

Activate Windows
Go to Settings to activate Windows.

UNIT – IV

SLO – 1 :

Passing array elements to function

Passing individual array elements

- In C programming, you can pass an entire array to functions.
- Passing array elements to a function is similar to passing variables to a function.

Example : Passing arrays to functions

// Program to calculate the sum of array elements by passing to a function

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

```
float calculateSum(float age[]);
```

```
int main() {
```

```
    float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18}; clrscr();
```

```
    // age array is passed to calculateSum()
```

```
    result = calculateSum(age);
```

```
    printf("Result = %.2f", result);
```

```
    getch();
```

```
    return 0;
```

```
}
```

```
float calculateSum(float age[]) {
```

```
    float sum = 0.0;
```

```
    int i;
```

```
    for (i = 0; i < 6; ++i) {
```

```
sum += age[i];
```

```
    }
```

```
    return sum;
```

```
}
```

OUTPUT

```
Result = 162.50
```

SLO – 2 :

Formal and actual parameters

Actual Parameters / arguments

- Arguments which are mentioned in the function call is known as the actual argument. For example:

```
func1(12, 23);
```

- here 12 and 23 are actual arguments.
 - Actual arguments can be constant, variables, expressions etc.
1. `func1(a, b);` // here actual arguments are variable
 2. `func1(a + b, b + a);` // here actual arguments are expression

Formal Parameters / arguments

- Arguments which are mentioned in the definition of the function is called formal arguments.
- Formal arguments are very similar to local variables inside the function.
- Just like local variables, formal arguments are destroyed when the function ends.

```
int factorial(int n)
{
    // write logic here
}
```

Here n is the formal argument.

Formal and actual parameters

- Things to remember about actual and formal arguments.
- Order, number, and type of the actual arguments in the function call **must match** with formal arguments of the function.
- If there is type mismatch between actual and formal arguments then the compiler will try to convert the type of actual arguments to formal arguments if it is legal, Otherwise, a garbage value will be passed to the formal argument.
- Changes made in the formal argument do not affect the actual arguments.

SLO – 1 :

Advantages of using functions

Advantages of using functions

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

SLO – 2 :
Processor Directives and #define Directives

Preprocessor Directives

- The preprocessor directives give instruction to the compiler to preprocess the information before actual compilation starts.
- All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line
- Any line in a program that begin with # indicates the Preprocessor directives .
- Followed by #, an identifier in the line of code indicates the directive name.
- For example, #define where “define” is the name of the directive

List of preprocessor directives

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #pragma

Preprocessor directives

#include

- The #include preprocessor directive is used to paste code of given file into current file.
- It is used to include system-defined and user-defined header files.
- If included file is not found, compiler renders error.
It has three variants:

#include <file>

- This variant is used for system header files.
- It searches for a file named file in a list of directories specified by us, then in a standard list of system directories.

Preprocessor directives

#include "file"

- This variant is used for header files of **your own program**.
- It **searches** for a file named file first in the **current directory**, then in the same directories used for system header files.
- The current directory is the directory of the current input file.

#include anything else

- This variant is called **computed #include**
- This can help in **porting the program to various operating systems** in which the necessary system header files are found in different places.

Preprocessor directives

Macro's (#define)

- A **macro** is a fragment of code which has been given a name.
- Whenever the name is used, it is replaced by the contents of the **macro**

#define token value There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros

- The object-like macro is an identifier that is replaced by value.
- It is widely used to represent numeric constants.

Preprocessor directives

For example:

```
#include<stdio.h>
#define PI 3.1415
main()
{
printf("%f",PI);
}
```

Output

3.14000

Preprocessor directives

Function-like Macros

- The function-like macro looks like function call.
- For example: `#define MIN(a,b) ((a)<(b)?(a):(b))`

Here, MIN is the macro name.

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

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

```
void main()
```

```
{
```

```
printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
```

```
}
```

Output:

Minimum between 10 and 20 is: 10

Preprocessor directives

#undef

To undefine a macro means to **cancel its definition**.
This is done with the #undef directive.

Syntax:

#undef token define and undefine

Example

```
#include <stdio.h>
#define PI 3.1415
#undef PI
main()
{   printf("%f",PI);
}
```

Output

Preprocessor directives

#ifdef

The #ifdef preprocessor directive checks if macro is defined by #define.

If yes, it executes the code.

Syntax:

```
#ifdef MACRO //code #endif
```

#ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code.

Syntax:

```
#ifndef MACRO //code #endif
```

Preprocessor directives

#if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code.

