Unit 5: functions

Compiled by: Er. Krishna Khadka

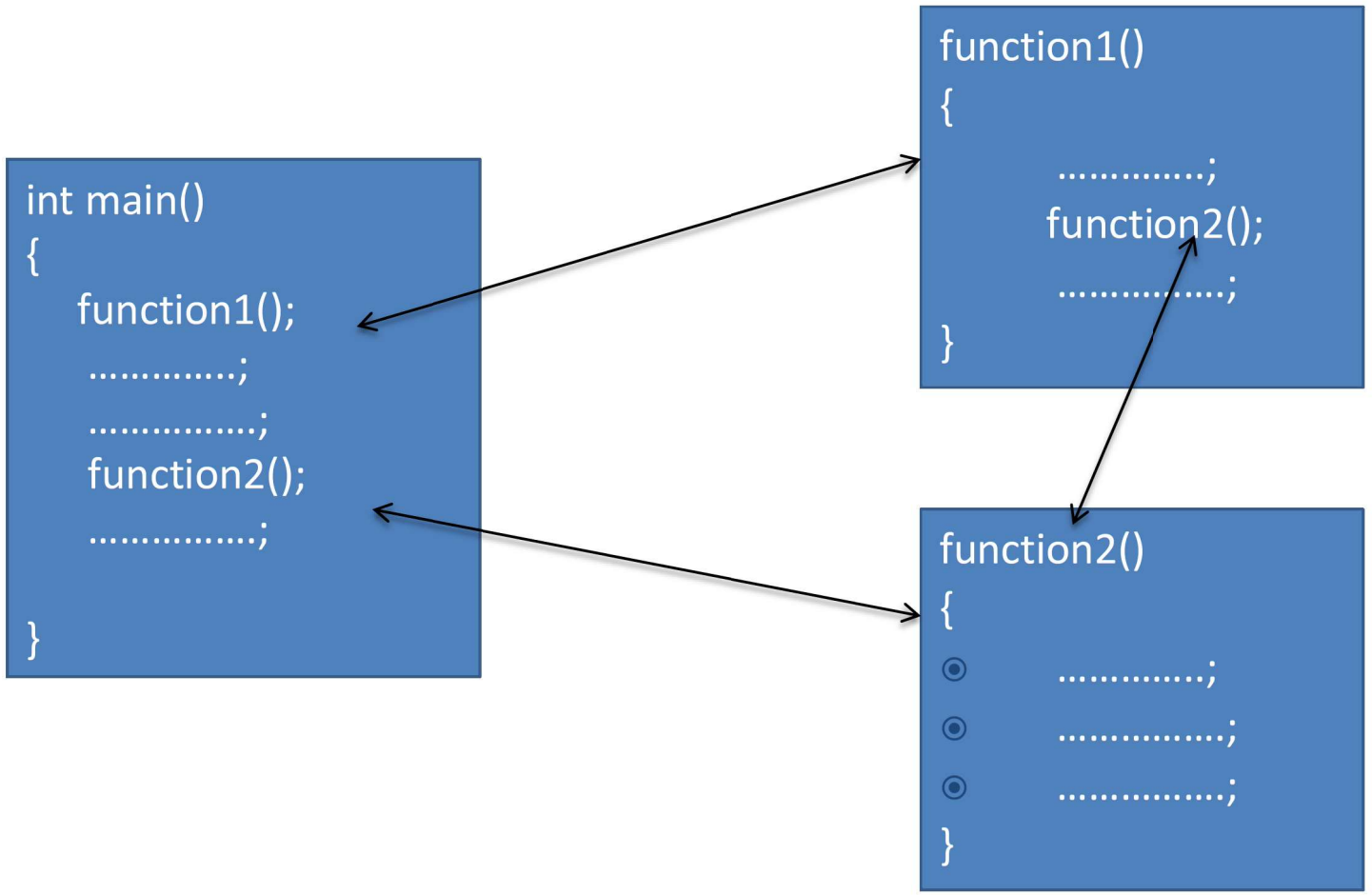
introduction

- A function is a self-contained program segment that carries out some specific, well- defined task.
- Every C program consists of one or more functions. One of these function must be main function.
- Execution of every C program always begins with main() function. Additional functions will be subordinate to main, and perhaps to one another.
- If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another.

