Pointers

- ***Pointers***
  - ***Pointer Representation in Memory***
  - ***Working with Pointers (pointer arithmetic)***
  - ***Working with Pointers to pointer***
  - ***Call by value/ call by reference***
  - ***Pointers and Array***
  - ***Array as a function argument***
  - ***Character arrays and pointers***
  - ***Pointers and multi- dimensional arrays***
  - ***Function pointer***
  - ***Pointers and dynamic Memory***
    - ***Malloc***
    - ***Calloc***
    - ***Realloc***

Pointers

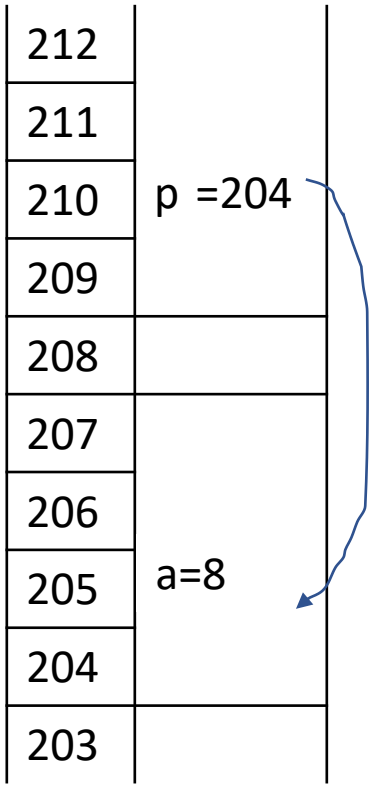
Suman Pandey

Pointer Representation in Memory

Pointers stores address of another variable

```
int a=5; /* compiler maintains a lookup table for all the variables
            in compiler lookup table it stores a and its address */

printf( "%d", a);  // this will print 5

int *p;
p=&a;
printf("%p", p); // this will print 204, why as pointer stores address of a
printf("%p", &a); // this will also print 204
printf("%p", &p); // this  will print 209
printf("%d", *p);
*p=8;
printf("%d", *p) ; // this will print 5 -> this concept is called dereferencing
print("%d", a);    // this will print 8
```

p -> address
**p -> value at the address*



Working with Pointers (pointer types)

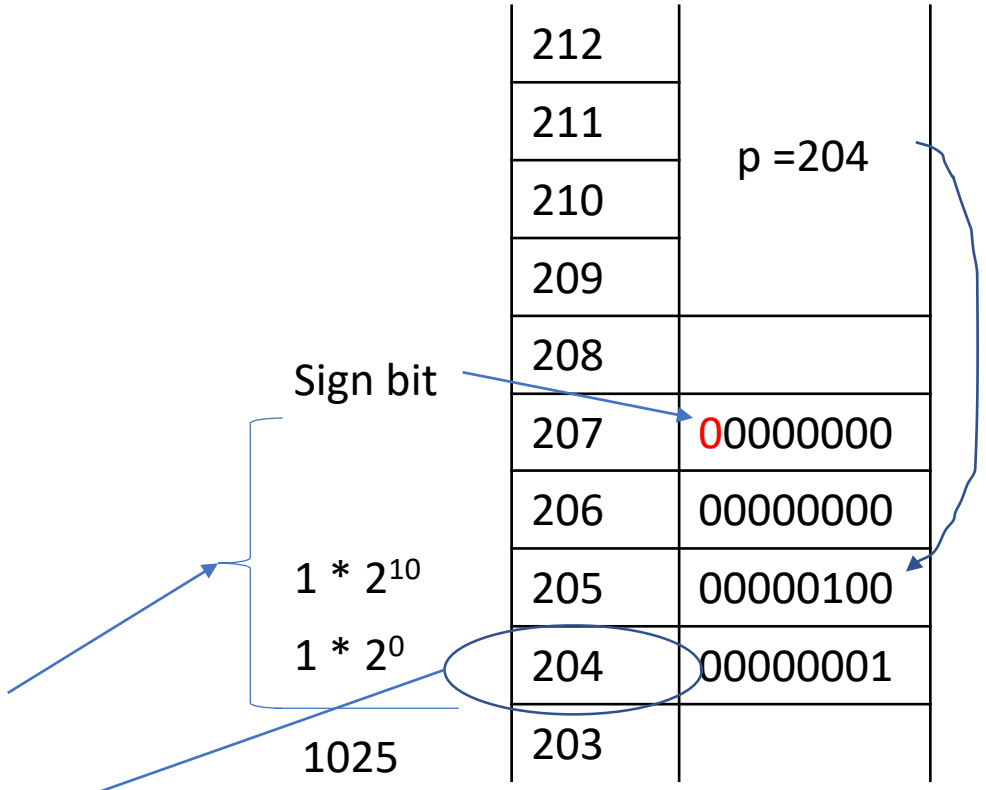
int -> int*

char -> char*

Why do we need strong types?

Why not generic type? Afterall pointer variables are only storing addresses.

Ans: because when we dereference, we are actually looking for values, hence the compiler needs to know how much data to access from memory.

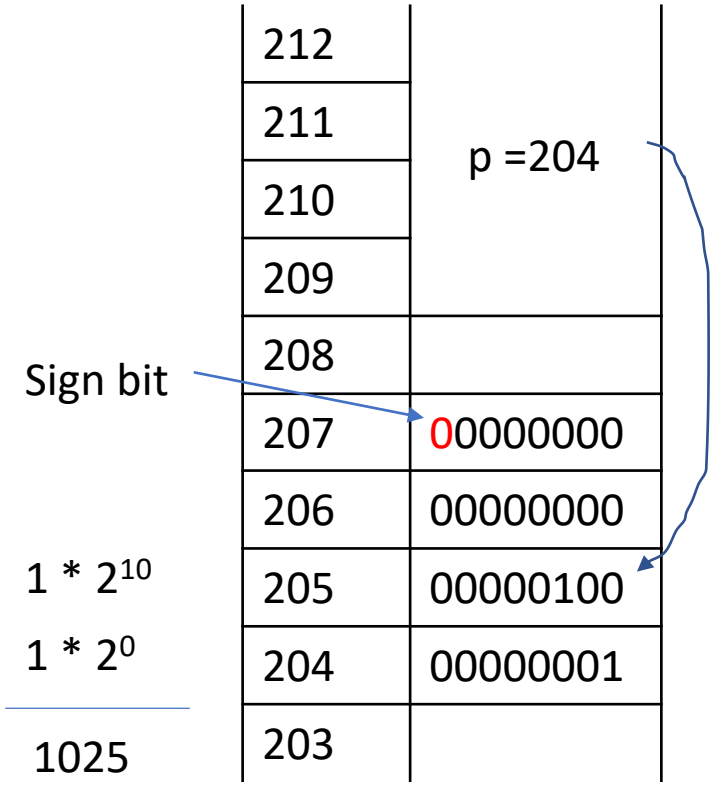



```
int a=1025;          // integer
int *p = &a;
printf("%p", p)  //204
printf( "%d", *p) // look at 4 bytes starting 204
printf("%p,%d", p, *p) //204, 1025
Char *p0; //size of char is 1 byte
p0 = (char*)p;  //typecasting  address 204 is stored here,
                // but when dereferenced *p0. then the machine says hey, p0 is a
                //pointer to character, and char is only 1 byte, hence I will look at
                //only 1 byte

printf("%p, %d", p0,*p0) //204 , 1
```

Working with Pointers (pointer arithmetic)

Only arithmetic allowed is to add/subtract a constant to a pointer

```
int a=1025;          // integer
int *p = &a;
printf("%p", p)  //204
printf("%d", *p) // look at 4 bytes starting 204
printf("%p, %d", p, *p) //204, 1025
printf("%p, %d", p+1, *(p+1)) //208, garbage value
// p+1 will print address of next 4 byte, and *(p+1) will print
the value at that address. because we have not assigned the
next 4 byte, it will most probably print a garbage.

char *p0; //size of char is 1 byte
p0 = (char*)p; //typecasting
printf("%p, %d", p0,*p0) //204 , 1
printf("%p , %d",  p0+1, *(p0+1)) //205, 4
Printf("%c , %c", *p0 , *(p0+1)) // this will print character
                                 //equivalent of these one bytes
```



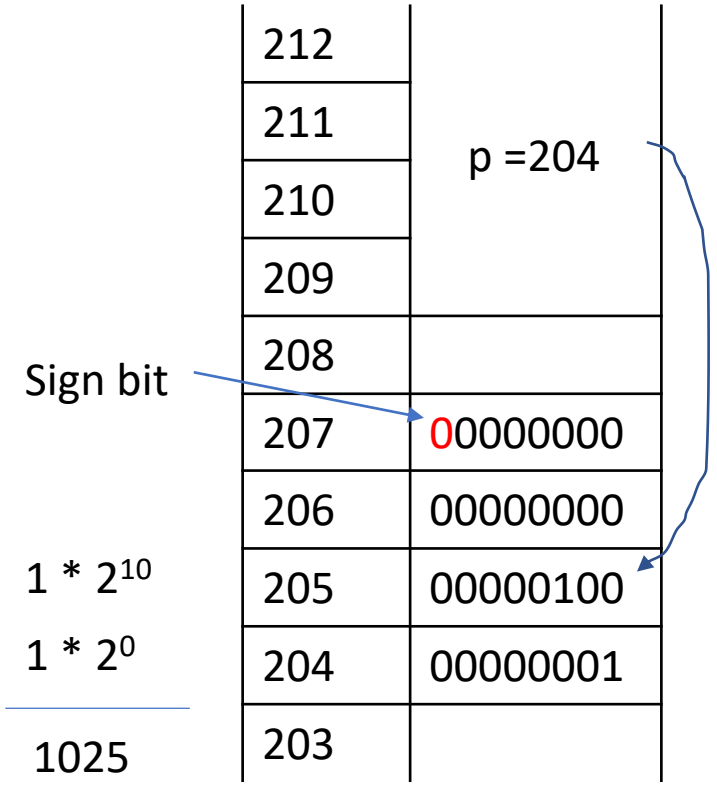
Working with Pointers (pointer arithmetic, void pointer)

