Introduction to pointers in C/C++

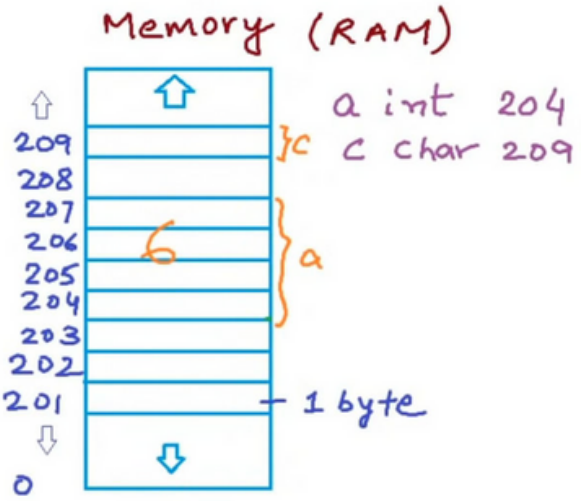
Introduction to pointers in C

int - 4 bytes

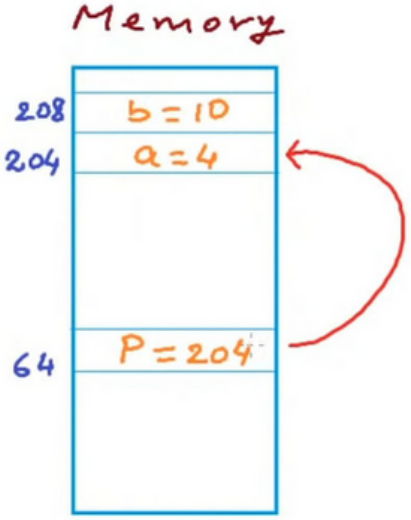
char - 1 byte

float - 4 bytes

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

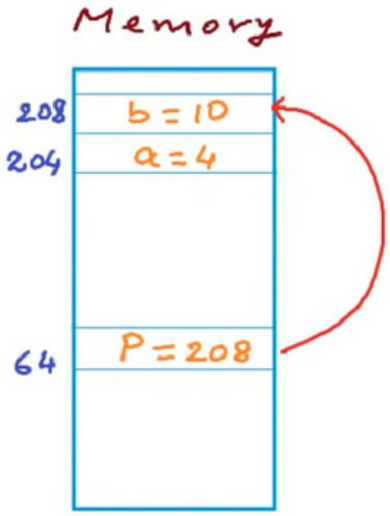


Pointers - variables that store address of another variable



Pointers - variables that store address of another variable

```
int a;
int *p;
p = &a;
```

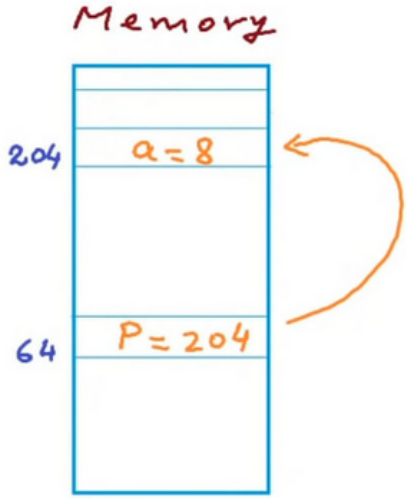


Pointers - variables that store address of another variable

p → address

*p → value at address

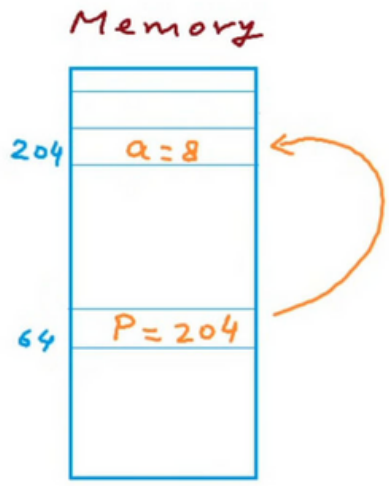
```
int a;
int *p;
⇒ p = &a;
a = 5;
⇒ Print p    // 204
  Print &a   // 204
  Print &p   // 64
⇒ print *p   // 5   ⇒ derefrencing
⇒ *p = 8
  Print a    // 8
```



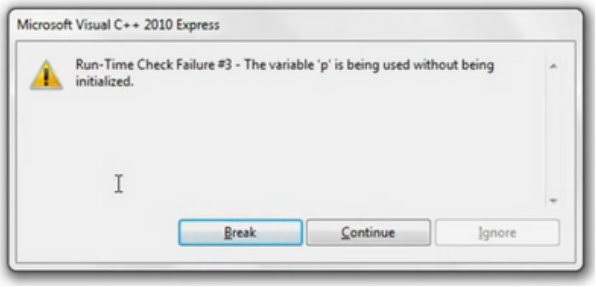
Working with pointers

Pointers – variables that store address of other variables

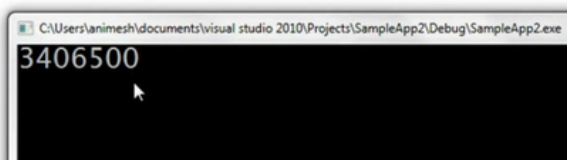
```
int a; //integer
int *p; //Pointer to integer
char c; // character
char *pc //pointer to character
double d  //double
double *p1 //Pointer to double
p = &a; a = 8; (*p) → 8
```




```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    printf("%d\n",p);
}
```



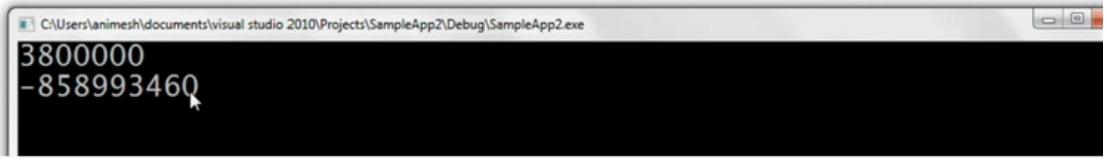

```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    p = &a; // &a = address of a
    printf("%d\n",p);
}
```



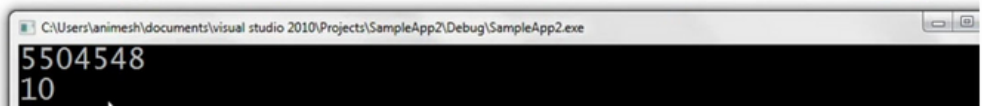
every time i run the code, new address is allocated

grabage value for a or *p because I don't initialize it

```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    p = &a; // &a = address of a
    printf("%d\n",p);
    printf("%d\n",*p); // *p- value at adress pointed by p
}
```




```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    a = 10;
    p = &a; // &a = address of a
    printf("%d\n",p);
    printf("%d\n",*p); // *p- value at adress pointed by p
}
```



```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    a = 10;
    p = &a; // &a = address of a
    printf("%d\n",p);
    printf("%d\n",*p); // *p - value at address pointed by p
    printf("%d\n",&a);
}
```

```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
3077992
10
3077992
```

```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    a = 10;
    p = &a; // &a = address of a
    printf("a = %d\n",a);
    *p = 12; // dereferencing
    printf("a = %d\n",a);
}
```

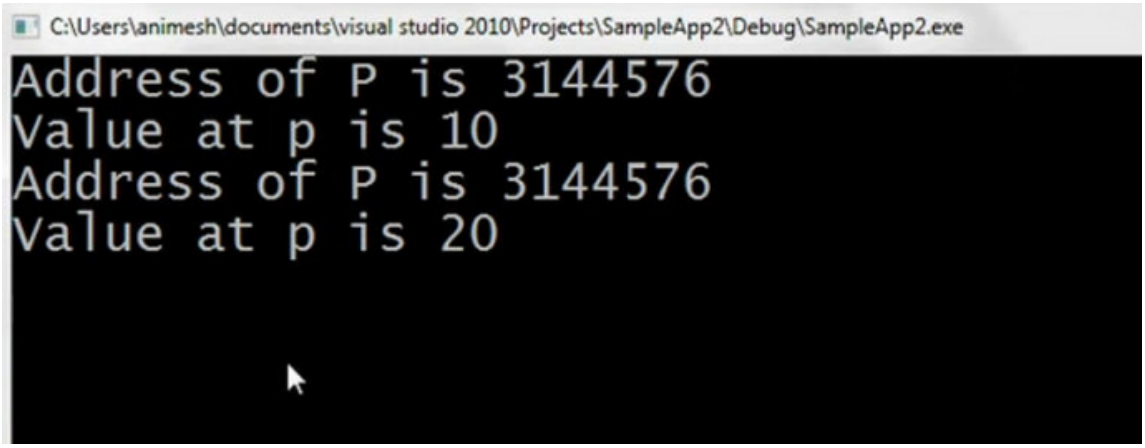
```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug
a = 10
a = 12
```

Will the address in p change to point p?

- No

```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    a = 10;
    p = &a; // &a = address of a
    int b = 20;
    *p = b; // Will the address in p change to point b??
    p = &b;
}
```

```c
#include<stdio.h>
int main()
{
    int a;  int *p;
    a = 10;
    p = &a; // &a = address of a
    printf("Address of P is %d\n",p);
    printf("Value at p is %d\n",*p);
    int b = 20;
    *p = b; // Will the address in p change to point b
    printf("Address of P is %d\n",p);
    printf("Value at p is %d\n",*p);
}
```



```c
#include<stdio.h>
int main()
{
    int a;
    int *p;
    a = 10;
    p = &a; // &a = address of a
}
```

```c
#include<stdio.h>
int main()
{
    int a = 10;
    int *p  = &a;
    // p = &a; // &a = address of a
}
```

```c
#include<stdio.h>
int main()
{
    int a = 10;
    int *p;
    p = &a;
    // Pointer arithmetic
    printf("%d\n",p); // p is 2002
    printf("%d\n",p+1); // p+1 is 2006
}
```

```c
#include<stdio.h>
int main()
{
    int a = 10;
    int *p;
    p = &a;
    // Pointer arithmetic
    printf("Address p is %d\n",p); // p is 2002
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address p+1 is %d\n",p+1); // p+1 is 2006
}
```

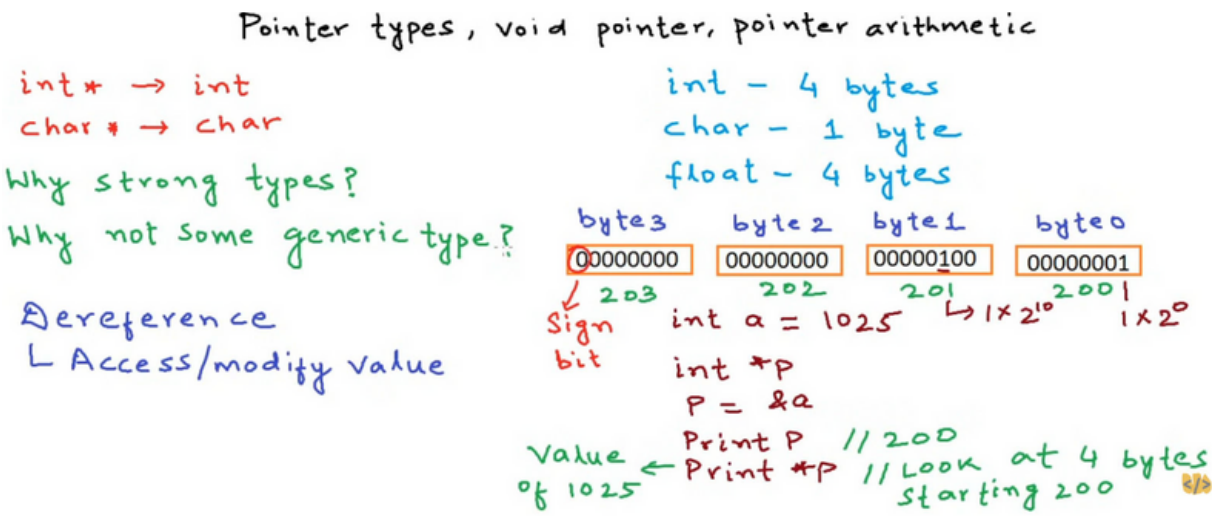
```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
Address p is 4586052
size of integer is 4 bytes
Address p+1 is 4586056
```

```c
#include<stdio.h>
int main()
{
    int a = 10;
    int *p;
    p = &a;
    // Pointer arithmetic
    printf("Address p is %d\n",p);
    printf("value at address p is %d\n",*p);
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address p+1 is %d\n",p+1);
    printf("value at address p+1 is %d\n",*(p+1));
}
```

```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
Address p is 2750832
value at address p is 10
size of integer is 4 bytes
Address p+1 is 2750836
value at address p+1 is -858993460
```

Pointer types, pointer arithmetic, void pointers

We don't use the pointer variables only to stroe memory addresses, we also use them to dereference these addresses and modify the values in it.




```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address = %d, value = %d\n",p,*p);
}
```

```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
size of integer is 4 bytes
Address = 4192228, value = 1025
```

compilation error

```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address = %d, value = %d\n",p,*p);
    char *p0;
    p0 = p;
}
```

```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address = %d, value = %d\n",p,*p);
    char *p0;
    p0 = (char*)p; // typecasting
    printf("size of char is %d bytes\n",sizeof(char));
    printf("Address = %d, value = %d\n",p0,*p0);
}

    // 1025 = 00000000 00000000 00000100 00000001
```

```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
size of integer is 4 bytes
Address = 5373032, value = 1025
size of char is 1 bytes
Address = 5373032, value = 1
```

```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of int
    printf("Address =
    printf("Address = %d, value = %d\n",p+1,*(p+1));
    char *p0;
    p0 = (char*)p; // typecasting
    printf("size of char is %d bytes\n",sizeof(char));
    printf("Address = %d, value = %d\n",p0,*p0);
    printf("Address = %d, value = %d\n",p0+1,*(p0+1));
    // 1025 = 00000000 00000000 00000100 00000001
}
```

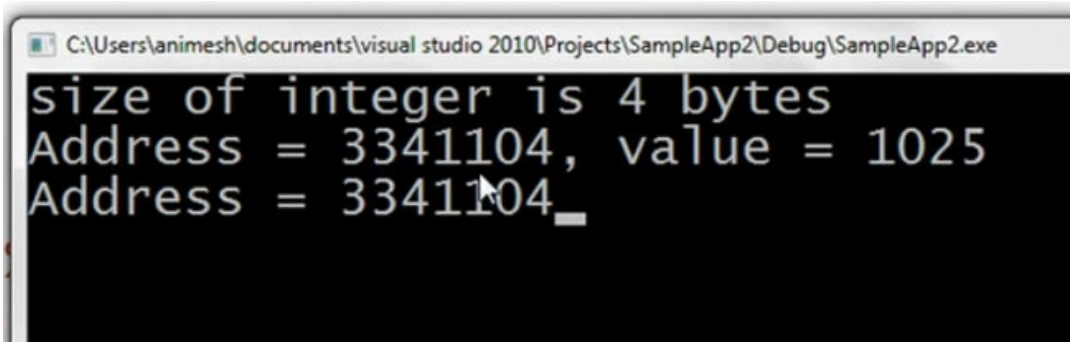
```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
size of integer is 4 bytes
Address = 4456036, value = 1025
Address = 4456040, value = -858993460
size of char is 1 bytes
Address = 4456036, value = 1
Address = 4456037, value = 4
```

Generic pointer

```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address = %d, value = %d\n",p,*p);
    // Void pointer - Genric pointer
    void *p0;
    p0 = p;
    printf("Address = %d, value = %d\n",p0,*p0);
}
```

Error: expression must be a pointer to a complete object type

We can only print the address

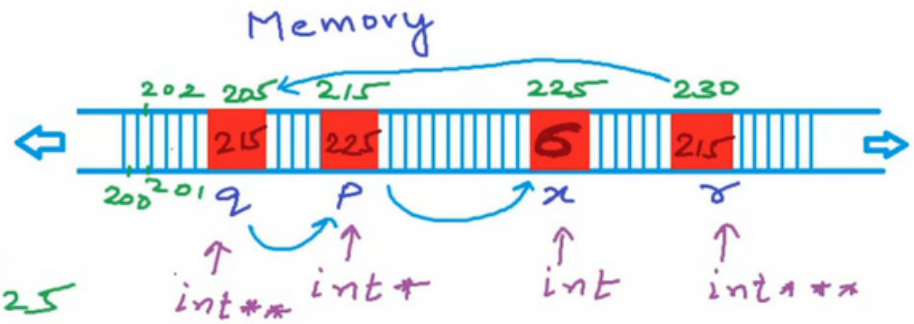


+1 is not also valid (compilation error)

```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address = %d, value = %d\n",p,*p);
    // Void pointer - Genric pointer
    void *p0;
    p0 = p;
    printf("Address = %d",p0);
    printf("Address = %d",p0+1);
}
```

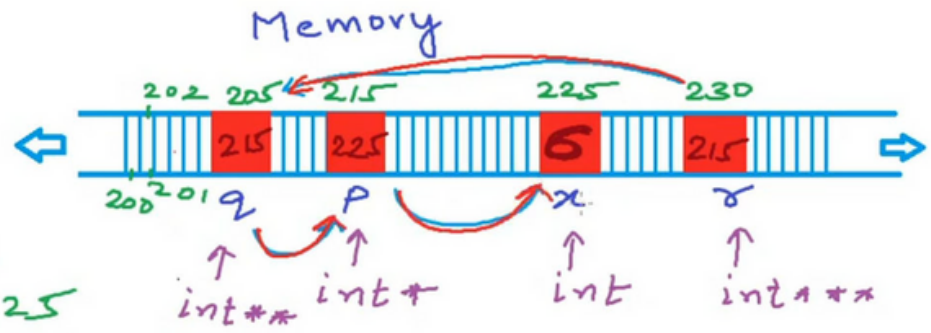
Pointers to Pointers in C/C++

```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
```

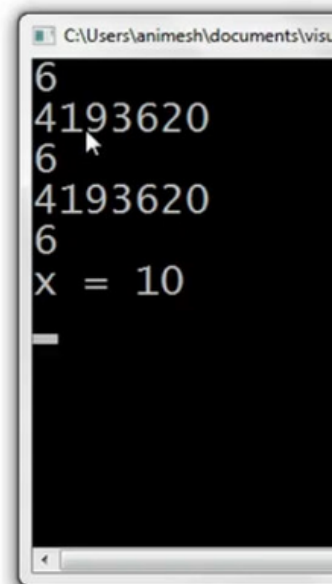


Pointer to pointer

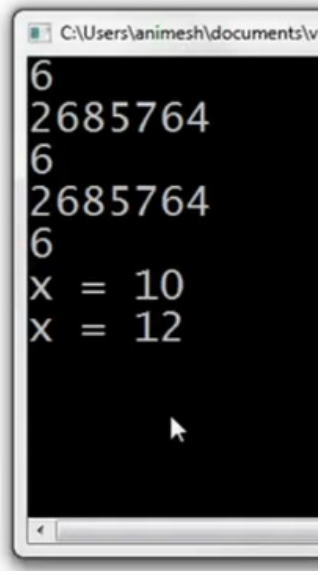
```c
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p); // 6
    printf("%d\n",*q); // 225
    printf("%d\n",*(*q)); // 6
    printf("%d\n", *(*r)) ; // 225
    printf("%d\n",*(*(*r))); // 6
}
```



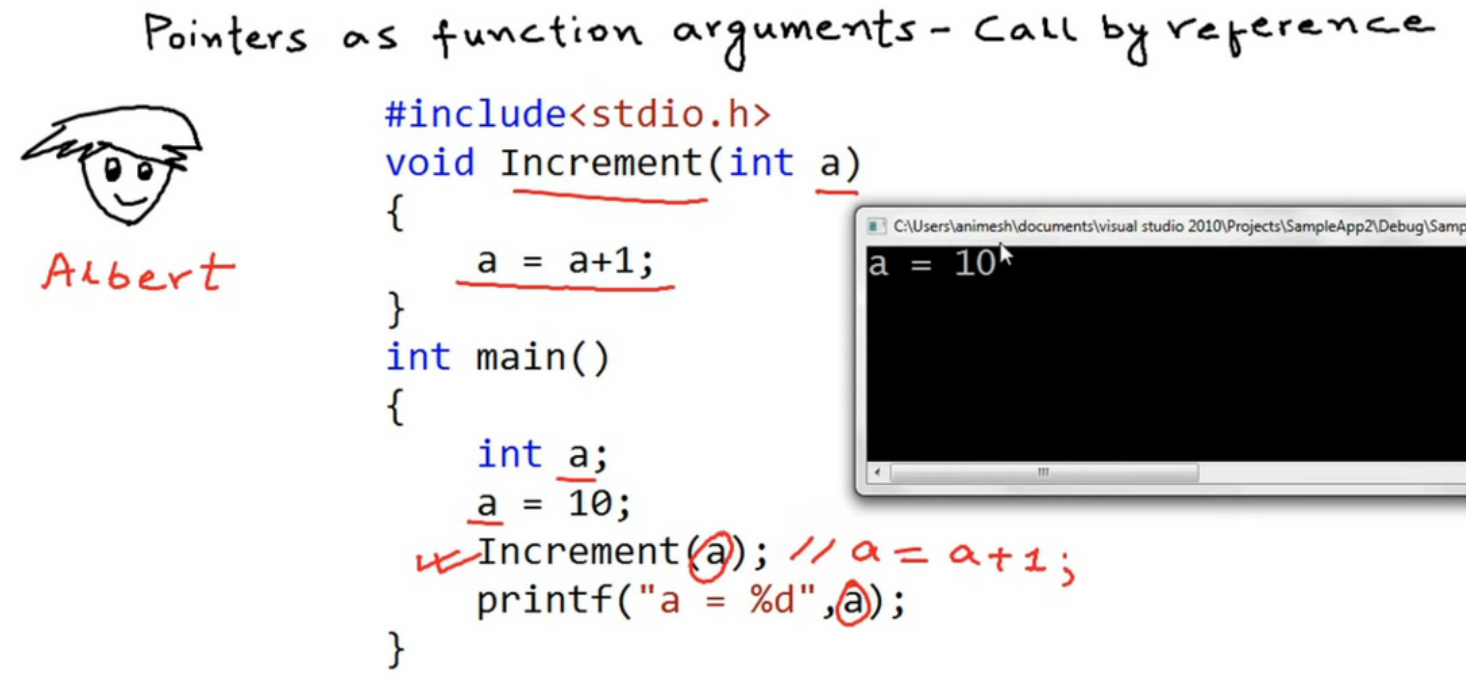

```c
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p);
    printf("%d\n",*q);
    printf("%d\n",**q);
    printf("%d\n",**r);
    printf("%d\n",***r);
    ***r = 10;
    printf("x = %d\n",x);
}
```



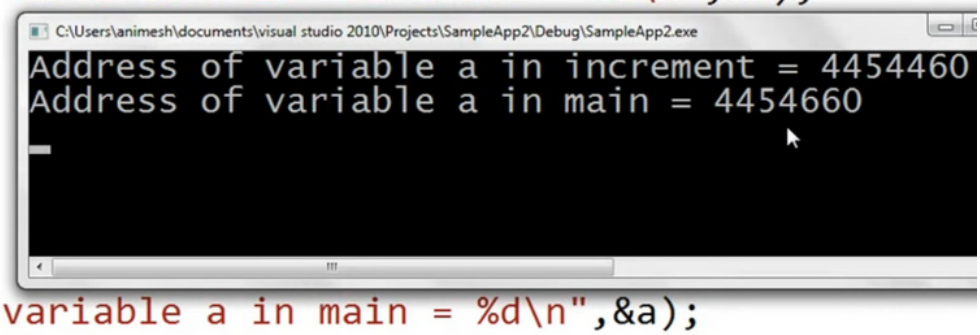
```c
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p);
    printf("%d\n",*q);
    printf("%d\n",**q);
    printf("%d\n",**r);
    printf("%d\n",***r);
    ***r = 10;
    printf("x = %d\n",x);
    **q = *p +2;
    printf("x = %d\n",x);
}
```



Pointers as function arguments - call by reference



```c
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
    printf("Address of variable a in increment = %d\n",&a);
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("Address of variable a in main = %d\n",&a);
    //printf("a = %d",a);
}
```



Global variables can be accessed and modified any where in the problemunlike local variables.

Code(text), Global/static and stack segments of allocated memory these three are fixed nd they are decided when the program starts executing

The application however can keep asking for more memory for its heap segment during its execution only.