Introduction contd.

- A function will carry out its intended action whenever it is accessed (called) from some other portion of the program.
- The same function can be accessed from several different places within a program.
- When the function has completed the assigned task, control will be returned to the point from which the function was accessed.
- A function will process information that is passed to it from the calling portion of the program and return a single value.

Introduction contd.



Advantages of function

- Function increases code reusability.
- Length of the source program can be reduced by using functions at appropriate places.
- Program development will be faster.
- Program debugging will be easier.
- Large number of programmer can be involved.
- Program can be developed in short period of time.
- Program can be developed in different places.
- Program can be tested and compiled independently by different member of a programming team.

Library function vs. user defined function

Library Function	User Defined Function
Library functions are pre defined functions.	User defined functions are functions which are created by the programmer according to the need.
Library function requires header file to use it.	User defined function requires function prototype to use it.
Program development time is faster.	Program development time is slower than library function.
The name of library function can't be changed.	The name of user defined function can be changed any time.
The program using library function will be usually short	The program using user defined function will be usually lengthy.
Example: printf(), scanf(), strupr(), strlen() etc.	**Example:** day (), product(), sum(), diff() etc.

Components of a function

- Function Prototype
- Function Call
- Function Definition

Function prototype

- Function prototype provides the following information to the computer.
  - The type of the value returned
  - The name of the function
  - The number and the type of the arguments that must be supplied in a function call
- **Syntax**

  data_type function_name (arg1, arg2 …..argn)
- **Example**

  int sum (int, int);
- In this example two arguments of 'int' type are passed inside function named 'sum' and it returns value of 'int' type
- Prototype only needed if function definition comes after use in program. They are generally written at the beginning of a program, ahead of the main () function.

Function call

- A function call is specified by the function name followed by the arguments enclosed in parenthesis and terminated by a semicolon.
- **Syntax**

  variable = function_name (arg1, arg2,.. argn);

  or function_name (arg1, arg2, ……., argn);

  where passing arguments arg1, arg2, arg3 ……. are optional.
- **Example**

  a = area (l, b);

OR

  area(l, b);

Function call contd.

```
main ()              // calling function
{
        --------------
        function ();       // function call
        --------------
}
function ()        //called function (body of the function)
{
   -----------------
   statements
   -----------------
}
```

In the above example the function () is called by the main () function and the code of the function (body of the function) is written after the main() function.

Function definition

- A function definition has two components:
  a. Function header
  b. Body of the function
- The first line of function definition is known as **'function header'** and is followed by the **'function body'**.
- Function header contains the value returned by the function followed by the function name and optionally a set of arguments, separated by commas and enclosed in parentheses.
- Each argument is preceded by its associated type declaration.
- An empty pair of parentheses must follow the function name if the function definition does not include any arguments.

Function definition contd.

- **Syntax**

  data_type function_name (data_type arg1, data_type arg2, ……..data_type argn)

  where data_type represents the data type of the item that is returned by the function, function_name represents the function name and arg1, arg2, arg3……. argn are variables (or arguments).

- **Example**

```
int sum( int a, int b) //Function header
{
        int c;
        c = a + b;
        retrun (c);
}
```

//Function body

example

```
#include<stdio.h>
int value(int);                    // function prototype
void main()                                  // calling function
{
        int a,b;
        printf("\nEnter a number");
        scanf("%d",&a);
        b = value(a);                        // function call
        printf("Square of %d = %d",a,b);
}
int value(int a)                   // function header
{
        int p;
        p = a * a;                 } function body (called function)
        return ( p );
}
```

Return statement

- Function returns a value to the calling function. For this, a return keyword is used.
- So, '*return*' keyword returns the program control from a function to the calling function.
- **Syntax**

  return expression;
- A function definition can contain multiple return statements. However, **only one** gets executed.
- **Example**

```
int great(int a, int b)
{       if ( a > b)
                return a;
        else
                return (b);
}
```

a program to print the greatest number from 2 nos. using function.

```
#include<stdio.h>
int great(int, int);    //prototyping
void main()
{
    int a,b,c;
    printf("\nEnter two numbers:");
    scanf("%d%d",&a,&b);
    c = great(a,b);                  // actual arguments
    printf("\nGreatest number= %d",c);
}
//function definition
int great(int a,int b)               // formal arguments
{   if(a>b)
            return(a);
    else
            return(b);
}
```

