Practical Workbook
CS-103
Programming Languages
(FD)

Name	: ______________________________
Year	: ______________________________
Batch	: ______________________________
Roll No	: ______________________________
Department:	______________________________
Teacher	: ______________________________

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

INTRODUCTION

The Practical Workbook for "Programming Languages" introduces the basic as well as advance concepts of programming using C language. C has been selected for this purpose because it encompasses the characteristics of both the high level languages; which give a better programming efficiency and faster program development; and the low level languages; which have a better machine efficiency.

Each lab session begins with a brief theory of the topic. Many details have not been incorporated as the same is to be covered in Theory classes. The Exercise section follows this section.

The Workbook has been arranged as thirty labs starting with a practical on the Introduction to programming environment and fundamentals of programming language.

Next lab session deals with single stepping; an efficient debugging and error detection technique.

Next three lab sessions cover the basic building blocks of programming. These practicals introduce the concepts of decision making; loops; function declaration and definition etc.

The next three experiments deal with the advance concepts like arrays, pointers, structures and unions. These features enable the users to handle not only large amount of data, but also data of different types (integers, characters etc.) and to do so efficiently.

Separate practicals have been included for different graphics and text modes, passing variable number of arguments to functions, and command line arguments.

Further two lab sessions covers the filing feature, which allows the user to store/retrieve the data on/from permanent storage like floppy or hard disk and various file and directory manipulation functions.

Finally, there are labs on hardware interfacing using ROM BIOS routines, which explain accessing the system color palettes, interfacing mouse etc. in programs using C.

Few changes have been made in the examples and exercises of this workbook since its last edition which was printed in 2011. Now these are more comprehensive than those of the previous issue. One lab session of the previous workbook is replaced by an appendix that discusses various date and time functions.

Practical Workbook
Programming Languages

CONTENTS

Lab Session 01

OBJECTIVE

Familiarization with Programming Environment using Turbo C

THEORY

The Development Environment - Integrated Development Environment (IDE):
The C compiler has its own built-in text editor. You may also use a commercial text editor or word processor that can produce text files. The important thing is that whatever you write your program in, it must save simple, plain-text files, with no word processing commands embedded in the text. The files you create with your editor are called source files, and for C++ they typically are named with the extension *.CPP*, *.CP*, or *.C*.

The C Developing Environment, also called as Programmers' Platform, is a screen display with windows and pull-down menus. Code of the program, error messages and other information are displayed in separate windows. The menus may be used to invoke the operations necessary to develop the program, debug and execute the program.

Invoking the IDE
To invoke the IDE from the windows you need to double click the TC icon.

To do so from the command prompt go in the specific directory and type 'tc'. This makes you enter the IDE interface, which initially displays only a menu bar at the top of the screen and a status line below will appear. The menu bar displays the menu names and the status line tells what various function keys will do.

Using Menus
If the menu bar is inactive, it may be invoked by pressing the [F10] function key. To select different menu, move the highlight left or right with cursor (arrow) keys. You can also revoke the selection by pressing the key combination for the specific menu.

Opening New Window
To type a program, you need to open an Edit Window. For this, open file menu and click "new". A window will appear on the screen where the program may be typed.

Writing a Program
When the Edit window is active, the program may be typed. Use the certain key combinations to perform specific edit functions.

Saving a Program
To save the program, select **save** command from the **file** menu. This function can also be performed by pressing the [F2] button. A dialog box will appear asking for the path and name of the file. Provide an appropriate and unique file name. You can save the program after compiling too but saving it before compilation is more appropriate.

Making an Executable File

The source file is required to be turned into an executable file. This is called "Making" of the *.exe* file. The steps required to create an executable file are:

1. Create a source file, with a *.cpp* extension.
2. Compile the source code into a file with the *.obj* extension.
3. Link your *.obj* file with any needed libraries to produce an executable program.

Compiling the Source Code
Although the source code in your file is somewhat cryptic, and anyone who doesn't know C will struggle to understand what it is for, it is still in what we call human-readable form. But, for the computer to understand this source code, it must be converted into machine-readable form. This is done by using a compiler. Hence, compiling is the process in which source code is translated into machine understandable language.

Creating an Executable File with the Linker
After your source code is compiled, an object file is produced. This file is often named with the extension *.OBJ*. This is still not an executable program, however. To turn this into an executable program, you must run your linker. C programs are typically created by linking together one or more OBJ files with one or more libraries. A library is a collection of linkable files that were supplied with your compiler.

Project/Make
Before compiling and linking a file, a part of the IDE called Project/Make checks the time and date on the file you are going to compile.

Compiling and linking in the IDE
In the Turbo C IDE, compiling and linking can be performed together in one step. There are two ways to do this: you can select Make *EXE* from the compile menu, or you can press the [F9] key.

Executing a Program
If the program is compiled and linked without errors, the program is executed by selecting Run from the Run Menu or by pressing the [Ctrl+F9] key combination.

The Development Cycle
If every program worked the first time you tried it, that would be the complete development cycle: Write the program, compile the source code, link the program, and run it. Unfortunately, almost every program, no matter how trivial, can and will have errors, or bugs, in the program. Some bugs will cause the compile to fail, some will cause the link to fail, and some will only show up when you run the program.Whatever type of bug you find, you must fix it, and that involves editing your source code, recompiling and relinking, and then rerunning the program.