Syntax:

```
#if expression //code #endif
```

#else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

```
#if expression //if code #else //else code #endif
```

Syntax with #elif

Preprocessor directives

Example

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main()
{
    #if NUMBER==0
    printf("Value of Number is: %d",NUMBER);
    #else printf("Value of Number is non-zero");
    #endif
    getch();
}
```

Output

Value of Number is non-zero

Preprocessor directives

#error

- The #error preprocessor directive indicates error.
- The compiler gives fatal error if #error directive is found and skips further compilation process.

//macro5.c

C #error example

```
#include<stdio.h>
#ifndef MATH_H
#error First include then compile
#else
void main()
{
float a; a=sqrt(7);
printf("%f",a);
}
```

Preprocessor directives

#pragma

- The #pragma preprocessor directive is used to provide additional information to the compiler.
- The #pragma directive is used by the compiler to offer machine or operating-system feature.
- Different compilers can provide different usage of #pragma directive.

Syntax:

#pragma token

Preprocessor directives

Example

```
#include<stdio.h>
#include<conio.h>
void func() ;
#pragma startup func
#pragma exit func
void main()
{
printf("\nI am in main");
getch(); }
void func(){
printf("\nI am in func");
getch();
}
```

Output

```
I am in func
I am in main
I am in func
```

SLO – 1 :

Nested Preprocessor Macros

Nested Preprocessor Macros

1. A macro defined within another macro is called nested macro
2. Macro name within another macro is called **Nesting of Macro**.
3. Whenever the **macro name is encountered** , the arguments are replaced by the **actual arguments from the program**.

Nested Preprocessor Macros -Example

```
#include<stdio.h>                                // nestedm1.c
#define SQU(x)((x)*x)
#define CUBE(x)(SQU(x)*x)
int main()
{
    int x; int y; x = SQU(3); // Argumented Macro
    y = CUBE(3); // Nested Macro
    printf("\nSquare of 3 : %d",x);
    printf("\nCube of 3 : %d",y);
    return(0);
}
```

Output :

Square of 3 : 9
CUBE of 3 : 27

Advantages and Disadvantages of Macros

Advantages of Macros

- Time efficiency.
- No need to pass arguments like function.
- It's preprocessed.
- Easier to Read

Disadvantages of Macros

- Very hard to debug in large code.
- Take more memory compare to function

SLO – 2 :

Advantages of using functions


Advantages of using functions

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

SLO – 1 :
Pointers and address operator

POINTERS

- A pointer is a variable that contains the address of the another variable.
- In other words, it points to the location of a variable and can indirectly access the variable.
- Pointers are declared using the * symbol.

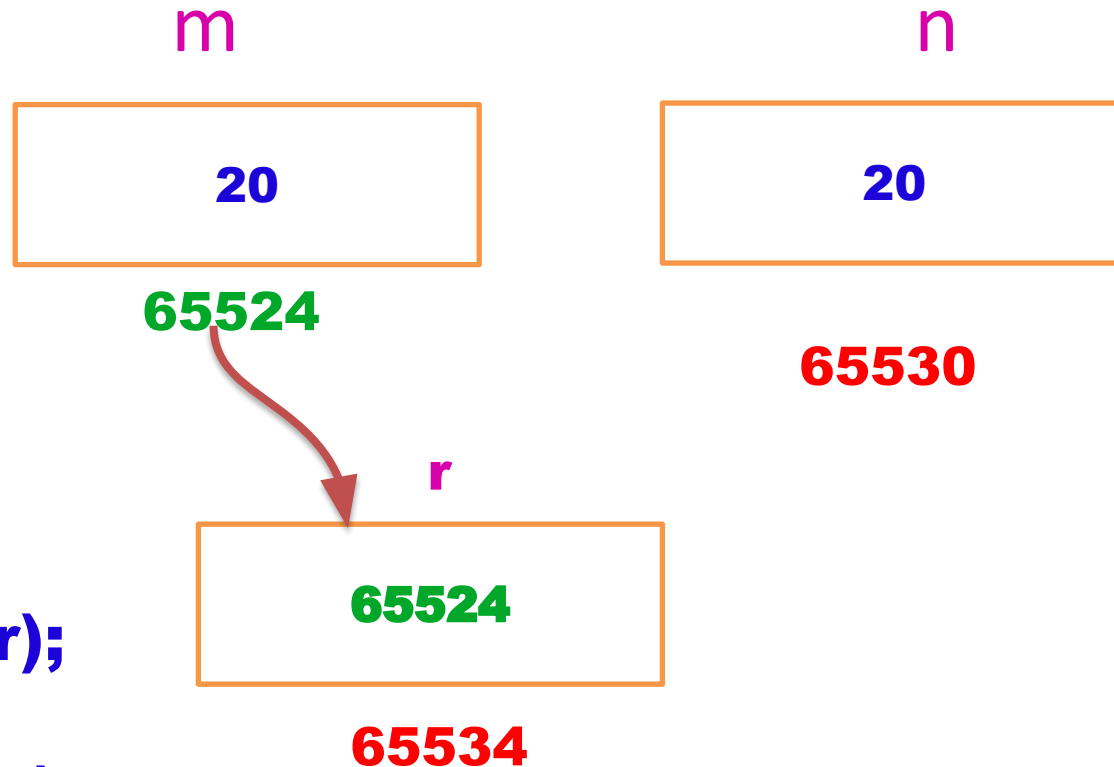
*  Asterisk

Example : `int *m;`

POINTERS

(pointerexample1.C)