- When a function does not have to return a value then we say that the function's return type is void.
- 'return' keyword is used at the end of the function to transfer the control.
- **Example**

```
void evenodd( int n)
{       if (n % 2 = = 0)
                printf("%d is even number", n);
        else
                printf("%d is odd number", n);
}
```

Categories of user defined functions

- **Function returning value and passing arguments**
- **Function returning value and passing no arguments**
- **Function returning no value and passing arguments**
- **Function returning no value and passing no arguments**

Function returning value and passing arguments

- Arguments are passed from the calling function to the called function and there will also be return statement in the called function.
- Example:

```
#include<stdio.h>
int sum(int, int); //function prototype
void main()
{
        int a,f;
        printf("\nEnter two numbers");
        scanf("%d%d",&a,&b);
        f=sum(a,b);                    //function call
        printf("\nSum is %d",f);
}
int sum(int x,int y)
{
        int c;                         } function definition
        c = x+y;
        return (c);
}
```

Function returning value and passing no arguments

- Arguments are not passed from the calling function to the called function but there is return statement in the called function.
- Example:

```
#include<stdio.h>
int sum();                  //function prototype
void main()
{
        int f;
        f=sum();            // function call
        printf("\nSum is %d",f);
}
int sum()
{       int a,b,c;
        printf("\nEnter two numbers");
        scanf("%d%d",&a,&b);
        c = a+b;
        return (c);
}
```

function definition

Function returning no value and passing arguments

- Arguments are passed from the calling function to the called function but there is no return statement in the called function.
- Example:

```
#include<stdio.h>
void sum(int,int);              //function prototype
void main()
{       int a,b,f;
        printf("\nEnter two numbers:");
        scanf("%d%d",&a,&b);
        sum(a,b);                       // function call
}
void sum(int a,int b)
{       int c;                                          //function definition
        c = a+b;
        printf("\nSum is %d",c);
}
```

Function returning no value and passing no arguments

- Arguments are not passed from the calling function to the called function and there is no return statement in the called function.
- Example:

```
#include<stdio.h>
void sum();                          //function prototype
void main()
{
        sum();                       //function call
}
void sum()
{
        int a,b,c;
        printf("\nEnter two numbers");
        scanf("%d%d",&a,&b);         //function definition
        c = a+b;
        printf("\nSum is %d",c);
}
```

RECURSION

- Recursion is a process by which a function calls itself repeatedly, until some specific condition has been satisfied.
- For the problem to be solved recursively two conditions must be satisfied:
  - The function should call itself.
  - The problem statement must include a stopping condition.
- When a recursive program is executed, the recursive function calls are not executed immediately. Rather, they are placed on a 'stack' until the condition that terminates the recursion is encountered.

Recursion contd.

- The function calls are then executed in reverse order, as they are 'popped' off the stack.
- Example: A program to calculate factorial of a no. using recursion

```
#include<stdio.h>
int fact(int);
int main()
{       int a,f,n;
        printf("\nEnter a number");
        scanf("%d",&n);
        f=fact(n);
        printf("\nfactorial of %d is %d",n,f);
}
int fact(int n)
{       if(n<=1)
                return( 1);
        else
                return(n*fact(n-1));
}
```

A stack is a Last-In, First-Out data structure in which successive data items are 'pushed down' upon preceding data items.

The data are later removed (i.e. they in reverse order, as are 'popped') from the stack indicated by the LIFO designation.

Recursion contd.

- While calculating factorial of a number recursively, the function calls will proceed in the following order.

  n! = n*(n-1)!

  (n-1)!= (n-1)*(n-2)!

  (n-2)!=(n-2)*(n-3)!

  ……..

  ……..

  2! = 2 * 1!

- The actual values will then be returned in the following reverse order.

  1! = 1

  2! = 2 * 1! = 2 * 1 = 2

  3! = 3 * 2! = 3 * 2 = 6

  4! = 4 * 3! = 4 * 6 = 24

  n! = n * (n-1)! = ----

Recursion contd.

