Date:
Page:

Unit - 4

Function :

* Formal and Actual Parameters *

- **Formal Parameter:** A variable and its type as they appear in the prototype of the function or method.

- **Actual Parameter:** The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling enviornment.

```
//Invoke (call) the method
int num1 = 25;
int num2 = 47;
int sum = add(num1, num2);  ---> Actual parameters

//method definition
public int add(int x, int y)  --> formal parameters
{
    return (x+y);
}
```

- Formal and Actual Parameters

One - Dimensional Array

- Actual Parameters:

```
int a[10];
```

Function call:

```
add (a);
```

here,

a is the base of the array
add is the function

- Formal Parameters:

within the body

```
void add (int a[])
{
    .......
}
```

int a[] - is a formal parameter

Date:
Page:

- F & A Parameters

Two-Dimensional Array

- Actual Parameters:

```
int a[10][10];
```

Function call:

```
add(a);
```

here,

a is the base address of the array 'a'
b is the base address of the array 'b'
add is the function.

- Formal Parameters:

within ~~body~~ function body

```
void add (int a[][10])
{
    .......
}
```

int a[][10] - is a formal parameter

Date:
Page:

- Passing array to function using call by value method.

```
#include<stdio.h>
void main()
{
    int a[] = {1,2,3,4,5};
    int i;
    for(i=0; i<=4; i++)
    {
        disp(a[i]);
    }
}

void disp(int n)
{
    printf("%d\t",n);
}
```

Output

1 2 3 4 5 exe. time = 0.095s

Date:
Page:

- Passing array to function using call by reference.

- When we pass the address of an array while calling a function then this is called function call by reference.

~~- When we pass the address of an array while calling a function then this.~~

- When we pass an address as an argument, the function declared should have a pointer as a parameter to receive the passed address

<u>pointer</u> - It is a variable that stores the address of another variable.

eg: A integer variable holds (or say stores) an integer value, however an integer pointer holds the address of a integer variable.

```
void main(){
    int a[]={1,2,3,4,5};
    int i;
    for (i=0;i<=4;i++){
        disp(&a[i]);
    }
}
void disp(int *n){
    printf("%d\t", *n);
}
```

Output:

1 2 3 4 5

exe ti: 0.075s

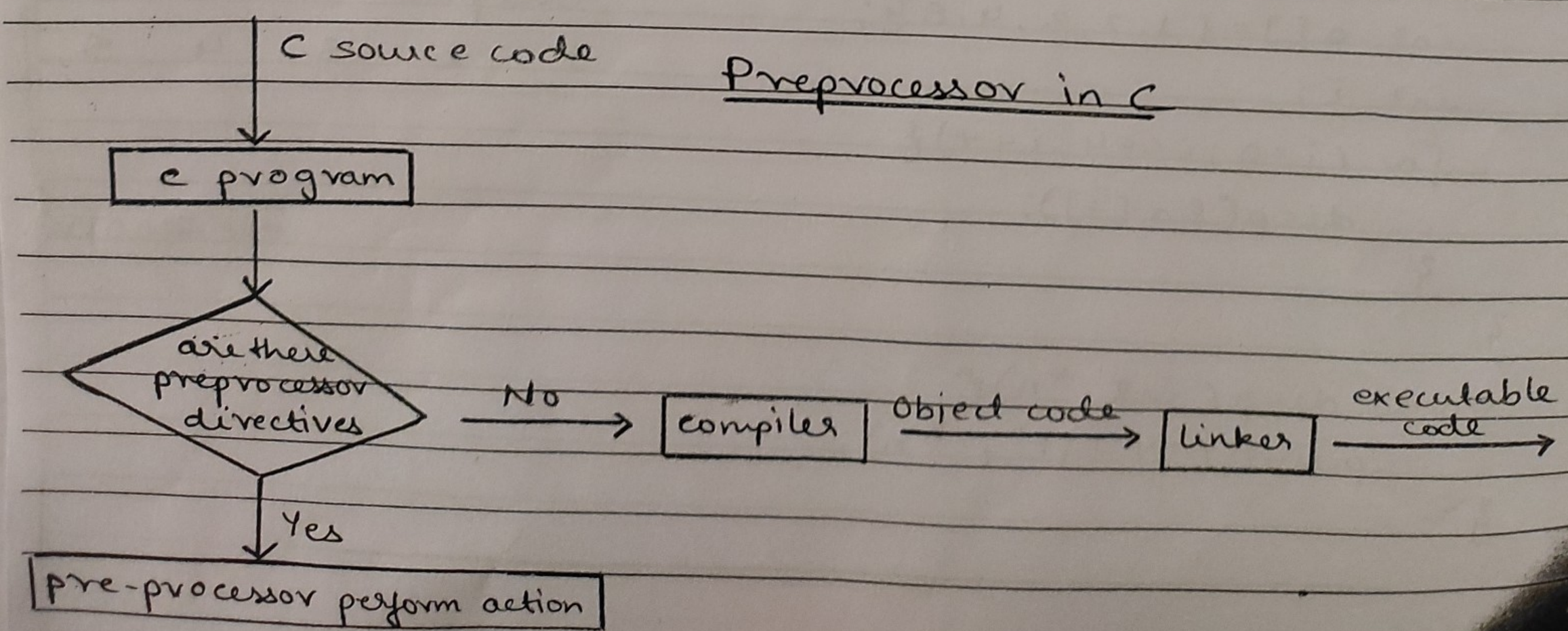
* Advantages of function *

- Avoid Code. Repitition
- 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.

* Preprocessor Directives *

- Preprocessor programs are programs that process our source code before compilation.

- There are a no. of steps involved b/w writing a program and executing a program in C.



Date:
Page:

- The C preprocessor is a micro ~~prep~~ processor that is used by compiler to transform your code before compilation.
It is called micro preprocessor because it allows us to add macros.

- Preprocessor directives are lines in your program that start with #.

- The # is followed by an identifier that is the directive name.

- Whitespace is not allowed before and after #.

- List of preprocessor directives -
  - #include
  - #define
  - #undef
  - #ifdef
  - #ifndef
  - #if
  - #else
  - #elif
  - #endif
  - #error
  - #pragma

* C Macros *

A macro is a segment of code which is replaced by the value of macro.
Macro is defined by #define derivative.

There are two types of macros:-
- Object - like Macros
- Function - like Macros

1) Object - like Macros

- The object - like macro is an identifier that is replaced by value.

It is widely used to represent numeric constants.
eg: # define PI 3.14

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

Output:
3.14000

Date:
Page:

2) Function-like Macros

- The function-like macros looks like function call.

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

- MIN is a macro name.

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
{
    printf("Min. b/w 10 & 20 is:%d\n",MIN(10,20));
}
```

Output:

Min b/w 10 & 20 is: 10

C #define

- It is used to define constant or micro substitution.
- It can use any basic data type

Syntax:

#define token value

Date:
Page:

#include

- It is used to paste code of given file into current file.
- Includes system-defined and user-defined header files.

#undef

- To undefine a macro means to cancel its definition.

Syntax:

```
#undef token
```

#ifdef

- Checks if macro is defined by #define

Syntax:

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

#ifndef

- Checks if macro is not defined by #define.

Syntax:

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

#if

- Evaluates the expression or condition

Syntax:

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

#else

- Evaluates the expression or condition, if condition of #if is false.

Syntax:

```
#if exp: //if code #else //elsecode #endif
```

error
- Indicates error

pragma
- Used to provide additional information to the compiler.

Syntax:
#pragma token

* Advantages of Macros *

- Time efficiency
- Not need to pass arguments like functions
- It's preprocessed
- Easier to Read.

* Disadvantages of Macros *

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

* Pointers *

- The pointer in C language is a variable which stores the address of another variable. i.e, direct address of the memory location.
- This variable can be of type int, char, array, function, or any other pointer.
- This is done by using [unary operator *] that returns the value of the variable located at the address specified by its operand.

Syntax:

```
Datatype *pointervariable;
```

eg:

```
1) int *ip        /* pointer to an integer */
2) double *dp     /* pointer to a double */
3) float *fp      /* pointer to a float */
4) char *ch       /* pointer to a character */
```

- Reference operator (&)

  & is called reference operator. It gives the address of a variable.

- Dereference operator (*)

  * is called dereference operator. It gives the value from the address.

Date:
Page:

How to use pointers:

- There are a few imp. operators, which we will do with the help of pointers very frequently.

(a) We define a pointer variable
(b) Assign the address of a variable to a pointer
(c) finally access the value at the address ~~variable~~ available in the pointer variable.