```
void main()
{
  int m=20,n;
  int *r;
  r=&m;
  printf("%u",r);
  n=*r;
  printf("%d",n);
}
```



POINTERS

- Examples of pointer declarations:

```
FILE *fptr;
```

```
int *a;
```

```
float *b;
```

```
char *c;
```

- The **asterisk**, when used as above in the declaration, tells the compiler that the variable is to be a pointer, and the type of data that the pointer points to, but **NOT** the name of the variable pointed to.

POINTERS

- Pointers are variables that contain *memory addresses* as their values.
- A variable name *directly* references a value. (m=10)
- A pointer *indirectly* references a value. Referencing a value through a pointer is called *indirection*.
- A pointer variable must be declared before it can be used.

Concept of Address and Pointers

- Memory can be conceptualized as a linear set of data locations.
- Variables reference the contents of a locations
- Pointers have a value of the address of a given location

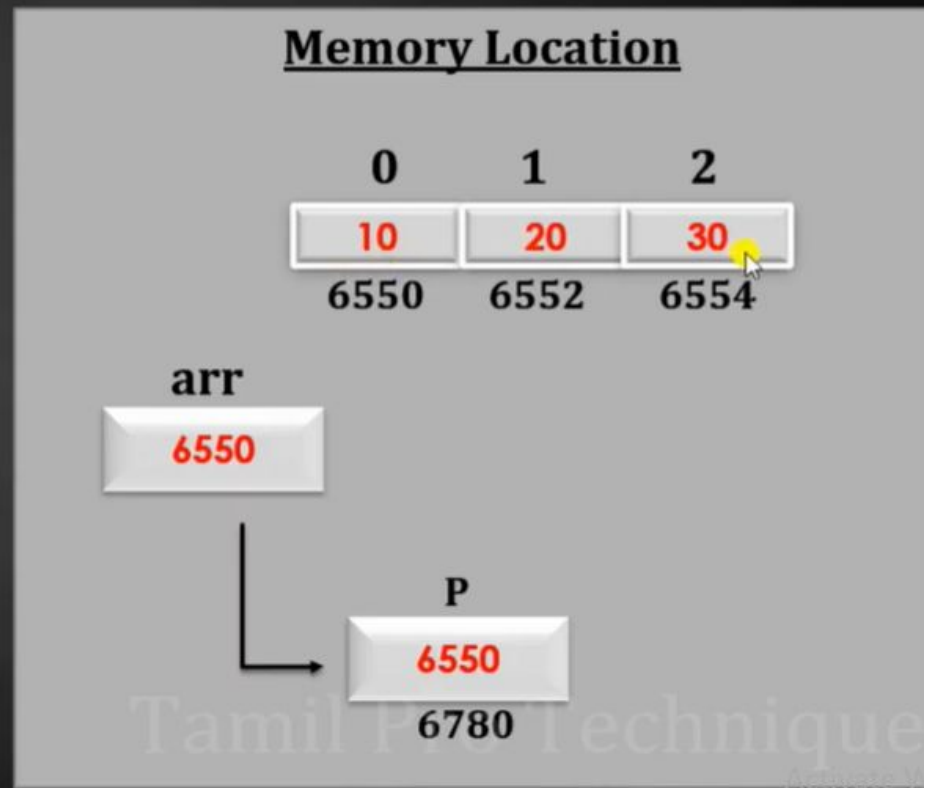
ADDR1	Contents1
ADDR2	
ADDR3	
ADDR4	
ADDR5	
ADDR6	
-	
-	
-	
ADDR11	Contents11
-	
-	
-	
ADDR16	Contents16

Act

Array and Pointers

Sample Program:

```
void main( )  
{  
    int arr[3]={10,20,30};  
    int *P;  
    P=arr; or P=&arr[0];  
  
    printf("%u", P);  
  
    printf("%d", *P);  
}
```



Array and Pointers

(pointere.C)

