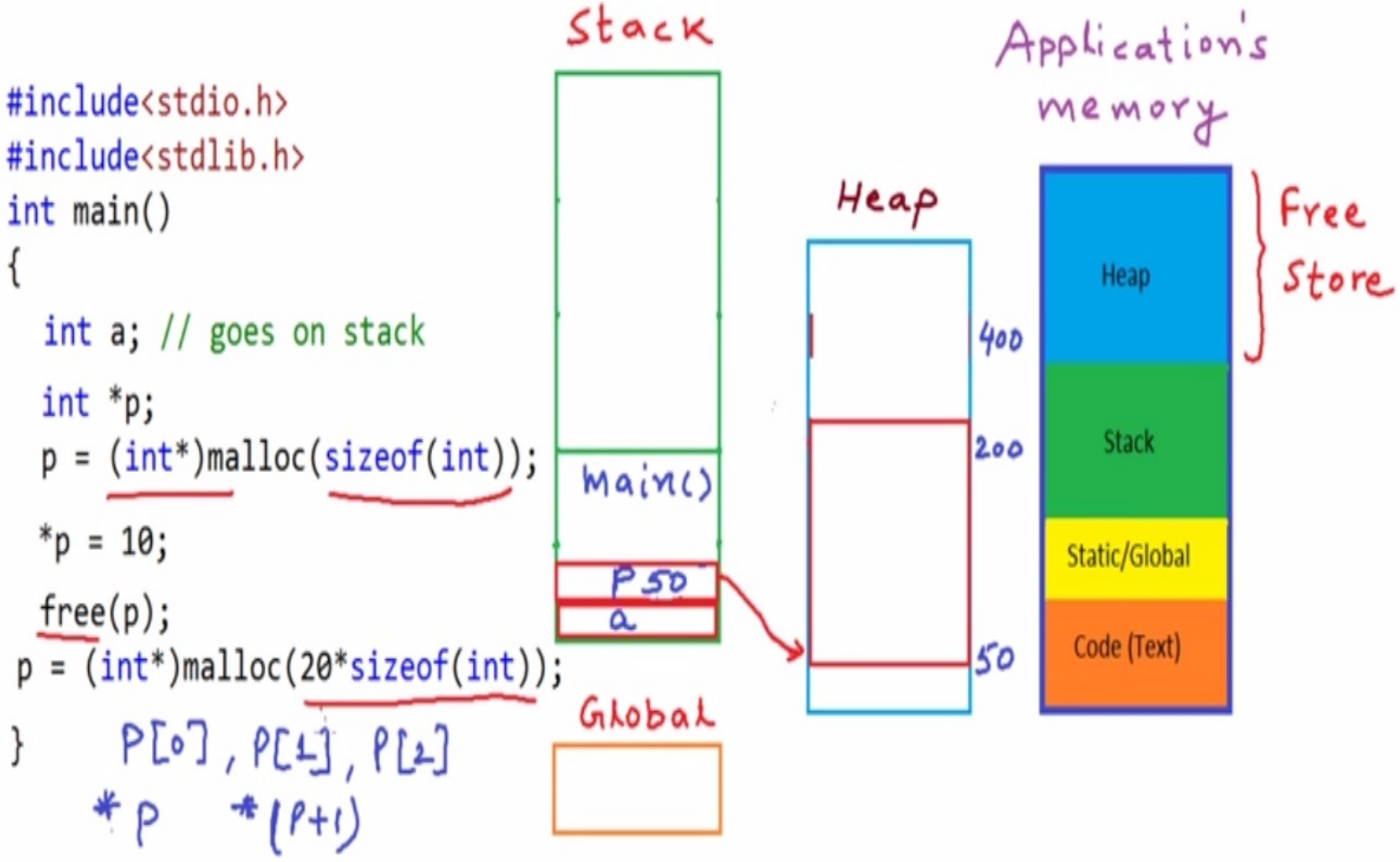
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int a; // goes on stack
  int *p;
  p = (int*)malloc(sizeof(int));
  *p = 10;
  p = (int*)malloc(sizeof(int));
  *p = 20;
```



- *If we want to fill in the location with address 200 in heap, we need to dereference the pointer p. So, the only way to fill the memory in heap is through reference.*
- *In the second call to malloc, one more block of 4 bytes is allocated in the heap with address 400 and p is now pointing to the address 400.*
- *The previous block with memory 200 is still kept and it is not cleared automatically. So, if we are not using it, we need to clear it.*

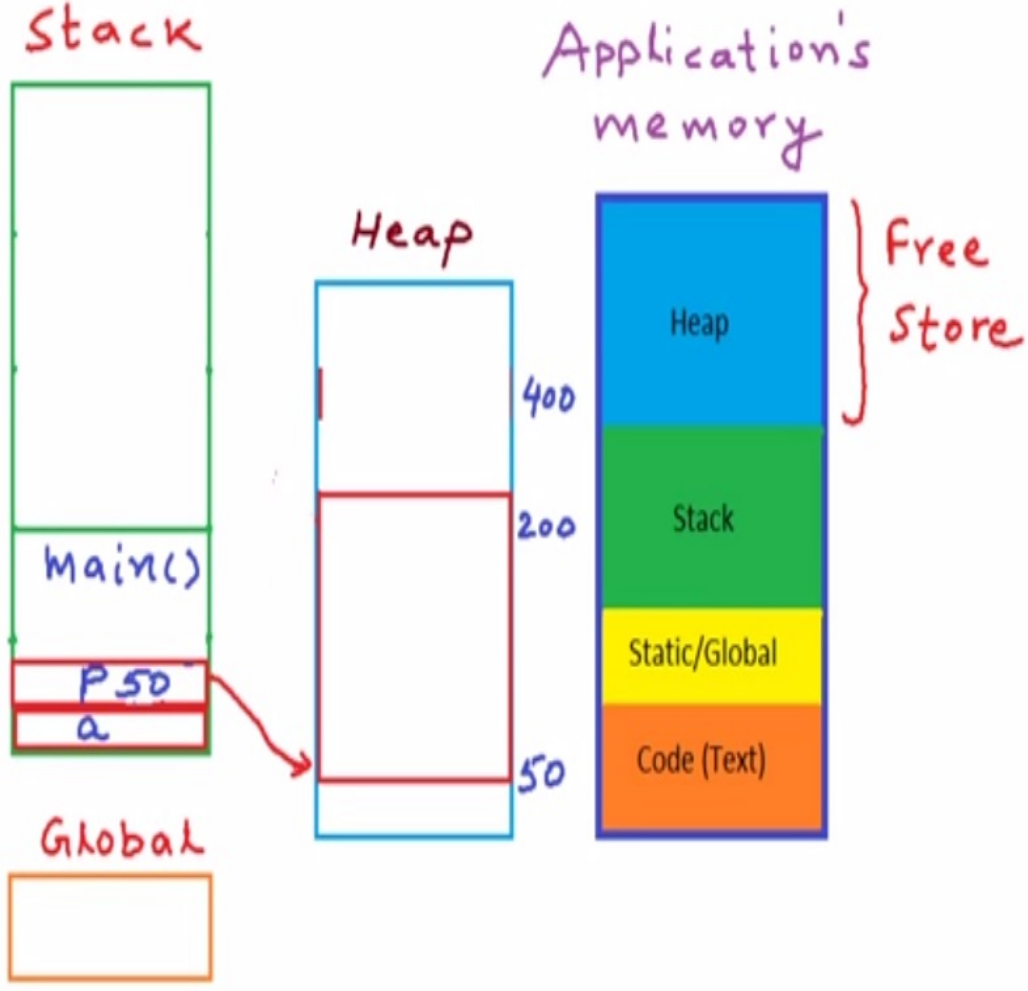


- *Once we are done using a particular memory, we should call the function free.*



- *Above, we reserve a block of 80 bytes from 50 to 130 in the heap.*
- *p will point to the base address of this block, which is 50. Above, p[0] is the value of p[0], which is same as *p.*
- *If no memory can be allocated in the heap, p returns NULL.*

