Lecture

Pointers

* Pointer is a derived data type that is used to store memory address of another variable.
* It can access and manipulate the variable's data stored at that location → call by reference
* It can be used to pass the data efficiently between the functions, and create dynamic data structures like linked lists, trees and graphs

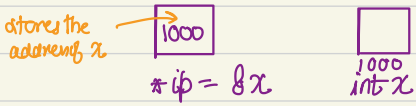
We use 'dereferencing operator' - '*' followed by the variable name

Syntax:

data_type *variable_name; → name of pointer variable.

Eg: int *ip → a pointer to an integer

double *dp → a pointer to a double



* Initialising the pointer with the address of another variable is known as referencing a pointer

Eg int *ptr = &x → integer variable; * → integer pointer

printf("%d", ptr) : 1000 → meaning the address of x is 1000.

Referencing a pointer: '&' 'address of' operator is used to store the address of a variable to a pointer

dereferencing a pointer:- '*' 'dereferencing operator' is used to

Outputs:

int x=10 (1000)
int *ptr = &x; (2000)
int *ptr2 = &ptr;

printf("%d", x) = 10
printf("%u", ptr) = 1000
printf("%u", ptr2) = 2000
printf("%u", *ptr2) = *(2000)
= *(1000)
= 10

Dereferencing a pointer: obtaining the value stored at the address stored in the pointer

& operator- "Address of" operator or "Referencing operator" used to store the address of an existing variable into a pointer.

* operator: "dereference operator" or "value at address" or "indirection operator" dereferencing a pointer is used to get the value from the memory address that is pointed out by a pointer.

* pointers are useful for returning multiple values from a function

Size of a pointer

The size of a pointer in C is same for every pointer and it doesn't depend upon what data type it is pointing to and the size of the pointer depends upon OS and CPU architecture.

→ it is 8byte for 64 bit system

⇒ it is 4 bytes for a 32 bit system.

size of int pointer = size of char pointer = size of void pointer.

```
int *p = &i;
printf("%p", p);
```
→ format specifier for pointers

Eg:
```
int *ptr = NULL;
printf("%p", ptr)
```
output: 0 ← value of address stored in pointer which is null.

Youtube

Pointer are variables that store the address of another variable.

```
int a;
int *p;
p = &a ; (p points to a and stores the address of a)
a = 5;
print p  // 204
print &a // 204
print &p // 64
print*p  // 5
```

a=5	204
p=204	64

* if we put * infront of a pointer then it gives us the value stored at the address stored in the pointer. This process is known as dereferencing.

p → address
*p → value at address

Types of pointers

or int* p;
char* p;

int *p; → pointer to an integer
char *p; → pointer to a character
double *p; → pointer to a double

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

```
3077992
10
3077992
```

Program to update values using pointers.

```
int main()
{
int a;
int *p;
 a=10;
 p= &a;
printf("%d\n", a);
*p=12; // updates the value at a.
printf("a=%d\n", a);
}
```

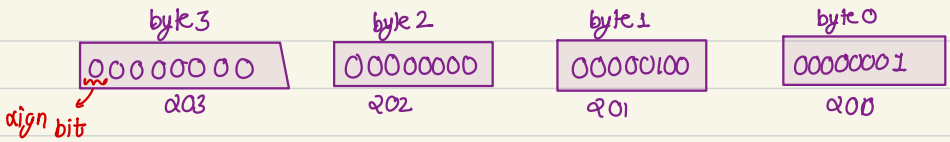
Outputs: 10
12

Increment of a pointer.

```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));
```

```
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
```

→ because sizeof int is 4 bytes



int a = 1025;

int *p;

p = &a

printf p //200

value of 1025 ← print *p // compiler looks that it is integer, which is of 4 bytes and then looks at 4 bytes starting at 200.

```c
#include <stdio.h>

int main()
{
int a=10234;
int* p=&a;
char *p0=(char*)p;// because p is a pointer to an integer
printf("size of integer is %d\n",sizeof(int));
printf("address of a is %d and value at that address is %d \n ",p,*p);
printf("size of character is : %d \n",sizeof(char));
printf("address: %d value at address : %d \n",p0,*p0);
return 0;
}
```

```
size of integer is 4
address of a is 1828614904 and value at that address is 10234
 size of character is : 1
address: 1828614904 value at address : -6
sarth@Sarthaks-Macbook-Air PointersInC %
```