Only arithmetic allowed is to add/subtract a constant to a pointer

int a=1025; // integer
*int *p = &a;*
printf("%p", p) //204
*printf("%d", *p) // look at 4 bytes starting 204*
*printf("%p , %d", p, *p) //204, 1025*
*printf("%p, "%d", p+1, *(p+1)) //208, garbage value*
*// p+1 will print address of next 4 byte, and *(p+1) will print the value at that address. because we have not assigned the next 4 byte, it will most probably print a garbage.*

*char *p0; //size of char is 1 byte*
p0 = (char)p; //typecasting*
*printf("%p, %d", p0,*p0) //204 , 1*
*printf("%p, %p", p0+1, *(p0+1)) //205, 4*

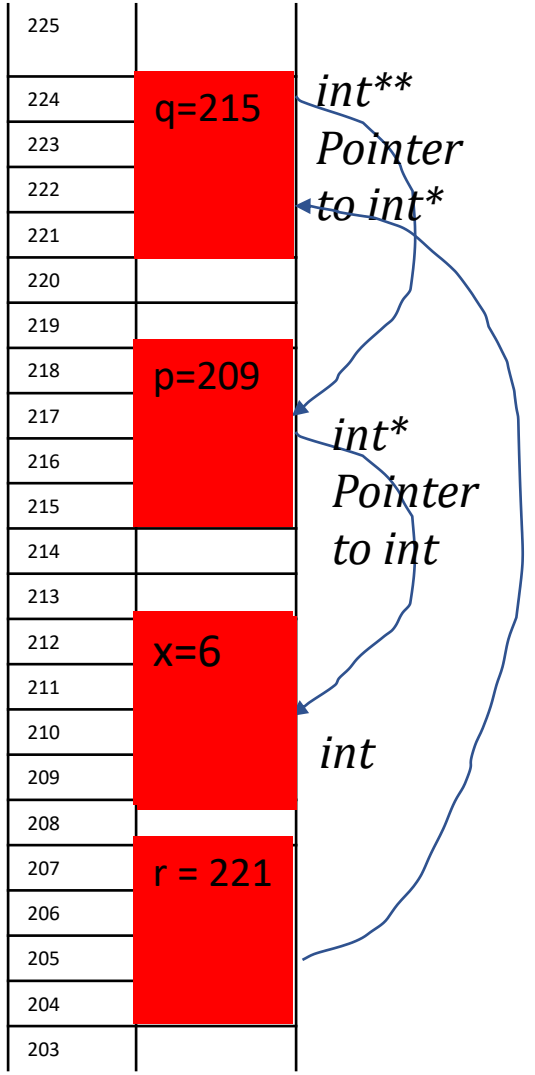
*void *p1;*
p1=p; //This is okai
*printf("%d", *p1) //this is not okai*



Working with Pointers to pointer

Only arithmetic allowed is to add/subtract a constant to a pointer

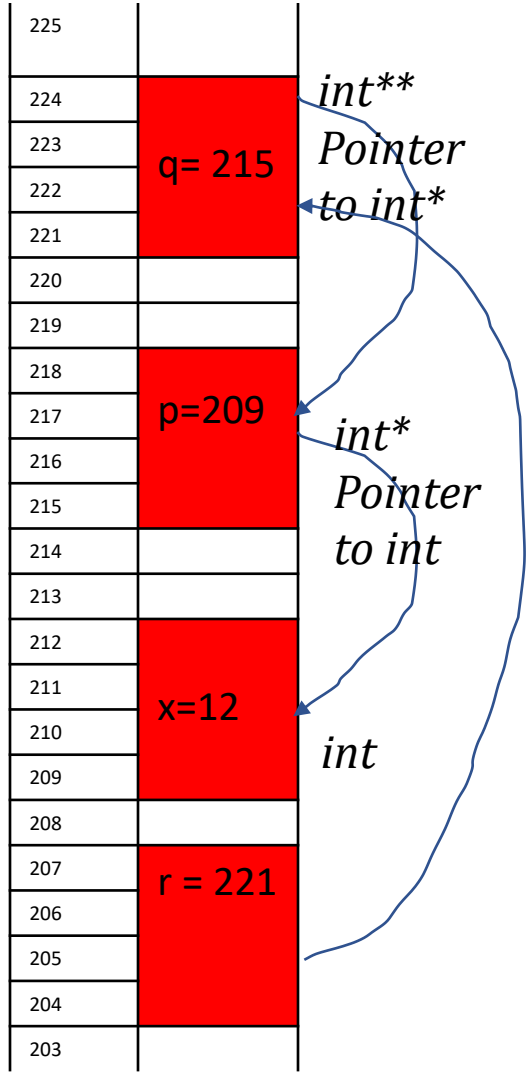
```
int x = 5;
int *p;     // pointer is also stored in 4 bytes
// we have int* for p because we also need to dereference
p=&x;
*p=6;
// Can we store the address of  pointer variable p?
//Ans: Yes
int **q; //
q = &p;
//We can go on like this
int ***r;
r = &q; //r can not store &p or &x , only r=&q will be valid
printf("%p", *p);       // 6 , value stored at the address in p
printf("%p",*q);        // 209, this will print the value stored at address p
printf("%d", *(*q)); // 6, first I will go to *q which is value stored at
                        address p, then I access the value stored at address 209, which is 6
printf(""%p", *r);          //215, value in stored at address q
printf("%p", *(*r));        //209, first go to value stored at address q, it has address 215, so
                            now go to value stored at address 215, which is 209.
printf("%d", *(*( *r)));    //6, first go to value stored at address q, it has address 215, then go to value stored at address 215, which is 209. then
                            again dereference one more time and go to value stored at address 209. which is 6. It dereferenced 3 times
```



Working with Pointers to pointer

Only arithmetic allowed is to add/subtract a constant to a pointer

```
int x = 5;
int *p;
p=&x;
*p=6;
int **q;
q = &p;
int ***r;
r = &q;
printf("%d", *p);
printf("%p",  *q);
printf("%d", *(*q));

printf("%p", *r);
printf("%p", *(*r));
printf( "%d", *(*( *r))) ;
***r=10    //chain of dereferencing
printf("%d", x);    //10
**q = *p + 2; //*p is also dereferencing x and **q is also dereferencing x
printf("%d", x)   //12
```



Call by Value

```
1  // Fig. 7.6: fig07_06.c
2  // Cube a variable using pass-by-value.
3  #include <stdio.h>
4
5  int cubeByValue(int n); // prototype
6
7  int main(void)
8  {
9     int number = 5; // initialize number
10
11    printf("The original value of number is %d", number);
12
13    // pass number by value to cubeByValue
14    number = cubeByValue(number);
15
16    printf("\nThe new value of number is %d\n", number);
17 }
18
19 // calculate and return cube of integer argument
20 int cubeByValue(int n)
21 {
22    return n * n * n; // cube local variable n and return result
23 }
```

When code execution is at line 21, memory looks like this

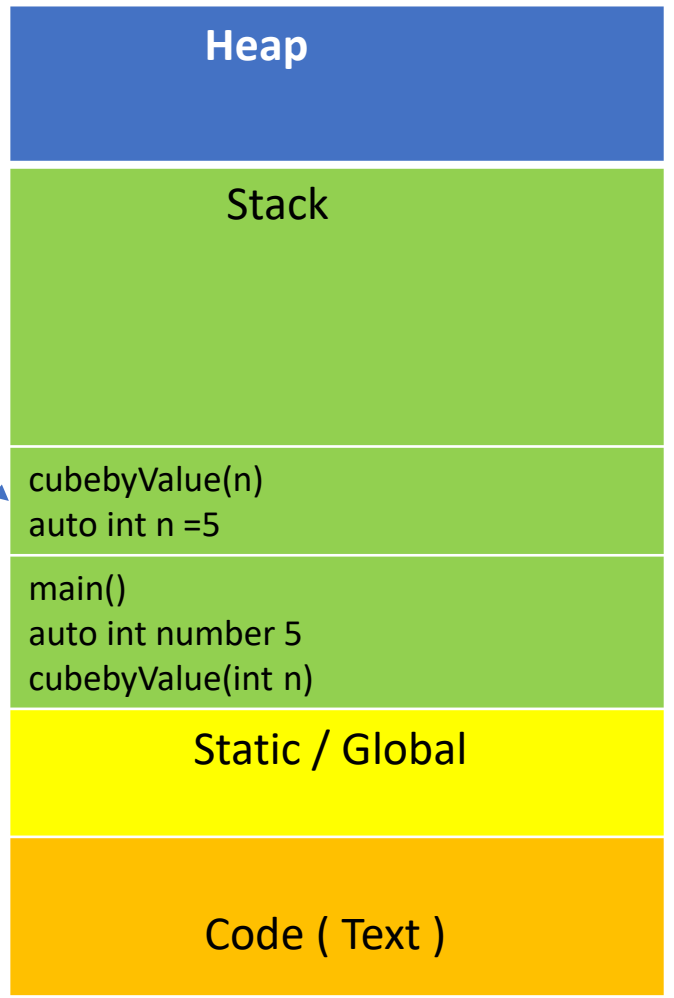


Fig. 7.6 | Cube a variable using pass-by-value. (Part 1 of 2.)