- A program to display the Fibonacci series up to n^{th} term using recursive function

```
#include<stdio.h>
int fibo(int n);
void main()
{       int i,n;
        printf("\nEnter a number");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
                printf("%d",fibo(i));
}
int fibo(int n)
{       if(n==0)
                        return 0;
        else if(n==1)
                        return 1;
        else
                return(fibo(n-1)+fibo(n-2));
}
```

Global & local variable

- A local variable is one that is declared inside a function and can only be used by that function. If you declare a variable outside all functions then it is a global variable and can be used by all functions in a program.
- #include<stdio.h>

```
int a;
int b;                    // Global variables
int add()
{
        return a + b;
}
int main()
{       int ans;          // Local variables
        a = 5; b = 7;
        ans = add();
        printf("%d\n",ans);
        return 0;
}
```

Functions with Arrays

- We can pass an entire array of values into a function just like passing individual variables.

- In this task it is essential to list the name of the array along with functions arguments without any subscripts and the size of the array as arguments.

Functions with Arrays contd.

- A program to input 5 numbers and calculate their sum using array and function.

```
#include<stdio.h>
int sum(int a[ ]);
void main()
{       int a[5],f,i;
        for (i=1;i<=5;i++)
        {
                printf("\nEnter number:");
                scanf("%d",&a[i]);
        }
        f =sum(a);
        printf("\nSum is %d",f);
}
int sum(int a[ ])
{       int c=0,i;
        for (i=1;i<=5;i++)
        {
                c = c+a[i];
        }
        return (c);
}
```

Functions with Arrays contd.

- A program to find the length of a string using user defined function.

```
#include<stdio.h>
int strlength(char str[]);
void main()
{       char str[100];
        printf("\nEnter a string :");
        gets(str);
        printf("\n The length of string is %d",strlength(str));
}
int strlength(char str[])
{       int i;
        for(i=0;str[i]!='\0';i++); //here the loop simply counts
        return(i);
}
```

Storage Classes in C

- There are four storage classes in C:

(a) Automatic storage class

(b) Register storage class

(c) Static storage class

(d) External storage class

A variable's storage class tell us;

- Where the variable would be stored.
- What will be the value of the variable, if the initial value is not specifically assigned. (i.e. the default initial value).
- What is the scope of the variable; i.e. in which functions the value of the variable would be available.
- What is the life of variable; i.e. how long would the variable exist.

Automatic storage class

- The features of a variable defined to have an automatic storage class are as under:

Storage	Memory.
Default initial value	An unpredictable value, which is often called a garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined.

- Following program shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

auto.c

```
1  #include<stdio.h>
2  void main()
3  {
4  auto int i,j;
5  printf("\n%d %d",i,j);
6  }
7
```

```
H:\My Drive\Cprogram slid

0 16
------------------------
Process exited after 11.
Press any key to continu
```

auto.c auto1.c

```c
#include<stdio.h>
void main()
{
auto int i=1;
{
auto int i=2;
{
auto int i=3;
printf("%d",i);
}
printf("%d",i);
}
printf("%d",i);
}
```

```
H:\My Drive\Cprogram slides program\s
321
--------------------------------
Process exited after 0.0483 secon
Press any key to continue . . .
```

Note that the compiler treats the three i's as totally different variables, since they are defined in different blocks.

Register Storage Class

- The features of a variable defined to have a register storage class are as under:

Storage	CPU registers..
Default initial value	Garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined.

A value stored in a CPU register can always be accessed faster than the one which is stored in memory. Therefore**, if** a variable is **used at many places** in a program it is better to **declare** its storage class as **register**.

auto.c | auto1.c | register.c | all.c | register1.c | [*] register2.c

```c
#include <stdio.h>
int main()
{
register int a;
// variable a is allocated memory
printf("%d",a);
}
```

H:\My Drive\Cprogram slides program\storageclasses\register2.exe

```
1
--------------------------------
Process exited after 0.05939 seconds with return value 0
Press any key to continue . . .
```

```c
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a);
// This will give a compile time error
//since we can not access the address of a register variable.
}
```

Static Storage Class

- The features of a variable defined to have a static storage class are as under:

Storage	Memory
Default initial value	Zero
Scope	Local to the block in which the variable is defined.
Life	Value of the variable persists between different function calls.