```
#include<stdio.h>
#include<conio.h>

void main()
{
int arr[5]={10,20,30,40,50};
int *p,i;
clrscr();
p=&arr[0];
for(i=0;i<5;i++)
printf("%u\n",*(p+i));
getch();
}
```

Pointer and address operator (use of & and *)

- When is & used?
- When is * used?

& -- "address operator" which gives or produces the memory address of a data variable

* -- "dereferencing operator" which provides the contents in the memory location specified by a pointer

Pointers and Functions

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.
- We looked earlier at a swap function that did not change the values stored in the main program because only the values were passed to the function swap.
- This is known as "call by value".

Pointers and Functions

- If instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference" since we are referencing the variables.
- The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now actually swapped when the control is returned to main function.

Pointers with Functions (example)

```
#include <stdio.h>
void swap ( int *a, int *b ) ;
int main ( )
{
    int a = 5, b = 6;
    printf("a=%d b=%d\n",a,b) ;
    swap (&a, &b) ;
    printf("a=%d b=%d\n",a,b) ;
    return 0 ;
}
```

```
void swap( int *a, int *b )
{
    int temp;
    temp= *a; *a= *b; *b =
temp ;
    printf ("a=%d b=%d\n", *a,
*b);
}
```

Results:

a=5 b=6

a=6 b=5

a=6 b=5

SLO – 2 :
**Size of Pointer Variables and
pointer operator**

To Find Size of Pointer Variables

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

```
int main()
```

```
{clrscr();
```

```
    printf("Size of int pointer = %d bytes.\n", sizeof(int*));
```

```
    printf("Size of char pointer = %d bytes.\n", sizeof(char*));
```

```
    printf("Size of float pointer = %d bytes.\n", sizeof(float*));
```

```
    printf("Size of double pointer = %d bytes.\n", sizeof(double*));
```

```
    printf("Size of long int pointer = %d bytes.\n", sizeof(long*));
```

```
    printf("Size of short int pointer = %d bytes.\n", sizeof(short*));
```

```
    getch();
```

```
    return 0;
```

```
}
```

OUTPUT

```
Size of int pointer = 2 bytes.  
Size of char pointer = 2 bytes.  
Size of float pointer = 2 bytes.  
Size of double pointer = 2 bytes.  
Size of long int pointer = 2 bytes.  
Size of short int pointer = 2 bytes.
```

To Find Size of Pointer Variables

- If you observe the output of C program you can see that pointer variables, irrespective of their data type, consume 2 ~~or 4~~ bytes of data in the memory.

Pointer Operators

- There are two special operators that are used with pointers * and &.
- The & is a unary operator that returns the memory address of its operand, for example

```
bal=&balance;
```

- outputs into bal the memory address of the variable balance. This address is the location of the variable in the computer's internal memory. It has nothing to do with the value of balance. The statements can be verbalized as bal receives the address of balance. This operator was used previously in the functions section.
- The second operator is *, and it is a complement of &. It is a unary operator that returns the value of the variable at the address specified by its operand. Consider the example below:
value=*balance;
- This operation will balance the value of balance into value

Example : Pointer Operators

```
#include <stdio.h> \nmain()\n{\nint balance;\nint *bal;\nint value;\nbalance=3200;\nbal=&balance;\nvalue=*bal;\nprintf("Balance is %d",value);\nreturn 0;\n}
```


OUTPUT

Balance is 3200

SLO – 1 :
**Pointers declaration and
dereferencing pointers**

Pointers declaration

- A pointer is a variable that contains the address of the another variable.
- In other words, it points to the location of a variable and can indirectly access the variable.
- Pointers are declared using the * symbol.

*  Asterisk

Example : `int *m;`

Dereferencing a void pointer

(getting the real content from a memory location)

- We have seen upto assigning an address to a pointer variable.
- Now how to get the content inside the memory location using the same pointer variable? For that we use the same **indirection operator *** used to declare a pointer variable.
- Consider the scenario:-

Dereferencing a void pointer

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

```
void main()  
{
```

```
int *ptr; // Declaring the pointer variable 'ptr' of data type int
```

```
int a=10; // Declaring a normal variable 'a' and assigning value 10 to  
it.
```

```
ptr=&a; // Assigning address of variable 'a' to pointer variable 'ptr'
```

```
printf("The value inside pointer is= %d",*ptr); // See the value is  
printed using *ptr;
```