```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;
}
```



- *In C++, instead of the functions malloc and free, we use the operators new and delete.*
- *As seen, the size of the array allocated 20 is put in brackets([20]). Again [] is used for deleting arrays of fixed size.*
- *Here(using C++), p will return an integer only unlike for C.*

malloc, calloc, realloc, free

Allocate block of memory

malloc -

calloc

realloc

deallocate block of memory

free

Malloc, Calloc, realloc, free

Allocate block of memory

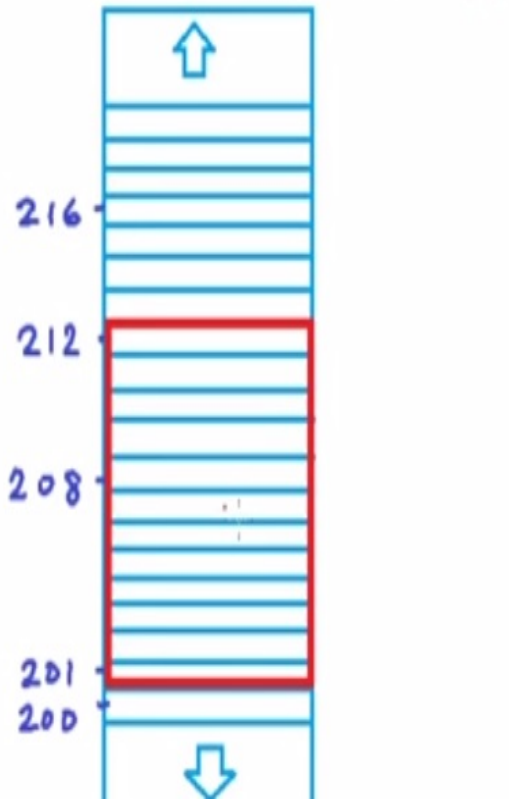
malloc - void* malloc(size_t size)

unsigned int