1025 in 32 bits

equivalent to 4

00000000 00000000 00000100 00000001

Byte 3 Byte 2 Byte 1 Byte 0

```c
int main()
{
int a=1024;
int* p=&a;
char *p0=(char*)p;// because p is a pointer to an integer
printf("size of integer is %lu\n",sizeof(int));
printf("address of a is %u and value at that address is %d \n ",p+1,*(p+1));
printf("size of character is : %lu \n",sizeof(char));
printf("address: %s value at address : %d \n",p0+1,*(p0+1));
return 0;
}
```

type casting in a pointer

```
size of integer is 4
address of a is 1875456764 and value at that address is 0
 size of character is : 1
address: 1875456761 value at address : 4
sarth@Sarthaks-Macbook-Air PointersInC %
```

Void pointer

syntax of declaration: `void *p;` ← generic pointer. can point to any data type and can be typecasted to any type.

```c
#include <stdio.h>

int main()
{
int a=1024;
int* p=&a;
void *p0;
p0=p;
printf(" address : %u value : %d/n",p0,*p0);
return 0;
}
```

→ gives error: void pointer cannot be dereferenced and can only be used to store address.

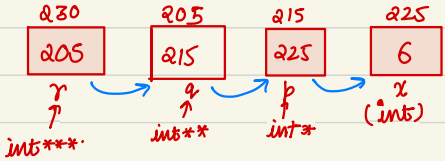
Pointer to Pointer

```
int x=5;
int *p;
p = &x;
int **q;
q = &p;
int ***r;
r = &q;
```

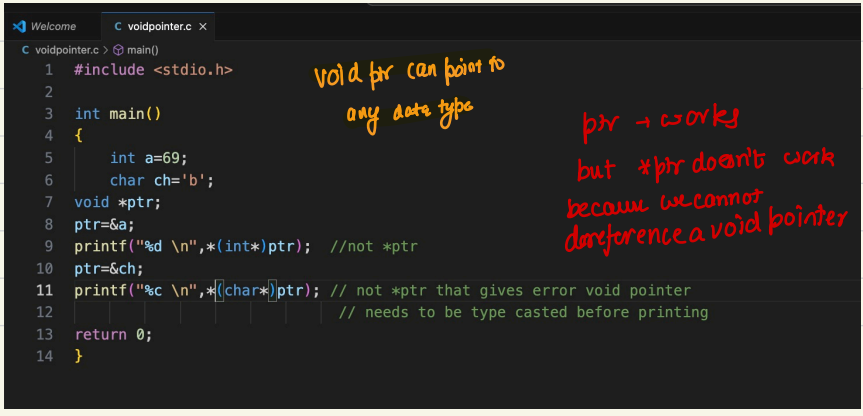
p points to integer x.

→ a pointer that points to a pointer 'p'. → having address of a pointer.

* we use integer pointer to store the address of integer variable as pointer are not only used to store address but they are allowed for dereferencing.




```
print (*p) = 6
print (*q) = 225
print (**q) = 6
print (**r) = 225
print (***r) = 6
***r = 10;
print (x) = 10;
**q = *p+2;
print (x) = 12
```



Pointers as function arguments - Call by reference

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

here address is dereferenced

Passing the reference.

Output: 11

call by reference.

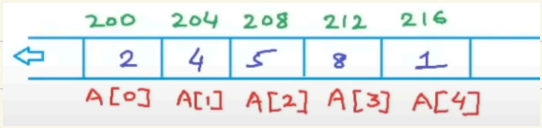
- helps us save some memory as copy of variables that is, local variables are not made which occupy more memory than reference of a variable.