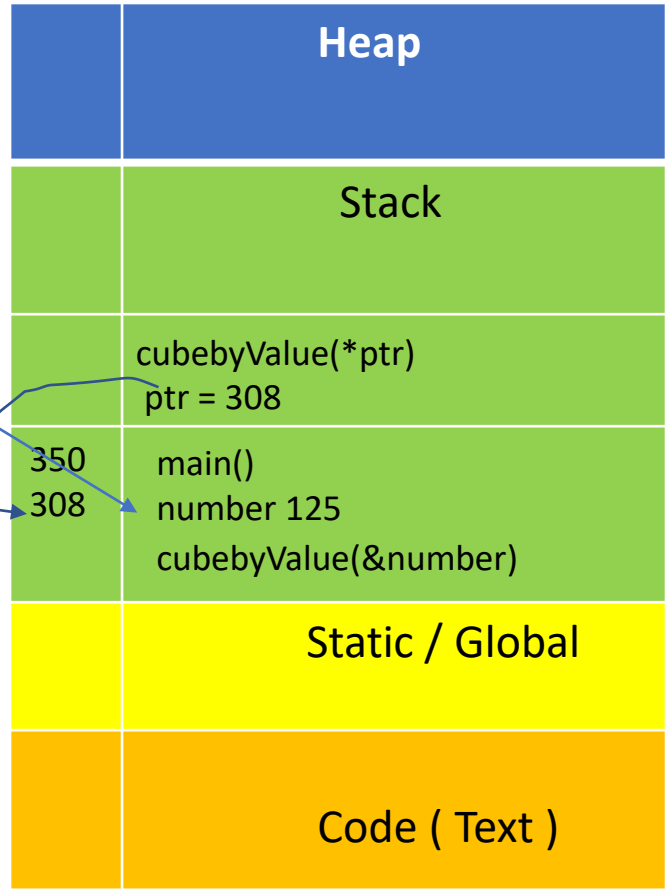
Call by reference

```
1  // Fig. 7.7: fig07_07.c
2  // Cube a variable using pass-by-reference with a pointer ar
3
4  #include <stdio.h>
5
6  void cubeByReference(int *nPtr); // function prototype
7
8  int main(void)
9  {
10    int number = 5; // initialize number
11
12    printf("The original value of number is %d", number);
13
14    // pass address of number to cubeByReference
15    cubeByReference(&number);
16
17    printf("\nThe new value of number is %d\n", number);
18 }
19
20 // calculate cube of *nPtr; actually modifies number in main
21 void cubeByReference(int *nPtr)
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 }
```

Instruction

*ptr = *ptr * *ptr * *ptr

Application Memory



ig. 7.7 | Cube a variable using pass-by-reference with a pointer argument. (Part

Pointers and Array

Pointers and array go together. There is strong relation

int A[5] int -> 4 bytes

*A ->5 *4 bytes*

A ={2,4,5,8,1}

int x=5;

*int *p*

p=&x

Print(p) //300

*Print (*p) // 5*

//pointer arithmetic

Print(p++) //304

Print((p++)) //garbage -> this is pointer arithmetic*

// Pointer arithmetic makes sense in the case of array

Print (A) //200, address of the first element of array

*Print(*A) // 2, dereferencing the first address will give first value*

/ For element at index i*

address -> & A[i] or A+i -> both mean same thing

*value -> A[i] or *(A+i) -> both mean same thing */*

for int i=0 ; i<5; i++

{ print(&A[i]);

print(A+i);

print(A[i]);

print((A+i));*

}

216	A[5]=1
212	A[3]=8
208	A[2]=5
204	A[1]=4
200	A[0]=2
300	X=5

Array as a function argument

Pointers and arrays go together. There is a strong relation

```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",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```

4 bytes 4 bytes

20 bytes 4 bytes

Why this difference?

To dive deep into this, we will have to understand how compiler treats array as function argument *-> check next slide*

216	A[5]=1
212	A[3]=8
208	A[2]=5
204	A[1]=4
200	A[0]=2
300	X=5

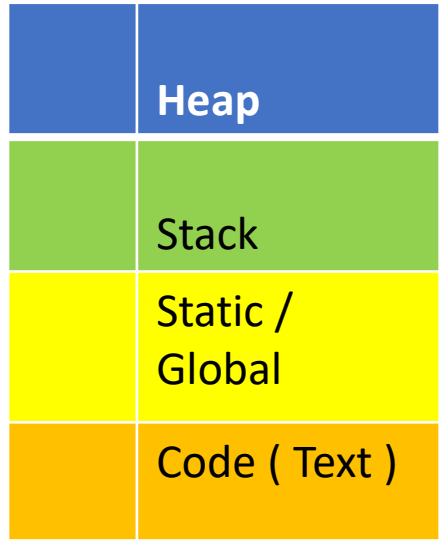
Array as a function argument

Pointers and arrays go together. There is a strong relation

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

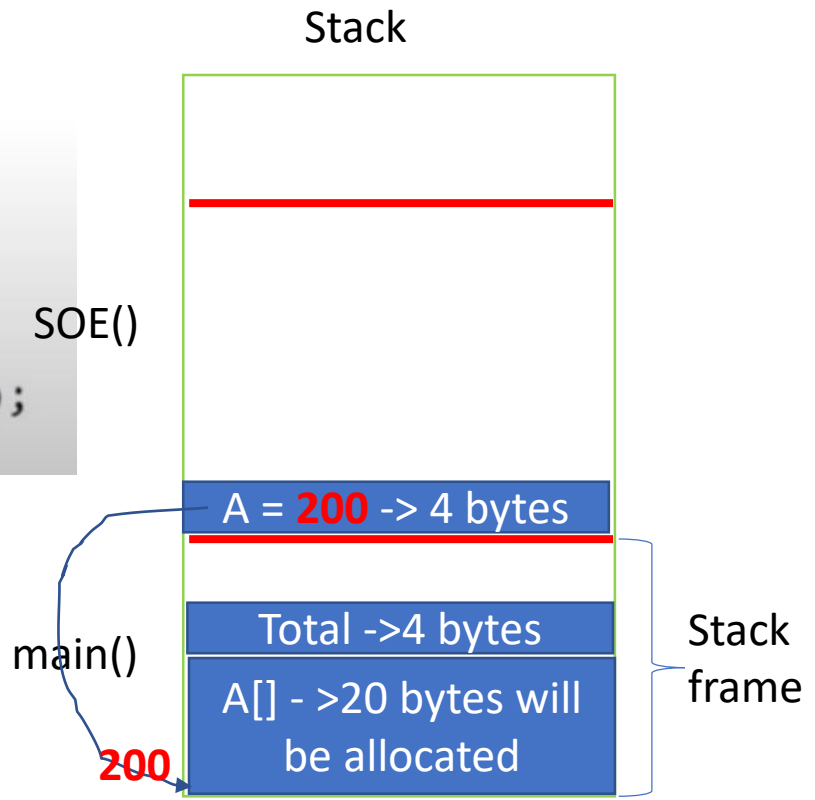
int *A

This is call by reference
Hence a copy of A is not made
Rather a pointer variable is made with the same name as A



For execution of the function calls we use stack memory.

4 bytes only



Character arrays and pointers

Pointers and arrays go together. There is a strong relation between character array and pointers too

Character array becomes more important because we can do many string manipulation using character array

1) *How to store Strings ?*

size of array >= no of character in string +1

"John" – size of array >= 5

char c[8];

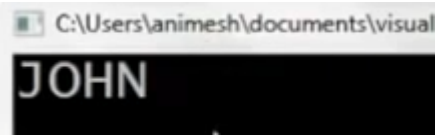
c[0] = 'j', c[1]='o', c[2] = 'h', c[3]='n' , c[4]='\0'

j	o	h	n	\0			

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

this will print some garbage value at end because we have broken the basic assumption of character string that it terminates with \0

```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);
}
```



This will behave properly.

We can even increase size of C

It will still behave okai.

Character arrays and pointers

We have a library for string

include <**string.h**>

All the functions in **string.h** assumes that strings are terminating with null character

```
#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);
}
```

At this point even if the length is 20, the len will print 4. because of basic assumptions attached to strings that it terminates with /0

Character arrays and pointers

Instead of writing character individuals,
We could use string literals.
String literals are implicitly terminated by '\0'
you don't need to explicitly put '\0' in string literals

```
//character arrays and pointers
#include<stdio.h>
#include<string.h>
int main()
{
    char C[20] = "JOHN";
    int len = strlen(C);
    printf("Length = %d\n",len);
}
```

This should be declared in one line, you cant declare it In the below form
~~Char C[20];~~
~~C= "JHON";~~

Remember C is a constant pointer.

Its okai to do
Char C[] = "JHON";
Q. What will be the size of C in bytes?

5 bytes // use sizeof(C)

Q. What will be the length of C ?

4 // use strlen(C)

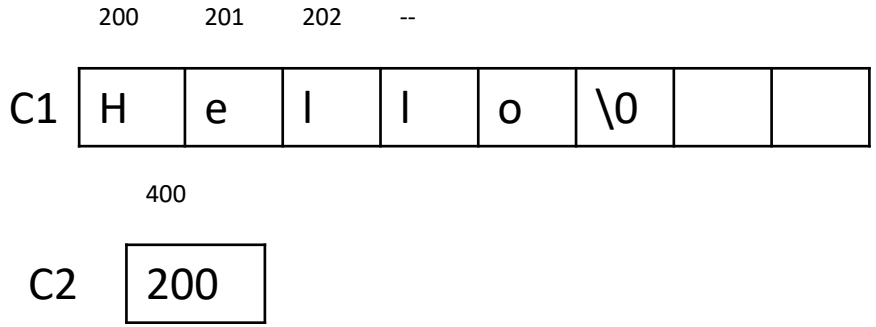
Its not okai to do
Char C[4] = "JHON";
This will give you compilation error

We can also declare like this
Char C[5] = {'J', 'H', 'O', 'N', '\0' }
But we need to explicitly add '\0' as the last character

Character arrays and pointers

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

char c1[8] =“Hello”;



declare a variable pointer to character

char c2*

can we do this ? Ans: yes

c2=c1

similarity

print(c2[1]) ; // e

c2[0] = ‘A’; //this is possible too, String will become “Aello”

*c2[i] is equal to * (c2 + i) // going to the next address and dereferencing*

*c1[i] is equal to the *(c1 + i) // going to the next address and dereferencing*

differences

can we do this ?

c1=c2 //not possible, its not valid, c1 is a constant pointer variable, we cant make it point to something else.

c1 = c1 + 1 ; //not possible

c2++; // this is possible. It will increment to next element, now c2 will point to 201.

// We can increment the pointer variable. Incrementing pointer variable means, we are

// pointing to the next element.

c1++; // not possible but c2++ is possible, incrementing the array itself is not possible.

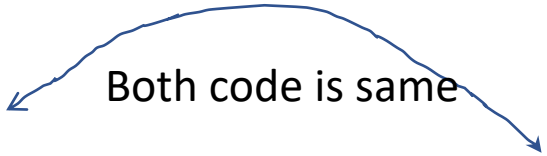
We must understand where we have array and where we have pointers and what we can do with each

Pointers part 2

Suman Pandey

Character arrays and pointers

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#include <string.h>
void print(char C[])
{
    int i = 0;
    while (C[i] != '\0')
    {
        C[i] = 'A';  //modifying these items
        printf("%c", C[i]);
        i++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    C[0] ='A';  // accessing each item
    print(C);
}
```