```
#include <stdio.h>
int main(){
    int var = 20;
    int *ip;
    ip = &var;
    printf("Address of var variable : %x\n", &var);

    printf("Address stored in ip variables: %x\n", ip);

    printf("Value of *ip variable : %d\n", *ip);
    return 0;
}
```

Output:
Address of var variable : bffd8b3c
Address of stored in ip variable : bffd8b3c
Value of *ip variable : 20

* Null pointers *

- NULL pointer is a pointer which is pointing to nothing.
- Pointer which is initialized with NULL value is considered as NULL pointer.

```
#include <stdio.h>
int main(){
    int *ptr = NULL;
    printf("The value of ptr is: %x\n", ptr);
    return 0;
}
```

Output:
The value of ptr is : 0

Date:
Page:

* Pointer Arithmetic *

- C allows you to perform some arithmetic operations on pointers, like addition, subtraction, etc.

- The result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.

- In pointer-from-pointer subtraction, the result will be an integer value.

- Following arithmetic operations are possible on pointer in C language:
  - Increment
  - Decrement
  - Addition
  - Subtraction
  - Comparison

- Incrementing pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location.

Rule to increment a pointer:

new_address = current_address + i * size_off (datatype).

- increases the no. of bytes of its ~~and~~ datatype

```
#include<stdio.h>
void main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    int i;
    printf("printing array elements ---\n");
    for(i=0; i<5; i++)
    {
        printf("%d", *(p+i));
    }
}
```

- <u>Decrementing pointer in C</u>

- If we decrement a pointer by 1, the pointer will start pointing to the previous location.
- Decreases the no. of bytes of it's datatype.

Rule to decrement a pointer:
new address = current address - i * size_of(data_type);

```
#include <stdio.h>
void main(){
int n=50;
int *p; //pointer to int
p=&n; //stores the address of n variable
printf("Address of p variable is %u \n", p);
p=p-1;
printf("After decrement : Address of p variable is %u \n", p);
} //p will now point to the immediate previous loca
```

- C pointer Addition

- we can add a value to the pointer variable.

formulae:

new_address = current_address + (number * size_of (datatype))

- C pointer Subtraction

- we can subtract a value from the pointer variable.
- Subtracting any no. from a pointer will give an address.

formulae:

new address = current_address - (number * size_of (datatype))

```
#include<stdio.h>
int main(){
int n=50;
int *p; int *pp; // pointer to int
p=&n; ~~int~~ pp=&n; // stores the address of n variable
printf("Address of p variable is %u\n",p);
p=p+3;   // adding 3 to pointer variable
pp=p-3;  // subtracting 3 from pointer variable
printf("After Adding 3: %u\n",p);
printf("After subtracting 3: %u\n",pp);
return 0;
}
```

Date:
Page:

* Limitations of pointer arithmetic *

- Addition of 2 pointers is not allowed.
- Subtraction of 2 pointers is applicable
- Addition of a pointer and an integer is commutative
- Subtraction of a pointer and an integer is not commutative.
- Only integers can be added to a pointer.

* Relation operations *

- A pointer can be compared with a pointer of same type or zero.
- Various relation operators are ==, !=, <, <=, >, >=

Ex.
```
float a = 1.0, b = 2.0, *fptr1, *fptr2;
fptr1 = &a;
fptr2 = &b;
int result;
result = fptr1 = fptr2;
```

* Function Pointers *

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

Syntax: [declaration of a function pointer
Calling a function through a function pointer]

return type (* ptr_name)(type 1, type 2 .....);

```
# include<stdio.h>
// a normal func. with an int parameter
// and void return type
void fun(int a)
{
    printf("value of a is %d\n", a);
}
int main()
{
    // fun_ptr is a pointer to function fun()
    void (* fun_ptr)(int) = &fun;

    //invoking fun() using fun_ptr
    (* fun_ptr)(10);
    return 0;
}
```

Date:
Page:

- Passing a function's address as an argument to other function.