Pointers and arrays

def int A[5];

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

(int occupies 4 bytes in modern compilers)




```
int *p;
p = &A[0];
print p    // 200
print *p   // 2
print(p+1) // 204
print *(p+1) // 4
```

print(*(p+2)) // 5

To access the element in the array at ith position

Address :- &A[i] or A+i

value :- A[i] or *(A+i)

* A gives us the base address of an array.

```
int A[5];
int *p;
p = A;
print A   //200
```

this points to the first element of the array

Pointer to base address.

```
#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);
}
```

```
2881420
2881420
2
2
```

```
#include<stdio.h>
int main()
{
    int A[] ={2,4,5,8,1};
    int i;
    for(int 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));
    }
}
```

```
# include <stdio.h>
int main()
{
int A[] = {2,4,5,6,1};
int i;
int *p = A;
p++; <- This is valid
A++; <- this is invalid.
}
```

```
2
3  int main()
4  {
5  int arr[]={1,2,3,4,5};
6  int *p=arr;
7  printf("%d \n",p);
8  p++; → increases by an integer.
9  printf("%d\n",p);
0  return 0;
1  }
```

Outputs.

```
1872081648
1872081652
sarth@Sarthaks-Macbook-Air PointersInC %
```

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);
}
```

when a compiler sees an array as function argument it doesn't copy the whole array. It actually creates a pointer variable by same name (same as data type of the array) and it points to the address of first element in the array. Thats why the out put is as such.

a implicitly converted to int* A

* arrays are always passed as reference parameters as in call by value a copy of array would be made which would occupy a large memory.

output.

```
SOE - Size of A = 4, size of A[0] = 4
Sum of elements = 1
Main - Size of A = 20, size of A[0] = 4
```

```
#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(A[0]);
    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 need to pass the sizeof the array as size is unknown if not passed.

this gives the correct output.

```
        |
        |
4 warnings generated.
SIZE OF S =8 and size of A[0]= 4
SUM OF ELEMNTS= 15
SIZE OF S =20 and size of A[0]= 4
sarth@Sarthaks-Macbook-Air PointersInC %
```

```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]);
    }
```

both pass the address reference of first element of the array.

```
2 4 6 8 10
```

since array is passed by reference we can actually modify it.

Character arrays and pointers (basically strings)

size of array ≥ no of characters in string + 1. → to tell that the string ends there.

Eg "JOHN" size ≥ 5

char C[8];

C[0] = 'J' C[1] = 'O' C[2] = 'H' C[3] = 'N' C[4] = '\0' ← null character

Rule:- a string in C has to be terminated by a null character.

```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);
}
```

even if it is C[5] - it will work.

```
JOHN_
```

if this is omitted then output is:

```
JOHN╠╠╠╠D√2_
```

printf function print the character till it finds a null character

Note: This is an invalid statement we cannot do like this.

```c
char C[20];
C = "JOHN";
```

we can avoid size in a single line i.e.

```c
char C[20] = "JOHN";
```

```c
//character arrays and pointers
#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);
```

```
Length = 4
```

this can be substituted by using string literals (which are enclosed in double quotes)

Eg: char C[20] = "JOHN";

↳ here null termination is implicit.

this function also works as printf function that it finds the length till a null character is not obtained.

here null character will not be implicit hence we have to enter it manually.

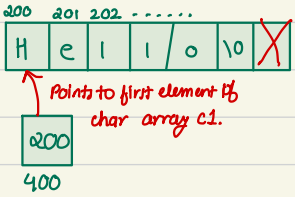
we can also declare a string as `char C[5]= {'J','O','H','N','\0'};`

* Arrays and pointers are different types that are used in similar manner

```
char c1[6] ="Hello";
char *c2;
c2= c1
```

// this is a valid statement and this returns the address of first element in the array

it will point to first element of the array C.



We cannot increment an array but we can increment a pointer.