The variables defined as static specifier can hold their value between the multiple function calls.

```c
#include<stdio.h>
int main()
{
    int i = 0;
    printf("\nLoop started:\n");
    for (i = 1; i < 5; i++) {
        // Declaring the static variable 'y'
        static int y = 5;
        // Declare a non-static variable 'p'
        int p = 10;
        // Incrementing the value of y and p by 1
        y++;
        p++;
        // printing value of y at each iteration
        printf("\nThe value of 'y', "
            "declared as static, in %d "
            "iteration is %d\n",
            i, y);

        // printing value of p at each iteration
        printf("The value of non-static variable 'p', "
            "in %d iteration is %d\n",
            i, p);
    }
}
```

```
H:\My Drive\Cprogram slides program\storageclasses\static1.exe

Loop started:

The value of 'y', declared as static, in 1 iteration is 6
The value of non-static variable 'p', in 1 iteration is 11

The value of 'y', declared as static, in 2 iteration is 7
The value of non-static variable 'p', in 2 iteration is 11

The value of 'y', declared as static, in 3 iteration is 8
The value of non-static variable 'p', in 3 iteration is 11

The value of 'y', declared as static, in 4 iteration is 9
The value of non-static variable 'p', in 4 iteration is 11
```

External Storage Class

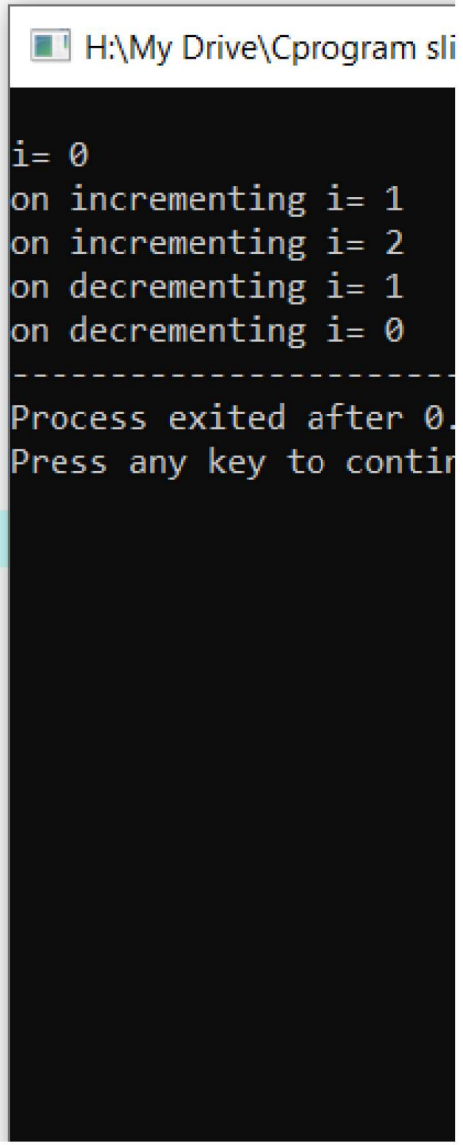
- The features of a variable defined to have an external storage class are as under:

Storage	Memory
Default initial value	Zero
Scope	Global
Life	As long as the program's execution doesn't come to an end.

- External variables differ from those we have already discussed in that their scope is global, not local.
- External variables are declared outside all functions, yet are available to all functions that care to use them.
- (Note:- the function that uses an external variable declare that variable external via the keyword **extern**)