```
void *p = malloc( 3 * sizeof(int))
Print p  // 208
```

no. of elements

size of one unit



- *Malloc returns a void pointer.*
- *We use malloc as above if we need the memory for 3 integers.*
- *The data size should be input to malloc as above, number of data units times size of data unit.*
- *The starting address should be that p is pointing 201 according to the figure.*

Malloc, Calloc, realloc, free

Allocate block of memory

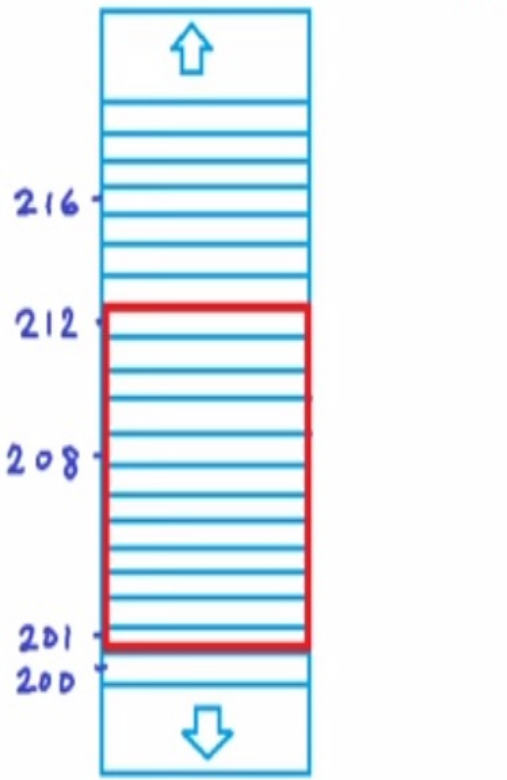
malloc - void* malloc(size_t size)

unsigned int