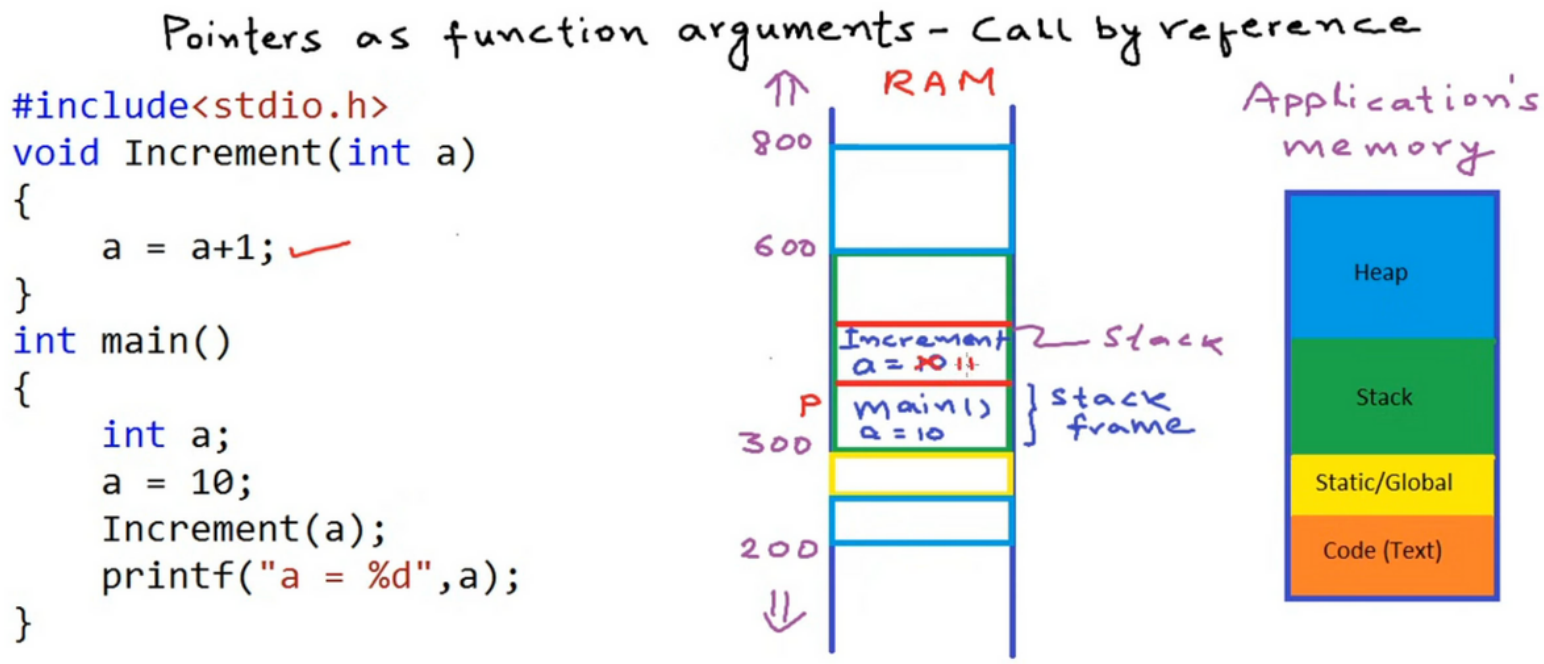
-

Let's say the memory allocated for our program if from address 200 to 800 (and these are the various segments of our application's memor)

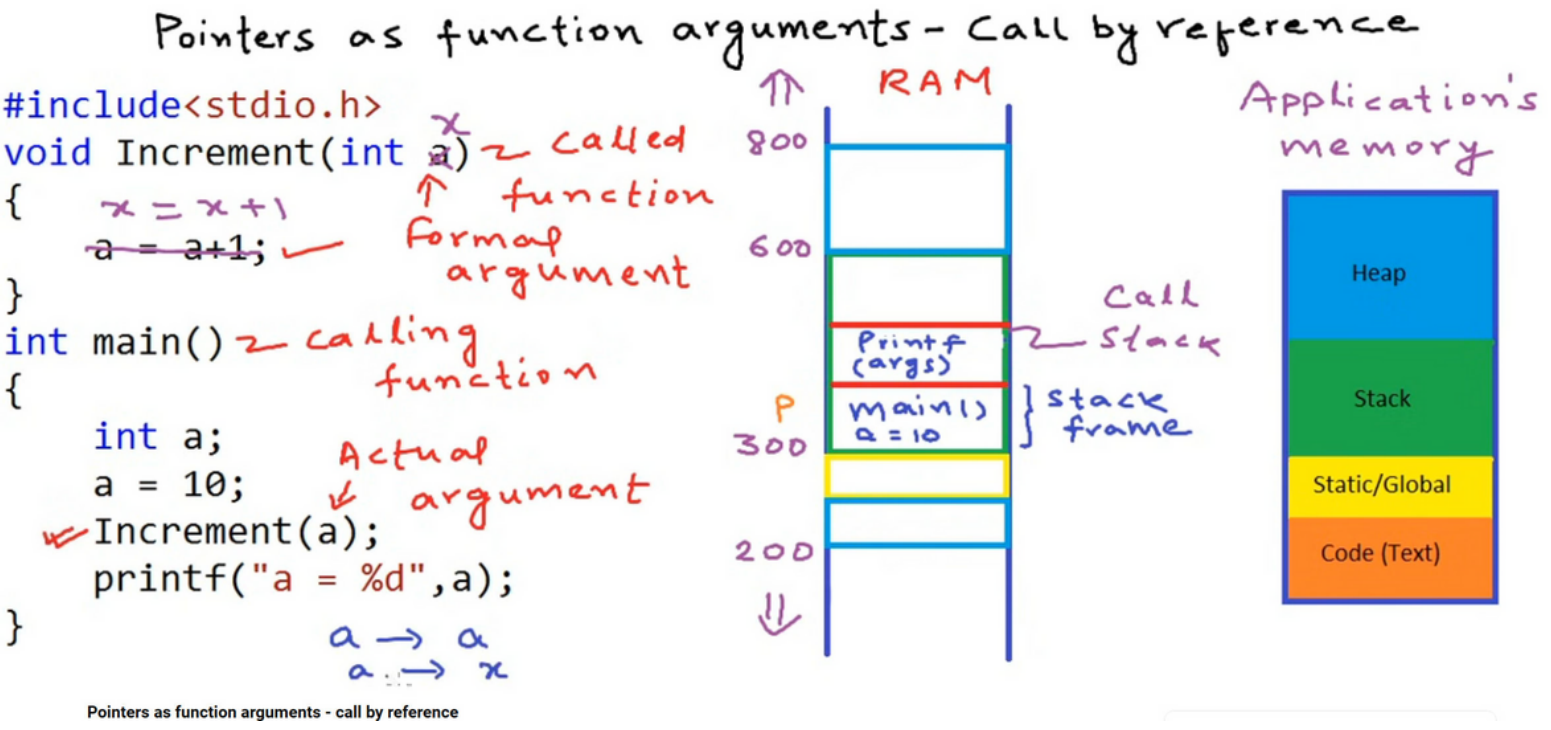
Ofcourse there is more memory in the RAM

When a function is invoked like when the program starts the main method is initially invoked, all the information bout the method call like its parameters, local variables, the calling function to which it should return, the current instruction at which is executing, all this information stored in the stack

each function will have a stack frame



and when increment finishes, the control returns to main method and what the machine does it clears the stack frame that was allocated for increment method (main method was paused during increment)



Pointers as function arguments - call by reference

Pointers as function arguments - Call by reference

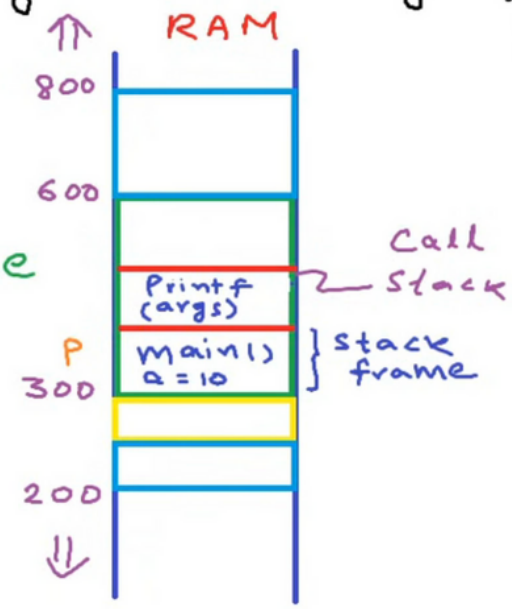
```c
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("a = %d",a);
}
```

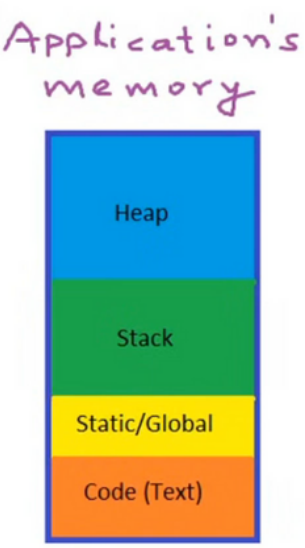
x

x = x + 1

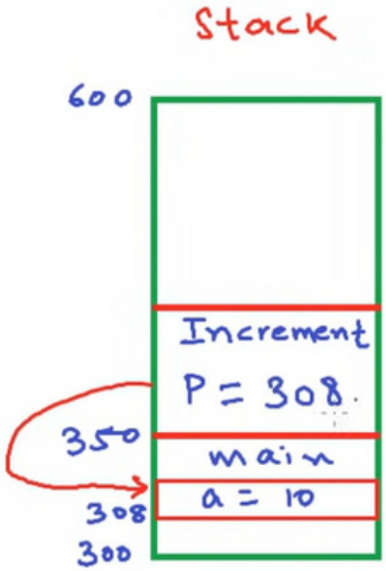
a → x

Call by value

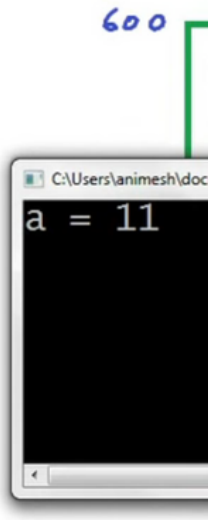




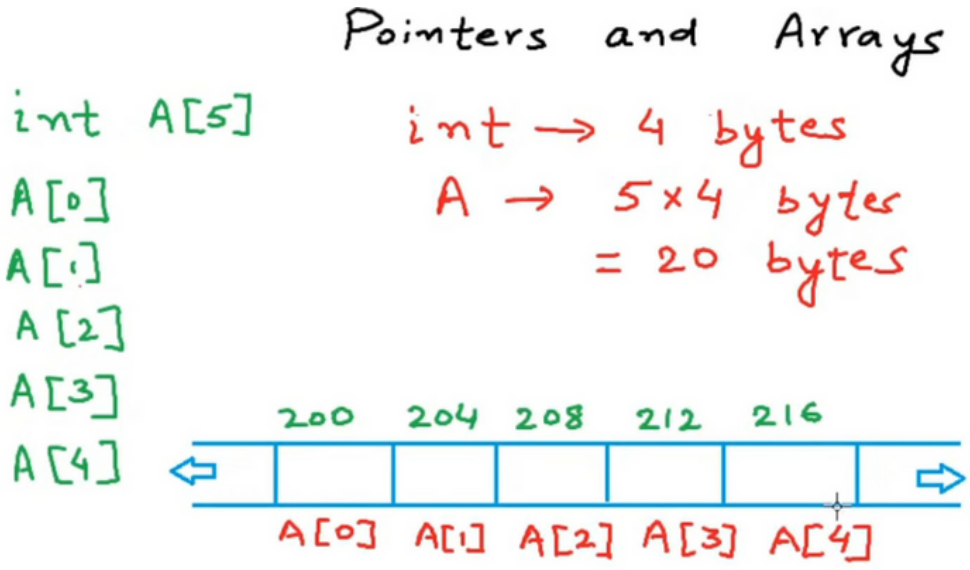

```c
#include<stdio.h>
void Increment(int *p)
{
    *p = (*p) + 1;
}
int main()
{
    int a;
    a = 10;
    Increment(&a);
    printf("a = %d",a);
}
```

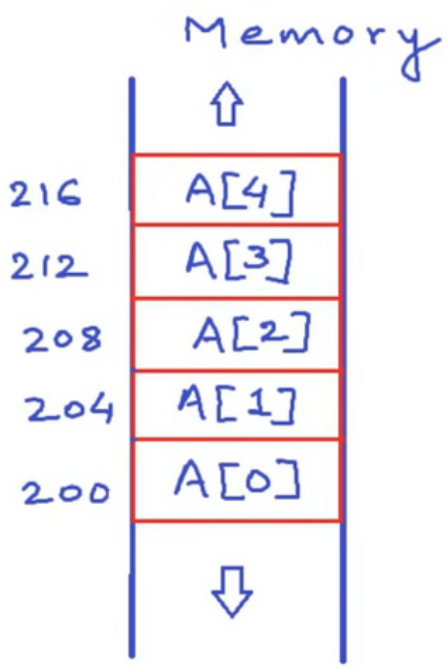


```c
#include<stdio.h>
void Increment(int *p)
{
    *p = (*p) + 1;
}
int main()
{
    int a;
    a = 10;
    Increment(&a);
    printf("a = %d",a);
}
```



Pointers and arrays





Pointers and Arrays

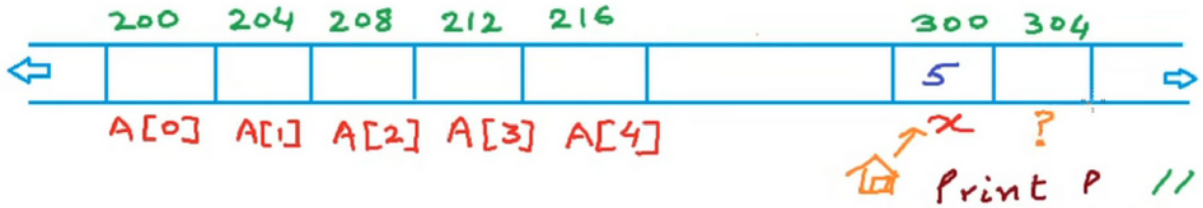
int A[5]

A[0]
A[1]
A[2]
A[3]
A[4]

int → 4 bytes

A → 5 × 4 bytes = 20 bytes

```
int x = 5
int *p
P = &x
Print P   // 300
Print *P  // 5
P = P + 1 // 304
```




```
Print P  // 304
Print *P
```

Pointers and Arrays

int A[5]

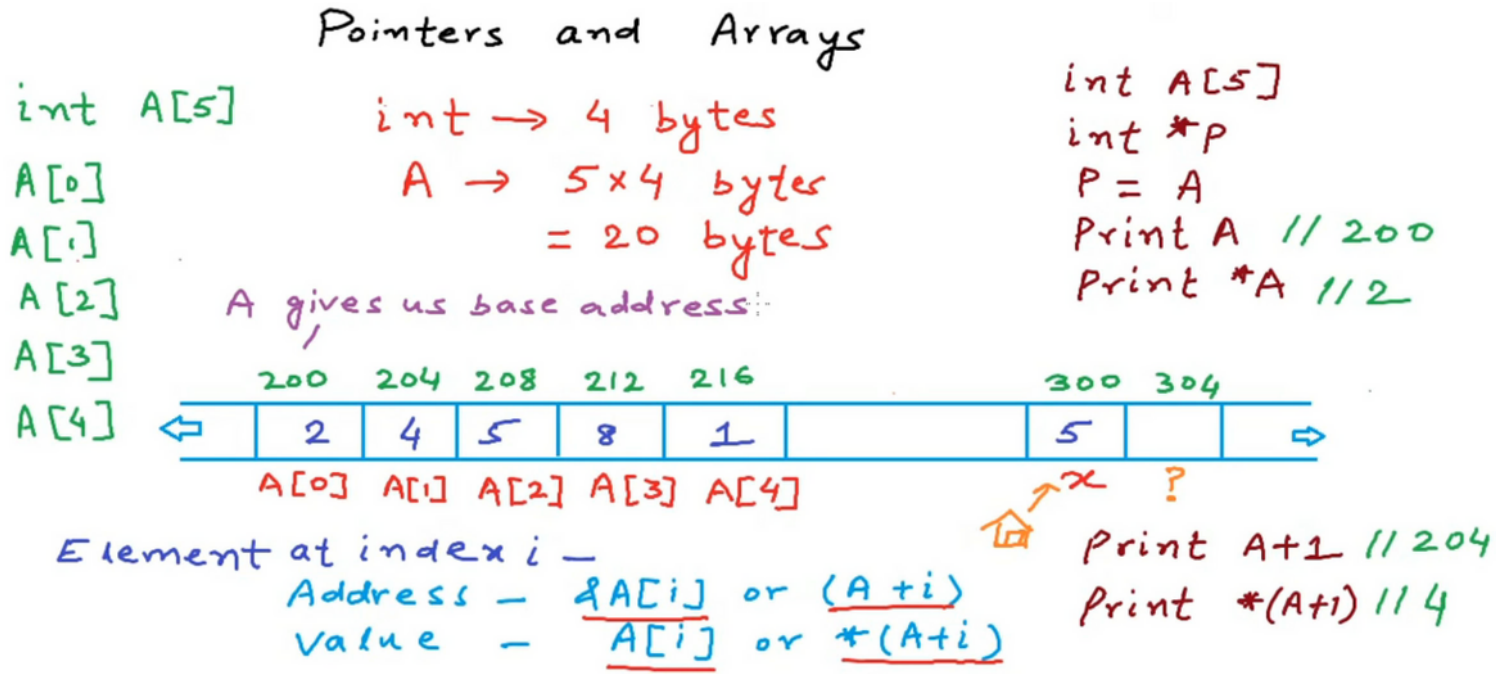
A[0]
A[1]
A[2]
A[3]
A[4]

int → 4 bytes

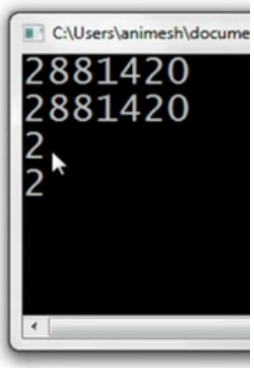
A → 5 × 4 bytes = 20 bytes

```
int A[5]
int *p
P = &A[0]
Print P   // 200
Print *P  // 2
```

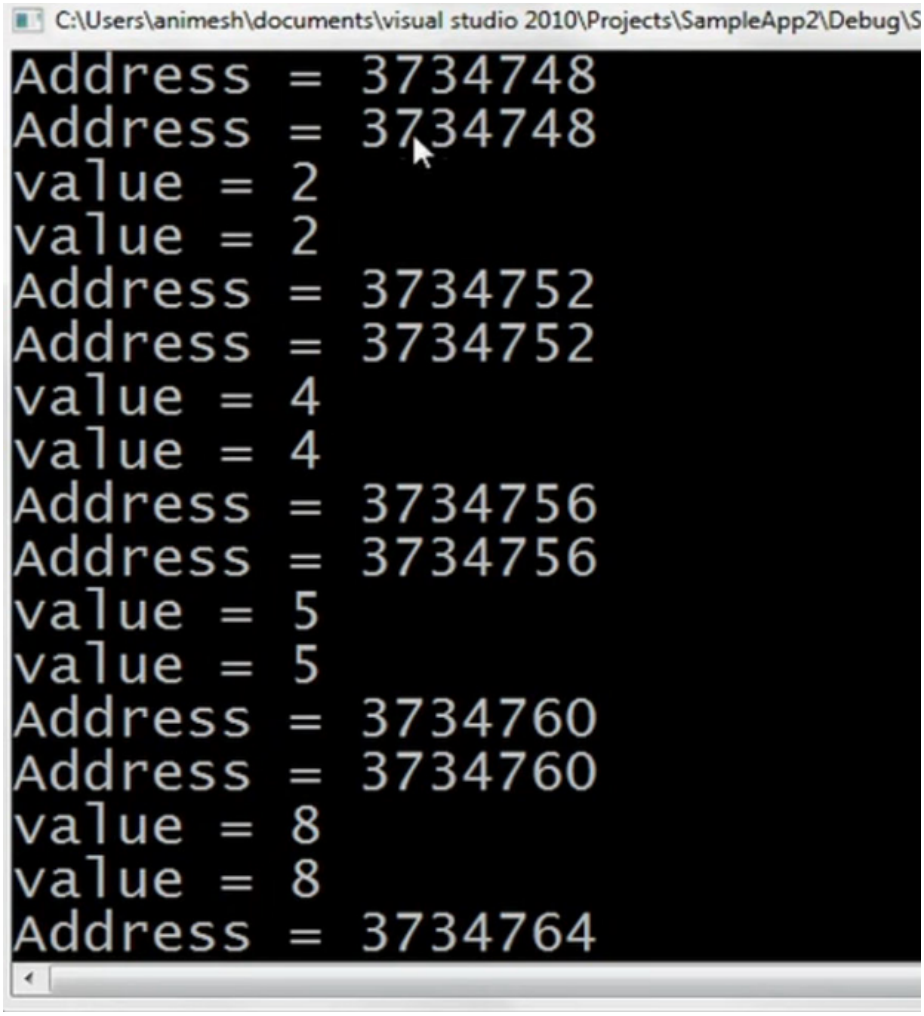

```
Print P+1   // 204
Print *(P+1) // 4
```




```c
// Pointers and Arrays
#include<stdio.h>
int main()
{
    int A[] ={2,4,5,8,1};
    printf("%d\n",A);
    printf("%d\n",&A[0]);
    printf("%d\n",A[0]);
    printf("%d\n",*A);
}
```



```c
// Pointers and Arrays
#include<stdio.h>
int main()
{
    int A[] ={2,4,5,8,1};
    int i;
    for(i = 0;i<5;i++)
    {
        printf("Address = %d\n",&A[i]);
        printf("Address = %d\n",A+i);
        printf("value = %d\n",A[i]);
        printf("value = %d\n",*(A+i));
    }
```



compilation error

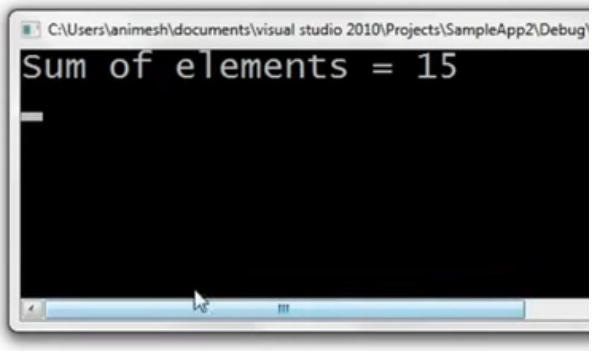
```
int A[] ={2,4,5,8,1};
int i;
int *p = A;
A++;
```

p++; is ok

```
int *p = A;
p++;
```

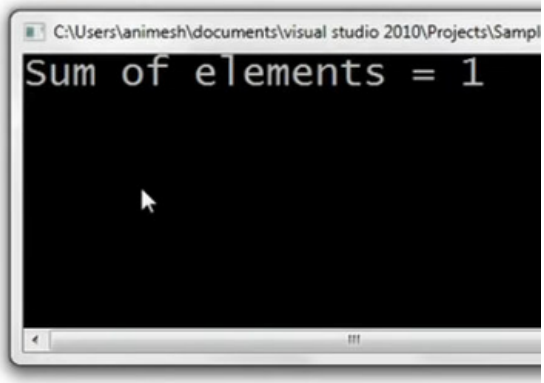
Arrays as function arguments

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



calculate the size of array inside the function

```c
// Arrays as function arguments
#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);
}
```

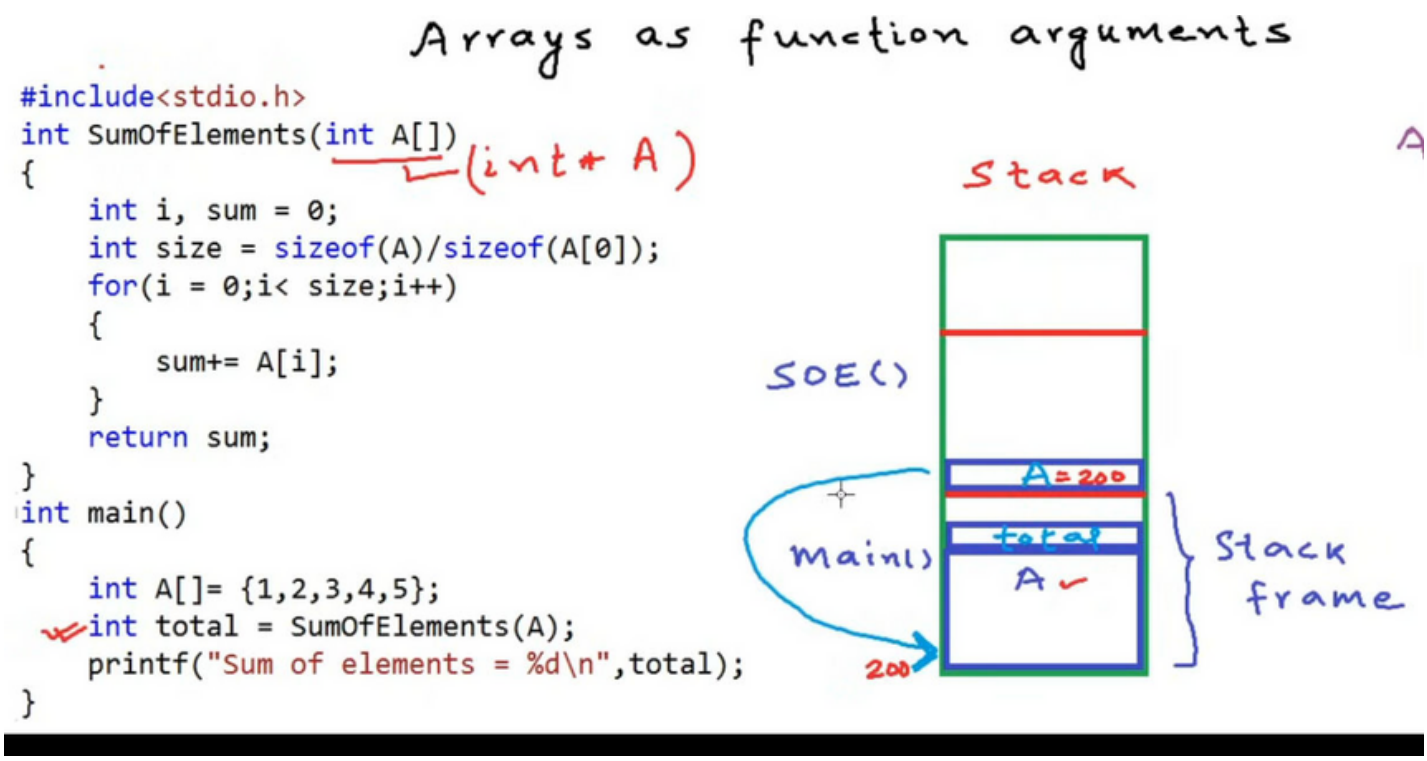


```c
#include<stdio.h>
int SumOfElements(int A[])
{
    int i, sum = 0;
    int size = sizeof(A)/sizeof(A[0]);
    printf("SOE - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
    for(i = 0;i< size;i++)
    {
        sum+= A[i];
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};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

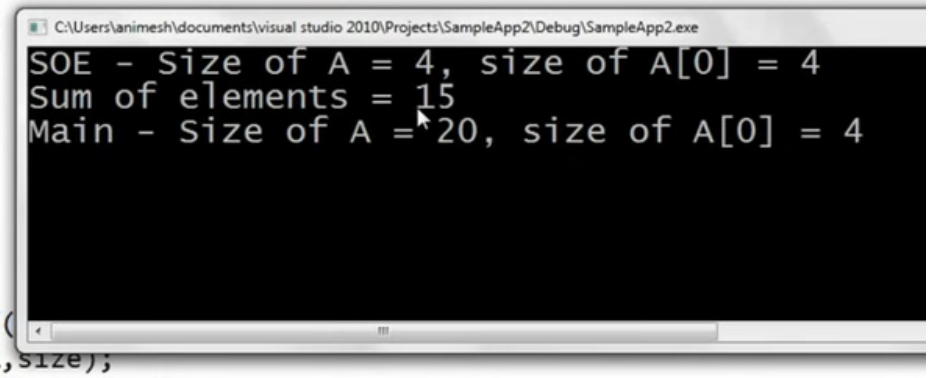
```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
SOE - Size of A = 4, size of A[0] = 4
Sum of elements = 1
Main - Size of A = 20, size of A[0] = 4
```

To understand why the size of A = 4 in SOE, we need to dive deeply into how compiler interprets an array as a function argument

the compiler only creates a pointer variable the same name instead of creating the whole array and just copy address of the first element of the array of the calling function
all that happens is a pointer to integer is created = the compiler implicitly converts this int A[] to int *A (not interpreted as an array but as pointer to integer)