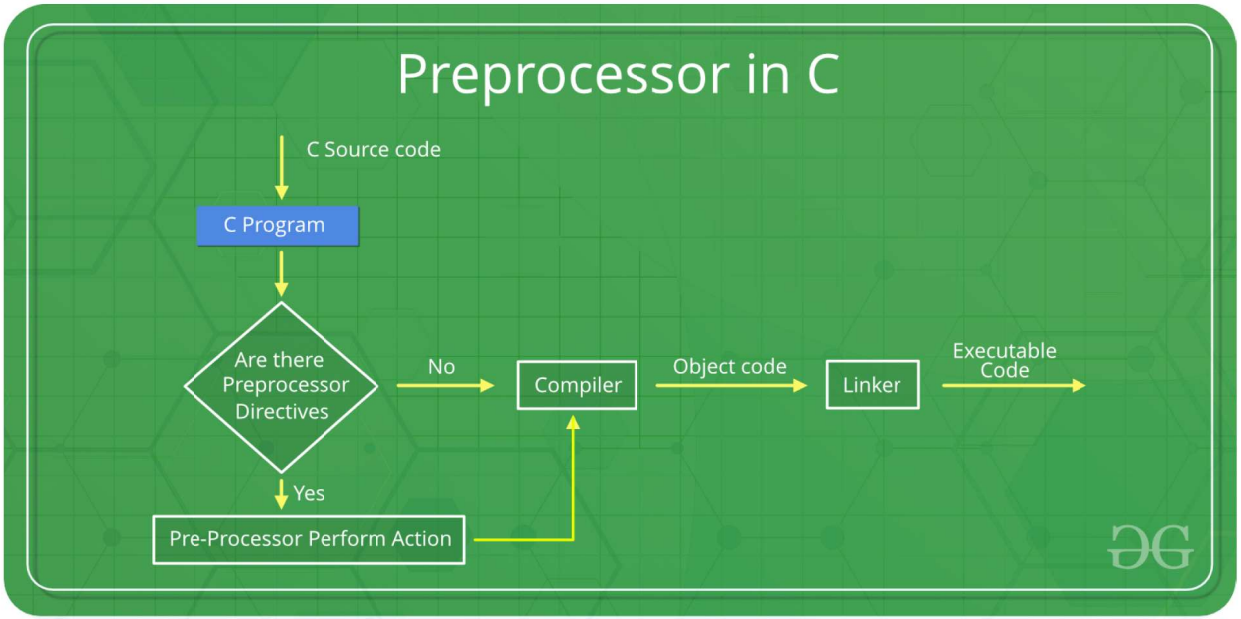
auto.c | auto1.c | register.c | all.c | register1.c | [*] register2.

```c
#include<stdio.h>
void increment();
void decrement();
int i;
void main()
{
printf("\ni= %d",i);
increment();
increment();
decrement();
decrement();
}
void increment()
{
i=i+1;
printf("\non incrementing i= %d",i);
}
void decrement()
{
i=i-1;
printf("\non decrementing i= %d",i);
}
```



Preprocessor Directives in C Programming

- As the name suggests Preprocessors are programs that process our source code before compilation.
- There are a number of steps involved between writing a program and executing a program in C



Program compilation and execution sequence

- The source code written by programmers is stored in the **file program.c**.
- This file is then processed by preprocessors and an expanded source code file is generated named program.
- This expanded file is compiled by the compiler and an object code file is generated named program .obj.
- Finally, the linker links this object code file to the object code of the library functions to generate the executable file program.exe.

Preprocessor Directives in C Programming

- Preprocessor programs provide preprocessors directives that tell the compiler to preprocess the source code before compiling.
- All of these preprocessor directives begin with a '#' (hash) symbol.
- The '#' symbol indicates that, whatever statement starts with #, is going to the preprocessor program, and the preprocessor program will execute this statement.
- Examples of some preprocessor directives are: *#include*, *#define*, etc.

There are 4 Main Types of Preprocessor Directives:

- Macros
- File Inclusion
- Conditional Compilation
- Other directives

1. Macros

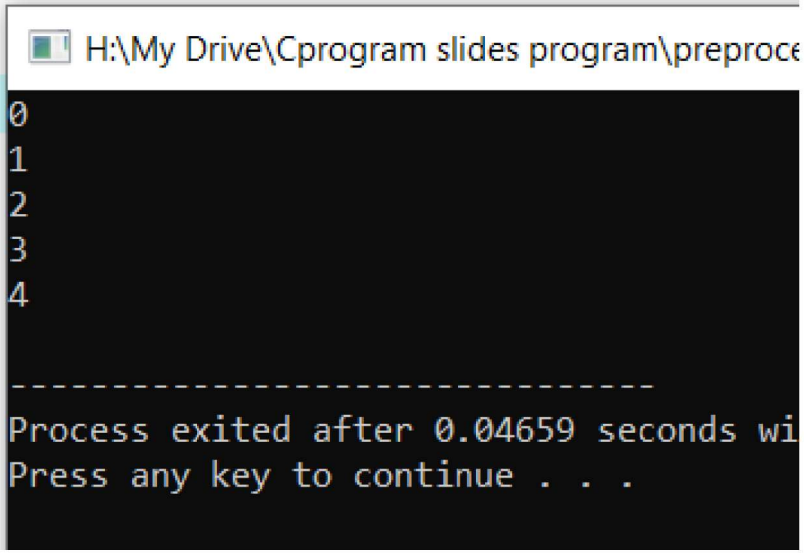
- Macros are a piece of code in a program that is given some name.
- Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code.
- The '#define' directive is used to define a macro.