```
void *p = malloc(3*sizeof(int))
Print p  // 201
*p = 2 X
```

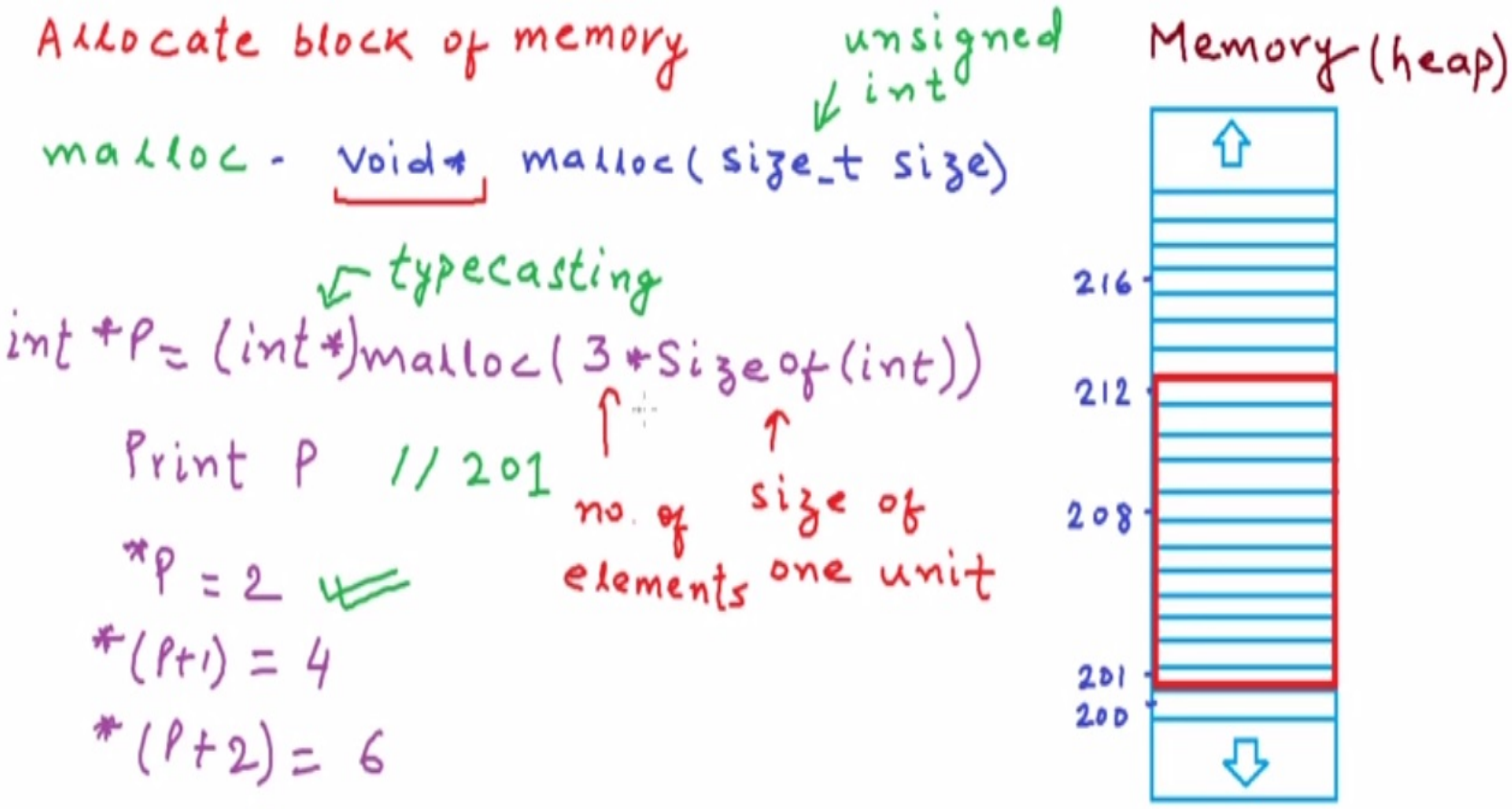
no. of elements

size of one unit



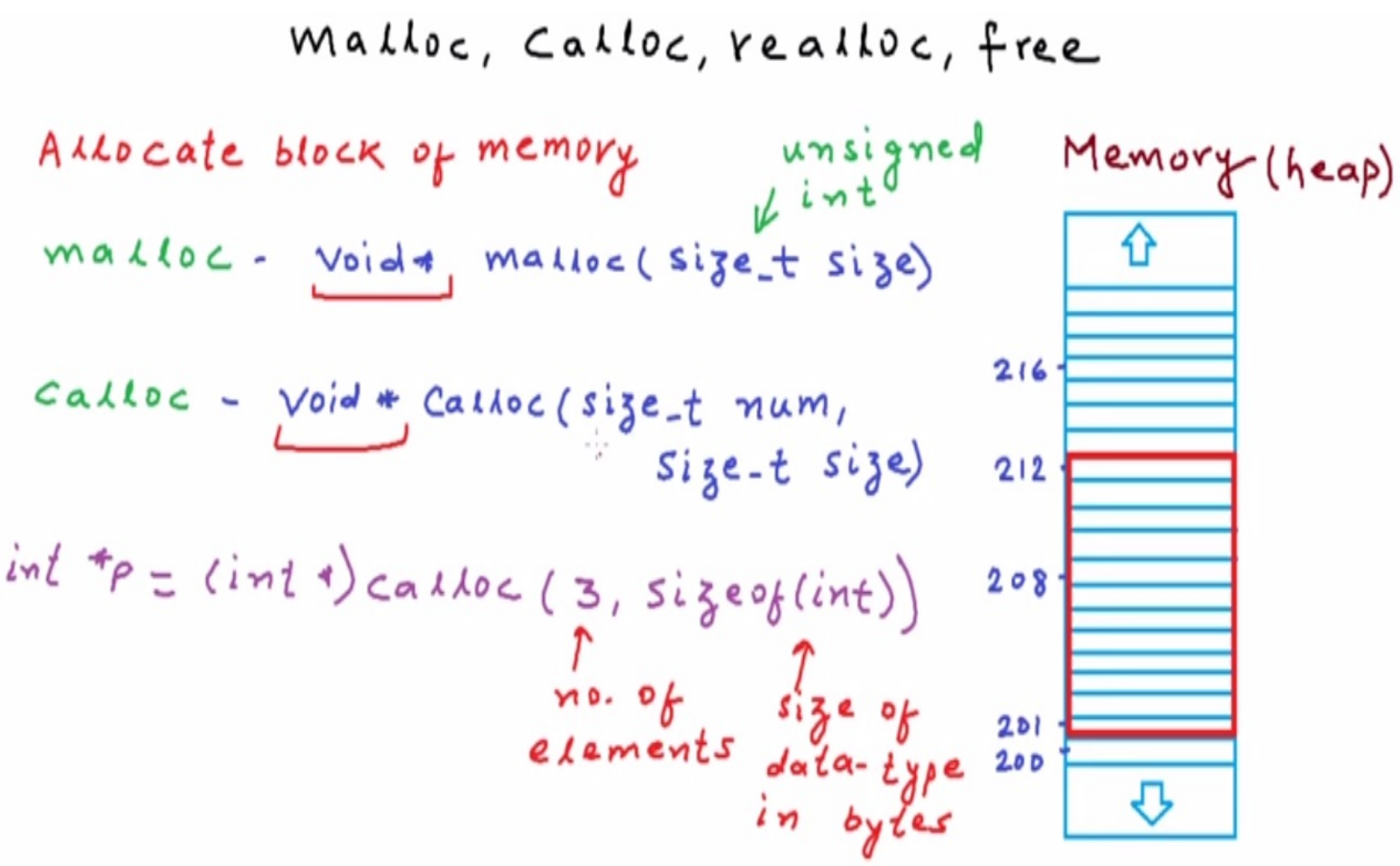
- *Malloc returns a void pointer. Therefore, we cannot dereference a void pointer.*
- *Malloc does not care if you are storing an integer or character or any other data type.*
- *To be able to dereference the pointer p, we need to determine the data type it is pointing, see next.*

malloc, calloc, realloc, free



- *Typecasting of the void pointer p is done above. Now, we can dereference p and (p+1) and (p+2).*
- *Also we can dereference p[0] and p[1] and p[2].*

Calloc: same job as Malloc



- *Takes two arguments instead of one as shown above.*
- *Malloc does not initialize the value of the data stored. Calloc initially sets these values to 0.*