```
}
```

Output:-

The value inside pointer is= 10

Dereferencing a void pointer

- We have seen about dereferencing a pointer variable.
- We use the **indirection operator** `*` to serve the purpose.
- But in the case of a void pointer we need to **typecast the pointer variable to dereference it**.
- This is because a void pointer has **no data type associated with it**.
- There is **no way the compiler can know** (or guess?) what type of data is pointed to by the void pointer.
- So to take the data pointed to by a void pointer we typecast it with the correct type of the **data holded inside the void pointers location**.

Dereferencing a void pointer

(voidpointer.c)

Example program:-

```
#include
```

```
void main()
```

```
{
```

```
int a=10;
```

```
float b=35.75;
```

```
void *ptr; // Declaring a void pointer
```

```
ptr=&a; // Assigning address of integer to void pointer.
```

```
printf("The value of integer variable is= %d",*( (int*) ptr) );// (int*)ptr - is used for type casting.  
Where as *((int*)ptr) dereferences the typecasted void pointer variable.
```

```
ptr=&b; // Assigning address of float to void pointer.
```

```
printf("The value of float variable is= %f",*( (float*) ptr) );
```

```
}
```

The output:-

The value of integer variable is= 10

The value of float variable is= 37.75

SLO – 2 :
**Void pointers and size of Void
pointer**

void pointers

- A pointer variable is usually declared with the data type of the “content” that is to be stored inside the memory location (to which the pointer variable points to).

Ex:- `char *ptr; int *ptr; float *ptr;`

- A pointer variable declared using a particular data type can not hold the location address of variables of other data types.
- It is **invalid** and will result in a **compilation error**.

void pointers

Ex:- char *ptr;

int var1;

ptr=&var1; // This is invalid because 'ptr' is a character pointer variable.

- Here comes the importance of a “void pointer”. A void pointer is nothing but a pointer variable declared using the reserved word in C 'void'.
- Ex:- void *ptr; // Now ptr is a **general purpose pointer variable**
- When a pointer variable is declared using keyword void – it becomes a **general purpose pointer variable**.
- Address of any variable of any data type (char, int, float etc.) can be assigned to a void pointer variable.

size of void pointer

(sizeofvoid.C)

- Any pointer be it a void, int, float pointer will be equal to the size of the word size (generally equal to the int size of the machine)

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

```
int main()  
{  
    void *p;  
    int n;  
    n = sizeof(p);  
    printf("%d\n",n);  
    return 0;  
}
```

SLO – 1 :

Arithmetic operations

Pointer arithmetic

- C programming **allow programmers** just like you to do arithmetic operations using pointers.
- So, programmers can perform any arithmetic operations using pointer variables. Performing arithmetic operations using pointer variables is said to be **arithmetic pointer**.
- **Note:** We **cannot add two pointers**. This is because pointers contain addresses, adding two addresses makes **no sense**, because you have no idea what it would point to.
- But we **can subtract two pointers**. This is because difference between two pointers gives the number of elements of its data type that can be stored between the two pointers.

C Program - Pointer Arithmetic (sumpointers.c)

```
#include <stdio.h>
int main()
{
    int a = 10, b = 6;
    int *ptr, *ptr1;
    clrscr();
    ptr = &a;
    ptr1 = &b;
    printf("sum of two pointers : %d \n", *ptr+b);
    printf("Subtraction of two pointers : %d \n", a-*ptr1);
    getch();
    return 0;
}
```

Note:

In the above program, two integer pointers ptr and ptr1 are declared and initialized by the address of a and b respectively. Then, *ptr + b and a - *ptr1 is used to perform Addition and Subtraction using pointers.

SLO – 2 :

Incrementing Pointer

Incrementing Pointer variable (increpointer.c)

```
#include <stdio.h>
int main()
{
    int arr[3] = {10, 11, 12};
    int *ptr, i;
    ptr = arr;
    for(i = 0; i < 3; i++)
    {
        printf("%d\t", *ptr);
        ptr++;
    }
    return 0;
}
```

Note:

Here pointer variable ptr is initialized with the address of **first element in an array arr**. We know that array stores the elements consecutively in it. Then its very simple for a pointer to point to the next memory location using increment operator.

Decrementing Pointer Variable (decrepointer.c)

```
#include <stdio.h>
int main()
{
int a[3] = {10, 11, 12};
int *ptr, i;
ptr = &a[2];
for(i = 3; i > 0; i--)
{
printf("%d\t", *ptr);
ptr--;
}
return 0;
}
```

Note:

Here pointer variable ptr is initialized with the address of **last element in an array arr**. We know that array stores the elements consecutively in it. Then its very simple for a pointer to point to the previous memory location using increment operator.

Comparing Pointer Variable

- Pointer variable can be compared either with other pointer variable or with normal variable. Let us using greater than operator to compare two pointer variables.

Comparing Pointer Variable (compointer.c)