macro.c

```c
#include <stdio.h>

// macro definition
#define LIMIT 5
int main()
{
    for (int i = 0; i < LIMIT; i++) {
        printf("%d \n",i);
    }

    return 0;
}
```



- **Macros With Arguments**: We can also pass arguments to macros. Macros defined with arguments work similarly to functions.

macro.c | macro1.c

```c
#include <stdio.h>

// macro with parameter
#define AREA(l, b) (l * b)
int main()
{
    int l1 = 10, l2 = 5, area;

    area = AREA(l1, l2);

    printf("Area of rectangle is: %d", area);

    return 0;
}
```

```
H:\My Drive\Cprogram slides program\preprocessor\
Area of rectangle is: 50
--------------------------------
Process exited after 0.05476 seconds with r
Press any key to continue . . .
```

2. File Inclusion

- This type of preprocessor directive tells the compiler to include a file in the source code program.
- There are two types of files that can be included by the user in the program:
- **Header File or Standard files:**
  - These files contain definitions of pre-defined functions like printf(), scanf(), etc.
  - These files must be included for working with these functions. Different functions are declared in different header files.
  - For example, standard I/O functions are in 'stdio' file whereas functions that perform string operations are in the 'string' file.
- **Syntax**:
  - #include< file_name >

2. File Inclusion

- **user-defined files:** When a program becomes very large, it is good practice to divide it into smaller files and include them whenever needed. These types of files are user-defined files. These files can be included as:
- #include"filename"

3. Conditional Compilation

- Conditional Compilation directives are a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions.
- This can be done with the help of two preprocessing commands 'ifdef' and 'endif'.
- Syntax:
- #ifdef macro_name

  statement1;

  statement2;

  statement3;

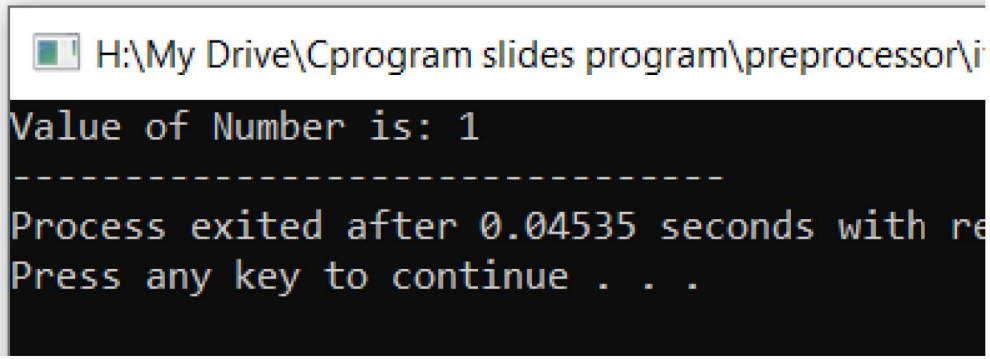
  .

  .

  statementN;
- #endif

ifdef.c

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



4. Other Directives

- Apart from the above directives, there are more directives that are not commonly used:
- **#undef Directive:**
  - The #undef directive is used to undefine an existing macro. This directive works as:
    - #undef LIMIT
- Using this statement will undefine the existing macro LIMIT. After this statement, every "#ifdef LIMIT" statement will evaluate as false.

undef.c

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

```
public int __cdecl printf (const char
```

undef.c

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

```
H:\My Drive\Cprogram slides program\
3.141500
--------------------------------
Process exited after 0.04764 seco
Press any key to continue . . .
```

Finished

Unit 5