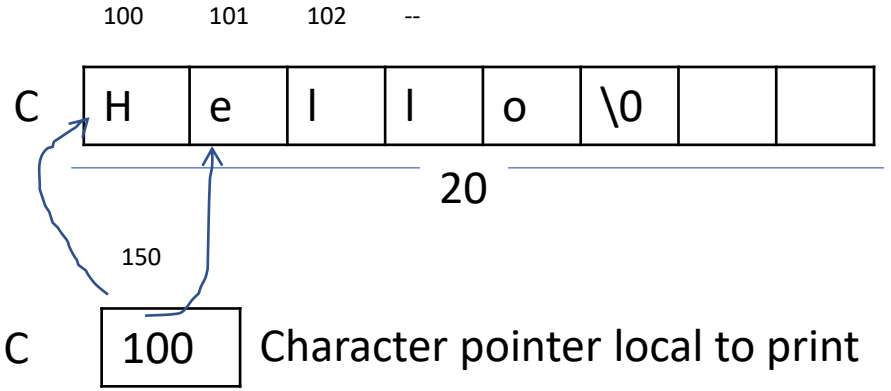


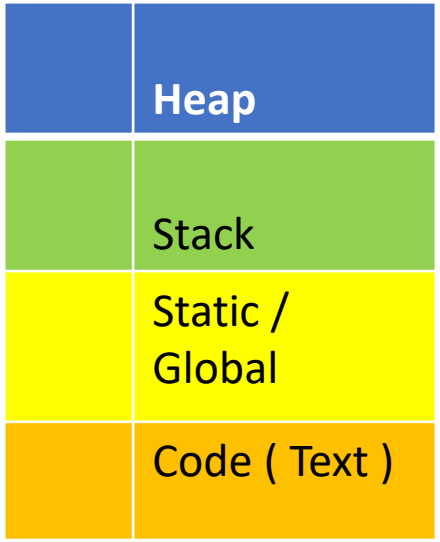

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

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

What will this print ? AAAAA

Character arrays and pointers

```
#define _CRT_SECURE_NO_WARNINGS
#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";
    print(C);
}
```





Let see the program control flow

C is not pointing to '\0'

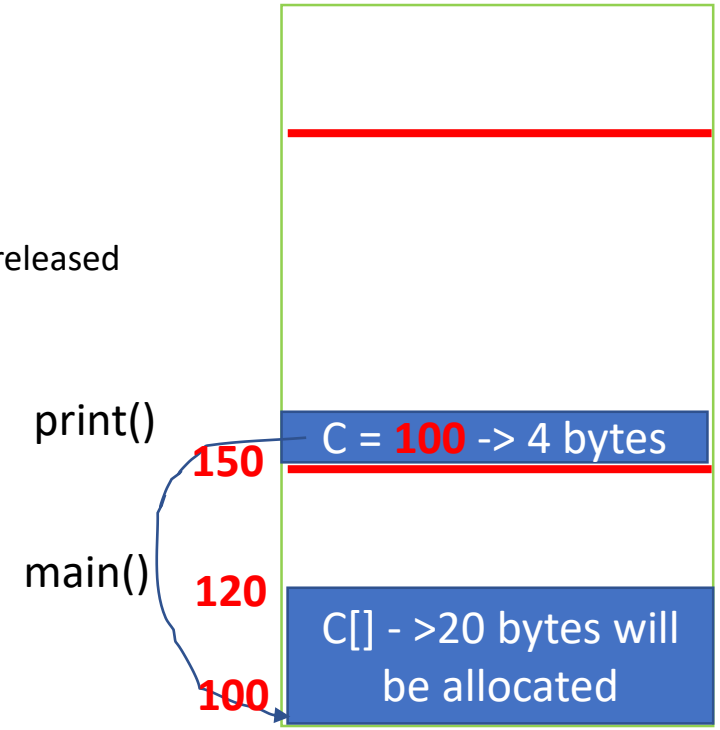
H will be printed

C++ will increment the address to 1 bytes, C = 101

…

We will keep on going until *C encounter '\0'

After print finish execution stack memory for print will be released



C in main and C in print are different

Character arrays and pointers

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

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

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

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

Fatal error at run time, why??

Character arrays and pointers

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

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

the string literal **"Hello"** is stored in read-only memory, and attempting to modify the contents of **C** (e.g., **C[0] = 'A';**) would result in undefined behavior. If you need to modify the string, you should use an array instead of a pointer to a string literal. For example: **char C[] = "Hello";**.

Character arrays and pointers , Constant Char array

Coming back to the character array, we stored a string literal in character array and function receives it in character pointer, we can modify the array elements using dereferencing of the pointer C

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

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

Sometimes, we want a function to just read the string and not write anything, to force this behavior we can take the argument as constant character pointer

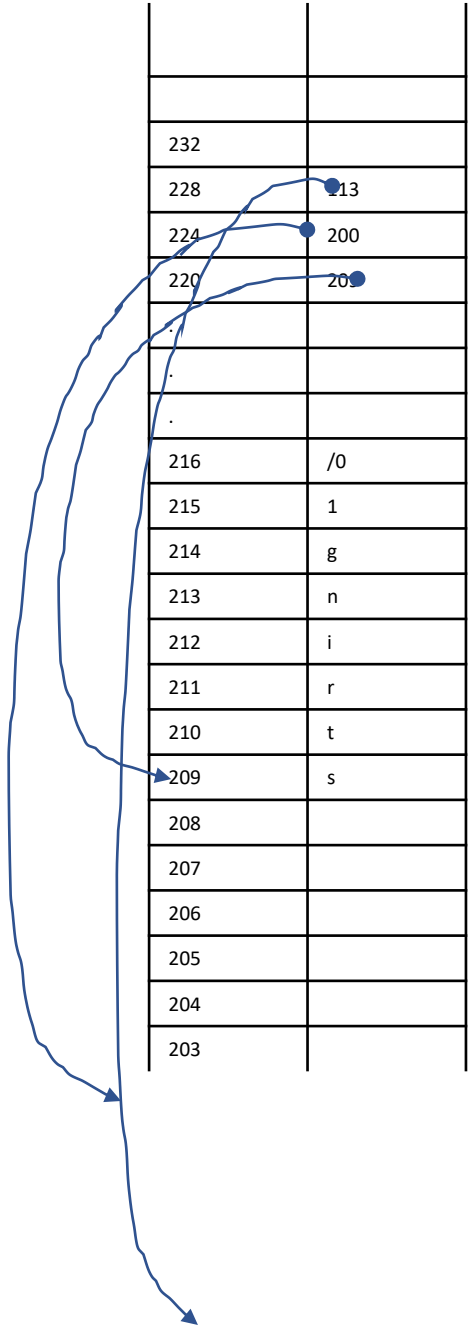
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#include <string.h>
void print(char const *C)
{
    while (*C != '\0')
    {                    // Compilation error, we can read but
        *C = 'A';        cant write in constant character pointer
        printf("%c", *C);
        C++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    C[0]='A';
    print(C);
}
```

Character arrays and pointers
What is possible and What is not possible?

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

int main()
{
    char *C = "Good";
    C[0] = 'A'; //can not modify  - run time exception
    *C = 'A'; //can not modify even this way - runtime exception

    char S[] = "Morning";
    S[0] = 'A';
    *S = 'A';
    *(S++) = 'A'; //can not modify the adress -compile time
    S = C; // can not modify address - compile time

    C = S; //can modify address but not value;
    printf("%c\n", *(++C)); // can modify address

}
```

Array of Pointers
Array of character Pointers

```
#include <stdio.h>
#include <string.h>
int main()
{
char* s[] = { "string1", "string2", "string3"};

printf("%s\n", s[0]);
printf("%s\n", s[1]);
printf("%s\n", s[2]);

printf("%s\n", *s);
printf("%s\n", *(s + 1));
printf("%s\n", *(s + 2));
}
```



How can you read complicated declarations ?

```
char *str[10]
```

```
char const *const C
```

str is an array 10 of pointers to char

```
char *(*fp)( int, float *)
```

```
char * const ptr;
```

fp is a pointer to a function passing an int and a pointer to float returning a pointer to a char

```
void (*signal(int, void (*fp)(int)))(int);
```

```
const char *ptr;
```

1.**signal** is a function that takes two parameters:
- •An integer parameter (**int**) representing the signal number.
- •A pointer to a function parameter (**void (*fp)(int)**) representing a function that takes an integer parameter and returns **void**.