Realloc: To change size of allocated memory

malloc, calloc, realloc, free

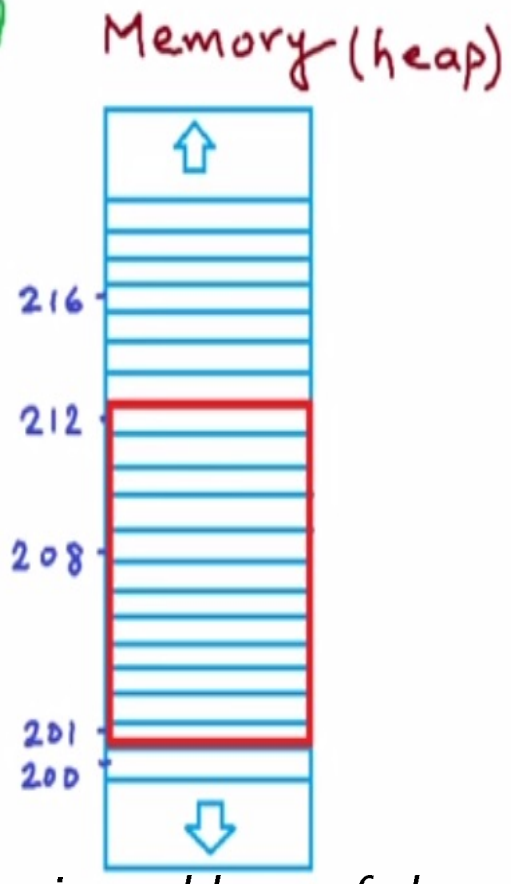
Allocate block of memory

unsigned int

malloc - void* malloc(size_t size)

calloc - void* calloc(size_t num, size_t size)

realloc - void* realloc(void* ptr, size_t size)



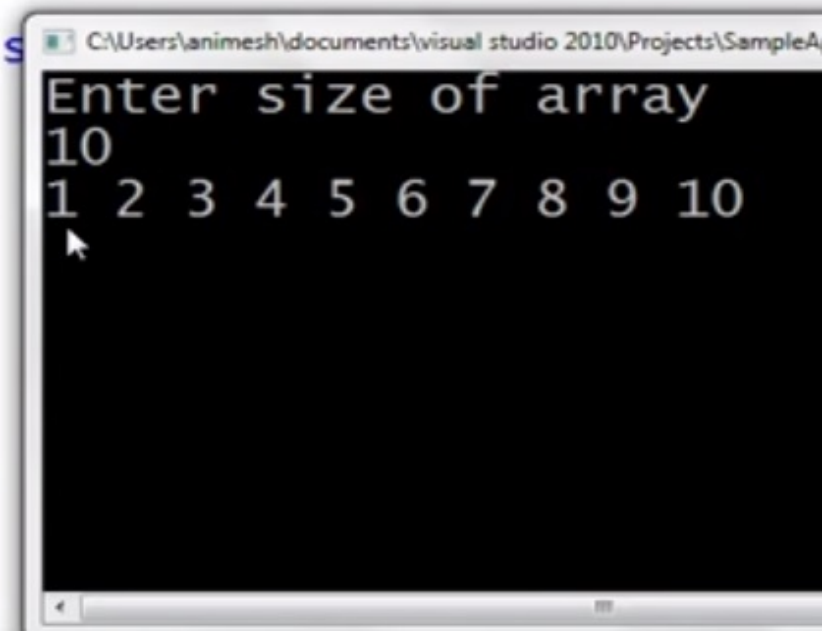
- *Realloc has two arguments: pointer to the starting address of the existing block and the size of the new block.*
- *If the new block has bigger size, then the machine copies the entire data in the old block to the newly created block.*

Application (See next slide for Output)

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*sizeof(int)); //dynamically allocated array
    for(int i = 0;i<n;i++)
    {
        A[i] = i+1;
    }
    for(int i =0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
```