```
#include <stdio.h>
int main()
{
    int a = 10, b = 6;
    int *ptr, *ptr1;
    ptr = &a;
    ptr1 = &b;
    if(*ptr > *ptr1)
        printf("value stored in a is greater than b ");
    else
        printf("value stored in b is greater than a ");
    return 0;
}
```

Note:

Here *ptr and *ptr1 will fetch the value from the address stored in it and greater than (>) operator helps us to find greatest among two pointer variables.

Program for pointer arithmetic arithmeticpointer.C

```
#include <stdio.h>

int main()
{
    int m = 5, n = 10, o = 0;

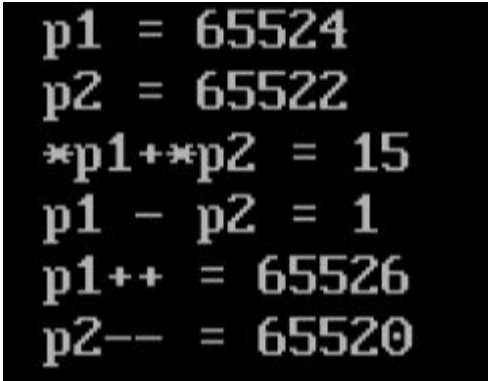
    int *p1;
    int *p2;
    int *p3;
    clrscr();
    p1 = &m; //printing the address of m
    p2 = &n; //printing the address of n

    printf("p1 = %d\n", p1);
    printf("p2 = %d\n", p2);

    o = *p1+*p2;
    printf("*p1+*p2 = %d\n", o); //point 1
```

Program for pointer arithmetic

```
p3 = p1-p2;  
printf("p1 - p2 = %d\n", p3); //point 2  
  
p1++;  
printf("p1++ = %d\n", p1); //point 3  
  
p2--;  
printf("p2-- = %d\n", p2); //point 4  
  
//Below line will give ERROR  
printf("p1+p2 = %d\n", p1+p2); //point 5  
getch();  
return 0;  
}
```



```
p1 = 65524  
p2 = 65522  
*p1+*p2 = 15  
p1 - p2 = 1  
p1++ = 65526  
p2-- = 65520
```

Pointer arithmetic

Explanation of the above program:

Point 1: Here, * means 'value at the given address'. Thus, it adds the value of m and n which is 15.

Point 2: It subtracts the addresses of the two variables and then divides it by the size of the pointer datatype (here integer, which has size of 2 bytes) which gives us the number of elements of integer data type that can be stored within it.

Point 3: It increments the address stored by the pointer by the size of its datatype(here 2).

Point 4: It decrements the address stored by the pointer by the size of its datatype(here 2).

Point 5: Addition of two pointers is not allowed.

SLO – 1 :

Constant Pointers

Constant Pointers

- A constant pointer is a pointer that cannot change the address its holding.
- In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.

A constant pointer is declared as follows :

```
<type of pointer> * const <name of pointer>
```

An example declaration would look like :

```
int * const ptr;
```

Example

(CONSTANTPTR1.C)

```
#include<stdio.h>
int main(void)
{
    int var1 = 0, var2 = 0;
    int *const ptr = &var1;
    clrscr();
    ptr = &var2;
    printf("%d\n", *ptr);
    getch();
    return 0;
}
```

In the above example :

- We declared two variables var1 and var2
- A constant pointer 'ptr' was declared and made to point var1
- Next, ptr is made to point var2.
- Finally, we try to print the value ptr is pointing to.

Constant Pointers

- we assigned an address to a constant pointer and then tried to change the address by assigning the address of some other variable to the same constant pointer.
- So we see that while compiling the compiler complains about 'ptr' being a read only variable.
- This means that we cannot change the value ptr holds.
- Hence we conclude that a constant pointer which points to a variable cannot be made to point to any other variable.

SLO – 2 :

Pointer to array elements and string

Pointer to array elements (ptrtoarray.C)

- Pointer to an array is also known as array pointer. We are using the pointer to access the components of the array

```
#include <stdio.h>
int main()
{
    int i;
    int a[5] = {10, 20, 30, 40};
    int *p = a;    // same as int*p = &a[0]
    clrscr();
    for (i = 0; i < 4; i++)
    {
        printf("%d", *p);
        p++;
    }
    getch();
    return 0;
}
```

Pointer to array elements

- Array of size 4 stores four element.
- Pointer variable "p" is declared, using "*" sign pointer is defined and stores the value of the array in to it.
- Using the for loop, *p is printed.
- Element at location of a[0], a[1], a[2], a[3] is printed which is stored in *p.

Pointer to string

(ptrtosting.c)

- pointer variable declared as string stores the string of the defined length and can be accessed as *variable[value]*