Important

```
print C2[1] //e
c2[0]= 'A'  // "Aello"
```

c2[i] is *(c2+i)

c1[i] or *(c1+i)

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

OR *str!='\0';

OR ("%c", *(str+i));

C++;

Outputs: HELLO

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

int main()
{
int n=printf("HELLO\n");
printf("%d\n",n);
return 0;
}
```

1 2 3 4 5 6

it returns the no of characters that it prints.

6 characters

```
HELLO
6
sarth@Sarthaks-Macbook-Air PointersInC %
```

```c
//character arrays and pointers
#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);
```

Bus error

can be substituted for C[20]

will give error as const cannot be updated

```c
//character arrays and pointers
#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);
}
```

just to read a array, we cannot make changes to it

C[0] = 'A'; will give error.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
void modify(const char* str)
{
    while(*str!='\0')
    {
        printf("%c",*str+1);
        str++;
    }
}
int main()
{
char str[20]="ABCDEFGH";
modify(str);
return 0;
}
```

here we are dereferencing and then using so no problem

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
void modify(char* str)
{
    while(*str!='\0')
    {
       *str=*str+1;
       str++;
    }
    return;
}
int main()
{
char* str="ABCDEFGH";
modify(str);
printf("%s",str);
return 0;
}
```

giving bus error

cannot modify

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
void modify(char* str)
{
    while(*str!='\0')
    {
       *str=*str+1;
       str++;
    }
    return;
}
int main()
{
char str[20]="ABCDEFGH";
modify(str);
printf("%s",str);
return 0;
}
```

copy of pointer created

can modify

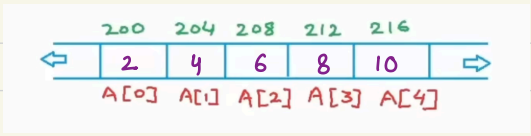
```
BCDEFGHI
sarth@Sarthaks-Macbook-Air strings %
```

Pointers and multidimensional arrays

```
int A[5];
int *p = A;
```

but we cannot say A = p;

only the name of the array returns the pointer to the first element of the array.




```
print (p)   // 200
print (*p)  // 2
print *(p+2) // 6
```

this would be the same as

```
print A     // 200
print *A    // 2
print *(A+2) // 6
```

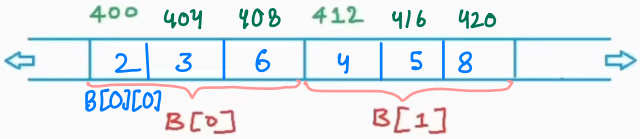
*(A+i) is same as A[i]

A+i is same as &A[i]

B[0][3] B[1][3]

```
int B[2][3]
```

→ creating array of array



B[0] - 1-D array of 3 integers

B[1] - " " "

```
int *p = B
```

→ will return a pointer to 1D array of 3 integers

wrong because it is not a pointer to an integer

correct syntax

pointer array.

```
int (*p)[3] = B;
```

→ will return a pointer to 1-D array of 3 integers

this will return a array

1D array of 3 integers

This is basically the name of second array of 3 integers as by the name of array of three integers we get the pointer to the first element of that array hence we get B[1][0]

// B[1][0]

```
print B or print &B[0]   // 400
print *B or B[0] or &B[0][0] // 400
print (B+1) or &B[1]     // 400 + 1x3x4
print *(B+1) or B[1] or &B[1][0]   // 412.
```

pointer to array of 3 integers

```
print *(B+1)+2 = (B[1])+2 or &B[1][2]
```

→ name of array so it gives &B[1][0] → pointer to first element of the array B[1]:

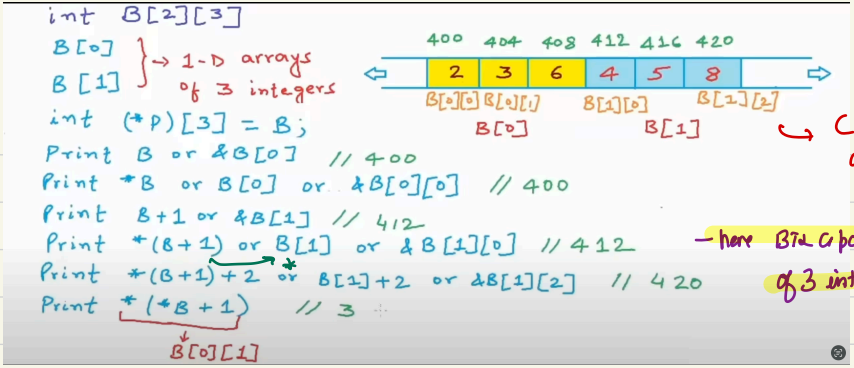
return the whole pointer array

output:- 420

B → int *[3];

```
print(**(B+1)) = B = &B[0] = *(&B[0]) = &B[0][0]
&B[0][0]+1 = &B[0][1]
*(&B[0][1]) = 3
```

→ output.