```c
#include<stdio.h>
int SumOfElements(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;
    printf("SOE - Size of A = %d, size of A[0] = %d\n",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 size = sizeof(A)/sizeof(
    int total = SumOfElements(A,size);
    printf("Sum of elements = %d\n",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```



we should always keep in mind that a variable name which is used as an array is different from a variable which is pointer to integer

even though the compiler gives us the privilages to use the name of the array to get the pointer to the first element like in this function sum of elements you could say something like to pass the first element &A[0] but if we just use A instead then that has allowed if A is an array (but cannot do something like incrementing or decrementing it like pointer variables)

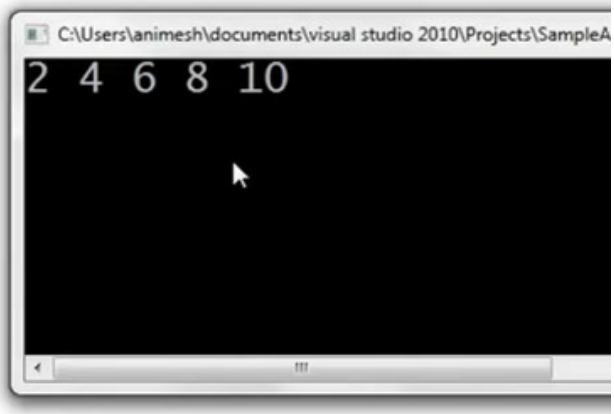
```c
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int total = SumOfElements(&A[0],size);
    printf("Sum of elements = %d\n",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```

and if we have a pointer to the starting address of the array we can use it as a variable name arraybecause A[i] is interpreted as value at address (A+i)

```c
#include<stdio.h>
int SumOfElements(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;
    printf("SOE - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
    for(i = 0;i< size;i++)
    {
        sum+= A[i]; // A[i] is *(A+i)
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int total = SumOfElements(A,size); // A can be used for &A[0]
    printf("Sum of elements = %d\n",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```

Also because the array is passed by reference we can modify the elements of it in the called function and it would reflected in the calling function

```c
#include<stdio.h>
void Double(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;
    for(i = 0;i< size;i++)
    {
        A[i] = 2*A[i];
    }
}
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int i;
    Double(A,size);
    for(i = 0;i< size;i++)
    {
        printf("%d ",A[i]);
    }
}
```



Character arrays and pointers - part 1

Character arrays and pointers

String :- group of characters

eg:- "John"

"Hello World"

"I am feeling lucky"

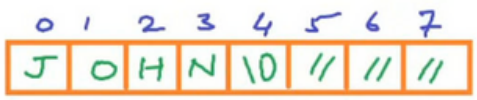
Character arrays and pointers

1) How to store strings

Size of array ≥ no. of characters in string + 1

"John" Size ≥ 5

char c[8];



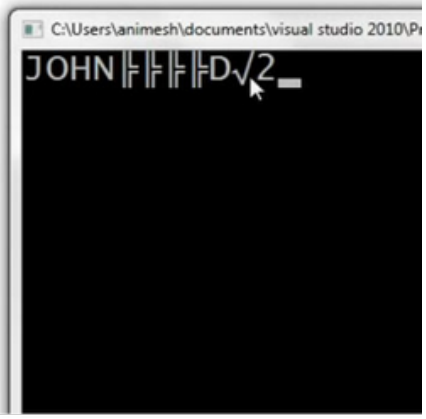
c[0] = 'J'; c[1] = 'O'; c[2] = 'H'; c[3] = 'N';

c[4] = '\0';

Rule :- A string in c has to be null terminated

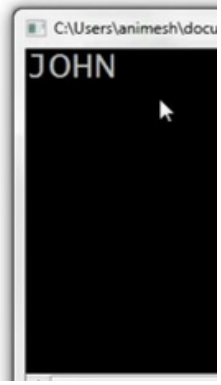
undefined behavior

```c
//character arrays and pointers
#include<stdio.h>
int main()
{
    char C[4];
    C[0] = 'J';
    C[1] = 'O';
    C[2] = 'H';
    C[3] = 'N';
    printf("%s",C);
}
```




```c
#include<stdio.h>
int main()
{
    char C[5];
    C[0] = 'J';
    C[1] = 'O';
    C[2] = 'H';
    C[3] = 'N';
    C[4] = '\0';
    printf("%s",C);
}
```

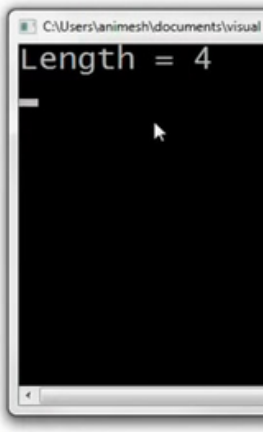
```c
#include<stdio.h>
int main()
{
    char C[20];
    C[0] = 'J';
    C[1] = 'O';
    C[2] = 'H';
    C[3] = 'N';
    C[4] = '\0';
    printf("%s",C);
}
```



not only print function

// take care null char don't count in lenth function

```c
#include<stdio.h>
#include<string.h>
int main()
{
    char C[20];
    C[0] = 'J';
    C[1] = 'O';
    C[2] = 'H';
    C[3] = 'N';
    C[4] = '\0';
    int len = strlen(C);
    printf("Length = %d\n",len);
}
```



in case of using

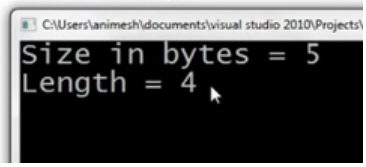
char C[20] = " JOHN";

we cannot replace it by

char C[20];

C = "JOHN"; // invalid

```c
#include<stdio.h>
#include<string.h>
int main()
{
    char C[] = "JOHN";
    printf("Size in bytes = %d\n",sizeof(C));
    int len = strlen(C);
    printf("Length = %d\n",len);
}
```



take care compilation error

```c
#include<stdio.h>
#include<string.h>
int main()
{
    char C[4] = "JOHN";
    printf("Size
    int len = strlen(C);
    printf("Length = %d\n",len);
}
```

Error: a value of type "const char [5]" cannot be used to initialize an entity of type "char [4]"

```c
#include<stdio.h>
#include<string.h>
int main()
{
    char C[5] = "JOHN";
    printf("Size in bytes = %d\n",sizeof(C));
    int len = strlen(C);
    printf("Length = %d\n",len);
}
```

another way of initialization

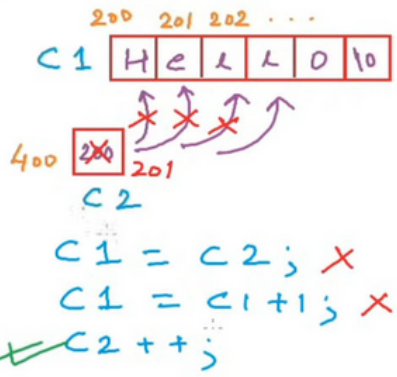
```c
#include<stdio.h>
#include<string.h>
int main()
{
    char C[5] = {'J','O','H','N','\0'};
    printf("Size in bytes = %d\n",sizeof(C));
    int len = strlen(C);
    printf("Length = %d\n",len);
}
```

2) Arrays and pointers are different types that are used in similar manner

```
char c1[6] = "Hello";
char* c2;
c2 = c1;  ✓
Print c2[1]; // l
c2[0] = 'A'; // "Aello"
```

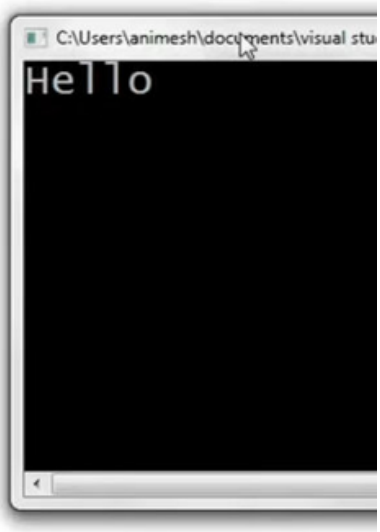
c2[i] is *(c2+i)

c1[i] or *(c1+i)



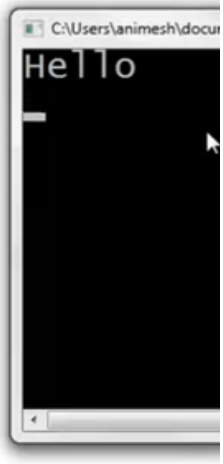
3) Arrays are always passed to function by reference

```c
//character arrays and pointers
#include<stdio.h>
void print(char* C)
{
    int i = 0;
    while(C[i] != '\0')
    {
        printf("%c",C[i]);
        i++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```

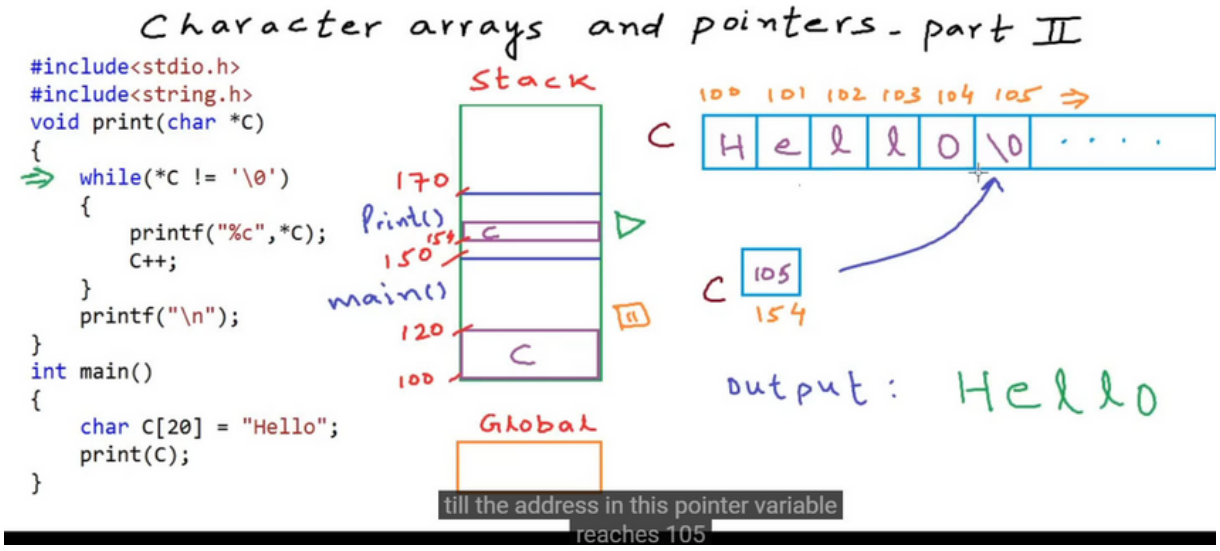


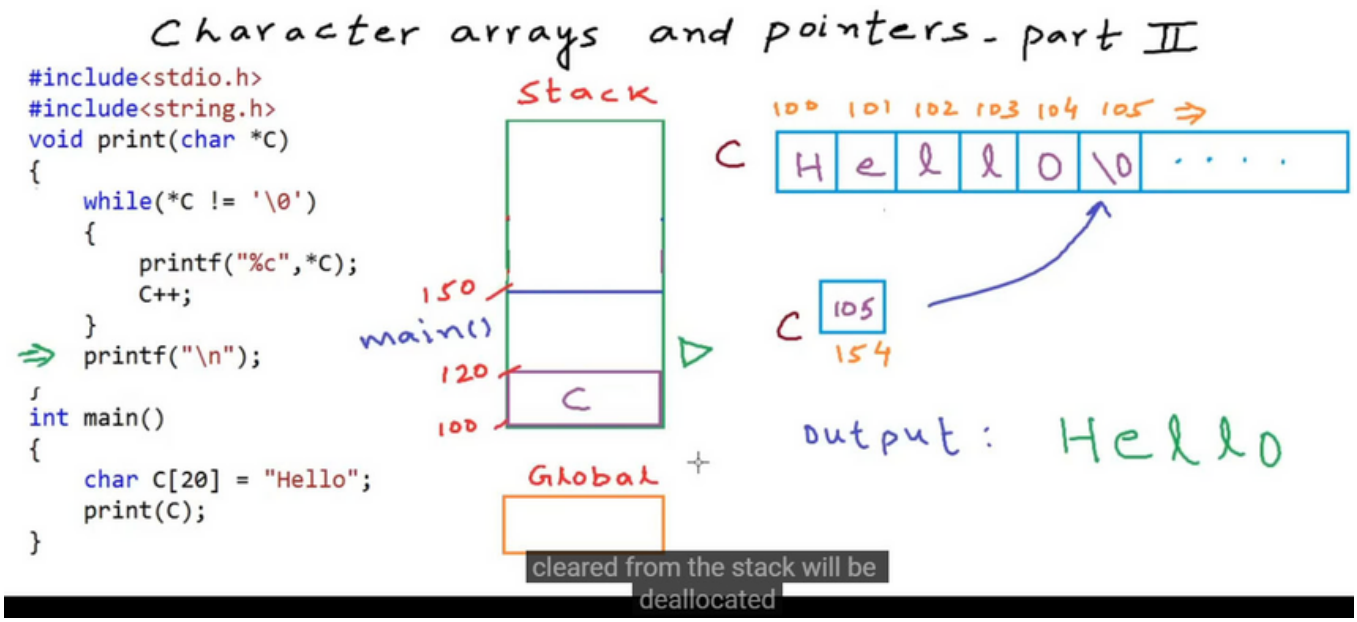

```c
#include<stdio.h>
void print(char* C)
{
    int i = 0;
    while(*(C+i)!= '\0')
    {
        printf("%c",C[i]);
        i++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```

```c
#include<stdio.h>
void print(char* C)
{
    while(*C != '\0')
    {
        printf("%c",*C);
        C++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```



Character arrays and pointers - part 2

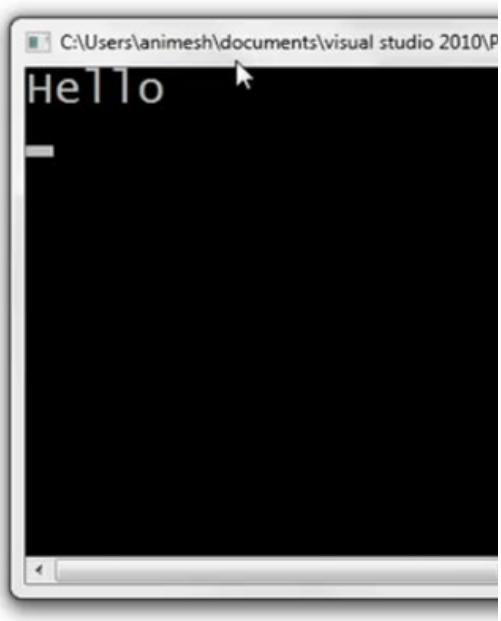




Now, instead of creating a character array of size 20 I'll create pointer named C and equate it aganist a string lateral in a statement

run the program and the result will be the same

```
#include<stdio.h>
#include<string.h>
void print(char *C)
{
    while(*C != '\0')
    {
        printf("%c",*C);
        C++;
    }
    printf("\n");
}
int main()
{
    char *C = "Hello";
    print(C);
}
```



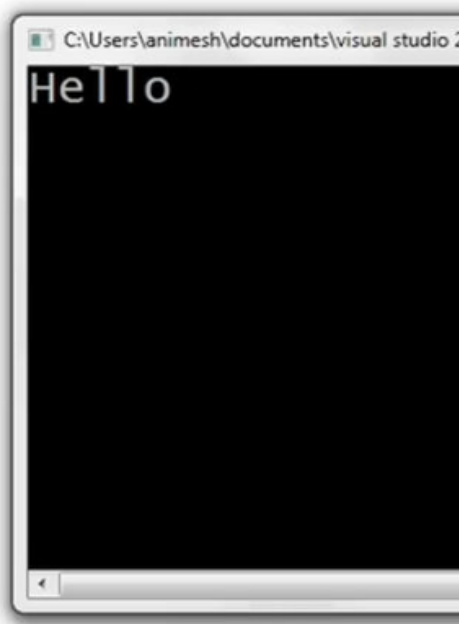
string gets stored as compile time constant and it cannot be modified

```c
#include<stdio.h>
#include<string.h>
void print(char *C)
{
    while(*C != '\0')
    {
        printf("%c",*C);
        C++;
    }
    printf("\n");
}
int main()
{
    //char C[20] = "Hello"; // string gets stored in the space for array
    char *C = "Hello"; // string gets stored as compile time costant
    C[0] = 'A';
    printf("Hello World");
    print(C);
}
```

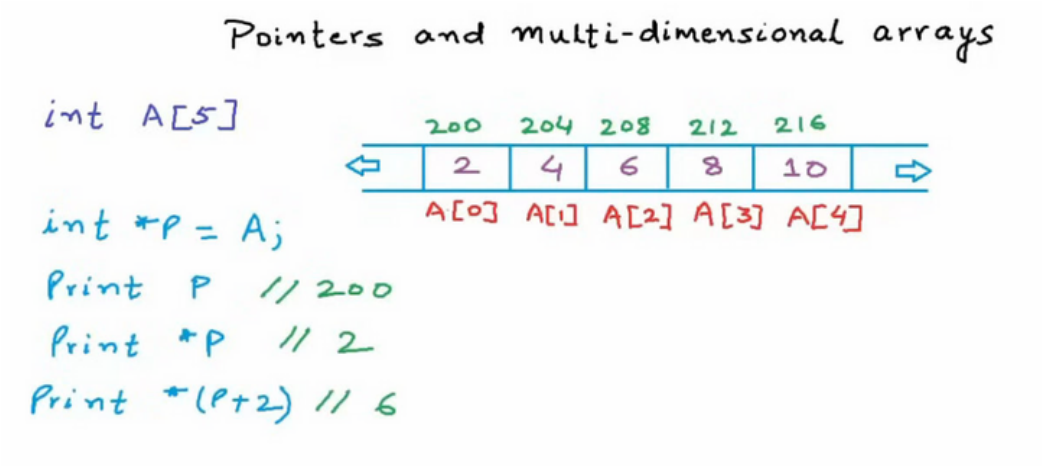

```c
#include<stdio.h>
#include<string.h>
void print(char *C)
{
    C[0] = 'A';
    while(*C != '\0')
    {
        printf("%c",*C);
        C++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```

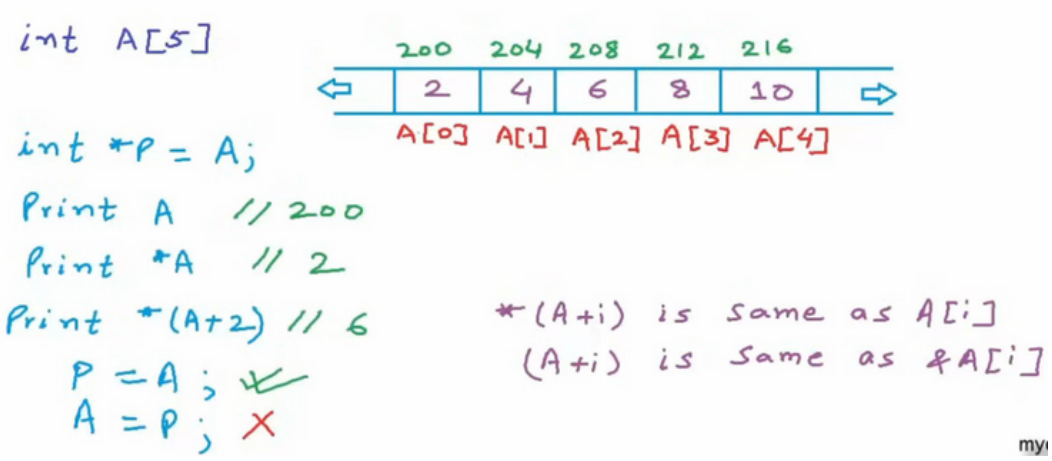
```c
#include<stdio.h>
#include<string.h>
void print(const char *C)
{
    C[0] = 'A';
    while(*C != '\0')
    {
        printf("%c",*C);
        C++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```

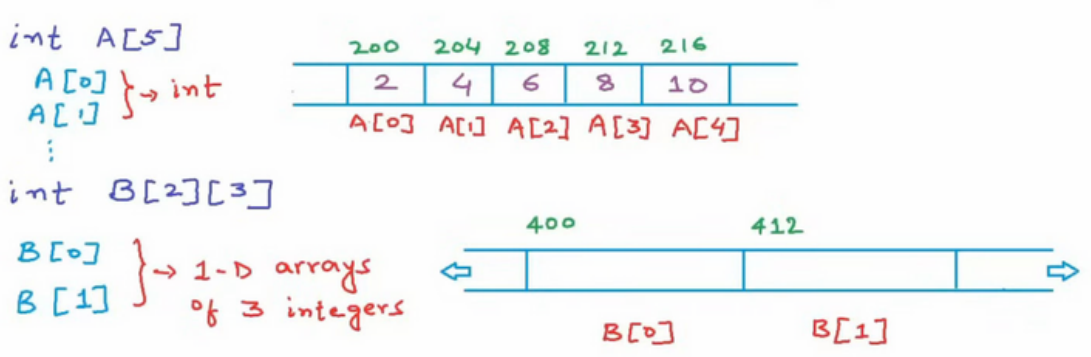

```c
#include<stdio.h>
#include<string.h>
void print(const char *C)
{
    while(*C != '\0')
    {
        printf("%c",*C);
        C++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```



Pointers and 2-D arrays





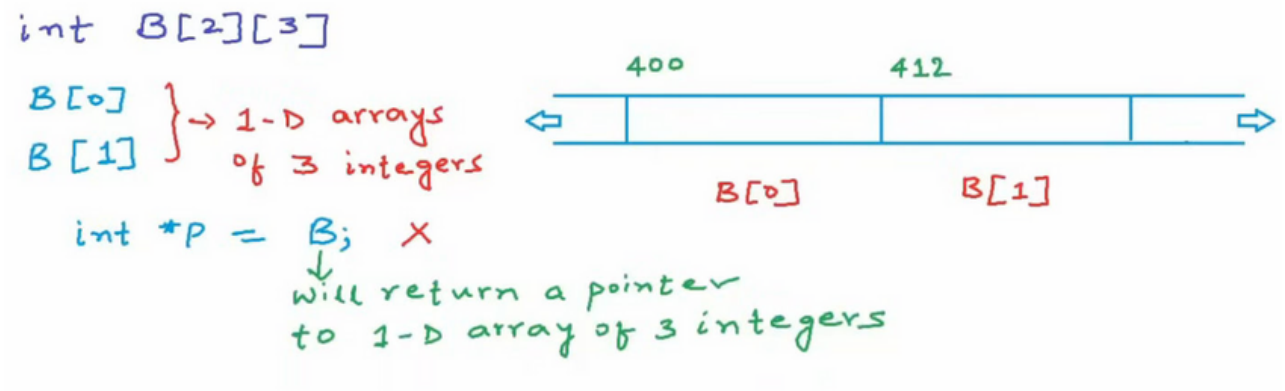


As we have said, name of the array returns a pointer to the first element in the array