```
#include <stdio.h>
#include <string.h>
int main()
{   char *str = "SRM UNIVERSITY";
    int i, j = strlen(str);
        clrscr()
        for(i = 0; i < j; i++)
            printf("%c", *str++);
    getch();
    return 0;
}
```

SLO – 1 :

Function Pointers

Function Pointers

- In C programming language, we can have a concept of **Pointer to a function** known as **function pointer in C**.

- **What is a function in C?**

A function is a block of code, implemented for a specific task, with a given function name.

- **What is a pointer in C?**

Pointer is a variable that stores the address of another variable.

Function Pointers

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions.

How to declare a function pointer?

function_return_type(*Pointer_name)(function argument list)
return_type function_pointer(argu)= &function_name

Examples :

```
double (*p2f)(double, char)    // p2f is name of the function
pointer                        //
void (*fun_ptr)(int) = &fun;    // fun_ptr is name of the
function pointer
```

Function Definition

```
void fun(int a)
{
    printf("Value of a is %d\n", a);
}
```


Example (funptr.c)

```
#include<stdio.h>
void fun(int a)
{
printf("a=%d\n",a);
}
void main()
{
void (*fun1)(int)=&fun;
(*fun1)(15);
}
```

Output:

a=15

Function Pointers

If we **remove bracket**, then the expression

“void (*fun_ptr)(int)”

becomes

“void *fun_ptr(int)”

which is declaration of a function that returns void pointer.

Interesting facts

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer **stores the start of executable code**.
- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- A function's name can also be used to get functions' address.

Example

(funptr1.c)

```
int sum (int num1, int num2)
{
    return num1+num2;
}
int main()
{
```

```
/* The following two lines can also be written in a single
 * statement like this: void (*fun_ptr)(int) = &fun;
 */
int (*f2p) (int, int);
int op1,op2;
f2p = sum;
```

Example

(funptr1.c)

//Calling function using function pointer

```
op1 = f2p(10, 13);
```

//Calling function in normal way using function name

```
op2 = sum(10, 13);
```

```
printf("Output1: Call using function pointer: %d",op1);
```

```
printf("\nOutput2: Call using function name: %d",  
op2);
```

```
return 0;
```

```
}
```

OUTPUT

```
Output1: Call using function pointer: 23  
Output2: Call using function name: 23
```

Some points regarding function pointer:

1. As mentioned in the comments, you can declare a function pointer and assign a function to it in a single statement like this:

```
void (*fun_ptr)(int) = &fun;
```

2. You can even remove the ampersand from this statement because a function name alone represents the function address. This means the above statement can also be written like this:

```
void (*fun_ptr)(int) = fun;
```

SLO – 2 :
Array of Function Pointers

Array of Function Pointers

- **Array**- group of items of homogeneous data type stored under a variable .
- **Functions** will receive arguments from the calling function.
- **Pointer** is a variable that stores the address of another variable.
- With **pointer parameters**, the functions can process actual data rather than a copy of data.

Array of Function Pointers

Syntax:

```
void(*fun_ptr_arrayname[])(data type1, datatype2 ) = {f_1,f_2,f_3,..};
```

Example:

```
void (*fun[])(int, int)={add, sub, mul};
```


Array of Function Pointers

- An array of function pointers can be created for the functions of same return type and taking same type and same number of arguments.
- For example,
 - if you have four function definitions for performing basic arithmetic operations (addition, subtraction, multiplication, and division) on two integer quantities, all these functions will take two integer arguments and will return an integer as a result.
- For such similar~return~type~and~argument operations you can create an array of function pointers and select a proper function by its index rather than its name.
- The following example demonstrates it.

Passing Function Pointer as an argument

- Whether , function can be passed as an argument?
- For achieving this we can pass the function pointer.
- Like normal data pointers, a function pointer can be passed as an argument to a function and can be returned from a function.

Passing Function Pointer as an argument

(funptr2.c)

```
#include<stdio.h>
void fun1(void(*test)())
{
int a=20; test(a);
}
```

```
void test(int a)
{
printf("a=%d\n",a);
}
```

```
void main()
{
fun1(&test);
}
```

Output:

a=20

SLO – 1 :
**Accessing Array of Function
Pointers**

Accessing Array of function pointer arrayfunptr.C

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