Output

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*s
    for(int i = 0;i<n;i++)
    {
        A[i] = i+1;
    }
    for(int i =0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```



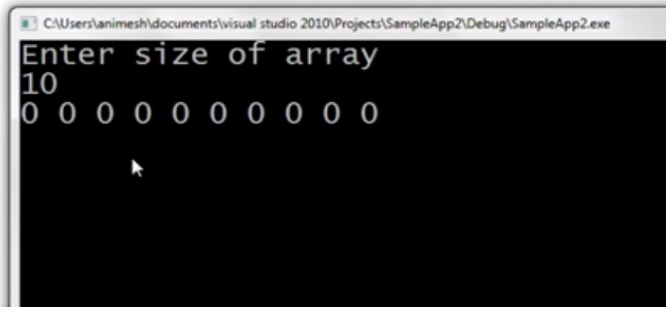
“Calloc” instead of “Malloc”

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)calloc(n,sizeof(int)); //dynamically allocated array
    for(int i = 0;i<n;i++)
    {
        A[i] = i+1;
    }
    for(int i =0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```

Initialization by malloc and calloc

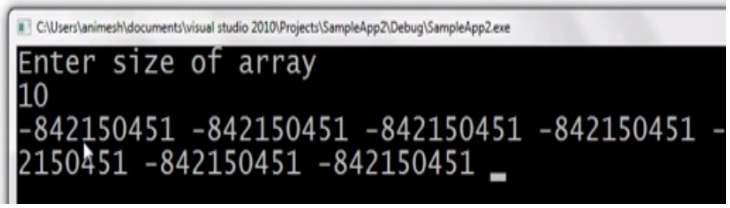
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)calloc(n,sizeof(int)); //dynamically allocated array