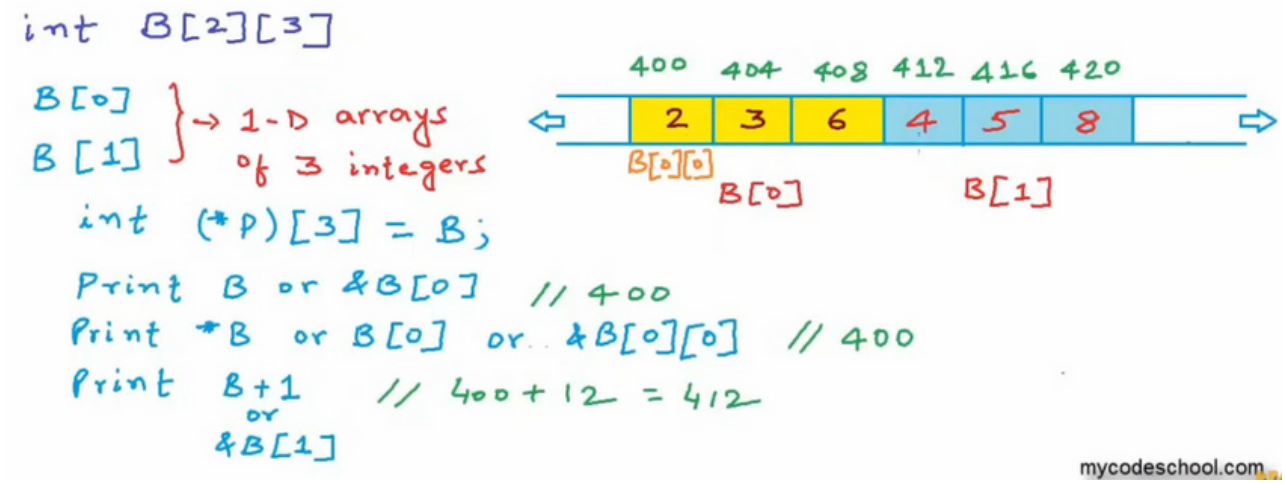
This time each element is not an integer but one dimensional array of 3 integers

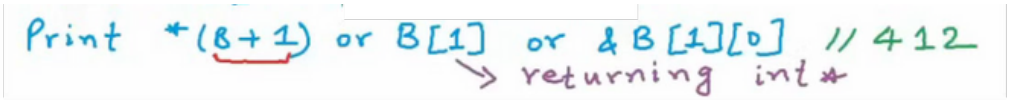
so int *p = B; // compilation error because B will return a pointer to 1-D array of 3 integers not just a pointer to integer

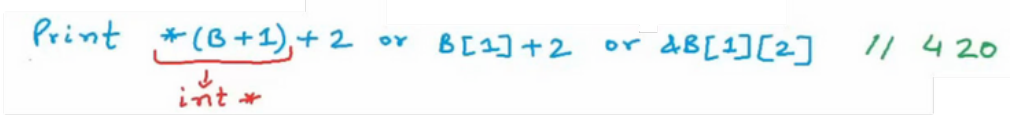
The type of a pointer matters not when you have to read the address but when you dereference or perform pointer arthimatic

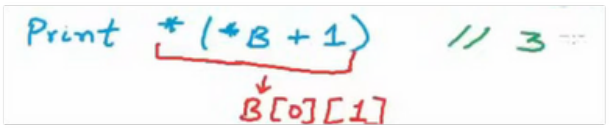


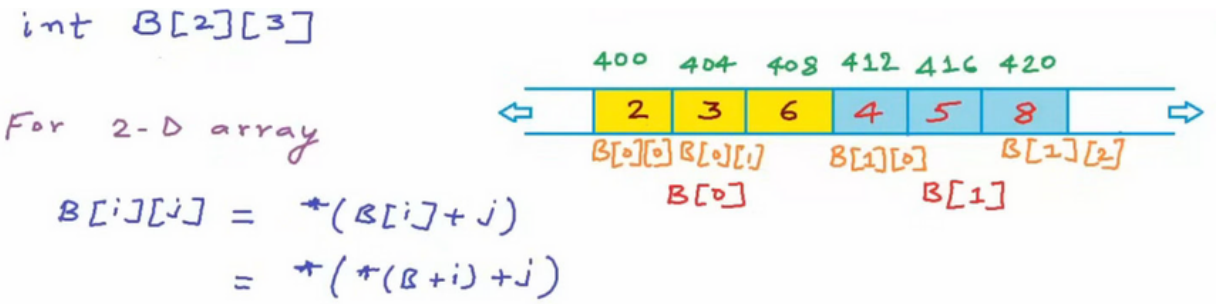
we can define a pointer to 1-D array od 3 integers like



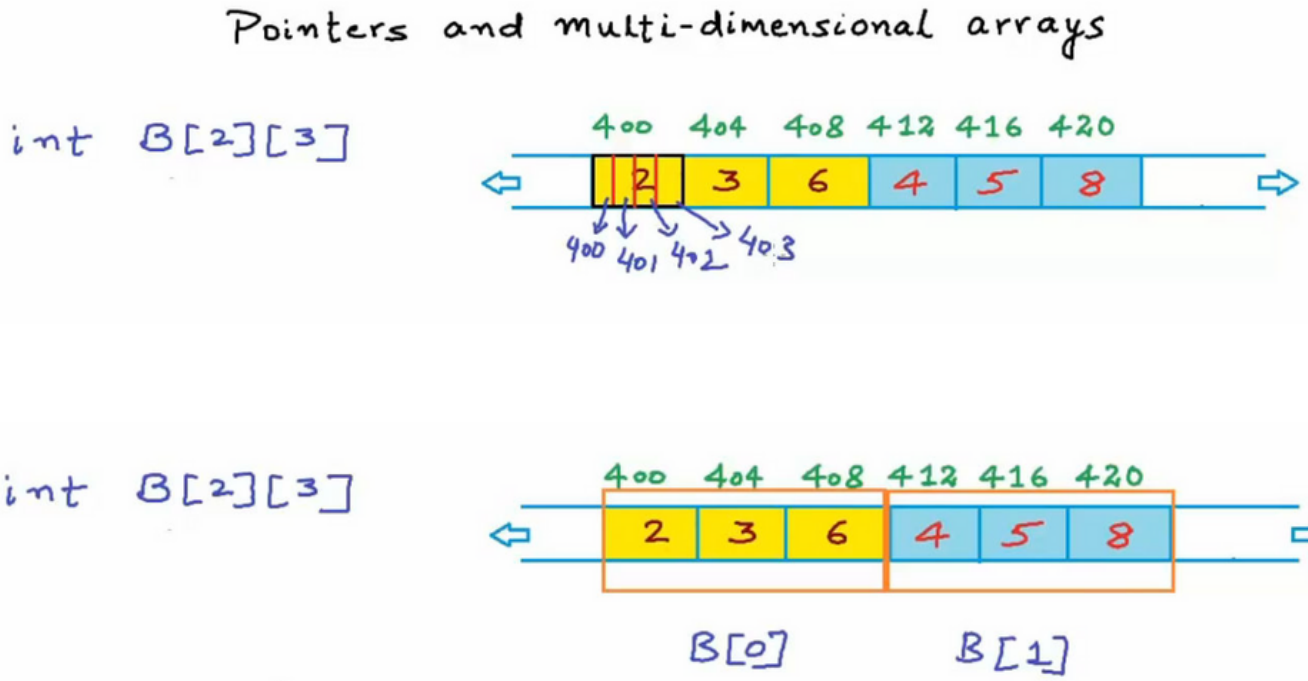


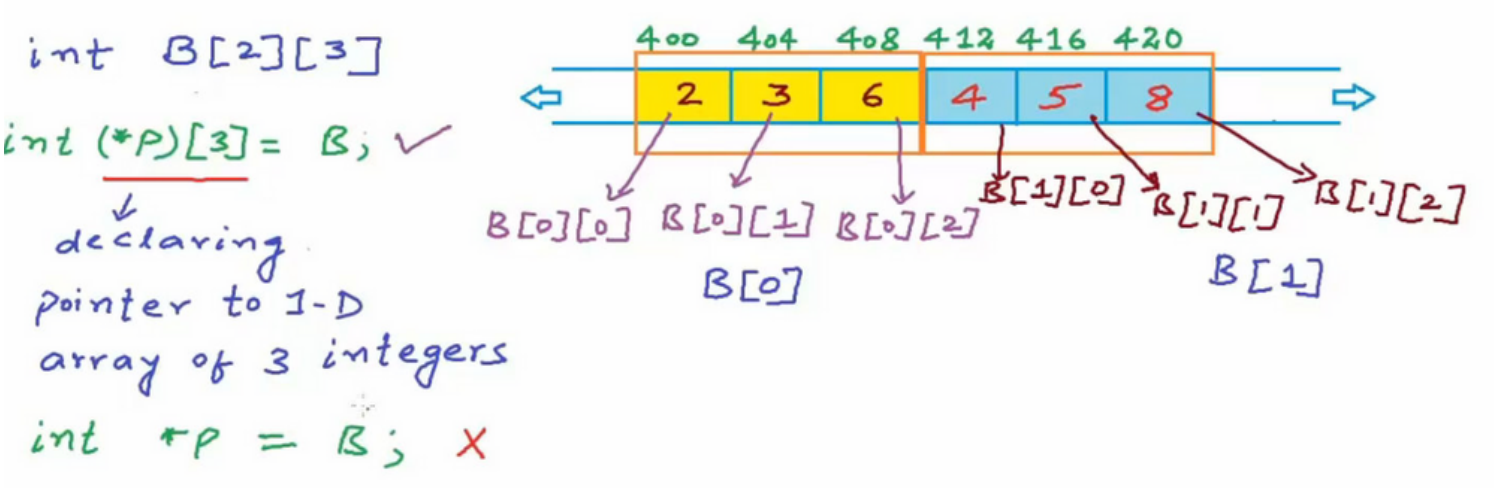






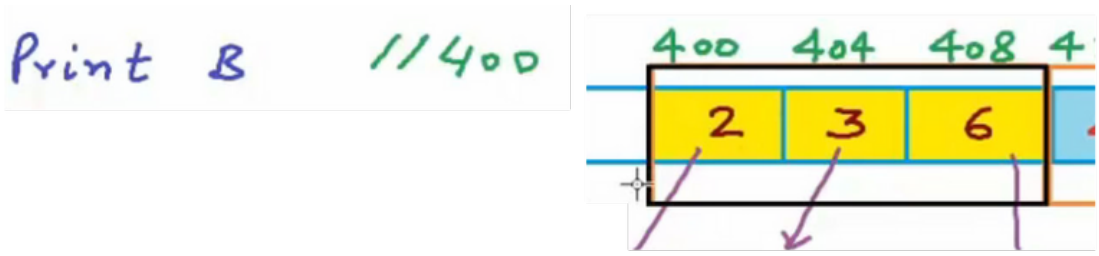
Pointers and multidimensional arrays

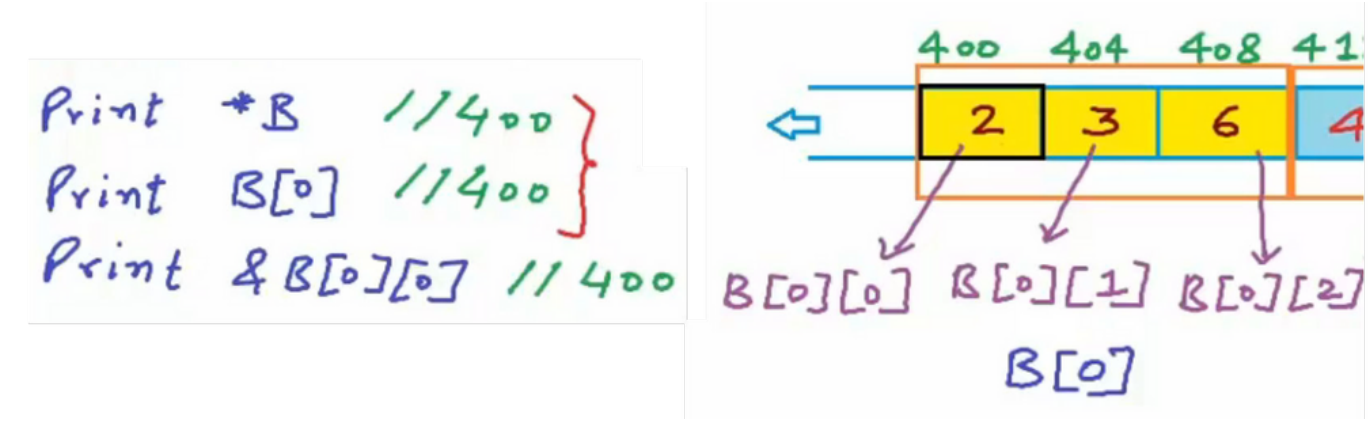


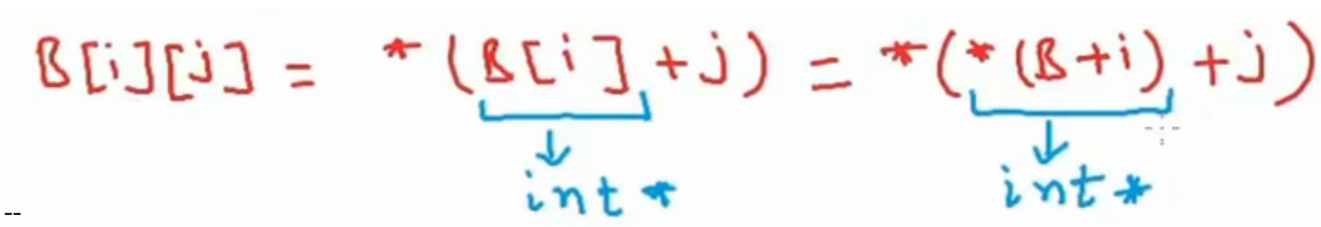


B is returning pointer to a 1-D array of three integers

while *B is returning pointer to an integer

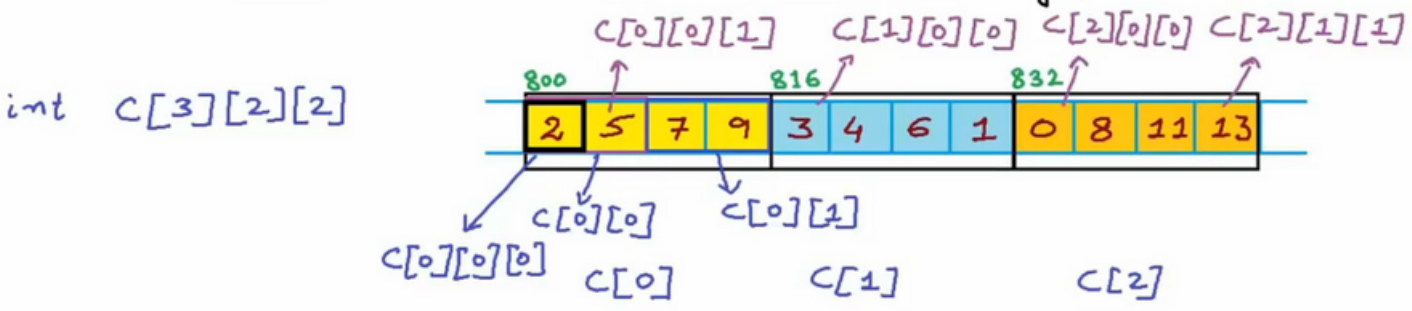


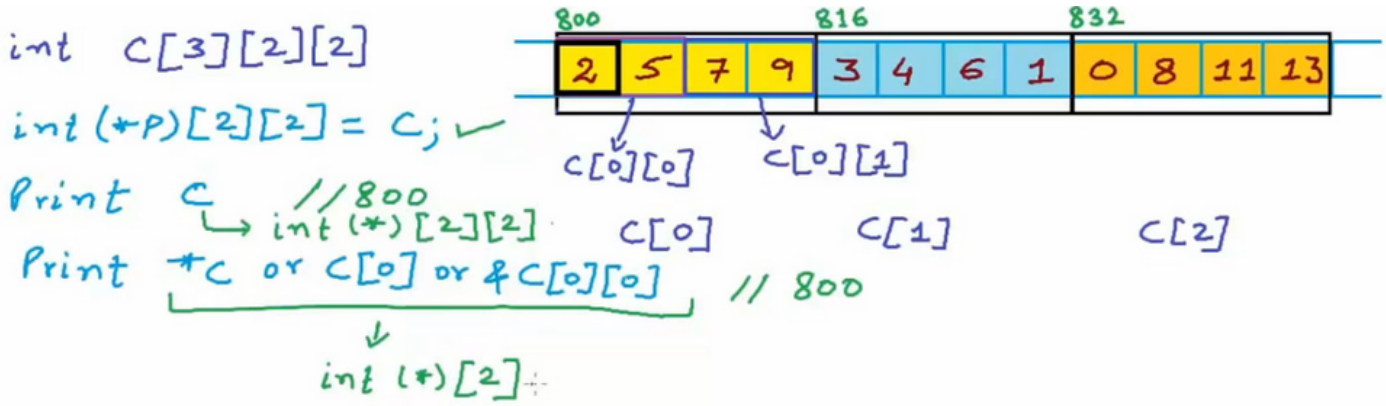




--

Pointers and multi-dimensional arrays

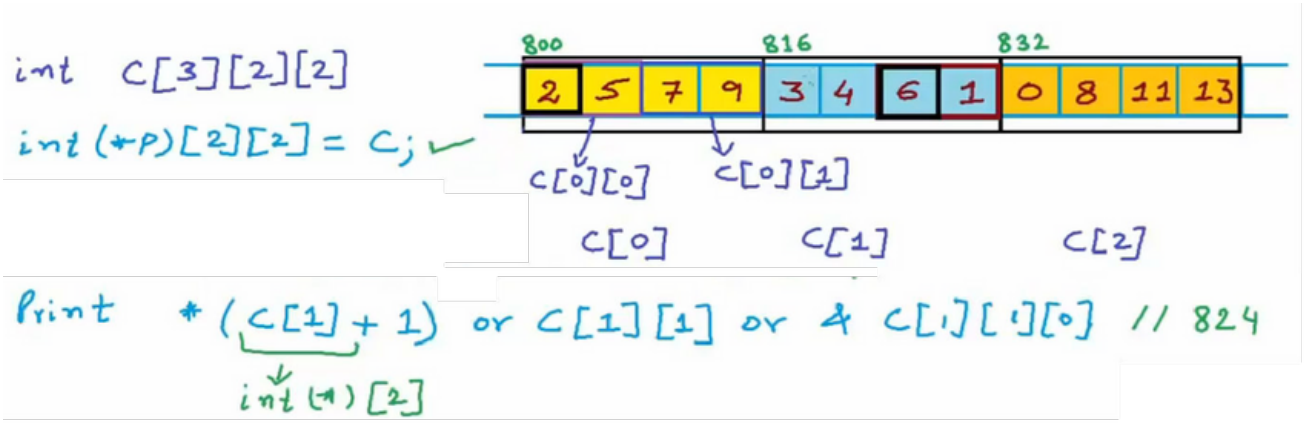




$$c[i][j][k] = *(c[i][j]+k) = *(*(c[i]+j)+k)$$
$$= *(*(*(c+i)+j)+k)$$

Print $*(\underbrace{c[0][1]}_{\downarrow\ (int*)} + 1)$

Print $*(c[0][1]+1)$ or $c[0][1][1]$ // 9




```c
// Pointers and multi- dimensional arrays
#include<stdio.h>
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    printf("%d %d %d %d", C, *C, C[0], &C[0][0]);
}
```

```
C:\Users\animesh\Documents\Visual Studio 2010\Projects\SampleApp5\Debug\SampleApp5.exe
4192172 4192172 4192172 4192172
```

```c
// Pointers and multi- dimensional arrays
#include<stdio.h>
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    printf("%d %d %d %d\n", C, *C,
    printf("%d\n",*(C[0][0]+1));
}
```

```
C:\Users\animesh\Documents\Visual Studio 2010\Projects\SampleApp5\Debug\SampleApp5.exe
3078752 3078752 3078752 3078752
5
```

Passing multi-dimensional array as function arguments

```c
// Pointers and multi- dimensional arrays
#include<stdio.h>
void Func(int *A) // Argument: 1-D array of integers
{
}
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    int A[2] = {1,2};
    int B[2][3] ={{2,4,6},{5,7,8}}; // B returns int (*)[3]
    Func(A); // A returns int*
}
```

```c
#include<stdio.h>
void Func(int (*A)[3]) // Argument: 2-D array of integers
{
}
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    int A[2] = {1,2};
    int B[2][3] ={{2,4,6},{5,7,8}}; // B returns int (*)[3]
    Func(A);
}
```

or

```c
// Pointers and multi- dimensional arrays
#include<stdio.h>
void Func(int A[][3]) // Argument: 2-D array of integers
{
}
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    int A[2] = {1,2};
    int B[2][3] ={{2,4,6},{5,7,8}}; // B returns int (*)[3]
    Func(B);
}
```

cannot do this

```c
// Pointers and multi- dimensional arrays
#include<stdio.h>
void Func(int A[][3]) // Argument: 2-D array of integers
{
}
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    int A[2] = {1,2};
    int B[2][3] ={{2,4,6},{5,7,8}}; // B returns int (*)[3]
    int X[2][4];
    Func(X); // X returns int (*)[4]
}
```

this is fine

```c
// Pointers and multi- dimensional arrays
#include<stdio.h>
void Func(int A[][3]) // Argument: 2-D array of integers
{
}
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    int A[2] = {1,2};
    int B[2][3] ={{2,4,6},{5,7,8}}; // B returns int (*)[3]
    int X[5][3];
    Func(X);
}
```

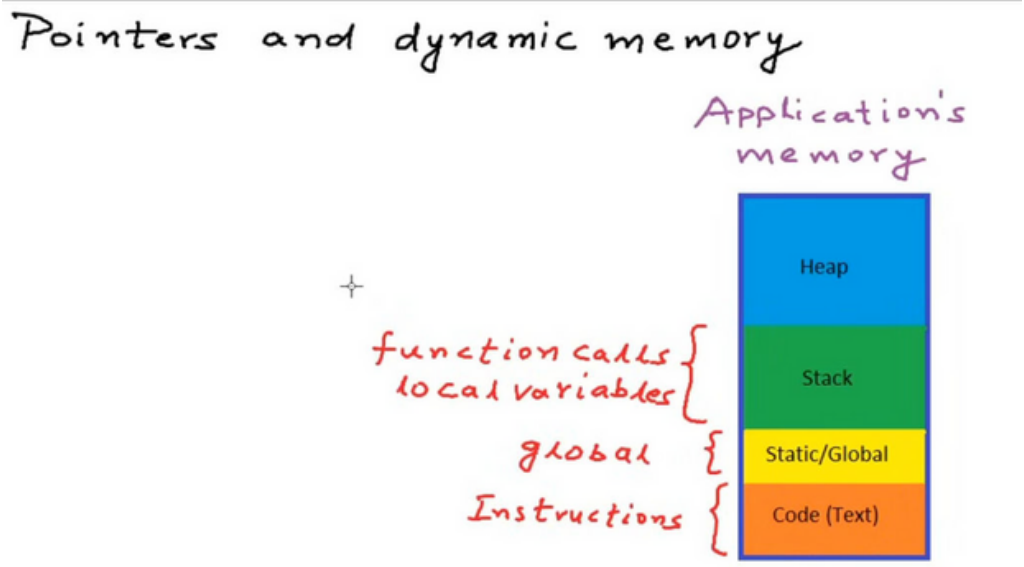
for C array

```c
void Func(int A[][2][2]) // Argument: 2-D array of integers
{
}
```

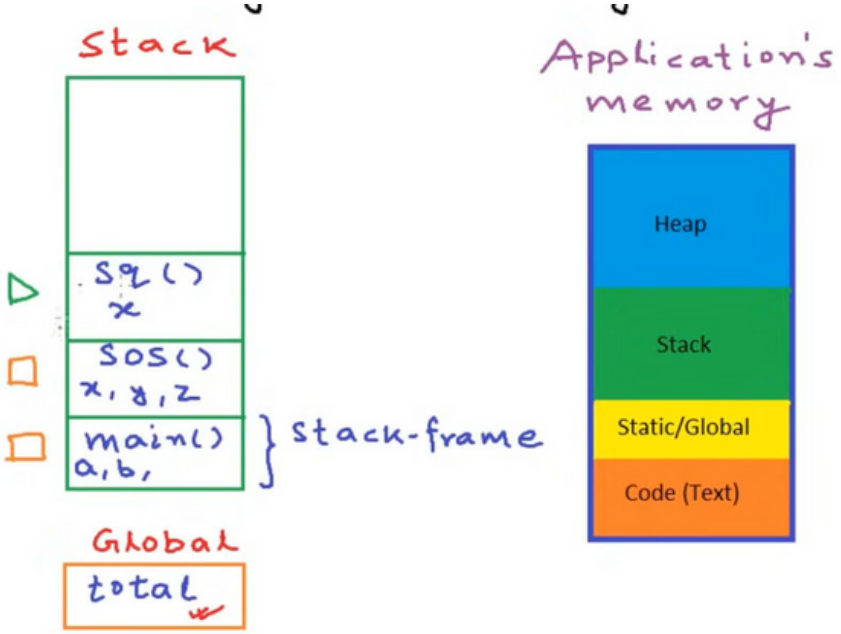
or

```c
// Pointers  and multi- dimensional arrays
#include<stdio.h>
void Func(int (*A)[2][2]) // Argument: 3-D array of integers
{
}
int main()
{
    int C[3][2][2]={{{2,5},{7,9}},
                    {{3,4},{6,1}},
                    {{0,8},{11,13}}};
    int A[2] = {1,2};
    int B[2][3] ={{2,4,6},{5,7,8}}; // B returns int (*)[3]
    Func(C);
}
```

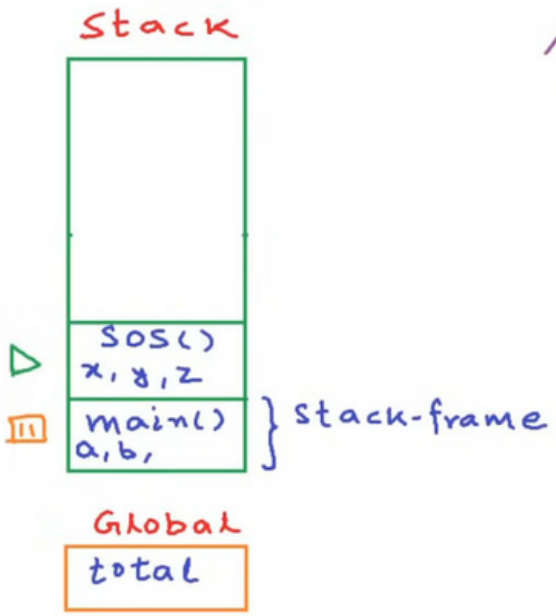
Pointers and dynamic memory - stack vs heap