2.The return type of the **signal** function is a pointer to a function that takes an integer parameter and returns **void**.

comp.lang.c by its author, David Anderson

Constant Pointer to Constant Data

access privilege

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int fun(const int * const l) {
    int b;
    l = &b;
    *l = 20;
}
int main()
{
    // Previledge 1
    const int v = 10;
    v = 20;

    int const v1 = 20;
    v1 = 20;

    // Previledge 2
    int* const v2 = &v1;
    v2 = &v;

    // Previledge 3

    const int* const v4 = &v;
    v4 = &v2;
    *v4 = 20;

    //Previledge 4
    int* v3 = &v;

    int l = 10;
    int* ptrl = &l;
    fun(l);
}
```

Can not change the data of the variable

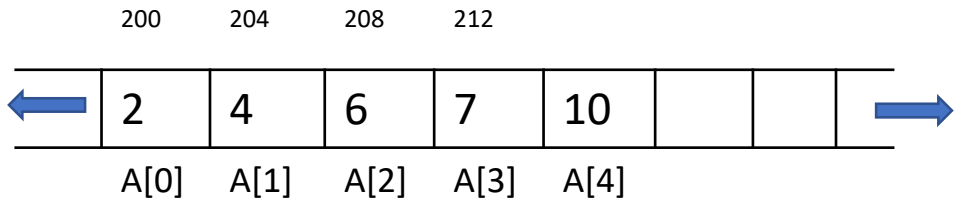
Can not change the address this variable is pointing to, but you can change the data at that address

Can not change the address this variable is pointing to, nor can you change the data at that address

Feel free to change the address this variable is pointing too, and also data at that address.

Same access privilege holds for the argument passing to a function also

Pointers and multi-dimensional arrays




```
int main()
{
    int A[5] = { 2,4,6,7,10 };
    int* P = &A[3];
    printf("%p\n", P);
    printf("%d\n", *P);
    printf("%d\n", *(P + 1));
```

Both will print same output

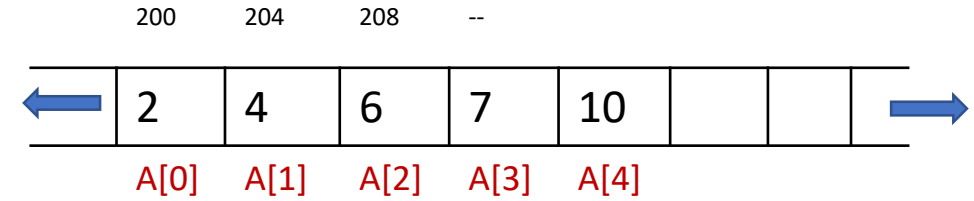
```
int main()
{
    int A[5] = { 2,4,6,7,10 };
    int* P = A;
    printf("%p\n", A);
    printf("%d\n", *A);
    printf("%d\n", *(A + 2));
}
```

Language gives us this flexibility

*(A + 2) is same as A[2]
(A+i) is same as &A[i]

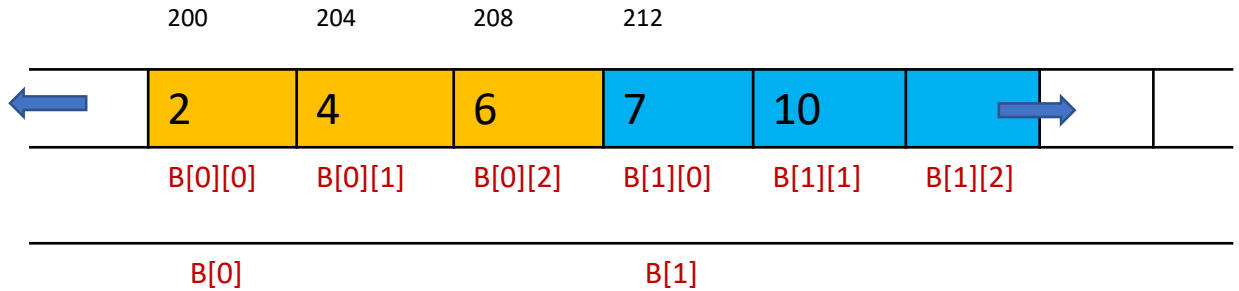
Remember that even though we can use the name of the array as the pointer for the pointer arithmetic, array is not same as a pointer variable
We can do P = A;
We can not do A = P;

Pointers and multi- dimensional arrays




```
int main()
{
    int A[5] = { 2,4,6,7,10 };

    int B[2][3] = { 2,4,6,7,10 };
    printf("%p\n", B[0]);
    printf("%p\n", B[1]);
}
```

1-D arrays of 3 integers



This will print
200
212

Pointers and multi- dimensional arrays

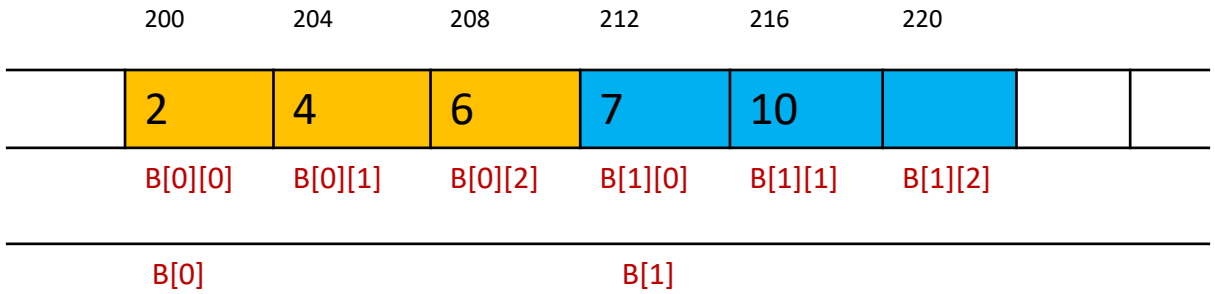
We must think of multidimensional array as **arrays of array**

int B[2][3]
B here is a collection of 2 one dimensional array of 3 elements
B will return us a pointer to one dimensional array of 3 integers

int (*p)[3] = B

*B gives us the address of B[0], When we access the value of B[0], its storing the address of first element of B[0] that is &B[0][0]
Hence *B will print us value of B[0] that is &B[0][0]

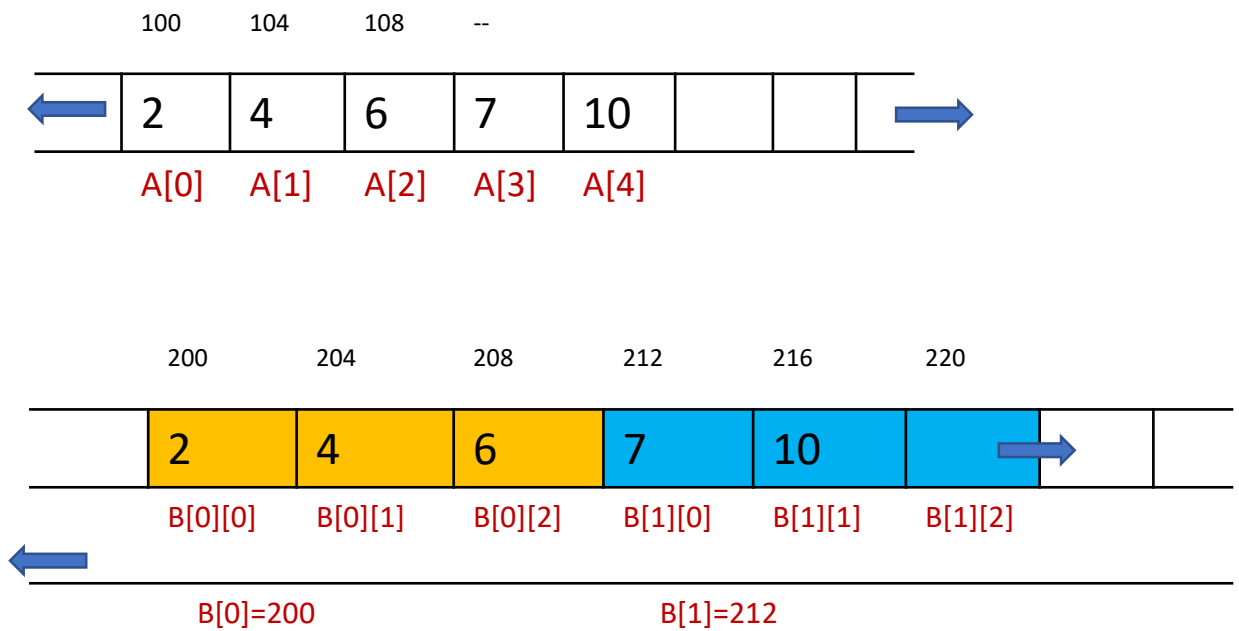
**B means, for first dereference we get value at B[0], which is adreess of &B[0][0], when we do second dereference, we go to &B[0][0] and get the value stored there, which is the actual value 2.



Pointers and multi- dimensional arrays

```
int main()
{
    int A[5] = { 2,4,6,7,10 };
    printf("%p\n", A);        //100
    printf("%p\n", &A);       //100
    printf("%p\n", *A);       //2
    int B[2][3] = { 2,4,6,7,10 };
    printf("%p\n", B[0]);     //200
    printf("%p\n", B[1]);     //212
    printf("%p\n", B);        //200
    printf("%p\n", &B[0]);    //200
    printf("%p\n", *B);       //200