C stores multi-d arrays in row major order.

— here B is a pointer to an array of 3 integers

* returned array.

B → int *[3] // pointer to integer pointer array

B[0] → int * // points to an integer pointer.

For 2-D array

pointer to 1 D array of 3 integers

B[i][j] = *(B[i]+j) = *(*(B+i)+j)

address of B[i]

p[2] is same as *(p+2)

Function pointers

A variable that stores the address of a function.

- Function pointers are useful when we want to call the function dynamically.

syntax: function return type (*pointer name)(function argument list)

Ex:

```
void hello()
{
printf("Hello world");
}
```

```
void (*ptr)() = &hello;
```

fn return type → void; arg list → (); pointer name → (*ptr)

```
(*ptr)();
```

→ calling a function using function pointer

```
int add(int x,int y)
{ return x+y;
}
```

```
int (*ptr)(int,int) = &add;
```

pointer name → (*ptr); arg list → (int,int); return type → int

```
int a = (*ptr)(3,10);
```

→ calling

↳ storing the returned value in a.

also be written as :-

```
int *ptr(int,int);
```

→ declaration

```
ptr = add;
```

→ initialization

using fn name returns the add of the function

functionptr.c > main()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
void print(int n)
{
    printf("HELLO WORLD!!");
}
int main()
{
void (*ptr)(int)=print;
(*ptr)(5);
return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
void function1()
{
    printf("THIS IS FUNCTION 1\n");
}
void function2()
{
    printf("THIS IS FUNCTION 2\n");
}
int main()
{
void (*arr[2])()={function1,function2};
arr[0]();
arr[1]();
return 0;
}
```

trc > main()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
int function1()
{
    printf("THIS IS FUNCTION 1\n");
    return 0;
}
int function2()
{
    printf("THIS IS FUNCTION 2\n");
    return 0;
}
int main()
{
int (*arr[2])()={function1,function2};
(arr[0])();
(arr[1])();
return 0;
}
```

or (*arr[0])();

```
void swap ( int* a , int* b) {
int c;
c = *a;
*a = *b;
*b = c; }
```

```
void (* ptr) ( int* , int* );   → declaration of function pointer
ptr = swap;                     → initialization of function pointer
(*ptr) ( &x, &y);               → calling of function pointer
```

Array of function pointers : Dynamic function calling without using if-else or switch case

```
float add( int a,int b) { return a+b;}
float subtract (int a,int b) { return a-b;}
float multiply (int a, int b) { return a*b;}
float divide ( int a, int b) { return a/b;}
int main()
{
float (*arr[4])(int, int) = { add, subtract, multiply, divide};
float ans = (*arr[1])(30, 10);
printf("%f", ans);
}
```

pointer array name, return type, initialization

calling ≈ `*(&subtract)(30,10);` (equivalent to)

return type of the functions

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
void swap(int* a,int* b)
{
    int c=*a;
    *a=*b;
    *b=c;
}
int main()
{
int a=9,b=6;
printf("BEFORE SWAPPING: a = %d b = %d \n",a,b);
void (*ptr)(int*,int*)=swap;
(*ptr)(&a,&b);
printf("AFTER SWAPPING: a = %d b = %d \n",a,b);
return 0;
}
```

Null pointer

It is a pointer that doesnt point to any location

syntax: `type ptr_name = NULL or 0;`

Uses of null pointer

* used to initialize pointers when it hasn't been assigned any valid memory address
* To check whether a pointer is null or not before de-referencing it
* It is used in DSA like trees, linked lists etc to indicate the end.
* To pass a null pointer to a function argument when we dont want to pass any valid memory address.

```
int *ptr = NULL;
printf("%d", *ptr);        // 0
if (ptr == NULL);          // returns true because pointer is null.

void passnull (int *value)
{ if( value== NULL)
  printf("Null pointer passed");
  return;
}
}
int main ()
{
passnull(NULL);
}
```

Passing null to a function

Null Pointer	Void Pointer
It is a value.	It is a type.
any pointer can be assigned null	It can only be of type void
A null pointer doesn't point to anything it just a special reserved value for pointers.	It points to a memory location that contains typeless data
All null pointers are equal	void pointers can be different

Dangling Pointers

They are the pointers that refer to a memory location that has been freed.

Uninitialized pointers / Wild Pointers

Using a pointer without assigning a valid address.