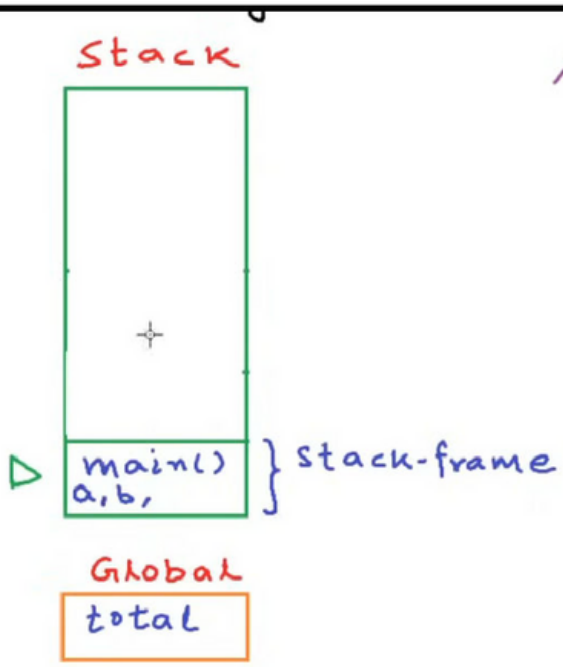

```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



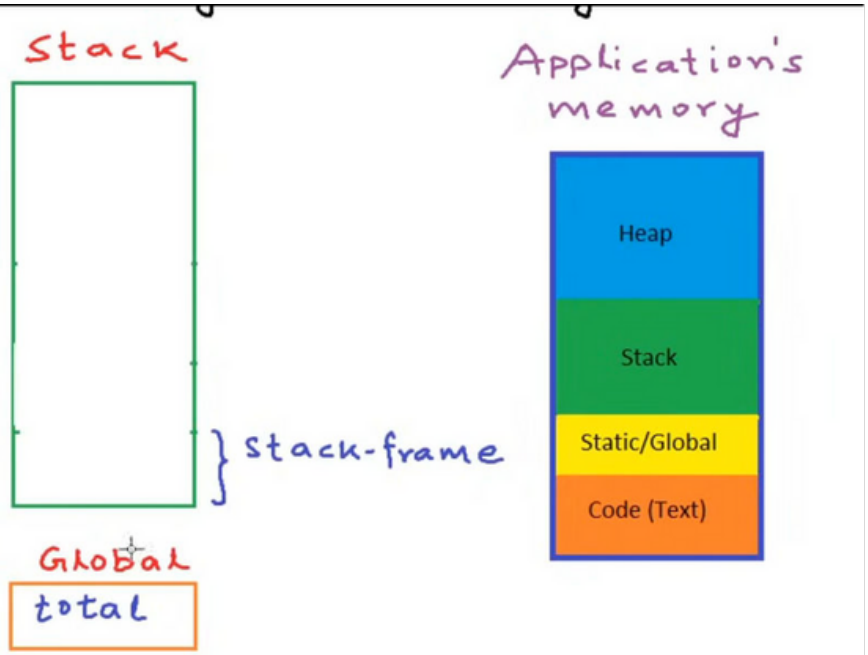
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



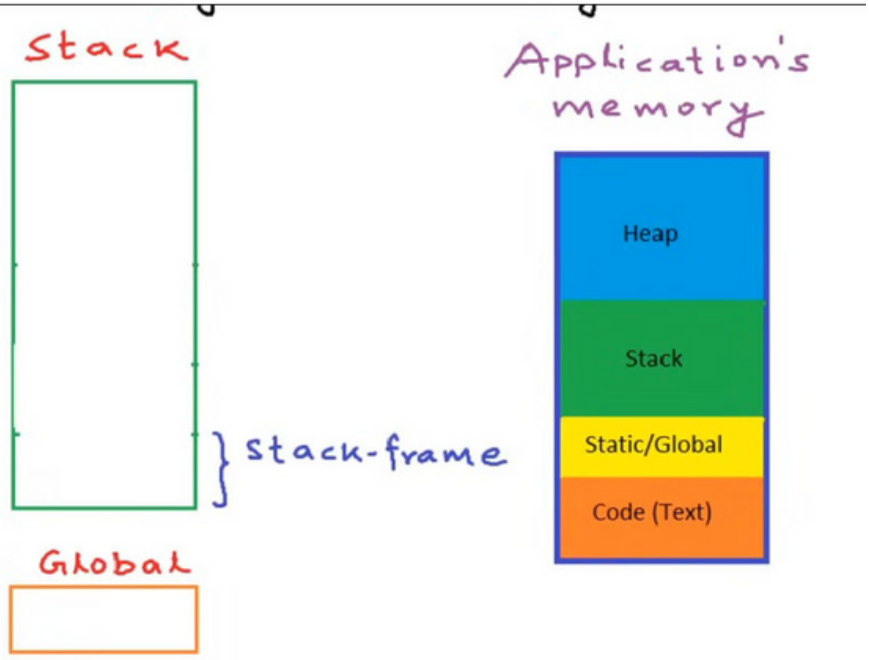

```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```

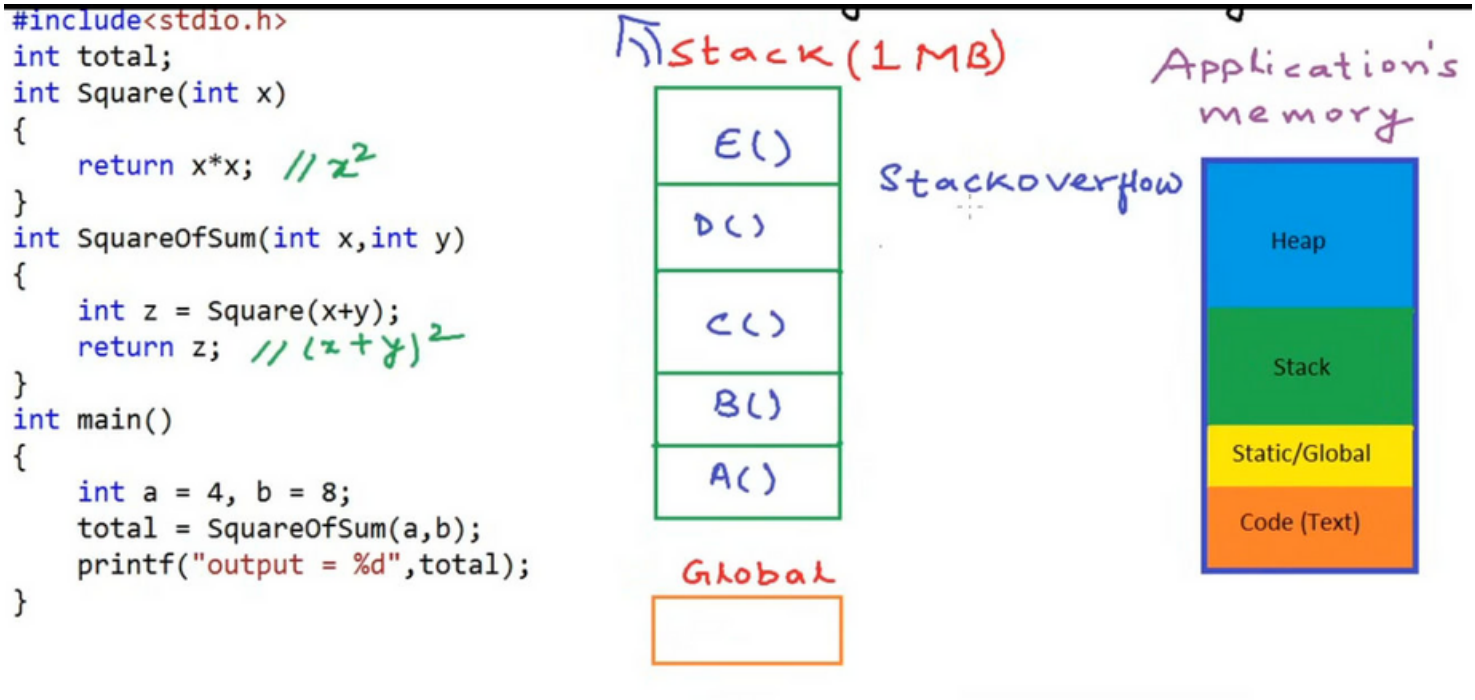


```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```




```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 overflow = and in this case our program will crash (because of bad recursion for example)

So, as we can see, there are some limitations of stack = the memory et aside for stack doesnot grow during runtime (Application cannot request more memory for stack)

It is not possible to manipulate the scope of a variable if it is on the stack

Another limitation is that, if we need to declare a large data type, like an array as local variable, then we need to know the size of the array at compile time only.

If we have a scenario like we have to decide how large the array will be based on some parameters during runtime then it is a problem with stack.

For that we have heap!

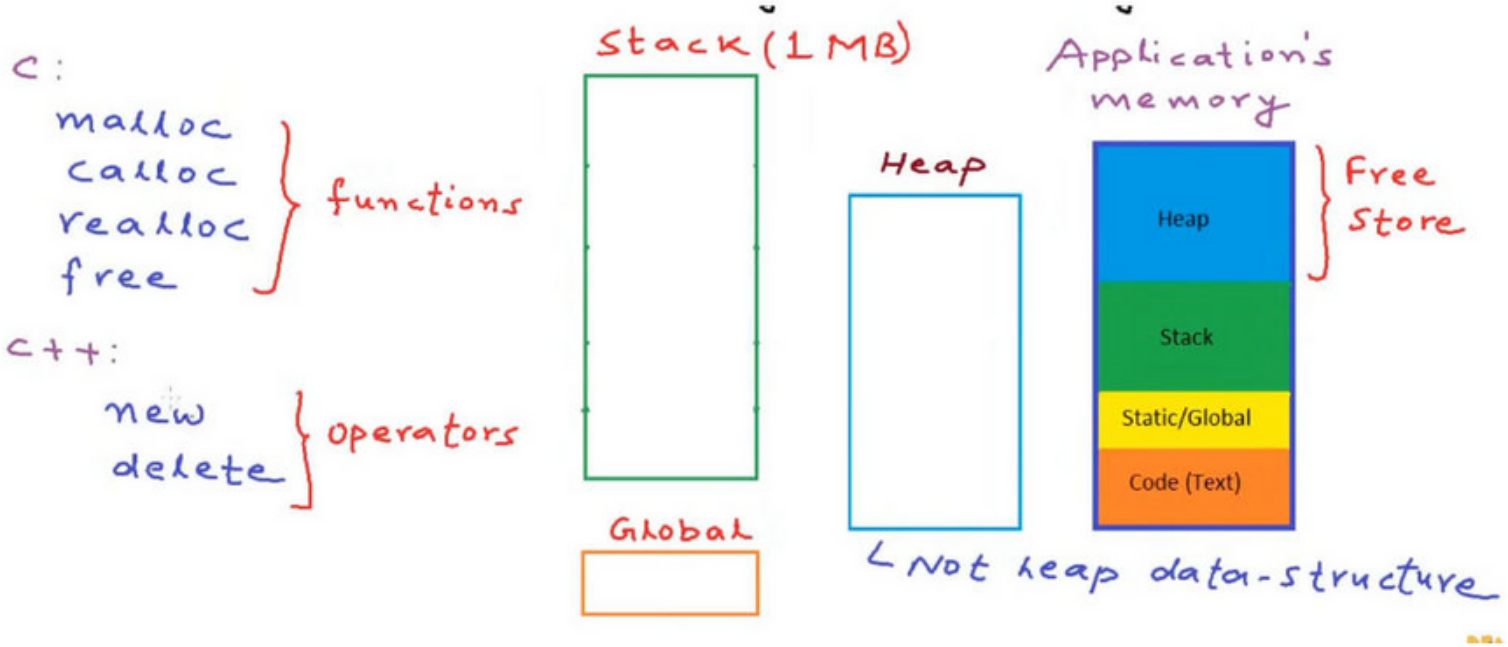
Unlike stack application heap is not fixed and there is no set rule for allocation and deallocation of the memory. A programmer can totally control how much memory to use from the heap, till what time to keep the data in the memory during the applications lifetime and heap can grow as long as you don't run out of memory on the system itself.

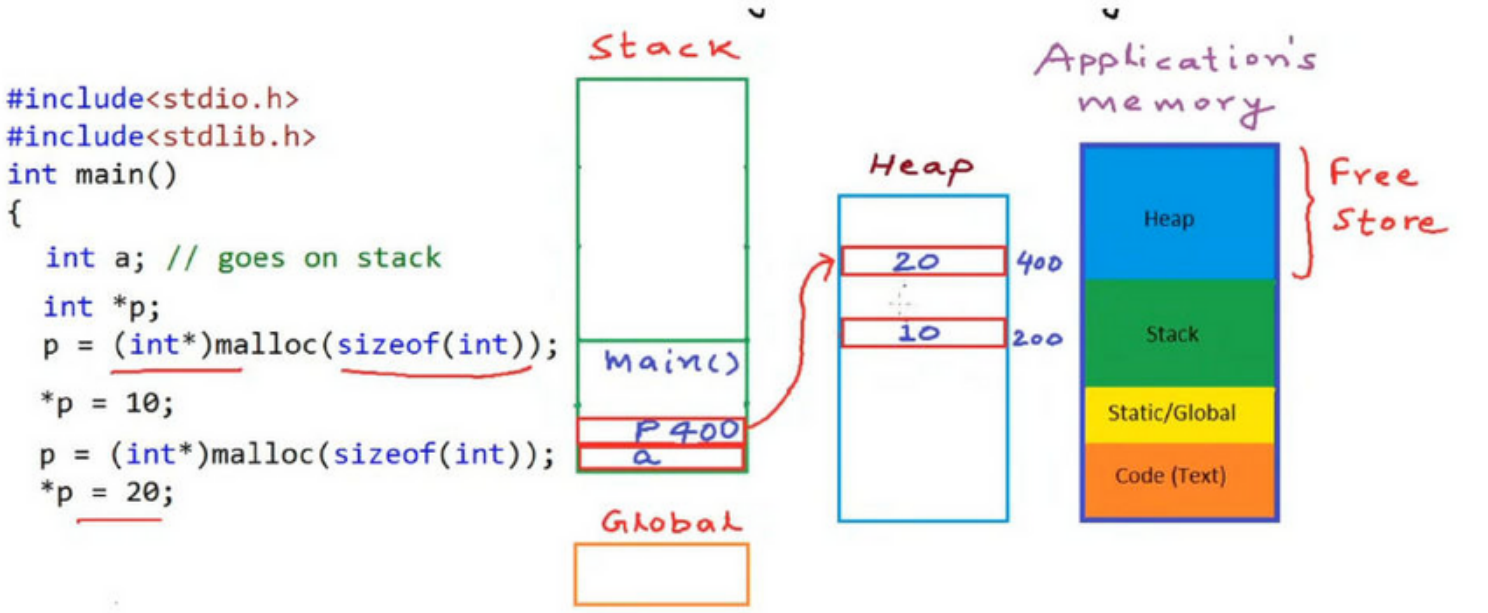
How heap is implemented by the operating system, language run time or the compiler is something which can vary, which is a thing of computer architecture

But an abstracted way of looking at the heap as a programmer is that this is one large free pool of memory available to us that we can use flexibly as per our need

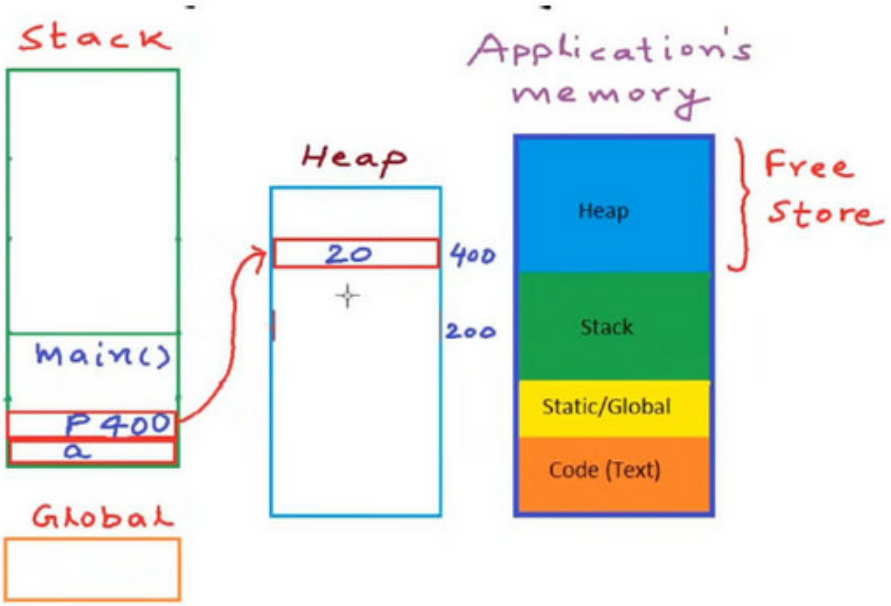
heap memory is not data structure (not implementation of heap data structure)

stack is also one data strcutre but the stack segment of the memory is actually an implementation of the stack data structure





```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;
```

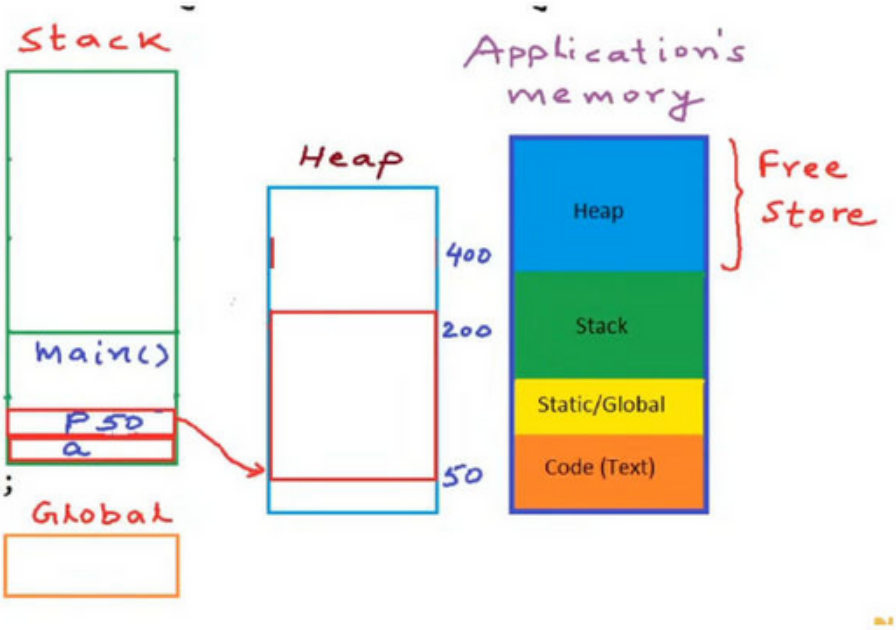


Store an arry in the heap

Note = If malloc cannot find/allocate enough memory in the heap it will return null

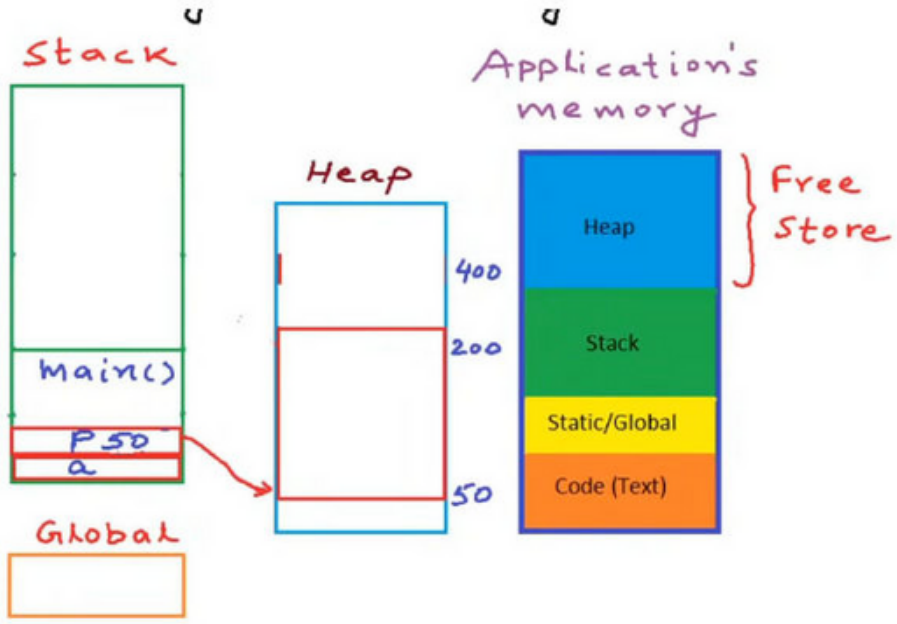
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int a; // goes on stack
   int *p;
   p = (int*)malloc(sizeof(int));
   *p = 10;
   free(p);
 p = (int*)malloc(20*sizeof(int));
}
```

P[0], P[1], P[2]
*P *(P+1)

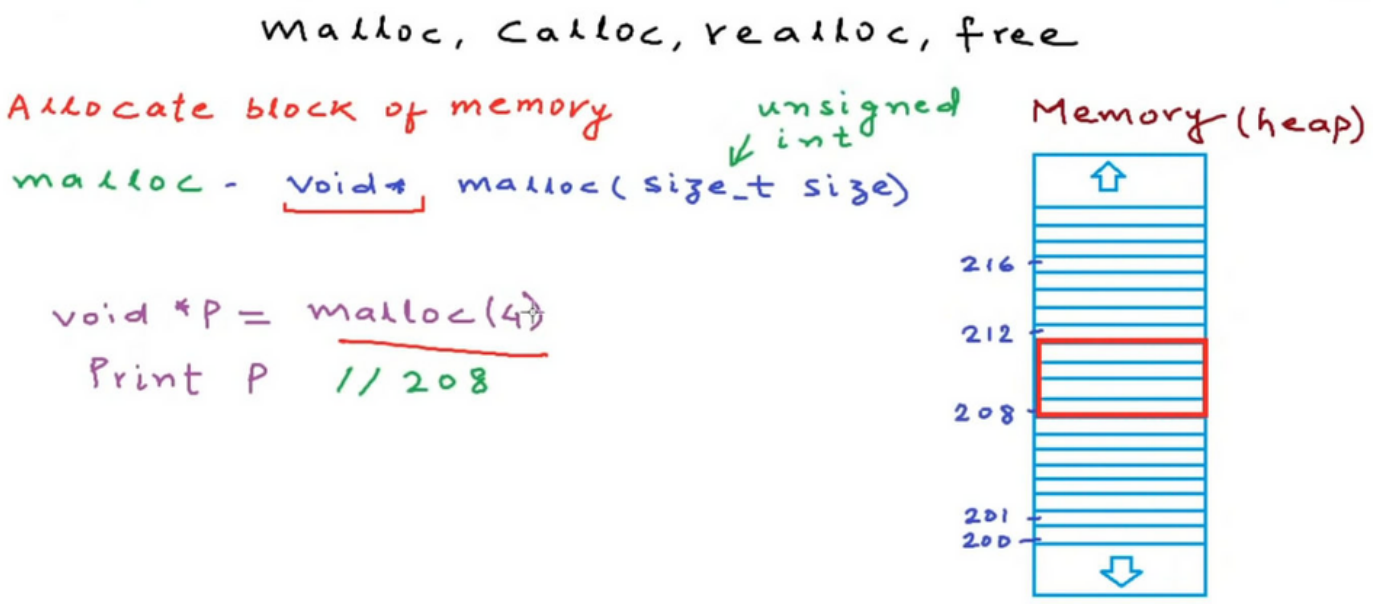


c++

```
#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;
}
```



Dynamic memory allocation in C - malloc calloc realloc free



malloc, calloc, realloc, free

Allocate block of memory

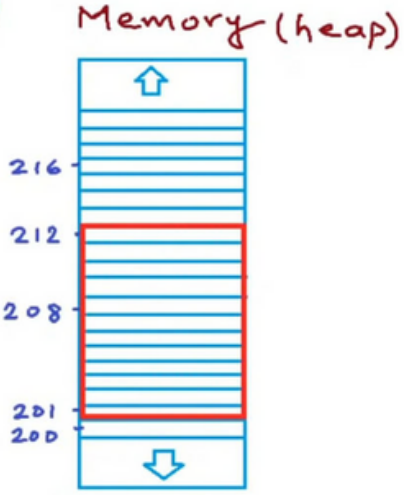
malloc - void* malloc(size_t size)

unsigned int