    printf("%p\n", &B[0][0]);//200
    printf("%p\n", B + 1);    //212
    printf("%p\n", &B[1]);    //212
    printf("%p\n", *(B + 1));//212
    printf("%p\n", B[1]);     //212
    printf("%p\n", &B[1][0]);//212
    printf("%p\n", *(B+1)+2 or B[1]+2 or &B[1][2]   //220
    printf("%p\n", *(*B+1)
}
```

same



B stores Address of B[0] **B will return 2

B is a name, B[0] and B[1] is also name

B stores Address of B[0], B+1 will jump size of B[0]

When we dereference then **type of the pointer becomes important**
B is a pointer to one dimensional array of 3 integers, Hence B+1 is also a pointer to one dimensional array of 3 integers
B+1 is same as &B[1] When we dereference we get whole one dimensional array of 3 integer's starting address

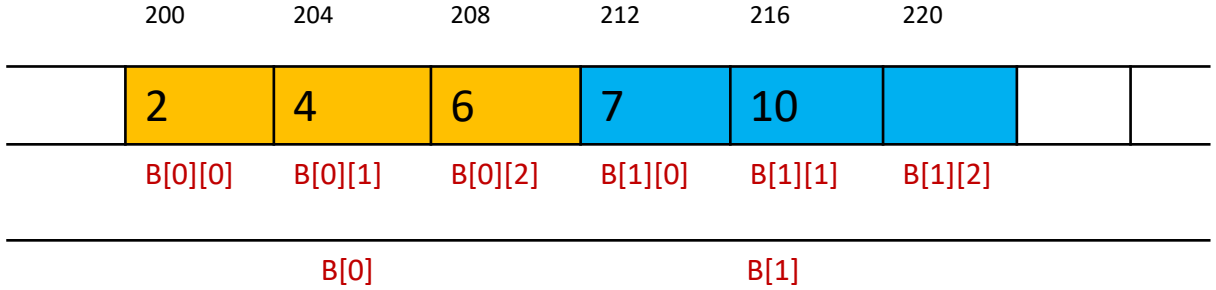
Here B -> int (*)[3] integer pointer to one dimensional array
Dereferencing it will give us one dimensional array *B =B[0]
B[0] in my expression returns me the pointer to the first integer in one dimentional array B[0] -> int *, at address 200
B[0] + 1, will take you to the pointer to next integer, that is 204, so when you dereference 204 you get 4 as answer

Pointers and dynamic Memory

We must think of multidimensional array as **arrays of array**

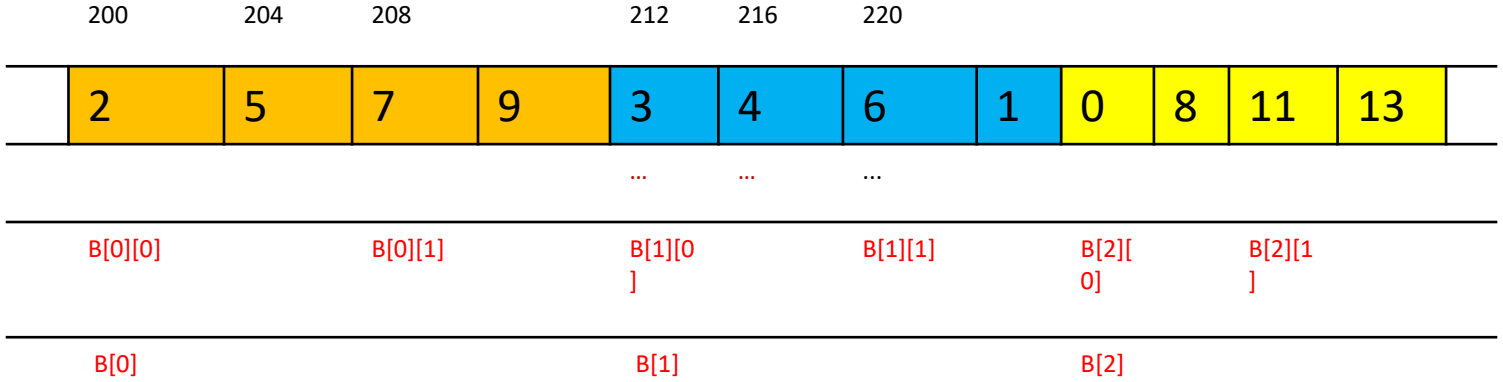
int B[2][3]
B here is a collection of 2 one dimensional array of 3 elements
B will return us a pointer to one dimensional array of 3 integers

int (*p)[3] = B

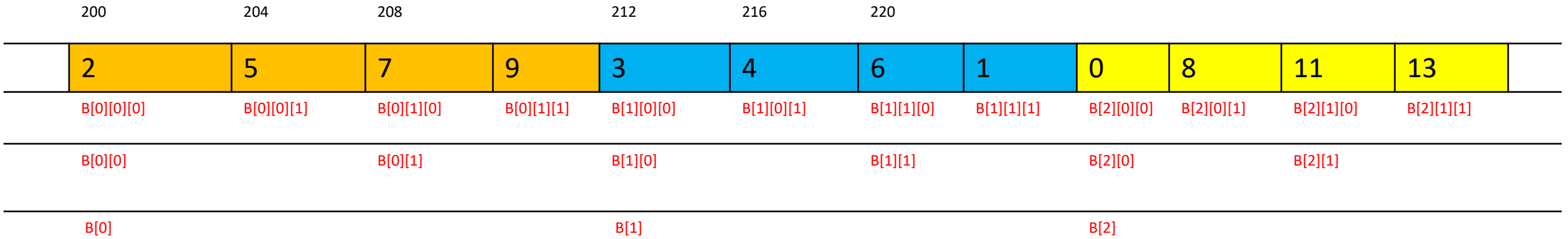


*B gives us the address of B[0], When we access the value of B[0],
its storing the address of first element of B[0] that is &B[0][0]
Hence *B will print us value of B[0] that is &B[0][0]

**B means, for first dereference we get value at B[0],
which is address of &B[0][0], when we do second dereference,
we go to &B[0][0] and get the value stored there,
which is the actual value 2.



int B[3][2][2] //B is a collection of 3 two dimensional array of size [2][2]
Int (*p)[2][2] = B ;
Print B //200
Print *B or B[0] or &B[0][0] or B[0][0] // 200
print B[i][j][k] = *** (B[i][j] + k) = * (* (B[i] + j) + k) = * (* (* (B + i) + j) + k)** // all these expression will give you same value



B gives us - (*)[2][2] -> pointer to 2 dimensional array
If we dereference B, *B -> this is same as B[0], B[0] is &B[0][0] -> pointer to one dimensional array

int (*) [2]

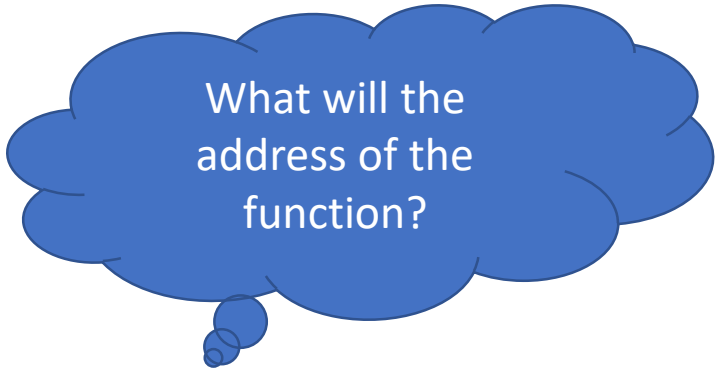
B gives pointer to two dimensional array
Dereferencing B once is giving the pointer to one dimensional array
Dereferencing B twice **B will give us the pointer to one element
Dereferencing B trice ***B will give us the value at that pointer
Dereferencing it once - > * (B + i) -> same as B[i] -> &B[i]
Dereferencing it twice -> * (* (B + i) + j) -> same as B[i][j] -> &B[i][j][0]
Dereferencing it trice -> * (* (* (B + i) + j) +k) ->same as B[i][j][k] -> this is also same as * (* (B[i] + j) + k) -> *(B[i][j] + k)

So what will be output to these ?
Print *(B[0][1] + 1) // same as B[0][1][1] //9
Print * (B[1] + 1) // B[1] gives us address of B[1][0], + 1 will jump 2 integers, address of B[1][1] -> dereferencing B[1][1] gives you address of B[1][1][0], so output 220

Function pointers

Pointers -> can point to a data
We can dereference and give us the value stored at that address

Pointers -> can point to a function

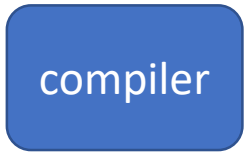


We write a program in high level language and we pass it to compiler

<< program.c >>

```
Int main()
{
 printf("Hello");
}
```

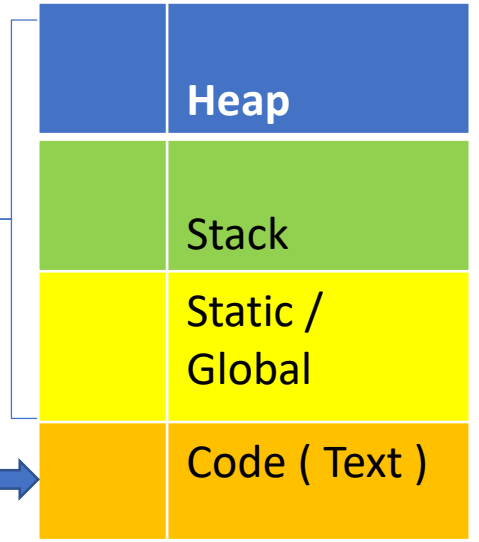
Source code



<< program.exe >>

10101010101
01010101010
11010101010

machine code



- Instructions will execute sequentially
- Unless the instruction itself says go and execute the instruction at address 212, this will happen in the case of function call
- Function will also be continuous block
- Address of fun1 is 212.
- When we say function pointers -> it stores address of function, means it stores address of the beginning of the block of memory location ,where all the instructions are stored for that function

fun1

216	Instruction 4
212	Instruction 3
208	Instruction 2
204	Instruction 1
200	Instruction 0

```c
#include <string.h>

void A()
{
    printf("Hello\n");
}

void B(void (*ptr)()) // function pointer as argument
{
    printf("call A through callback function B\n");
    ptr(); // call-back function that "ptr" points to
}
int main()
{
    void (*p)();
    void (*q)();
 // (*p)() and *p() is different thing. The second syntax
means you are calling a function that returns a type pointer
    p = A; //p=&A; same thing, two syntax
    (*p)(); //p(); same thing, two syntax

    B(p);
    q=p;
    //instead of all the above syntax, you can symbol call
    B(A);
    // A can be called back by B through a function pointer.
}
```

Function pointers example1

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int main() {

    int (*operation)(int, int);
    int op;

    printf("Enter 1 fir addition and 2 for subtraction \n");
    scanf("%d", &op);

    if(op == 1)
        operation = &add;
    else
        operation = &subtract;

    int result = operation(4, 2);
    printf("Result of operation: %d\n", result);

    return 0;
}
```

Function pointers example 2
As callback function

```c
#include <stdio.h>
#define SIZE 10

void bubble( int work[], const int size,
             int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order; int counter;
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Enter 1 for ascending 2 for descending order,\n“);
   scanf( "%d",  &order );

   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
      printf( "\nData items in ascending order\n" );
   } else {
      bubble( a, SIZE, descending );
      printf( "\nData items in descending order\n" );
   }
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   } /* end for */
 return 0;
}
void bubble( int work[], const int size,
             int (*compare)( int a, int b ) )
{
   int pass; int count;
   void swap( int *element1Ptr, int *element2ptr );
   for ( pass = 1; pass < size; pass++ ) {
            for ( count = 0; count < size - 1; count++ ) {
            if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
            swap( &work[ count ], &work[ count + 1 ] );
         }
      }
   }
}
```

```c
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold; /* temporary holding variable */

   hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} /* end function swap */

int ascending( int a, int b )
{
   return b < a; /* swap if b is less than a */

}

int descending( int a, int b )
{
   return b > a; /* swap if b is greater than a */

}
```

we can **use function pointers** to avoid code redundancy.
when you want to create callback mechanism, and need to pass address of a function to another function.