```c
#include <stdio.h>
int main() {
  int *p;  // Uninitialized pointer
  *p = 10;  // Attempt to dereference and assign value
  printf("Value of *p: %d\n", *p);  // Undefined behavior
  return 0;
}
```

Output:
Segmentation fault

correct way

```c
int x=10;
int *p;
p= &x;
*p= 20;
```

p= &x; → a pointer pointing to x

*p= 20; → modifies x

Common mistakes while using pointers

* using uninitialized pointers
* Dangling pointers
* incorrect pointer arithmetic.

ADVANTAGES OF POINTERS

1. Passing pointers to a function saves memory as duplicates are not made.
2. Pointers can be used to easily manipulate and access data
3. Pointers can be used to return multiple vales from a function.
4. It can be used to access and manipulate data stored at the location stored in the pointer.

FUNCTION POINTERS

2. They are useful when we want to call a function dynamically.

ARRAY OF FUNCTION POINTERS

1. When we want to call functions dynamically.

NULL POINTERS

1. It is a value assigned to a pointer.
2. Used in data structures to indicate an end.
3. Used when we dont want to assign a pointer any valid memory address

DANGLING POINTERS

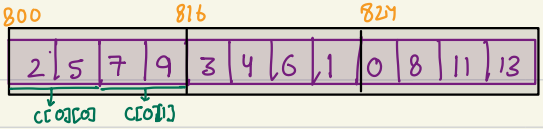
It is a pointer that refers to a memory location that has been freed.

WILD POINTERS

Using a pointer without [redacted]ferencing it.

Continued

Q) int C[3][2][2] =

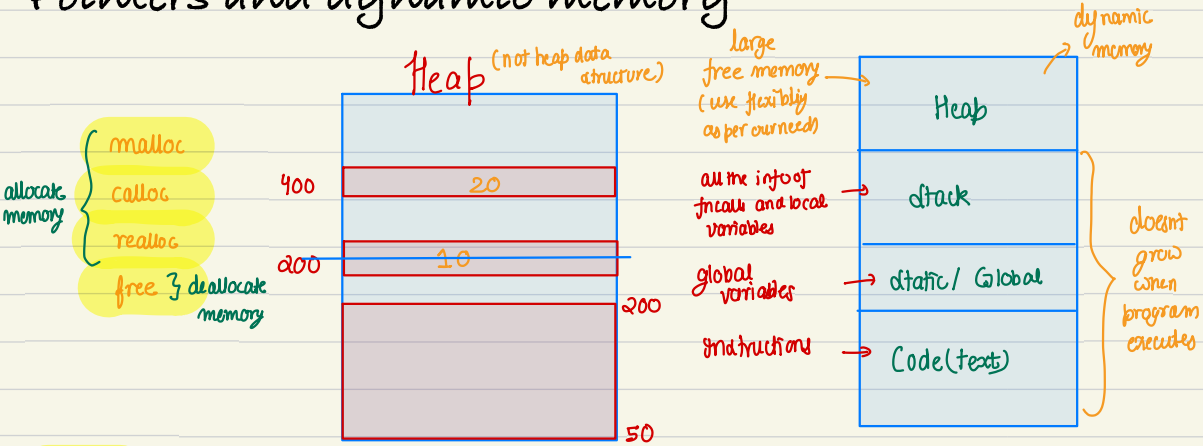


Print(*(C[0][1]+1)) output 9

print *(C[1]+1)) give the name of the array {6,1} name → pointer to first element of the array

= 824

Pointers and dynamic memory



Malloc

int *p = (int*) malloc (sizeof(int)); ← Hey! give me a block of 4 bytes on heap // returns 200

*p = 10; free(p); ← Passing the add of the memory needed to be free

p = (int *) malloc (sizeof(int)); // returns 400

*p = 20; ← it returns a void pointer so we need to type cast it

These allocated memory doesn't clean on its own therefore we also need to free this memory

To allocate memory for an array

p = (int*) malloc (20 * sizeof(int)); point to 50

const void a → address cannot be modified

malloc - `void* malloc ( size_t size)` → asks for size in bytes → returns the pointer to the first byte of the allocated memory

returns the void pointer

unsigned integer

`void *p = malloc ( 3 * sizeof (int))`

- `void *` cannot be dereferenced
- `3` → no of elements
- `sizeof(int)` → size of one unit

`int *p = (int*) p` or `(int *) malloc(3 * sizeof(int))`

- doesn't initialise allocated memory so there would be garbage value stored there
- if memory is not allocated then null pointer is returned

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

- contiguous memory allocation
- num → no of units/elements
- size → size of datatype
- initialise all allocated memory to 0.

To change the size of pre allocated block of allocated memory

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

- ptr → pointer to the starting address of existing block
- size → size of new block

- extended block of previous block (if contiguous memory available)
- make new block and copy the prev block. (if contiguous memory not available)

dynamic memory > C malloc.c > main()

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n;
printf("ENTER THE SIZE OF THE 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]);
}
return 0;
}
```