```
void *P = malloc( 3 * Sizeof(int))
Print P   // 201
*P = 2 X
```

no. of elements

size of one unit



malloc, calloc, realloc, free

Allocate block of memory

malloc - void* malloc(size_t size)

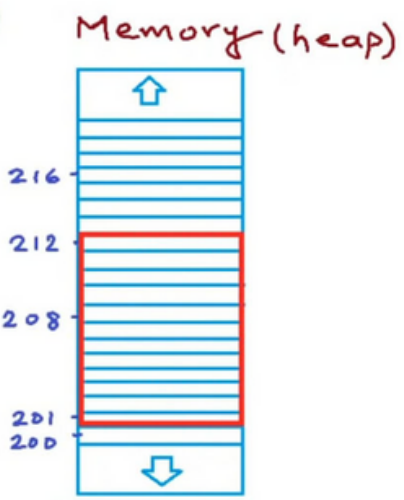
unsigned int

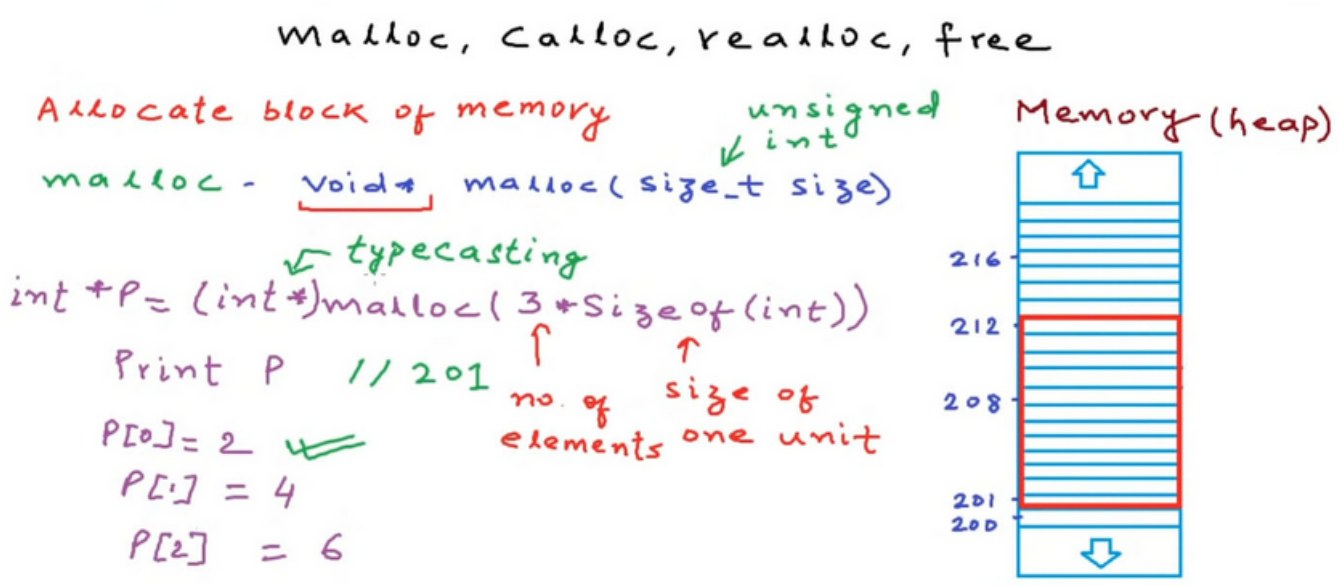
typecasting

```
int *P = (int *)malloc( 3 * Sizeof(int))
Print P   // 201
*P = 2
*(P+1) = 4
*(P+2) = 6
```

no. of elements

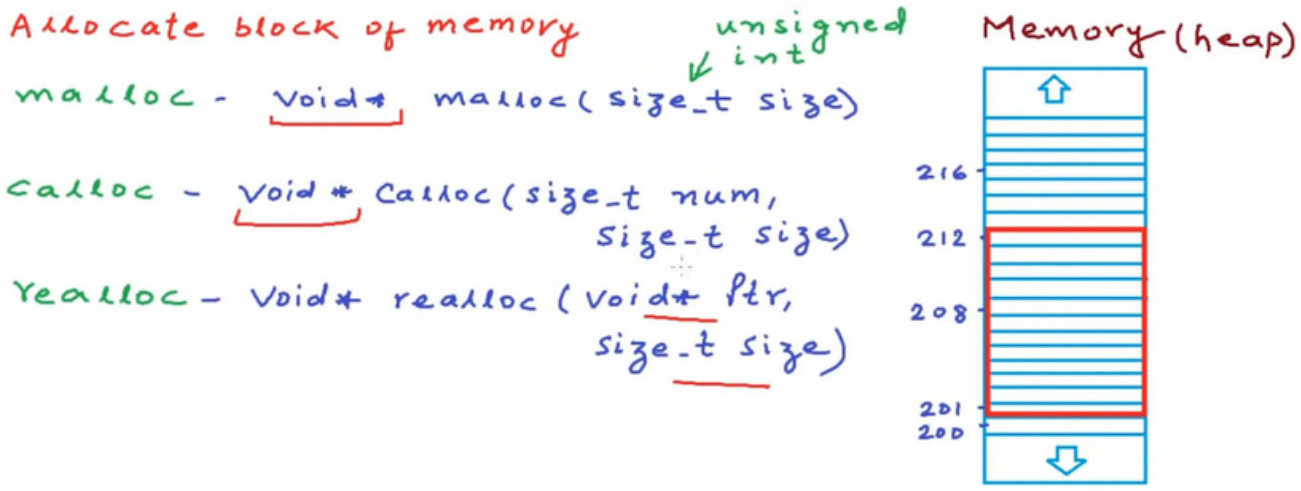
size of one unit





malloc = garbage intialization

calloc = 0



I want to declare an array with size n

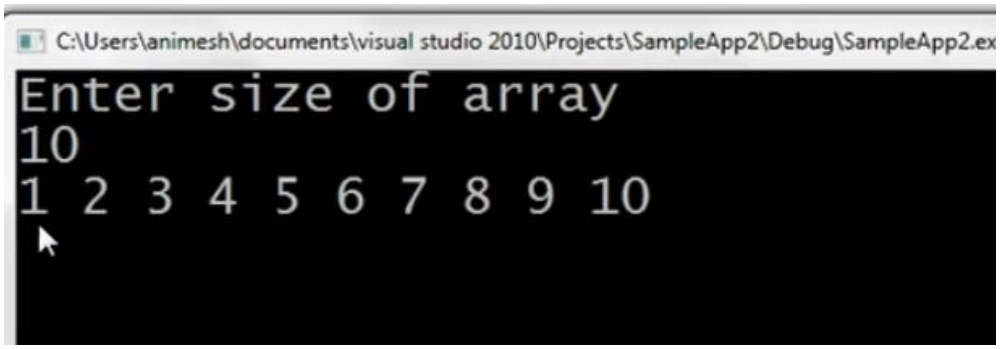
we cannot do that = we cannot know the size of the array in the run time = compilation error

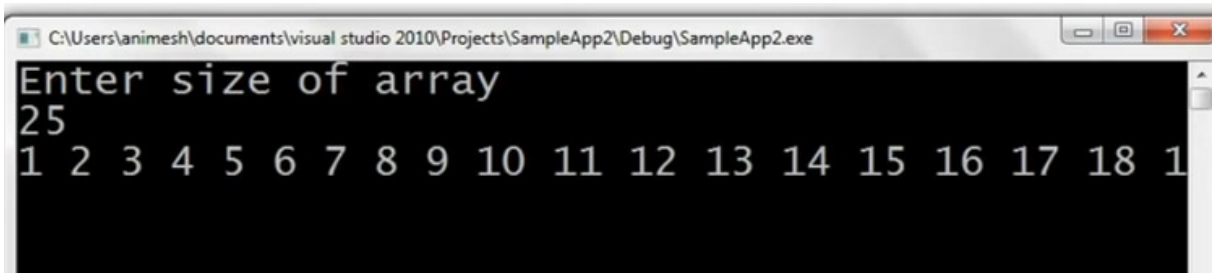
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int A[n];
}
```

allocate the memory dynimcally

don't forgate to cast the return 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*)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]);
    }
}
```

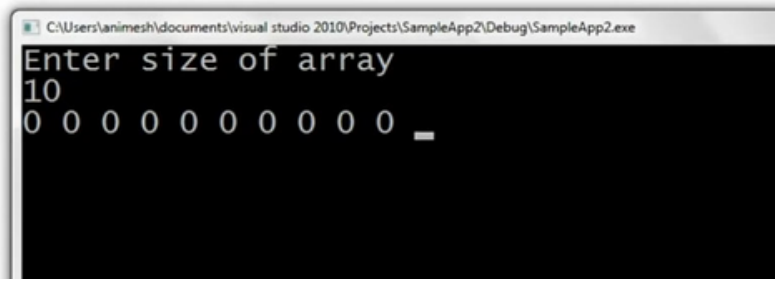





```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)calloc(n,sizeof(int)); //dynamically allocated array
    for(int i = 0;i<n;i++)
    {
        A[i] = i+1;
    }
    for(int i =0;i<n;i++)
    {
        printf("%d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

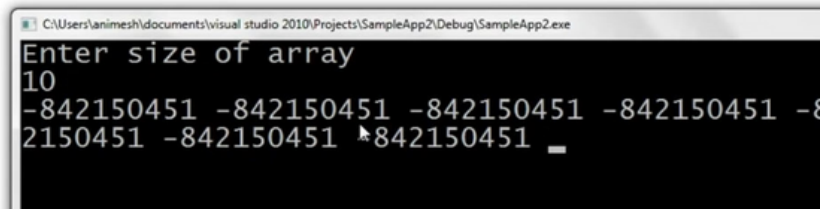
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)calloc(n,sizeof(int)); //dynamically allocated array

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



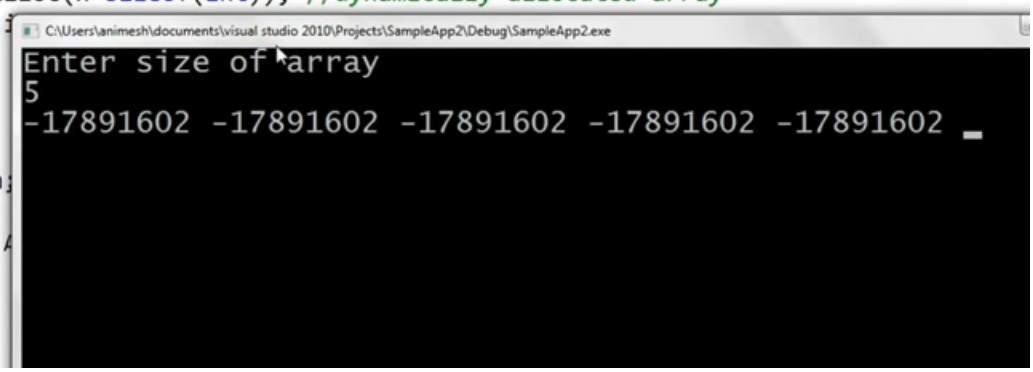
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*sizeof(int)); //dynamically allocated array

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

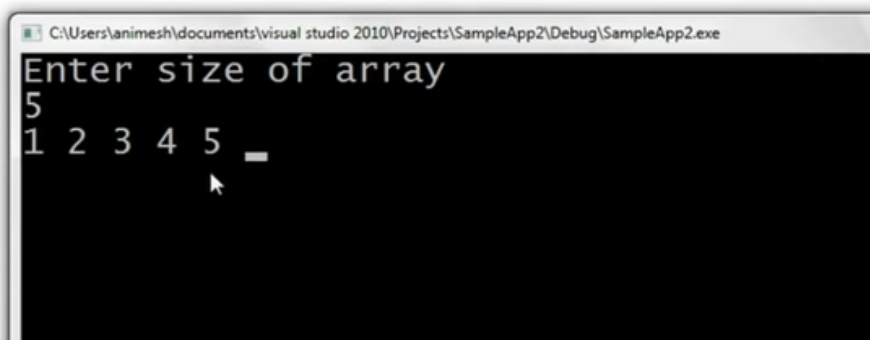


free

```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;
    {
        printf("%d ",A
    }
}
```




```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]);
    }
}
```

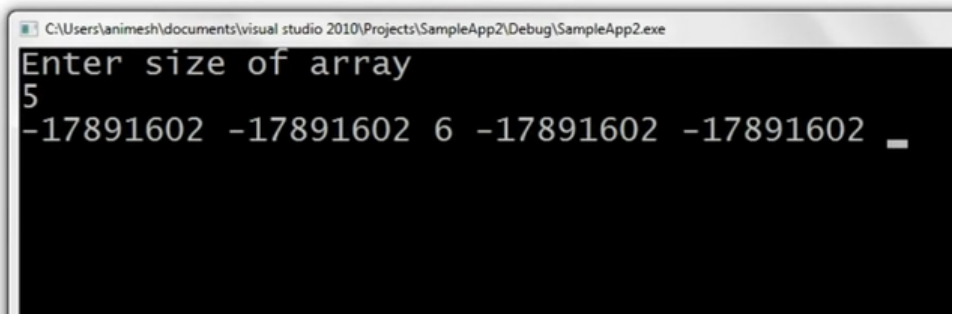


Note: even though we are freeing the memory here, we are able to access that, the value at the particular memory location (very dangerous thing about pointers)

you should read and write to that addresses only if the address is llocated to you.

check this even after freeing the memory

```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]);
    }
}
```

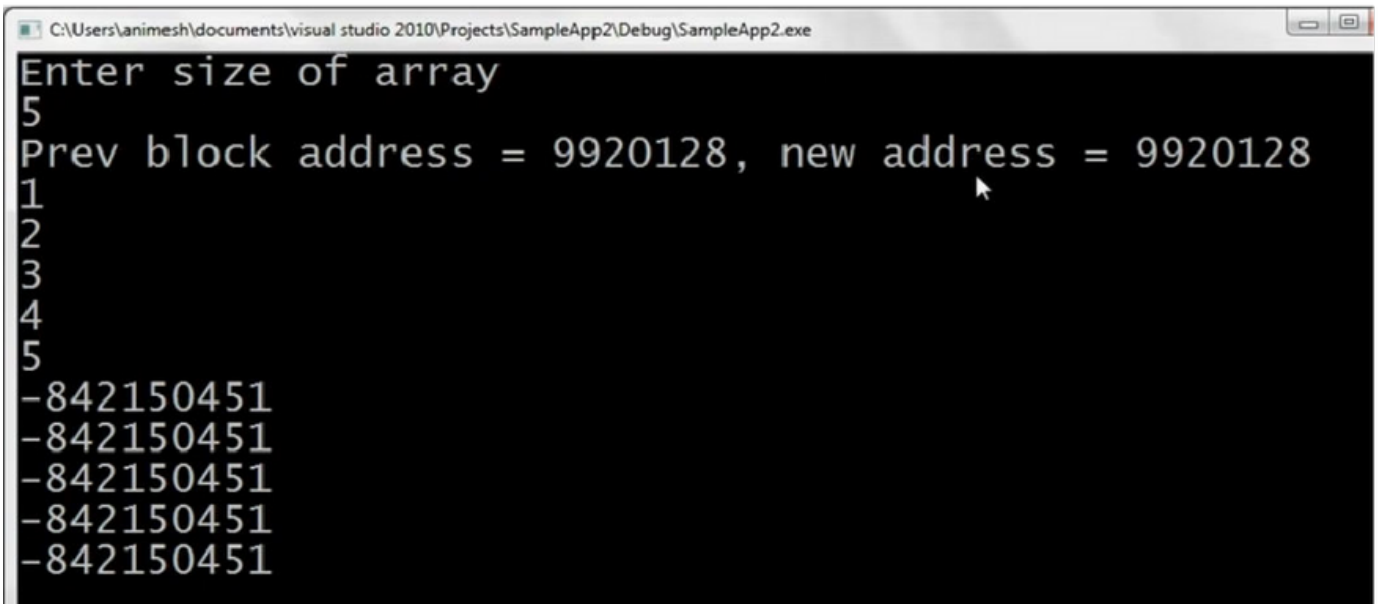


so don't forget null

```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 = NULL; // After free , adjust pointer to NULL
    for(int i = 0;i<n;i++)
    {
        printf("%d ",A[i]);
    }
}
```

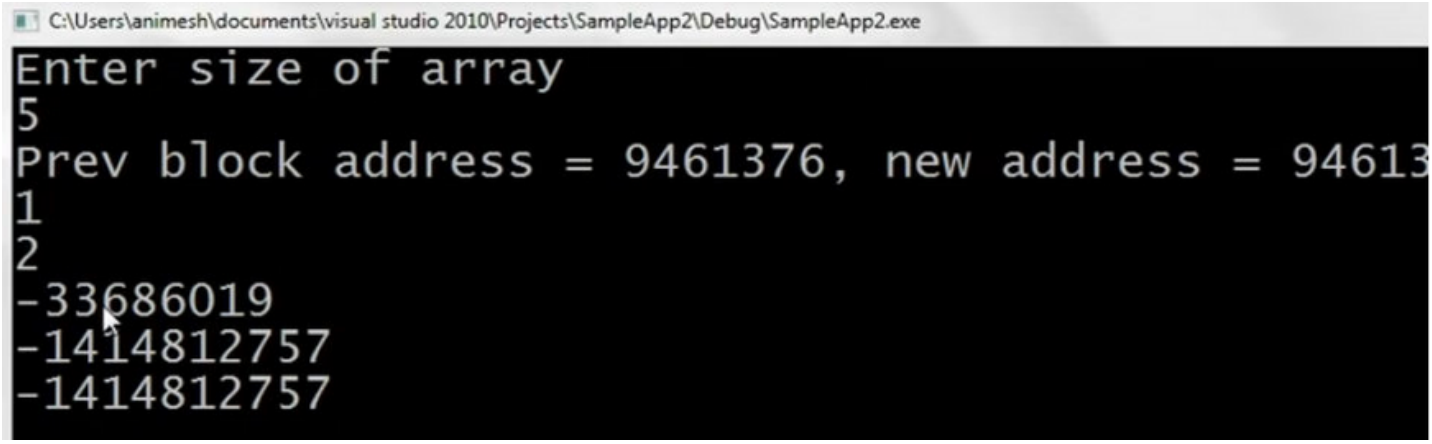
realloc

```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;
    }
    int *B = (int*)realloc(A, 2*n*sizeof(int));
    printf("Prev block address = %d, new address = %d\n",A,B);
    for(int i = 0;i<2*n;i++)
    {
        printf("%d\n",B[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;
    }
    int *B = (int*)realloc(A, (n/2)*sizeof(int));
    printf("Prev block address = %d, new address = %d\n",A,B);
    for(int i = 0;i<2*n;i++)
    {
        printf("%d\n",B[i]);
    }
}
```

deallocate the memory of 3 elements



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

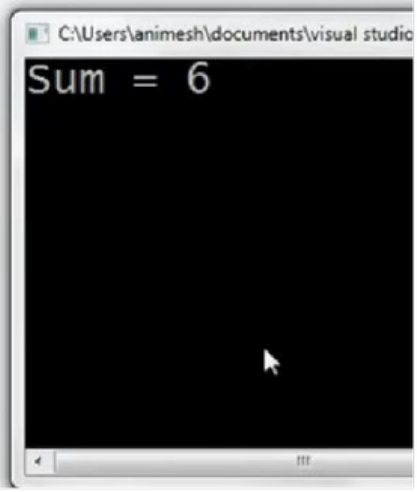
```c
int main()
{
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);
    int *A = (int*)malloc(n*sizeof(int)); //dynamically allocated array
    for(int i =0;i<n;i++)
    {
        A[i] = i+1;
    }
    int *A = (int*)realloc(A, 0); //equivalent to free(A)
    printf("Prev block address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

we can pass the first element as NULL = equivalent to calling malloc = new block dosn't copy any thing

```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;
    }
    int *B = (int*)realloc(NULL, n*sizeof(int)); //equivalent to malloc
    printf("Prev block address = %d, new address = %d\n",A,B);
    for(int i = 0;i<n;i++)
    {
        printf("%d\n",B[i]);
    }
}
```

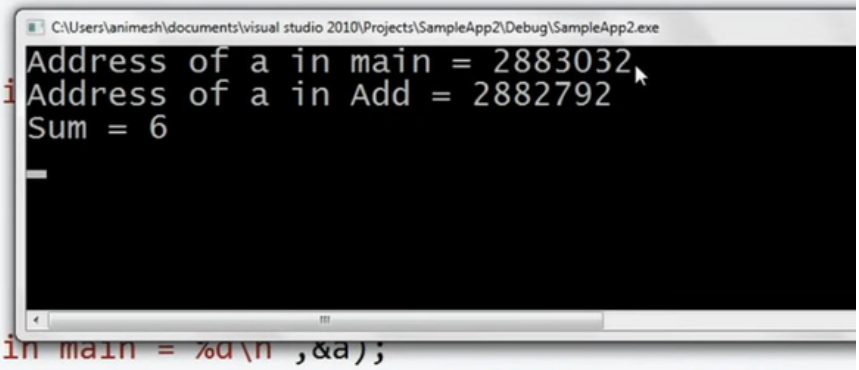
Pointers as function returns in C/C++

```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int Add(int a,int b){
    int c = a+b;
    return c;
}
int main() {
    int x = 2, y =4;
    int z = Add(x,y);
    printf("Sum = %d\n",z);
}
```

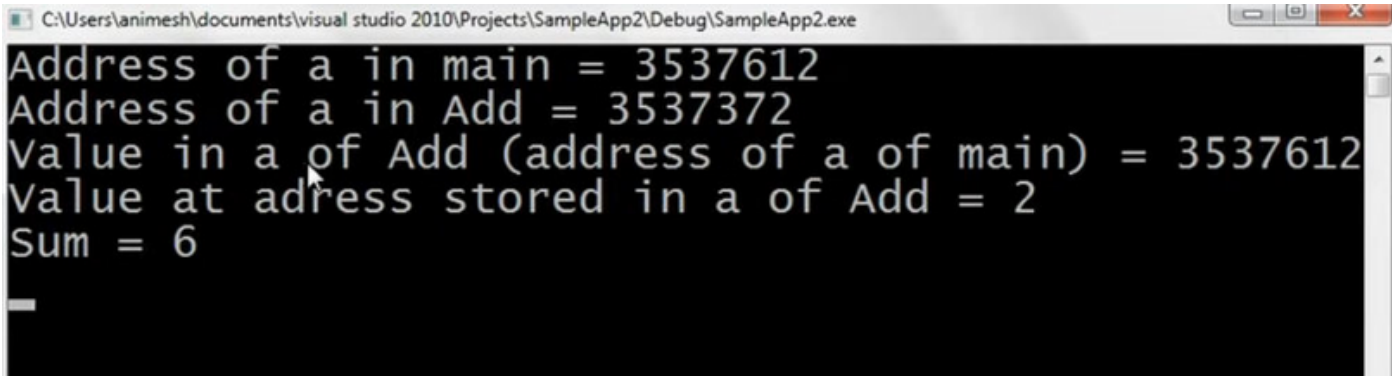



```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int Add(int a,int b){ // Called funtion
    printf("Address of a in Add = %d\n",&a);
    int c = a+b;
    return c;
}
int main() {  //Calling function
    int a = 2, b =4;
    printf("Address of a in main = %d\n",&a);
    //Call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

```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int Add(int a,int b){
    printf("Address of a i
    int c = a+b;
    return c;
}
int main() {
    int a = 2, b =4;
    printf("Address of a in main = %d\n",&a);
    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);
}
```

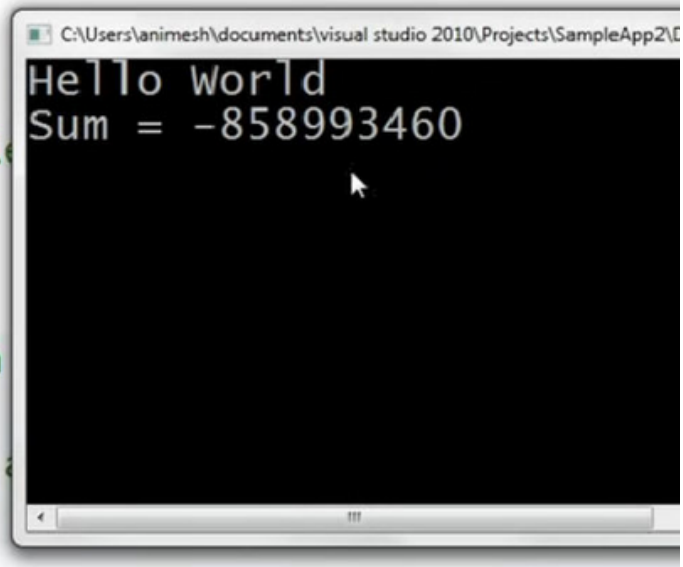



```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int Add(int* a,int* b){ // Called funtion
    // a and b are pointer to integers local to Add
    printf("Address of a in Add = %d\n",&a);
    printf("Value in a of Add (address of a of main) = %d\n",a);
    printf("Value at adress stored in a of 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 reference
    int c = Add(&a,&b); // a and b are integers local to Main
    printf("Sum = %d\n",c);
}
```