Function pointers example 3
Array of function pointer

```c
#include <stdio.h>

void function1( int a );
void function2( int b );
void function3( int c );

int main( void )
{
    void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice; /* variable to hold user's choice */

   printf( "Enter a number between 0 and 2, 3 to end: " );
   scanf( "%d", &choice );

   /* process user's choice */
   while ( choice >= 0 && choice < 3 ) {

      /* invoke function at location choice in array f and pass
         choice as an argument */
      (*f[ choice ])( choice );

      printf( "Enter a number between 0 and 2, 3 to end: ");
      scanf( "%d", &choice );
   } /* end while */

   printf( "Program execution completed.\n" );
   return 0; /* indicates successful termination */
} /* end main */
```

```c
void function1( int a )
{
   printf( "You entered %d so function1 was called\n\n", a );
} /* end function1 */

void function2( int b )
{
   printf( "You entered %d so function2 was called\n\n", b );
} /* end function2 */

void function3( int c )
{
   printf( "You entered %d so function3 was called\n\n", c );
} /* end function3 */
```

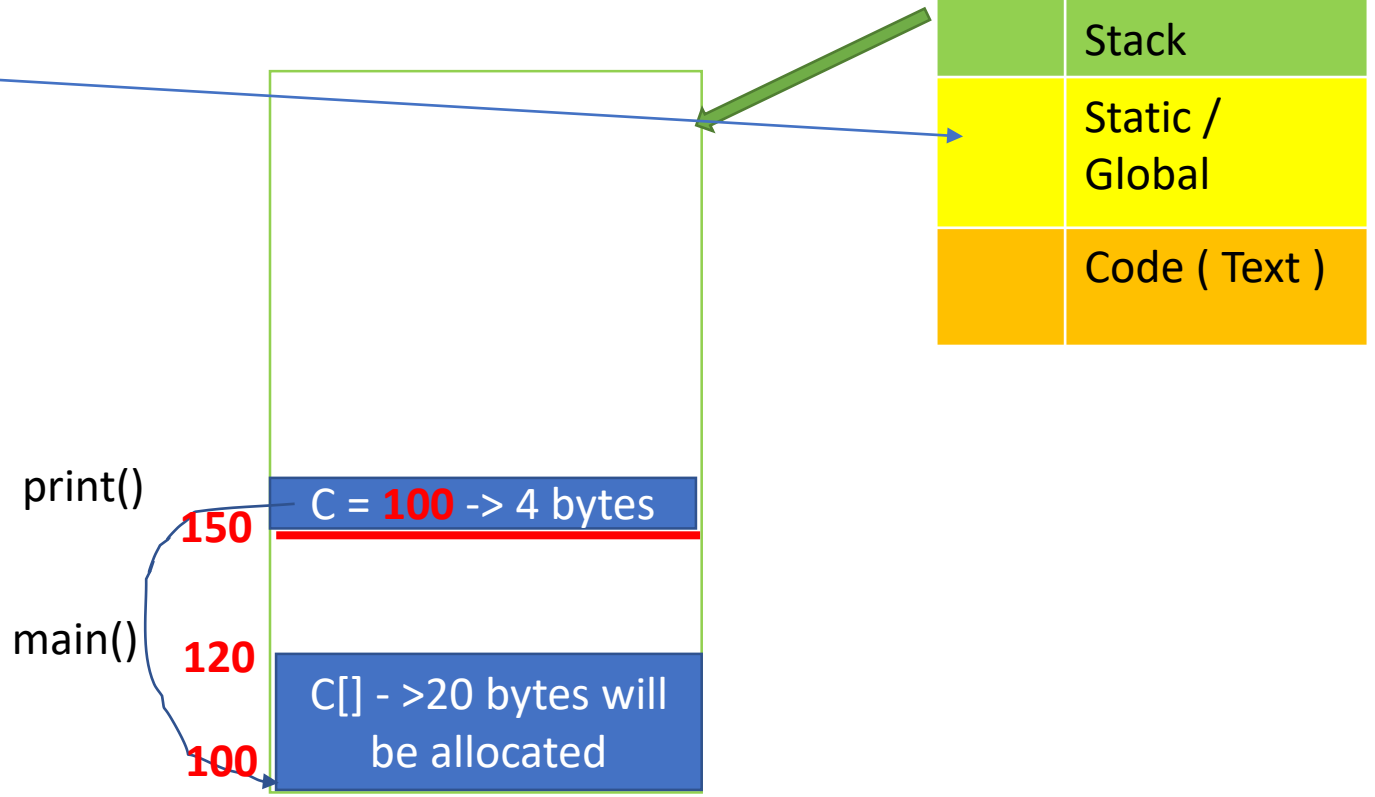
Finite State Machines where the elements of (multi-dimensional) arrays indicate the routine that processes/handles the next state. This keeps the definition of the FSM in one place (the array).

Heaps Malloc

Suman Pandey

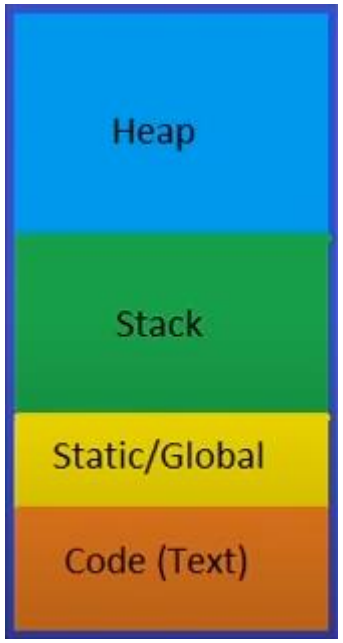
Character arrays and pointers

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int total=10;
#include <string.h>
void print(char* C)
{
    while (*C != '\0')
    {
        printf("%c", *C);
        C++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```



What is Heap?

- Application Memory



- **Dynamic Memory Allocation**

- Function calls/ Local variables / automatic variables/ lives only until the function is executing

- Lives for the entire program, until application executes

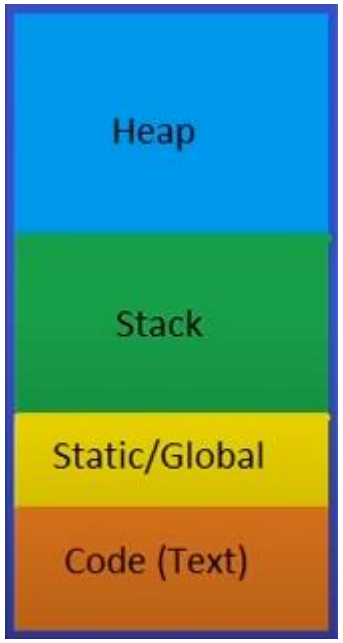
- Instructions

- Does not Grow while application is running

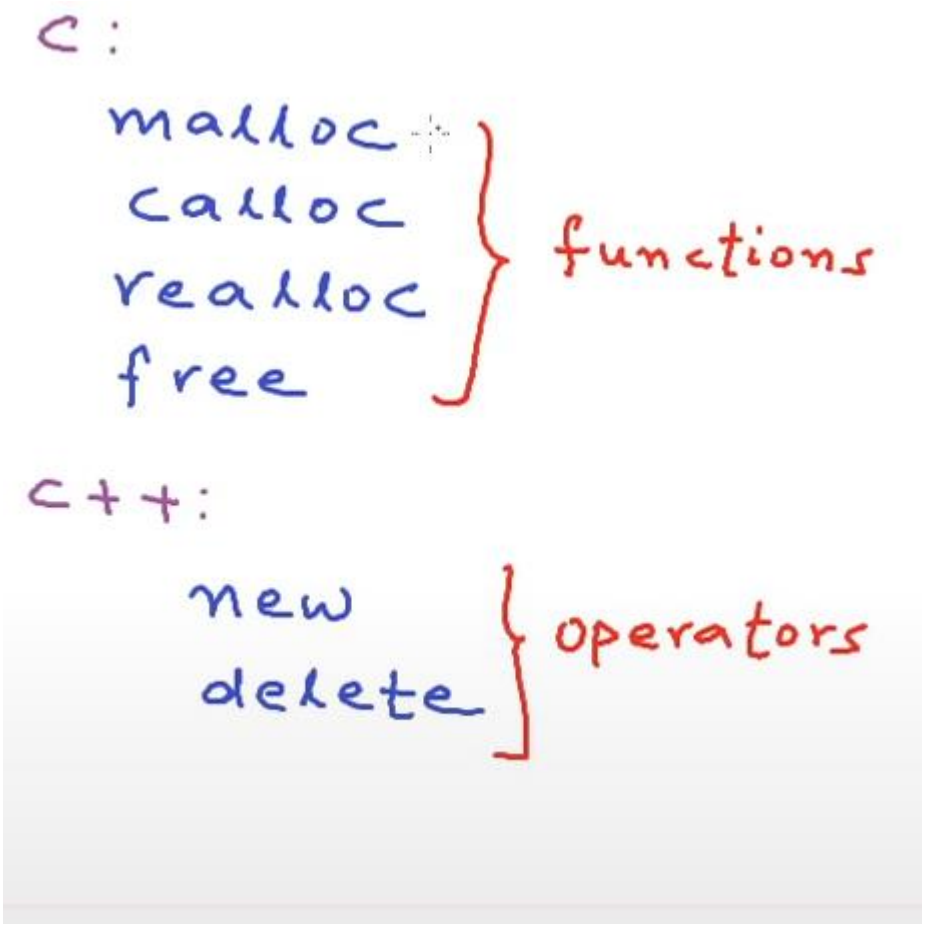
- Stack functions like a Stack Data structure
- Heap doesn't have anything to do with heap data structure
- Heap memory is assigned by compiler and Operating systems, so can be assigned in the way OS work.