Dynamic memory > C malloc.c > main()

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n;
printf("ENTER THE SIZE OF THE 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]);
}
return 0;
}
```

Output: 0 0 0 0 0 0

↳ if malloc used then garbage values

dynamic memory > C malloc.c > main()

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n;
printf("ENTER THE SIZE OF THE 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]);
}
return 0;
}
```

free the memory

Output: garbage value as memory has been freed

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n;
printf("ENTER THE SIZE OF THE 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;
}
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<n/2;i++)
{
    printf("%d ",B[i]);
}
return 0;
}
```

important

here previous block of memory is halved

if we increase the size of the ptr then either previous block of memory is allocated or a new block is created and a copy is created

int *A = (int*)realloc(A,0) will be equivalent to free(A)

int *B = (int*) realloc(NULL, n*sizeof(int)); will be equivalent to int *A = (int*)malloc(n*sizeof(int));

While returning pointers from a fn we have to be very careful and check that memory isn't being freed and reallocated as in the stack - here we can use heap in which we have full control over the memory allocation and deallocation.

```c
#include <stdio.h>
#include <stdlib.h>
int* Add(int *a,int *b)
{
    int* c=(int*)malloc(sizeof(int));
    *c=*a+*b;
    return c;
}
void print()
{
    printf("HELLO WORLD!!!\n");
}
int main()
{
    int x,y;
x=20;
y=60;
int *ans=(Add(&x,&y));
print();
printf("%d \n",*ans);
return 0;
}
```

Static memory allocation	Dynamic memory allocation
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

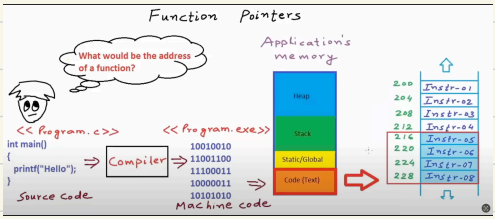
S.No.	malloc()	calloc()
1.	A function that creates one block of memory of a fixed size.	A function that assigns a specified number of blocks of memory to a single variable.
2.	It only takes one argument	Takes two arguments.
3.	It is faster than calloc.	slower than malloc()
4.	It is used to indicate memory allocation	Used to indicate contiguous memory allocation
5.	Syntax : void* malloc(size_t size);	Syntax : void* calloc(size_t num, size_t size);
6.	It does not initialize the memory to zero	Initializes the memory to zero
7.	Does not add any extra memory overhead	Adds some extra memory overhead

calloc → multiple blocks of memory

Pointers to functions

Function is nothing but instructions stored in contiguous block of memory

address of the function would be the address of first instruction of the function (entry point of the function)



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

argument type

int (*p)(int,int) = &Add; or Add

↓ return type of the fn

dereferencing and executing the fn. → int c = (*p)(2,3);

print(c) // 5

If int *p(int,int) were declaring a function which returns a pointer to integer.

```c
//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()
{
    B(A);
}
```