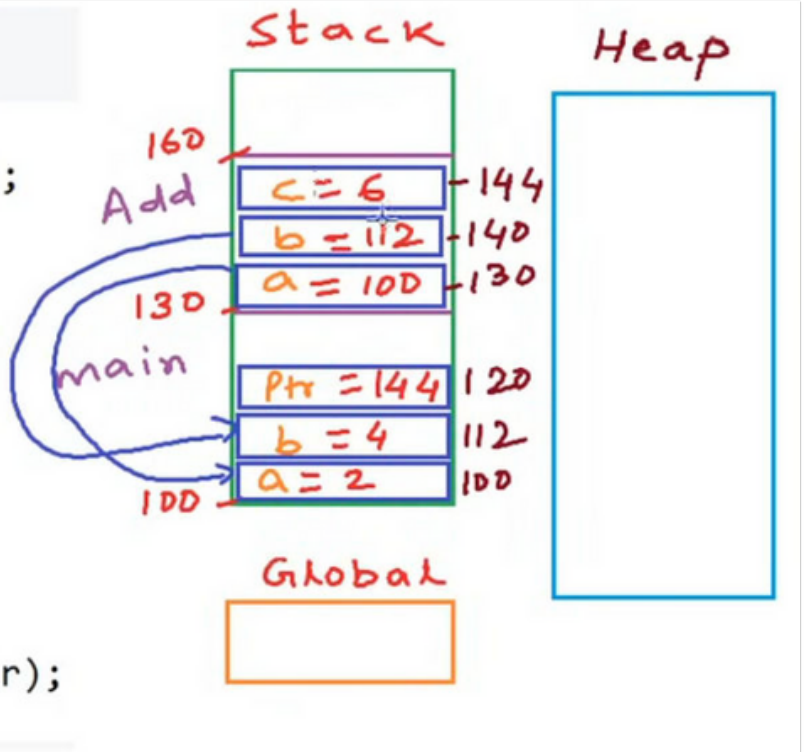



```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
int *Add(int* a,int* b){ // Called funtion  - returns pointer to integer
    int c = (*a) + (*b);
    return &c;
}
int main() {  //Calling function
    int a = 2, b =4;
    int* ptr = Add(&a,&b); // a a
    printf("Sum = %d\n",*ptr);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

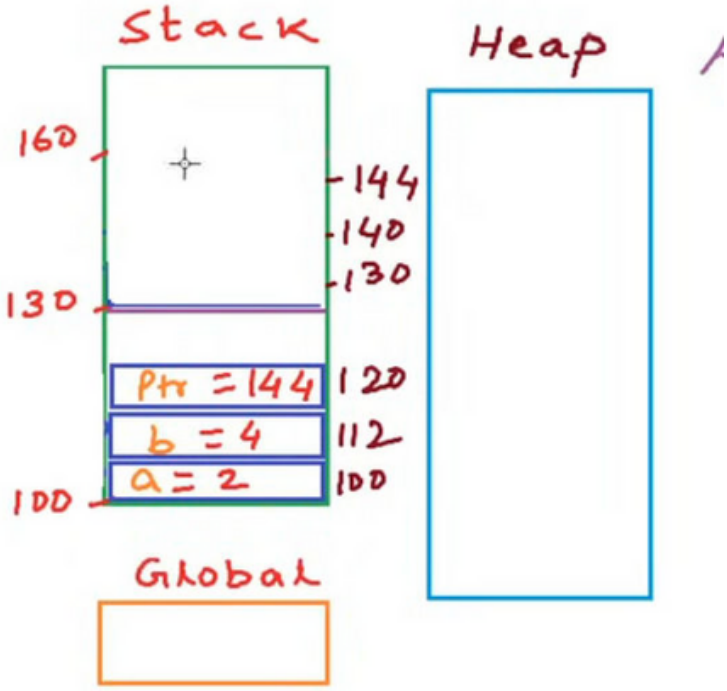
```c
// Pointers as function returns
#include<stdio.h>
#include<stdlib.h>
void PrintHelloWorld(){
    printf("Hello World\n");
}
int *Add(int* a,int* b){ // Calle
    int c = (*a) + (*b);
    return &c;
}
int main() {  //Calling function
    int a = 2, b =4;
    int* ptr = Add(&a,&b); // a
    PrintHelloWorld();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




```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);
}
```

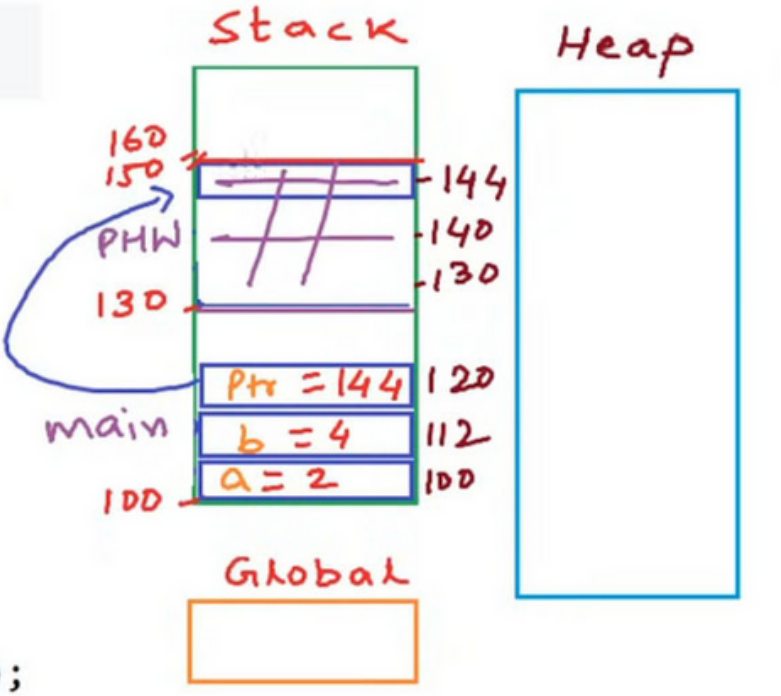


```c
#include<stdio.h>
#include<stdlib.h>
void PrintHelloWorld(){
    printf("Hello World\n");
}
int *Add(int* a,int* b){
    int c = (*a) + (*b);
    return &c;
}
int main() {
  ✓int a = 2, b =4;
  ✓int* ptr = Add(&a,&b);
    PrintHelloWorld();
    printf("Sum = %d\n",*ptr);
}
```



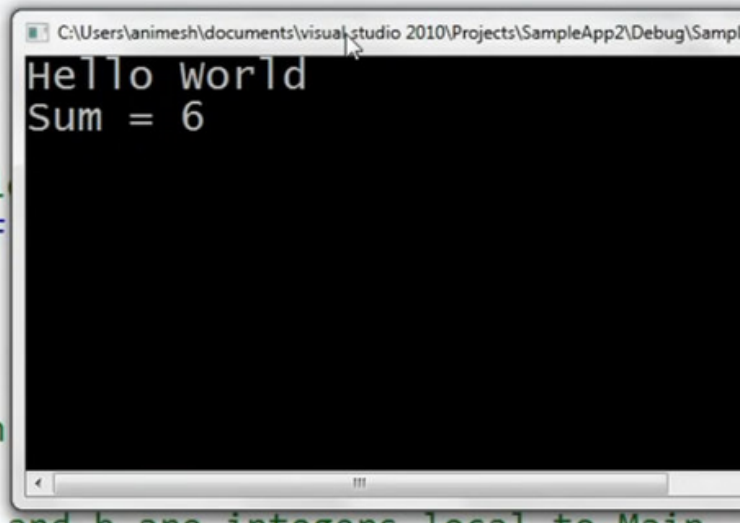
over written by print function

```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);
}
```

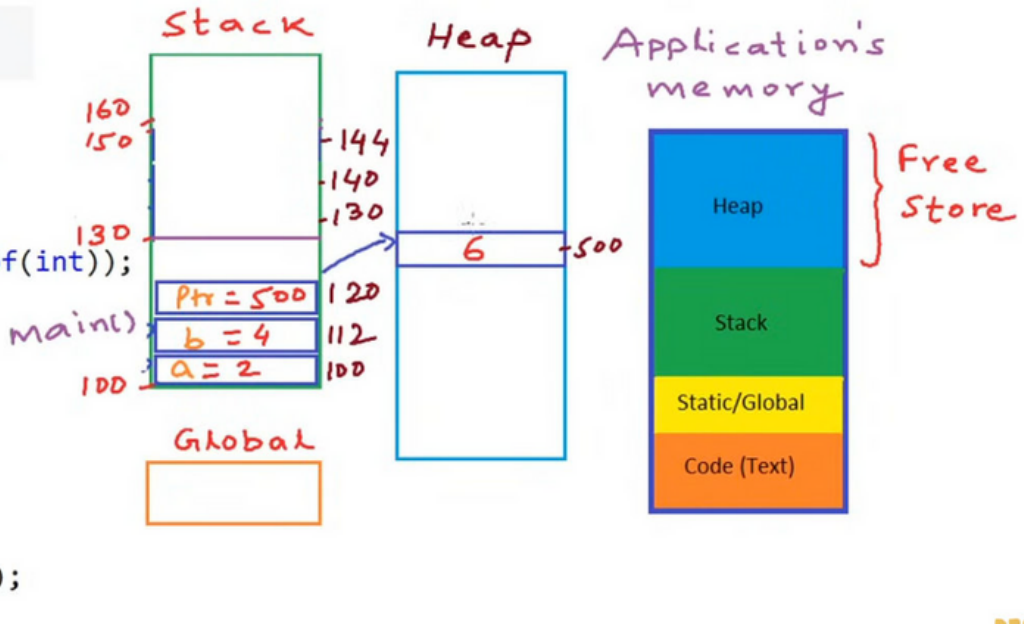


using malloc

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

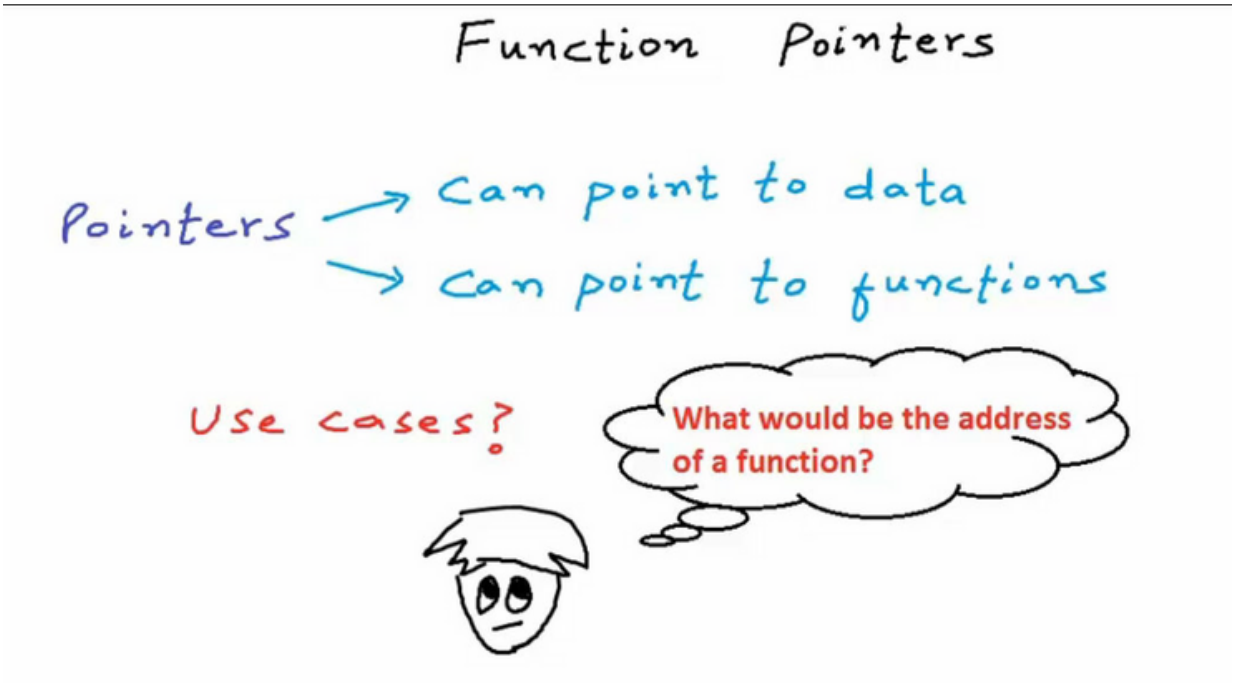


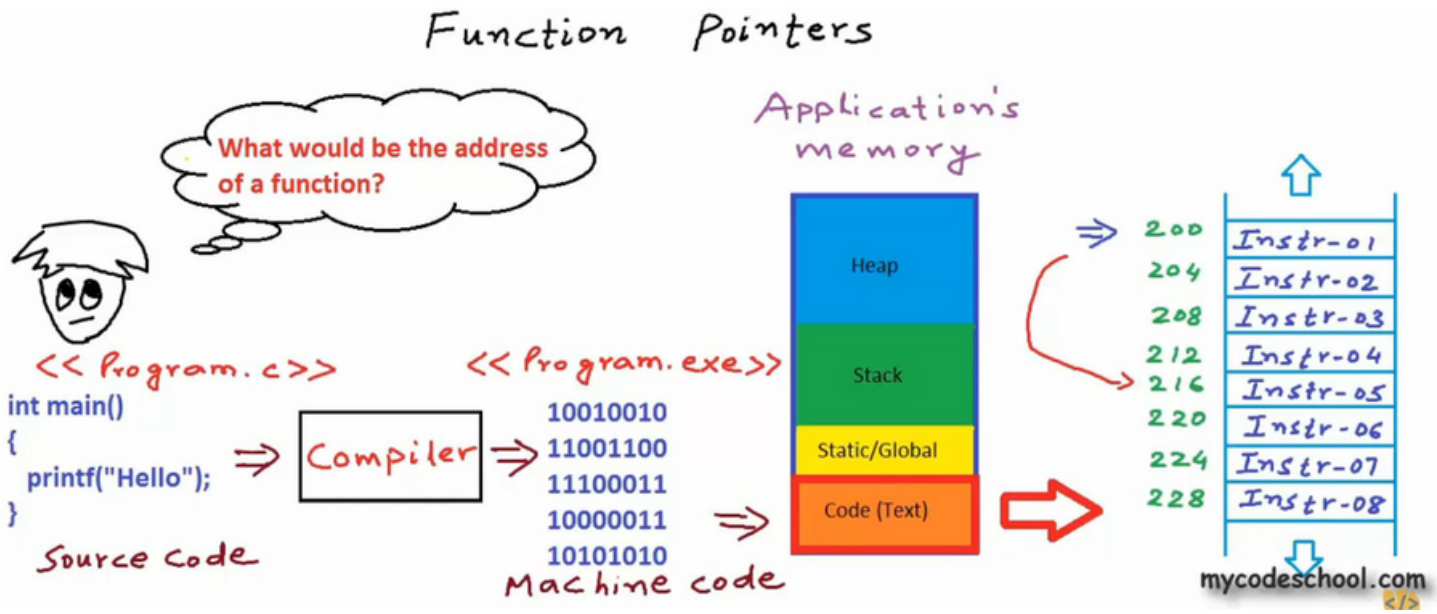

```c
#include<stdio.h>
#include<stdlib.h>
void PrintHelloWorld(){
    printf("Hello World\n");
}
int *Add(int* a,int* b){
  int* c = (int*)malloc(sizeof(int));
  *c = (*a) + (*b);
  return c;
}
int main() {
  int a = 2, b =4;
  int* ptr = Add(&a,&b);
  PrintHelloWorld();
  printf("Sum = %d\n",*ptr);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

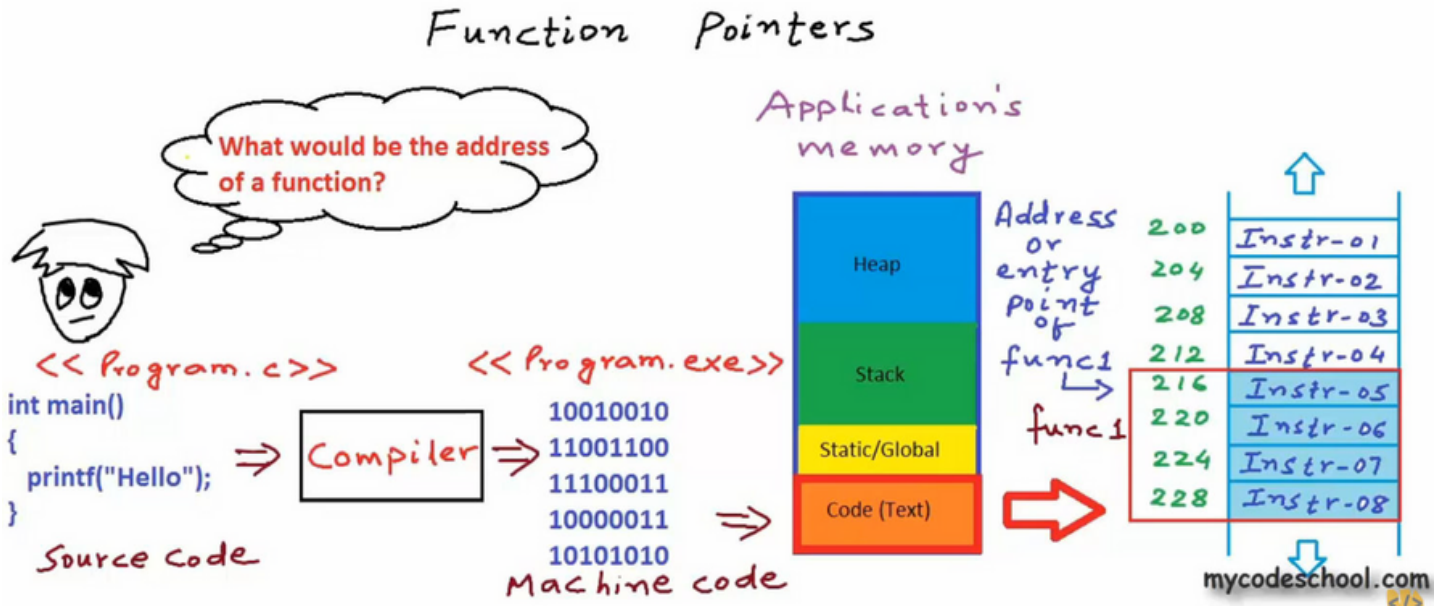


you need to explicitly deallocated

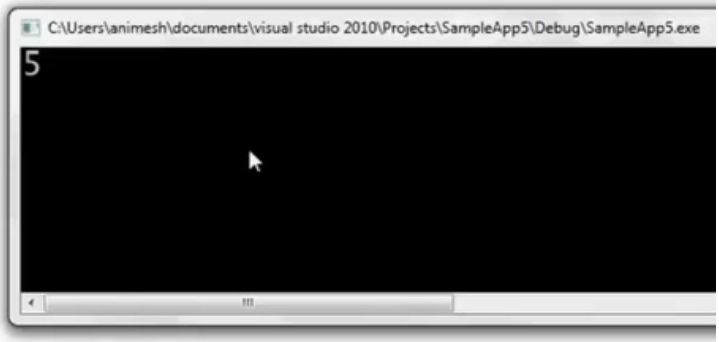
Function Pointers in C / C++



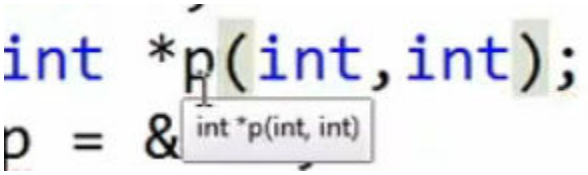





```c
//Function Pointers in C/C++
#include<stdio.h>
int Add(int a,int b)
{
    return a+b;
}
int main()
{
    int c;
    int (*p)(int,int);
    p = &Add;
    c = (*p)(2,3); //de-referencing and executing the function.
    printf("%d",c);
}
```



If we wouldn't use parentheses, then it will mean that we are declaring function that will return pointer to integer



And if I would write a declration like this, then that would return pointer to integer

```c
//Function Pointers in C/C++
#include<stdio.h>
int Add(int a,int b)
{
    return a+b;
}
int *Func(int,int); //declaring a function that would return int*
int main()
{
    int c;
    int (*p)(int,int);
    p = &Add;
    c = (*p)(2,3); //de-referencing and executing the function.
    printf("%d",c);
}
```

and this is declaring a function pointer

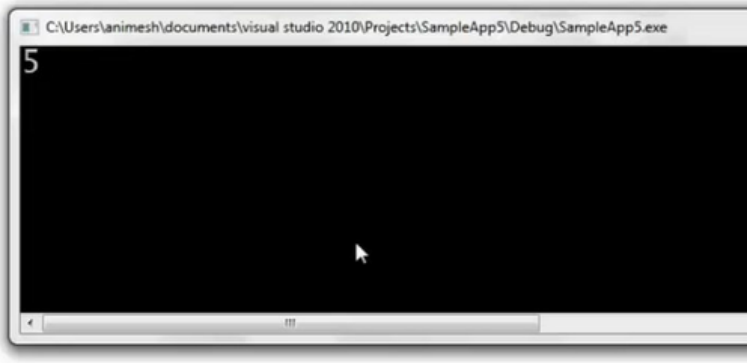
```c
//Function Pointers in C/C++
#include<stdio.h>
int Add(int a,int b)
{
    return a+b;
}
int (*Func)(int,int); //declaring a function pointer
int main()
{
    int c;
    int (*p)(int,int);
    p = &Add;
    c = (*p)(2,3); //de-referencing and executing the function.
    printf("%d",c);
}
```

function name also return address of the function or in other words, an appropriate pointer.

```c
int c;
int (*p)(int,int);
p = Add; // function name will return us pointer
```

To dereference, instead of using this parentheses and * operator with function pointer name, we can simply use the function pointer name

```c
//Function Pointers in C/C++
#include<stdio.h>
int Add(int a,int b)
{
    return a+b;
}
int main()
{
    int c;
    int (*p)(int,int);
    p = Add; // function name will return us pointer
    c = p(2,3); //de-referencing and executing the function.
    printf("%d",c);
}
```



One final thing, to be able to a function type of the function pointer must be appropriate

compilation error

```c
//Function Pointers in C/C++
#include<stdio.h>
int Add(int a,int b)
{
    return a+b;
}
int main()
{
    int c;
    void (*p)(int,int);
    p = Add; // function name will return us pointer
    c = p(2,3); //de-referencing and executing the function.
    printf("%d",c);
}
```

```c
//Function Pointers in C/C++
#include<stdio.h>
void PrintHello()
{
    printf("Hello\n");
}
int Add(int a,int b)
{
    return a+b;
}
int main()
{
    void (*ptr)();
    ptr = PrintHello;
    ptr();
}
```

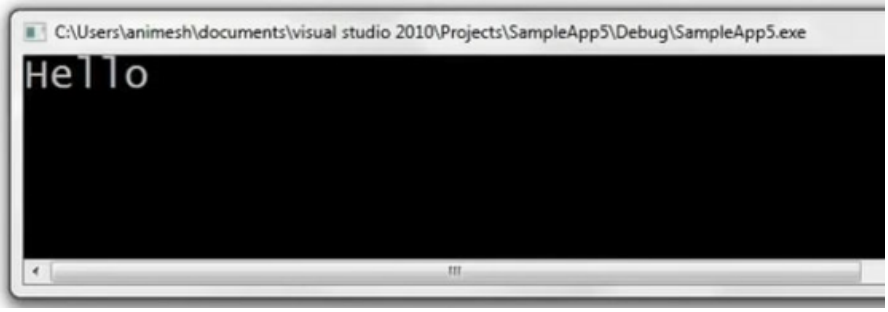

```c
//Function Pointers in C/C++
#include<stdio.h>
void PrintHello(char *name)
{
    printf("Hello %s\n",name);
}
int Add(int a,int b)
{
    return a+b;
}
int main()
{
    void (*ptr)(char*);
    ptr = PrintHello;
    ptr("Tom");
}
```

Function pointers and callbacks

Function pointer can be passed as arguments to functions, and then a function that would receive a function pointer as arguments can call back the function that this pointer will point to.

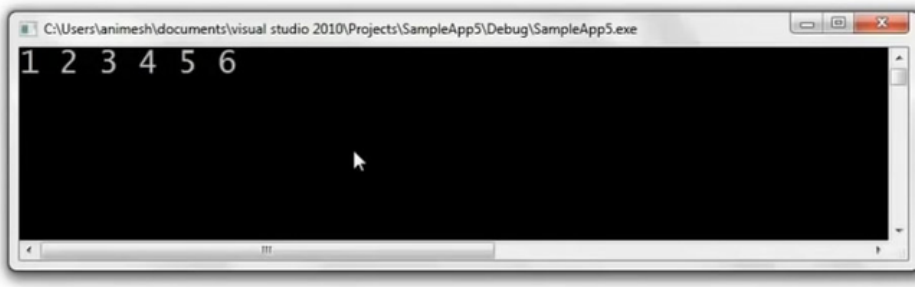
```
//Function Pointers and callbacks
#include<stdio.h>
void A()
{
    printf("Hello");
}
void B(void (*ptr)()) // function pointer as argument
{
    ptr(); //call-back function that "ptr" points to
}
int main()
{
    void (*p)() = A;
    B(p);
}
```

or




```
#include<stdio.h>
void A()
{
    printf("Hello");
}
void B(void (*ptr)()) // function pointer as argument
{
    ptr(); //call-back function that "ptr" points to
}
int main()
{
    B(A); // A is callback function.
}
```

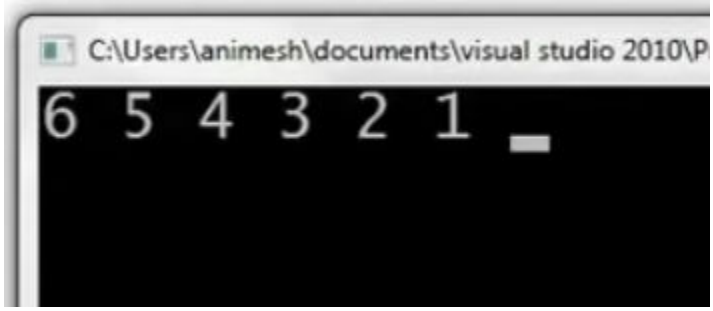
```
void BubbleSort(int *A,int n) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(A[j] > A[j+1]) { //compare A[j] with A[j+1] and SWAP if needed
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={3,2,1,5,6,4};
    BubbleSort(A,6);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```



let's say I want to sort my list in decreasing order of the value of integers so what change should go in my code here?

```
void BubbleSort(int *A,int n) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(A[j] < A[j+1]) { //compare A[j] with A[j+1] and SWAP if needed
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={3,2,1,5,6,4};
    BubbleSort(A,6);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```

mycodeschool.



Requiremets:

I want to sort a list of integers in increasing order and sometimes I want to sort a list in decreasing order of the value of integers.

So, what all can I do?

2 sort functions (duplicate code = same function except < or > only one character)

So can we do something better?

we can pass a flag to select

```
void BubbleSort(int *A,int n,int flag) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(A[j] < A[j+1]) { //compare A[j] with A[j+1] and SWAP if needed
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={3,2,1,5,6,4};
    BubbleSort(A,6);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```