Functions to manage Heap

- Application Memory



- Can grow, that's why its dynamic memory
- Does not Grow while application is running

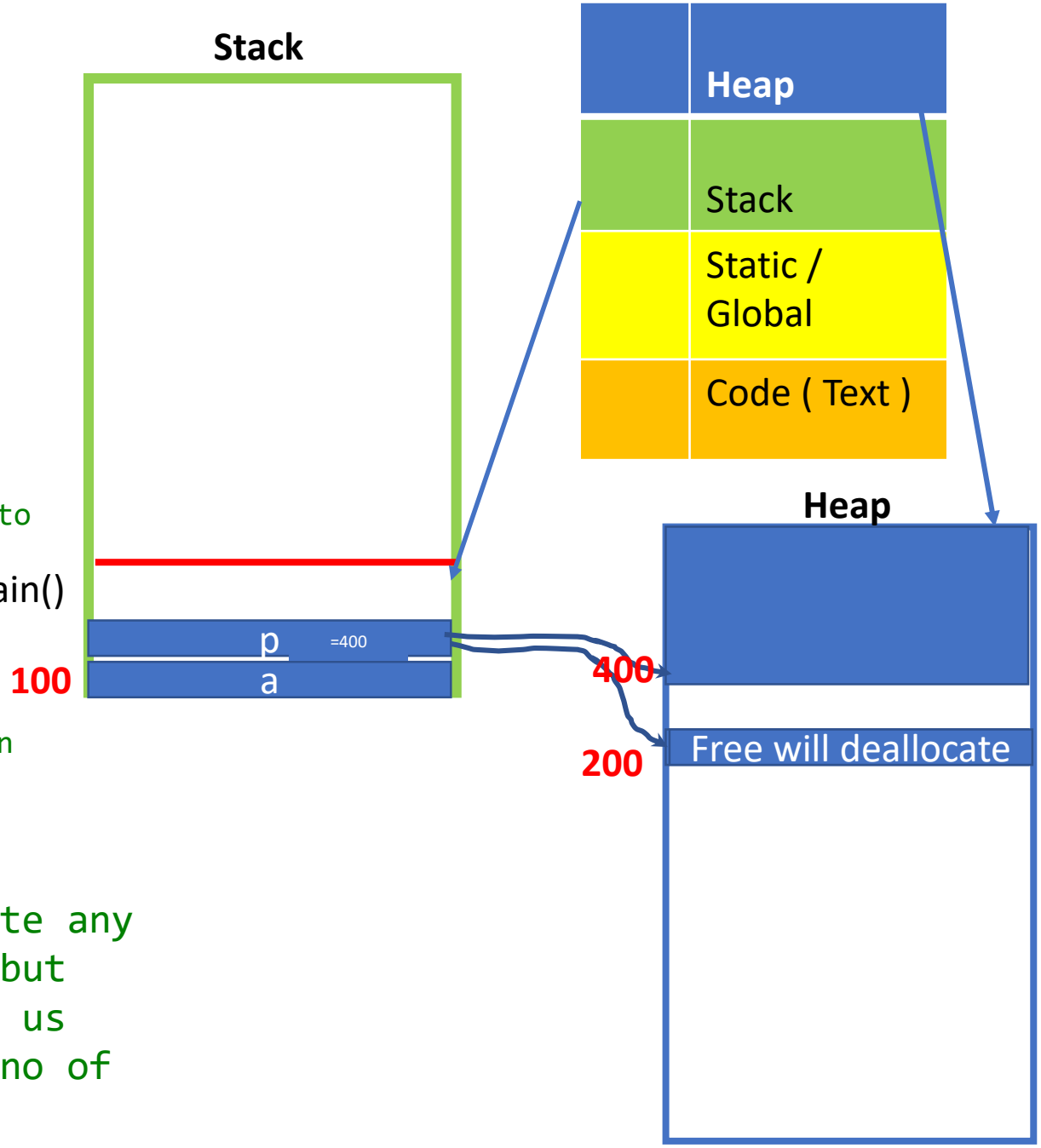


Heap in action

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

int main()
{
    int a; // goes on stack
    int* p; // goes in stack
    p = (int*)malloc(sizeof(int));
    // What does malloc function do ?
    // it looks for the free memory in the heap
    // books it for you ,and returns u the pointer to
that memory location in pointer variable p
    *p = 10;
    free(p); // any memory allocated through malloc
should get free.
    p = (int*)malloc(20*sizeof(int));
// the previous block is still in the memory, it can
not be cleared out automatically

}
```

malloc returns a void pointer, hence we need to typecast the return with (int*)

malloc can allocate any number of bytes, but this syntax helps us define the exact no of bytes.



Heap : for variable size array

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main() {
    int size;

    printf("Enter the size of the array: ");
    scanf("%d", &size);


    int* array = (int*)malloc(size * sizeof(int));

    if (array == NULL) {
        printf("Memory allocation failed. Exiting...\n");
        return 1;
    }

    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &array[i]);
    }
    printf("The elements you entered are:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Free the dynamically allocated memory
    free(array);

    return 0;
}
```

calloc and realloc

```
int *p = (int*)malloc(20*sizeof(int));   // no initialization, so it will have garbage value

int *p = (int*)calloc(20, sizeof(int));  // calloc initialize memory with 0


int *p = (int*)realloc(void * ptr, sizeof(int));  // for reallocating

int *A = (int*)malloc(20*sizeof(int));
int *B = (int*)realloc(A,40*sizeof(int)); // it will try to find consecutive memory and
                                          //allocate
                                          // if it doesn't find consecutive memory, it will
                                          // allocate new memory and copy the previous
                                          //stuffs into the new memory
```

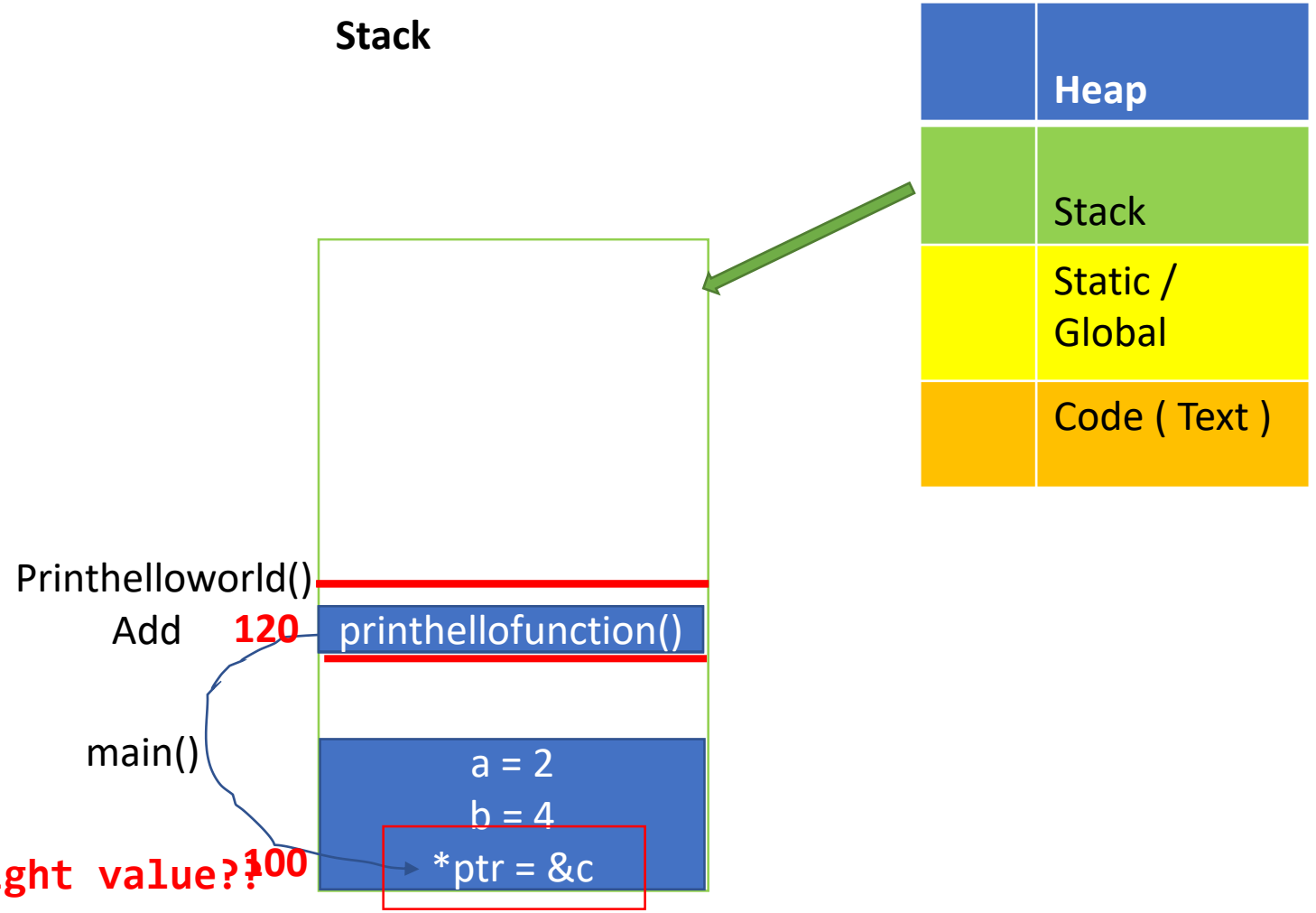
Heap : for variable size array when you decide to extend the existing array size

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main() {
    int size;

    printf("Enter the size of the array: ");
    scanf("%d", &size);


    int* array = (int*)malloc(size * sizeof(int));

    if (array == NULL) {
        printf("Memory allocation failed. Exiting...\n");
        return 1;
    }

    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &array[i]);
    }
```

```c
//Extend the size of array
    int exsize;
    printf("Enter the extra size to add to the arra: ");
    scanf("%d", &exsize);

    array = (int*)realloc(array, (size+exsize)*sizeof(int));

    printf("Enter the extra elements:\n");
    for (int i = size; i < (size + exsize); i++) {
        scanf("%d", &array[i]);
    }
    printf("\n");
    for (int i = 0; i < (size + exsize); i++) {
        printf("%d %d\n",i, array[i]);
    }

    free(array);

    return 0;
}
```

Pointers as function return

```
void printhelloworld() {
    printf("Helloworld\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);
    // Can the above print you the right value??
    // Ans : No , Why???

}
```



Pointers as function return with heap

```c
#include <stdio.h>
#include <stdlib.h>

void printhelloworld() {
    printf("Helloworld\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);
    free(ptr);
}
```