    for(int i =0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```



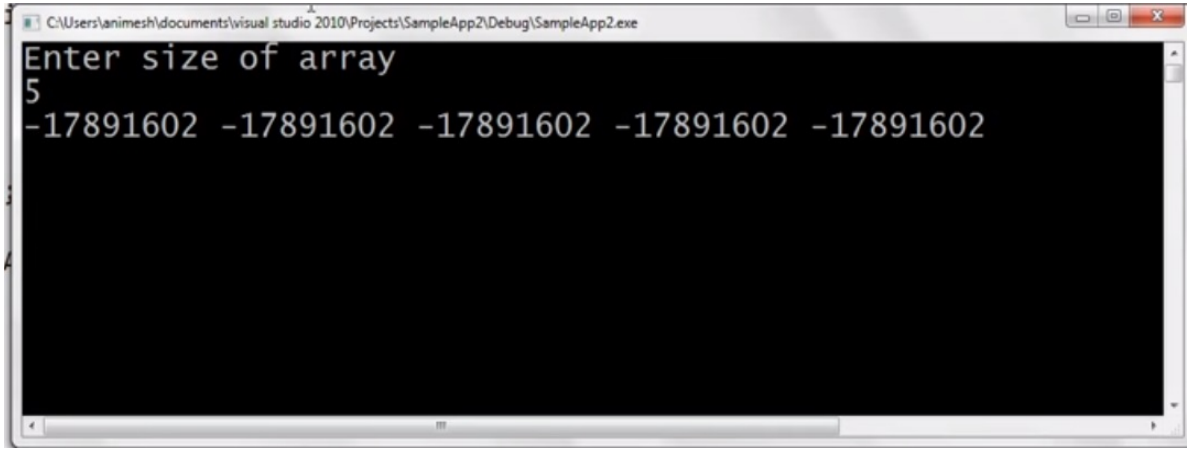

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*sizeof(int)); //dynamically allocated array

    for(int i =0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```



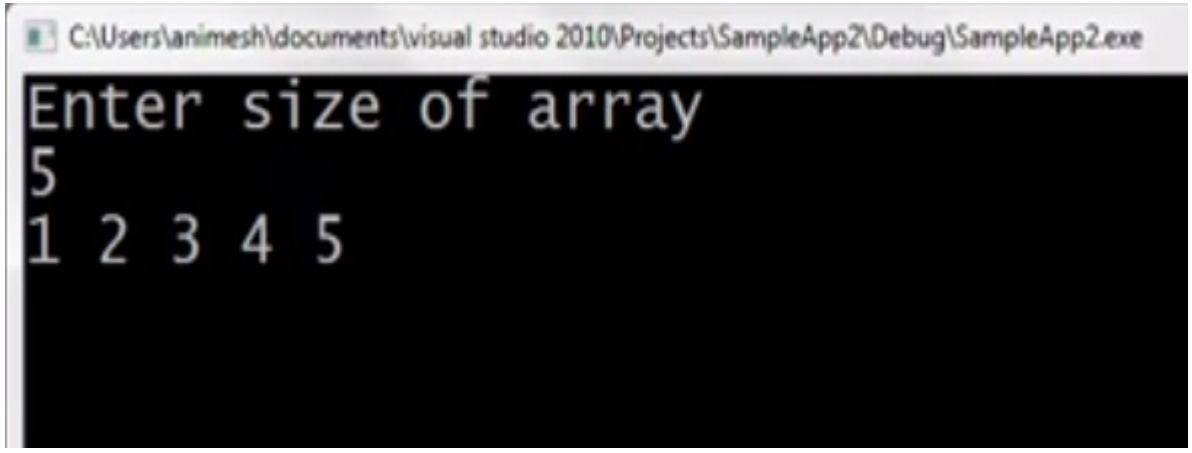
Deallocation using “free” function

```c
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*sizeof(int)); //dynamically allocated array
    for(int i =0;i<n;i++)
    {
        A[i] = i+1;
    }
    free(A);
    for(int i = 0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```



Deallocation using “free” function

```c
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*sizeof(int)); //dynamically allocated array
    for(int i =0;i<n;i++)
    {
        A[i] = i+1;
    }
    //free(A);
    for(int i = 0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```



```c
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*sizeof(int)); //dynamically allocated array
    for(int i =0;i<n;i++)
    {
        A[i] = i+1;
    }
    free(A);
    A[2] = 6;
    for(int i = 0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```

- *After using "free", we are able to modify the value of A[2]. But in some machines, this program may crash.*
- We should always use the memory that is allocated.