```
#include <stdio.h>
void func1(void (*ptr)()) //contains func. pointer as argument
void func 2()
int main()
{
    func1(func2);
    return 0;
}
void func1( void (*ptr)())
{
    printf("Function 1 is called");
    (*ptr)();
}
void func 20
{
    printf("\n Function2 is called");
}
```

Date:
Page:

* Array of Function pointers *

- Function pointers are used in those applications where we do not know in advance which function will be called.

- In array of function pointer, array takes the addresses of different functions, and the appropriate function will be called based on the index no.

Eg:

```
#include <stdio.h>
void add (int a, int b)
{
    printf (" add = %d\n", a+b);
}
void sub(int a, int b)
{
        printf(" sub = %d\n", a-b);
}
void mul(int a, int b)
{
    printf (" mul = %d\n", a*b);
}
void main ()
{
    void (*fun [] ) (int, int) = { add, sub, mul };
    int ch;
    int a,b;
```

```
    printf("Enter a and b\n");
    scanf("%d%d", &a, &b);
    printf("Enter the operation\n");
    scanf("%d", &ch);
    if (ch>2)
        printf("Wrong Input\n");
    else
        (*fun[ch])(a,b);
}
```

Output:

```
Enter a and b
4 5
Enter the operation
2
mul = 20
```

- Unlike normal pointers, a function pointer points to code, not data.

- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

Date:
Page:

* facts about function pointers *

- A function's name can also be used to get function's address.

```
#include <stdio.h>
// a normal function with an int parameter
// and void return type.
void fun(int a)
{
    printf("value of a is %d\n", a);
}
int main()
{
    void (*fun-ptr)(int) = fun;   // & removed
    fun-ptr(10);   // * removed
    return 0;
}
```

Date:
Page:

* Constant Pointers *

- A constant pointer in c cannot change the address of the variable to which it is pointing, i.e., the address will remain constant.

Therefore, we can say if a ~~poin~~ constant pointer is pointing to some variable, then it cannot point to any other variable.

Syntax:

<type of pointers> * const <name of pointer>;

```
#include<stdio.h>
int main()
{
    int a=1, b=2;
    int * const ptr;
    ptr = &a;
    ptr = &b;
    printf("Value of ptr is: %d", *ptr);
    return 0;
}
```

Output:

error.

Date:
Page:

- Pointer to Constant

- A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed.

- The value of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

Syntax:

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

```
#include<stdio.h>
int main()
{
    int a =100;
    int b=200;
    const int* ptr;
    ptr=&b;
    *ptr=300;
    printf("value : %d", *ptr);
    return0;
}
```

the code shows error "assignment of read-only location '*ptr'".

- Means we cannot change the value of the variable to which the pointer is pointing.

Output:

error.

Date:
Page:

- <u>constant pointer to a constant</u>

- A constant pointer to a constant is a pointer, which is a combination of the above two pointers.

- It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

```
#
int main()
{
    int a=10, b=90;
    const int * const ptr = &a;
    *ptr =12;
    ptr = &b;
    printf("Value : %d", *ptr);
    return 0;
}
```

code shows error.

Date:
Page:

* sizeof () operator in C *

- The sizeof() operator is commonly used in C.
- It determines the size of the exp. or the data type specified in the number of char-sized storage units.

```
#incl
int main()
{
    int a;
    float b;
    double c;
    char d;
    short e;
    printf("size of int is %d \n", sizeof(a)); //4
    printf(" size of float is %d \n", sizeof(b)); //4
    printf(" size of double is %d \n", sizeof(c)); //8
    printf(" size of char is %d \n", size of(d)); //1
    printf(" size of short is %d \n", sizeof(e)); //2
    return 0;
}
```

* Dynamic array allocation *

- The storage size of the data type is constant, it varies when implemented in different platforms.

- Eg: we dynamically allocate the array space by using sizeof operator:

```
int *ptr = malloc(10 * sizeof(int));
```

* C dereference pointer *

- It is known as an indirection operator, which is represented by (*).

- When indirection operator (*) is used with pointer variable, then it is known as dereferencing a pointer.

- When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Date:
Page:

* void pointers in C *

- A pointer to void means a generic pointer that can point to any data type.

- We can assign the address of any datatype to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

Syntax:

```
void *pointer name

int a=10;
char b= 'x';
void *p=&a //void pointer holds address of int 'a'
p=&b //void pointer holds address of char 'b'
```

→ Advantages

- malloc() and calloc() return void *type and this allows these functions to be used to allocate memory of any data type (just becoz of void*)