Use a function pointer

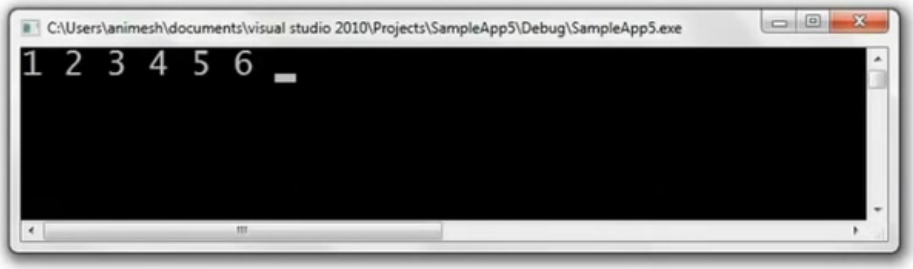
Sorting of a list is always done on basis of some ranking mechanism so based on some property we should be able to compare elements and say that this should come first and this should come later

I will perform the comparison through a function

```c
#include<stdio.h>
// callback function should compare two integers, should return 1 if first element has
// higher rank, 0 if elements are equal and -1 if second element has higher rank
void BubbleSort(int *A,int n,int (*compare)(int,int)) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(compare(A[j],A[j+1]) > 0) { //compare A[j] with A[j+1] and SWAP if needed
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={3,2,1,5,6,4};
    BubbleSort(A,6);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```

mycode

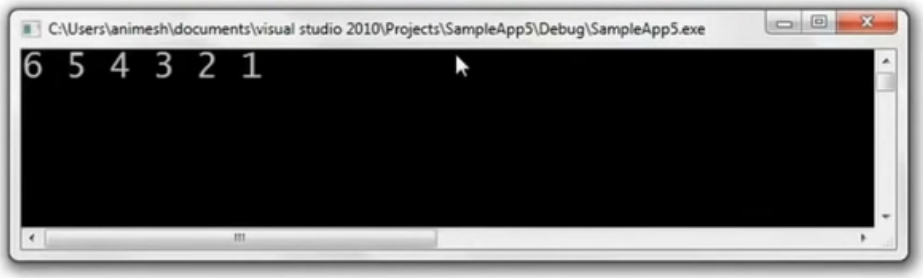
```c
#include<stdio.h>
int compare(int a,int b)
{
    if(a > b) return 1;
    else return -1;
}
void BubbleSort(int *A,int n,int (*compare)(int,int)) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(compare(A[j],A[j+1]) > 0) { //compare A[j] with A[j+1] and SWAP if needed
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={3,2,1,5,6,4};
    BubbleSort(A,6,compare);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```



mycodesch

The numbers are sorted in increasing order.

```c
#include<stdio.h>
int compare(int a,int b)
{
    if(a > b) return -1;
    else return 1;
}
void BubbleSort(int *A,int n,int (*compare)(int,int)) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(compare(A[j],A[j+1]) > 0) { //compare A[j] with A[j+1] and SWAP if needed
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={3,2,1,5,6,4};
    BubbleSort(A,6,compare);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```



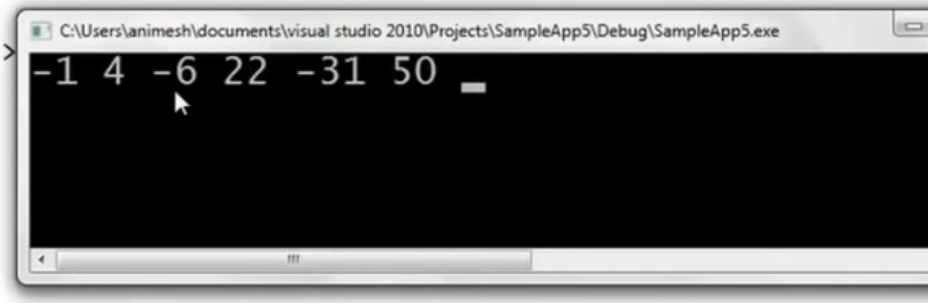
The array is now sorted in decreasing order of the value of integers.

mycodesch

```c
int main() {
    int i, A[] ={-31,22,-1,50,-6,4}; // {-1,4,-6,22,-31,50}
    BubbleSort(A,6,compare);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```

```c
#include<math.h>
int compare(int a,int b)
{
    if(a > b) return -1;
    else return 1;
}
int abosulte_compare(int a,int b)
{
    if(abs(a) > abs(b)) return 1;
    return -1;
}
```

```c
int absolute_compare(int a,int b)
{
    if(abs(a) > abs(b)) return 1;
    return -1;
}
void BubbleSort(int *A,int n,int (*compare)(int,int)) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(compare(A[j],A[j+1]) >
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={-31,22,-1,50,-6,4}; // => {-1,4,-6,22,-31,50}
    BubbleSort(A,6,absolute_compare);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```



The bubble sort function here can take only an array of integers, but we have a library function that can take any array

This library function is in stdlib.h and its named qsort = quick sort

to this function you should pass an array (can be any array, array of integers, char, or even a complex data types (structure))

second argument will be number of elements in array

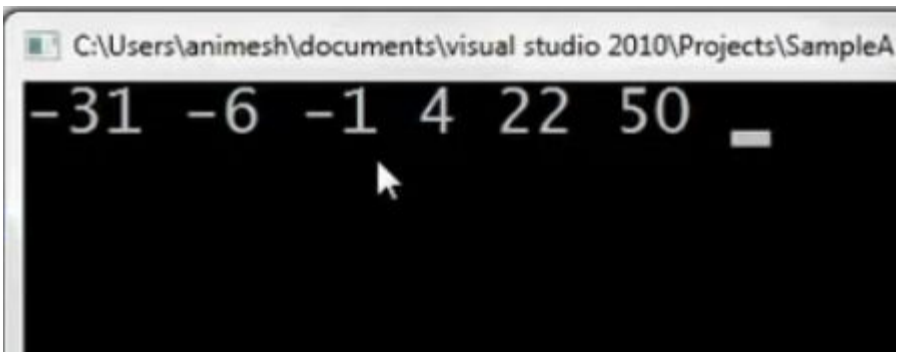
third argument will be size of the data type

last argument should be a function pointer to comparison function

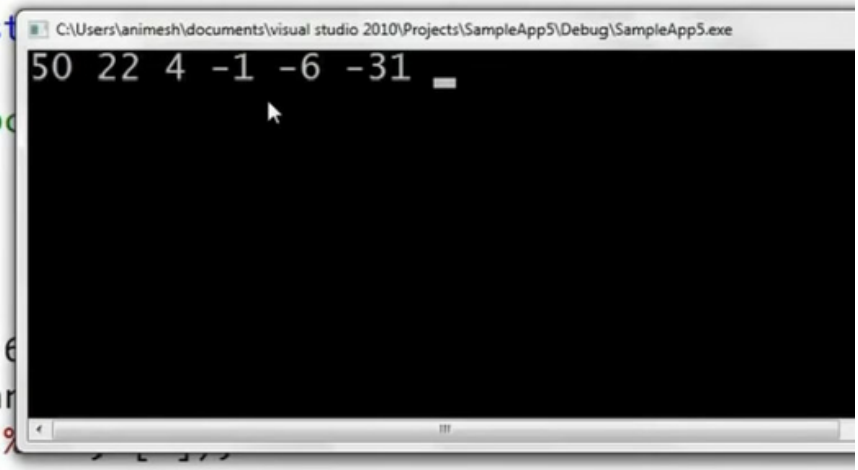
This comparison function must return any positive integer if A is ranked higher, a nigative integer if A is ranked lower and 0 if both are ranked same we can simply return A-B and it will be the same

So, basically this comparison function can be used to sort the array in increasing order of value of integers

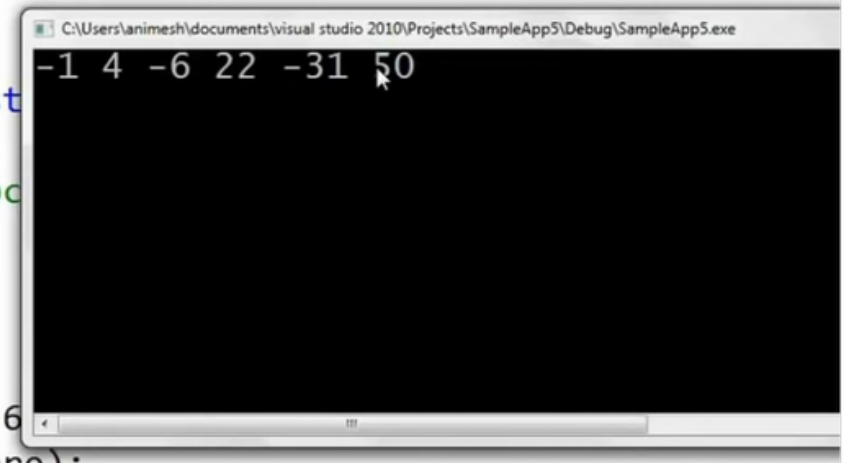
```c
//Function Pointers and callbacks
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int compare(const void* a, const void* b)
{
    int A = *((int*)a); // tyepcasting to int* and getting value
    int B = *((int*)b);
    return A-B;
}
int main() {
    int i, A[] ={-31,22,-1,50,-6,4}; // => {-1,4,-6,22,-31,50}
    qsort(A,6,sizeof(int),compare);
    for(i = 0;i<6;i++) printf("%d ",A[i]);
}
```



```c
//Function Pointers and callbacks
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int compare(const void* a, const
{
    int A = *((int*)a); // tyepc
    int B = *((int*)b);
    return B-A;
}
int main() {
    int i, A[] ={-31,22,-1,50,-6
    qsort(A,6,sizeof(int),compar
    for(i = 0;i<6;i++) printf("%
}
```

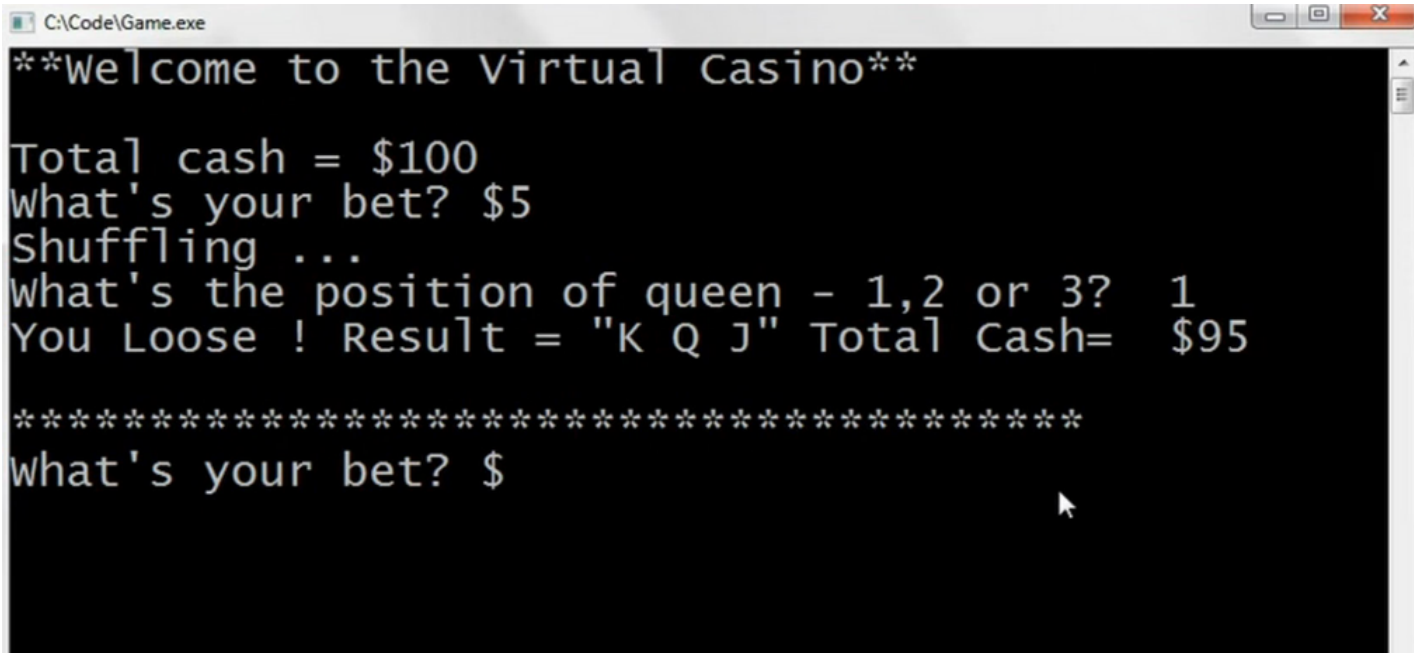


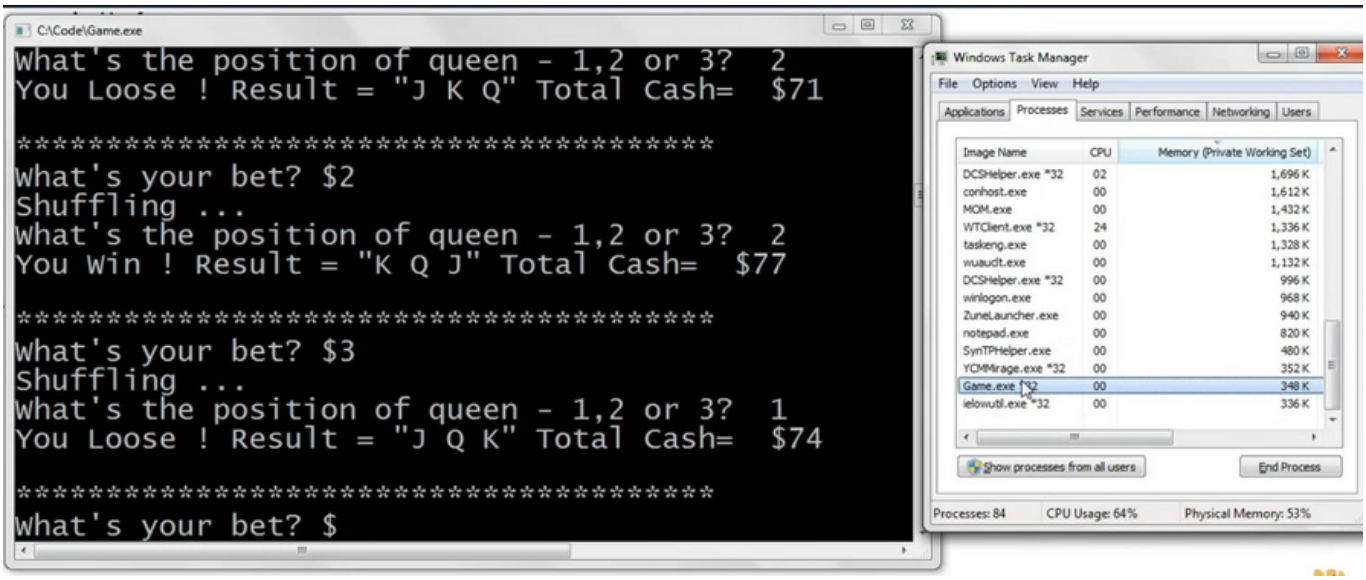

```c
//Function Pointers and callbacks
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int compare(const void* a, const
{
    int A = *((int*)a); // tyepc
    int B = *((int*)b);
    return abs(A) - abs(B);
}
int main() {
    int i, A[] ={-31,22,-1,50,-6
    qsort(A,6,sizeof(int),compare);
    for(i = 0;i<6;i++) printf("%d ",A[i]);
}
```



Memory leak in C/C++

```
#include<stdlib.h>
#include<time.h>
int cash = 100;
void Play(int bet){
    char C[3] = {'J','Q','K'};
    printf("Shuffling ...\n");
    srand(time(NULL)); // seeding random number generator
    int i;
    for(i = 0;i<5;i++)
    {
        int x = rand() % 3;
        int y = rand() % 3;
        int temp = C[x];
        C[x] = C[y];
        C[y] = temp; // swaps characters at position x and y
    }
    int playersGuess;
    printf("What's the position of queen - 1,2 or 3?  ");
    scanf("%d",&playersGuess);
    if(C[playersGuess - 1] == 'Q') {
        cash += 3*bet;
        printf("You Win ! Result = \"%c %c %c\" Total Cash=  $%d\n",C[0],C[1],C[2],cash);
    }
    else {
        cash -= bet;
        printf("You Loose ! Result = \"%c %c %c\" Total Cash=  $%d\n",C[0],C[1],C[2],cash);
    }
}
int main() {
    int bet;
    printf("**Welcome to the Virtual Casino**\n\n");
    printf("Total cash = $%d\n",cash);
    while(cash > 0 ) {
        printf("What's your bet? $");
        scanf("%d",&bet);
        if(bet == 0 || bet > cash) break;
        Play(bet);
        printf("\n*************************************\n");
    }
}
```

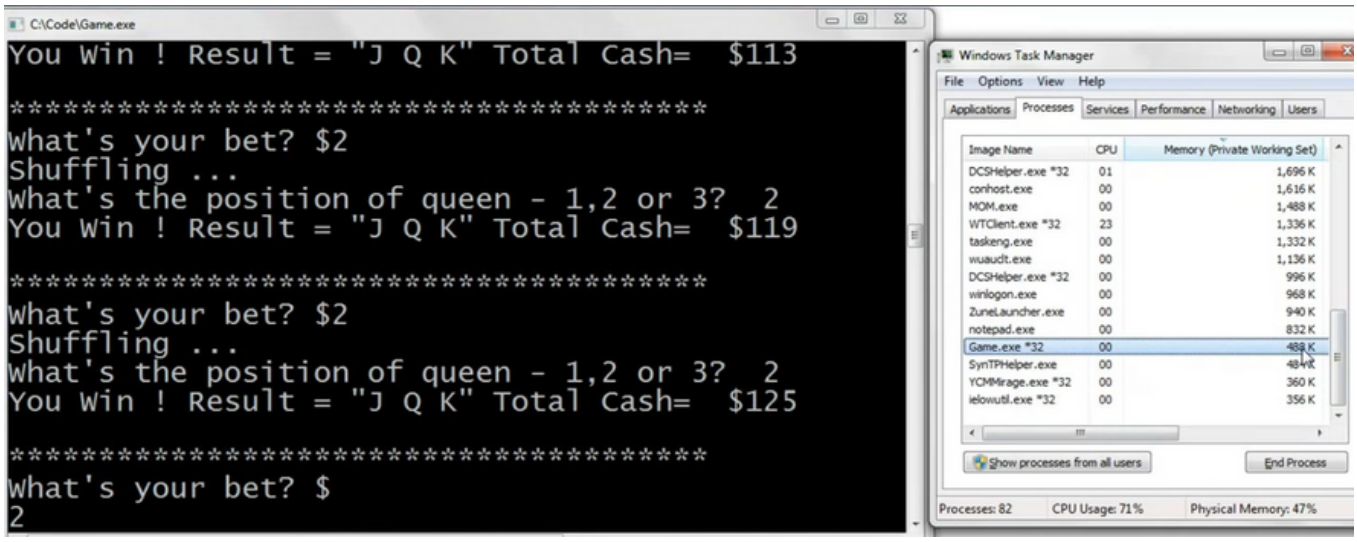




AND AS i GO AND PLAYING THE MEMORY IS NOT INCREASING ALWAYS 348K

```
void Play(int bet){
    char *C = (char*)malloc(3*sizeof(char)); // c++ :  char *C = new char[3];
    C[0] = 'J'; C[1] = 'Q'; C[2] = 'K';
    printf("Shuffling ...\n");
    srand(time(NULL)); // seeding random number generator
    int i;
    for(i = 0;i<5;i++)
    {
```

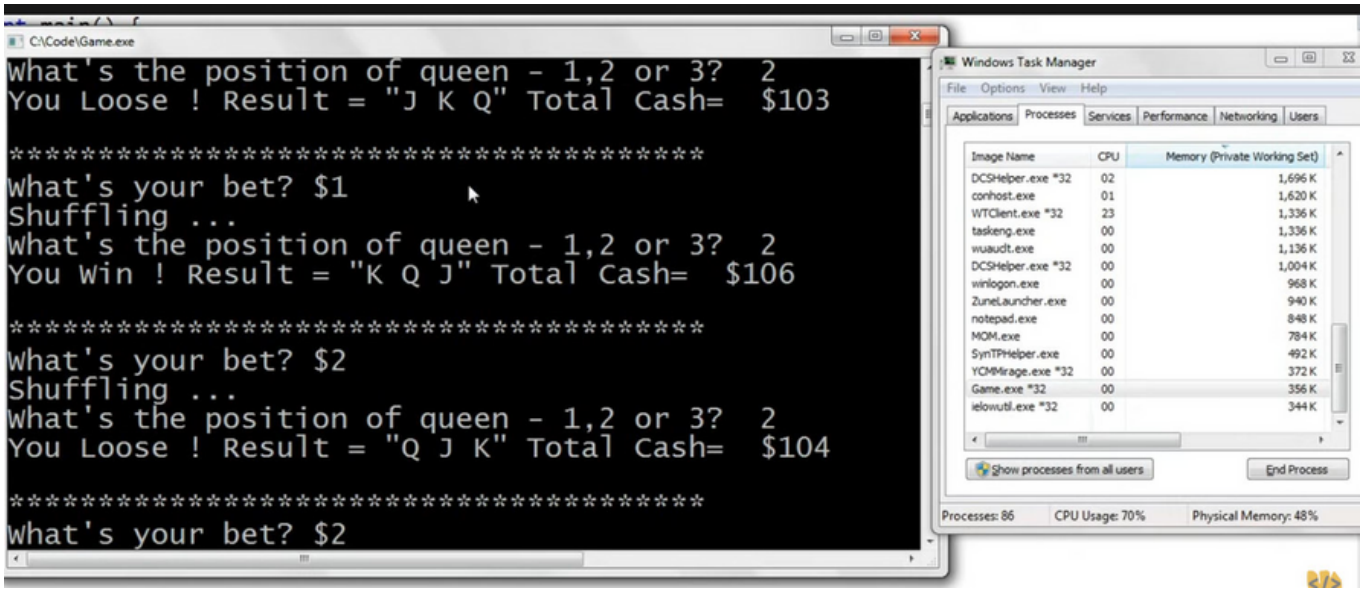
MEMORY INCREASE!



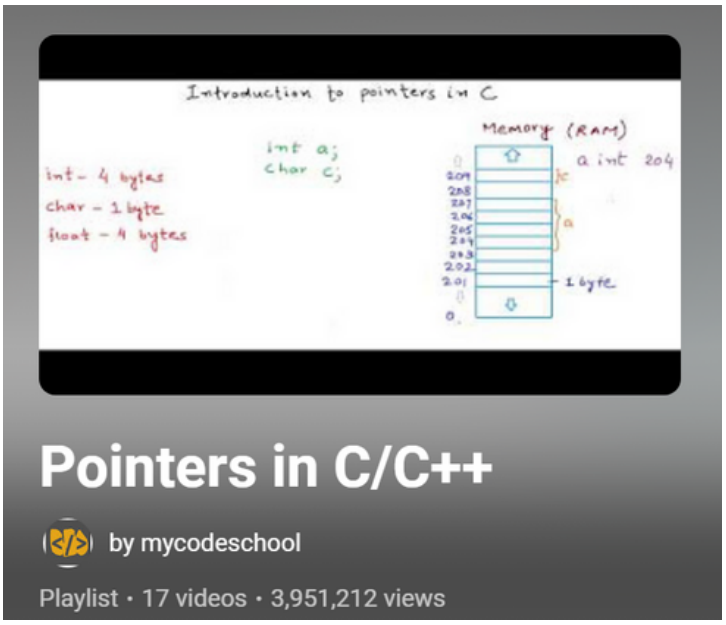
SOLUTION

```
void Play(int bet){
    char *C = (char*)malloc(10000*sizeof(char)); // c++ :  char *C = new char[3];
    C[0] = 'J'; C[1] = 'Q'; C[2] = 'K';
    printf("Shuffling ...\n");
    srand(time(NULL)); // seeding random number generator
    int i;
    for(i = 0;i<5;i++)  {
        int x = rand() % 3;
        int y = rand() % 3;
        int temp = C[x];
        C[x] = C[y];
        C[y] = temp; // swaps characters at position x and y
    }
    int playersGuess;
    printf("What's the position of queen - 1,2 or 3?  ");
    scanf("%d",&playersGuess);
    if(C[playersGuess - 1] == 'Q') {
        cash += 3*bet;
        printf("You Win ! Result = \"%c %c %c\" Total Cash=  $%d\n",C[0],C[1],C[2],cash);
    }
    else {
        cash -= bet;
        printf("You Loose ! Result = \"%c %c %c\" Total Cash=  $%d\n",C[0],C[1],C[2],cash);
    }
    free(C);
}
```

ALWAYS 356K


C:\Code\Game.exe
What's the position of queen - 1,2 or 3? 2
You Loose ! Result = "J K Q" Total Cash= $103
****
What's your bet? $1
Shuffling ...
What's the position of queen - 1,2 or 3? 2
You Win ! Result = "K Q J" Total Cash= $106
****
What's your bet? $2
Shuffling ...
What's the position of queen - 1,2 or 3? 2
You Loose ! Result = "Q J K" Total Cash= $104
****
What's your bet? $2
Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Image Name CPU Memory (Private Working Set)
DCSHelper.exe *32 02 1,696 K
conhost.exe 01 1,620 K
WTClient.exe *32 23 1,336 K
taskeng.exe 00 1,336 K
wuaudt.exe 00 1,136 K
DCSHelper.exe *32 00 1,004 K
winlogon.exe 00 968 K
ZuneLauncher.exe 00 940 K
notepad.exe 00 848 K
MOM.exe 00 784 K
SynTPHelper.exe 00 492 K
YCMMirage.exe *32 00 372 K
Game.exe *32 00 356 K
ielowutil.exe *32 00 344 K
Show processes from all users
End Process
Processes: 86 CPU Usage: 70% Physical Memory: 48%

Pointers in C/C++



Notes By linkedin/ahmedmagdyhafez/