```
int sum(int a, int b);
```

```
int subtract(int a, int b);
```

```
int mul(int a, int b);
```

```
int div(int a, int b);
```

```
int (*p[4]) (int x, int y);
```

```
int main(void)
```

```
{
```

```
    int result;
```

```
    int i, j, op;
```

Accessing Array of function pointer

```
p[0] = sum; /* address of sum() */  
p[1] = subtract; /* address of subtract() */  
p[2] = mul; /* address of mul() */  
p[3] = div; /* address of div() */  
  
printf("Enter two numbers: ");  
scanf("%d %d", &i, &j);  
  
printf("0: Add, 1: Subtract, 2: Multiply, 3: Divide\n");  
do {  
    printf("Enter number of operation: ");  
    scanf("%d", &op);  
} while(op<0 || op>3);
```

Accessing Array of function pointer

```
result = (*p[op]) (i, j);  
printf("%d", result);
```

```
    return 0;  
}
```

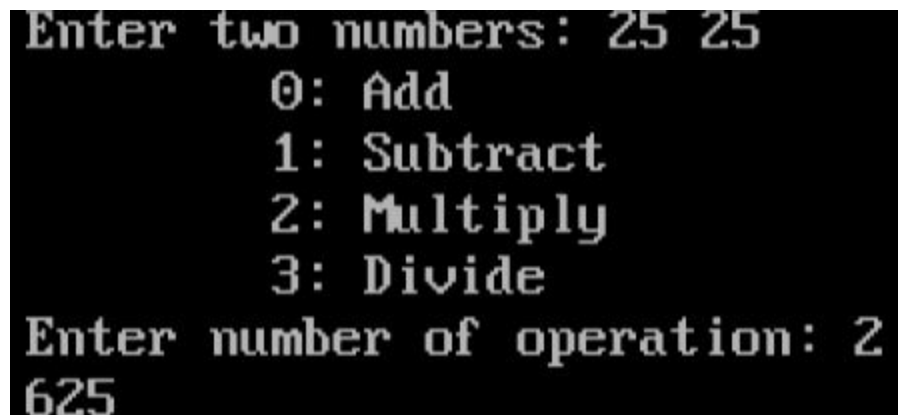
```
int sum(int a, int b)  
{  
    return a + b;  
}
```

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

Accessing Array of function pointer

```
int mul(int a, int b)
{
    return a * b;
}
int div(int a, int b)
{
    if(b)
        return a / b;
    else
        return 0;
}
```

OUTPUT

A screenshot of a terminal window showing the output of a C program. The text is as follows:
Enter two numbers: 25 25
0: Add
1: Subtract
2: Multiply
3: Divide
Enter number of operation: 2
625
The text is displayed in a monospaced font on a black background.

```
Enter two numbers: 25 25
0: Add
1: Subtract
2: Multiply
3: Divide
Enter number of operation: 2
625
```


SLO – 2 : **NULL pointer**

NULL pointer

- NULL Pointer is a pointer which is pointing to nothing.
- Pointer which is initialized with NULL value is considered as NULL pointer.
- The word "**NULL**" is a constant in C language and its value is **0**. In case with the pointers - if any pointer does not contain a valid memory address or any pointer is uninitialized, known as "**NULL pointer**".
- We can also assign **0** (or **NULL**) to make a pointer as "**NULL pointer**".

NULL pointer

Syntax :

data type *pointer_variable=0;

datatype *pointer_variable=NULL;

Example:

```
int *p= NULL
```

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

```
//nullptr1.c
```

```
int main()
```

```
{
```

```
int *p= NULL; //initialize the pointer as null.
```

```
printf ("The value of pointer is %u", p);
```

```
return 0;
```

```
}
```

output :

The value of pointer is

NULL pointer

Example:

- In this example, there are 3 integer pointers ptr1, ptr2 and ptr3.
- ptr1 is initialized with the address of the integer variable num.
- Thus ptr1 contains a valid memory address. ptr2 is uninitialized and ptr3 assigned 0. Thus, ptr2 and ptr3 are the **NULL pointers**.

NULL pointer

NULLPTR.C

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

```
int main(void) {  
    int num = 10;  
  
    int *ptr1 = &num;  
    int *ptr2;  
    int *ptr3=0;  
  
    clrscr();  
    if(ptr1 == 0)  
        printf("ptr1: NULL\n");  
    else  
        printf("ptr1: NOT NULL\n");  
  
    if(ptr2 == 0)  
        printf("ptr2: NULL\n");  
    else  
        printf("ptr2: NOT NULL\n");  
  
    if(ptr3 == 0)  
        printf("ptr3: NULL\n");  
    else  
        printf("ptr3: NOT NULL\n");  
  
    getch();  
    return 0;  
}
```



Uses of Null Pointer

- To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
- To pass a null pointer to a function argument when we don't want to pass any valid memory address.
- To check for null pointer before accessing any pointer variable. So that, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.