Correcting Errors
If the compiler recognizes some error, it will let you know through the Compiler window. You'll see that the number of errors is not listed as 0, and the word "Error" appears instead of the word "Success" at the bottom of the window. The errors are to be removed by returning to the edit window. Usually these errors are a result of a typing mistake. The compiler will not only tell you what you did wrong; they'll point you to the exact place in your code where you made the mistake.

Exiting IDE

An Edit window may be closed in a number of different ways. You can click on the small square in the upper left corner, you can select **close** from the **window** menu, or you can press the [Alt][F3] combination. To exit from the IDE select **Exit** from the **File** menu or press [Alt][X] combination.

EXERCISES

1. Load and execute the following program using Turbo C environment. Explore different features available for execution.

```
#include<stdio.h>
int main()
{
Print (“Hello World \n”)
Return 0
}
```

Lab Session 02

OBJECTIVE

Fundamentals of Programming Language

Building Blocks of Programming Language:
In any language there are certain building blocks:

- Constants
- Variables
- Operators
- Methods to get input from user (scanf( ), getch( ) etc.)
- Methods to display output (Format Specifier, Escape Sequences etc.) and so on.

Variables and Constants
If the value of an item can be changed in the program then it is a variable. If it will not change then that item is a constant. The various variable types (also called *data type*) in C are: *int*, *float*, *char*, *long* etc. For constants, the keyword ***const*** is added before declaration.

Operators
There are various types of operators that may be placed in three categories:

Basic:	+ - * / %
Assignment:	= += -= *= /= %=
	(++, -- may also be considered as assignment operators)
Relational:	< > <= >= == !=

Format Specifiers
Format Specifiers tell the *printf* statement where to put the text and how to display the text. The various format specifiers are:

%d	=>	integer
%c	=>	character
%f	=>	floating point etc.

Field Width Specifiers
They are used with **%** to limit precision in floating point number. The number showing limit follows the radix point.

Escape Sequences
Escape Sequence causes the program to **escape** from the normal interpretation of a string, so that the next character is recognized as having a special meaning. The back slash "\" character is called the **Escape Character"**. The escape sequence includes the following:

\n	=>	new line
\b	=>	back space
\r	=>	carriage return
\"	=>	double quotations
\\	=>	back slash etc.

Getting Input From the User
The input from the user can be taken by the following techniques: scanf( ), getch( ), getche( ), getchar( ) etc.

Examples

1. Implementing a Simple C Program

```
                #include<conio.h>
                #include<stdio.h>
                void main(void)
        {
                clrscr();
         printf("\n Hello World");
                getch();
        }
```

2. Demonstrating the fundamentals of C Language

```
                #include<conio.h>
                #include<stdio.h>
               void main(void)
        {
                clrscr();
         int num1,num2,sum,product;
         printf("\tThe program takes two numbers as input and
                prints their sum and product");
         printf("\n Enter first number:");
         scanf("%d",&num1);
         printf("\n Enter second number:");
         scanf("%d",&num2);
         sum=num1+num2;
         product=num1*num2;
         printf("\n%d+%d=%d",num1,num2,sum);
         printf("\n%d*%d=%d",num1,num2,product);
                getch();
        }
```

EXERCISES

1. Type the following program in C Editor and execute it. Mention the Error (if any).

```
void main(void)
{
   printf("This is my first program in C");
}
```

---

2. Add the following line at the beginning of the above program. Recompile the program. What is the output?

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

3. Make the following changes to the program. Mention the Errors observed, in your own words:
   i. Write Void instead of void.

   ii. Remove the semi colon ';'.

   iii. Erase any one of brace '{' or '}' .

4. Write a program to calculate the Area (A= πr^2) and circumference of a circle (***C=2πr***), where r = radius is taken as input and π is declared as a constant. The precision of π should be the number of characters in your name. Display the result to 4 decimal places.

5. Write a single C statement to output the following on the screen:

My name is “*Your Name*”
And my roll number is “00*Your_roll_no*”
I am a student of ‘Computer and Information System Department’

Lab Session 03

OBJECTIVE

Debugging and Single-Stepping of Programs

THEORY

One of the most innovative and useful features of Turbo C++ is the integration of debugging facilities into the IDE.

Even if your program compiles perfectly, it still may not work. Such errors that cause the program to give incorrect results are called **Logical Errors**. The first thing that should be done is to review the listing carefully. Often, the mistake will be obvious. But, if it is not, you'll need the assistance of the Turbo C Debugger.

One Step at a Time

The first thing that the debugger can do for you is slow down the operation of the program. One trouble with finding errors is that a typical program executes in a few milliseconds, so all you can see is its final state. By invoking C++'s single-stepping capability, you can execute just one line of the program at a time. This way you can follow where the program is going.

Consider the following program:

```
          void main(void)
      {
        int number, answer=-1;
        number = -50;
        if(number < 100)
              if(number > 0)
                answer = 1;
            else
                answer = 0;
       printf("answer is %d\n", answer);
      }
```

Our intention in this program is that when **number** is between 0 and 100, **answer** will be 1, when the **number** is 100 or greater, **answer** will be 0**,** and when **number** is less than 0, **answer** will retain its initialized value of –1. When we run this program with a test value of -50 for **number,** we find that **answer** is set to 0 at the end of the program, instead of staying –1.

We can understand where the problem is if we single step through the program. To do this, simply press the [F7] key. The first line of the program will be highlighted. This highlighted line is called the **run bar**. Press [F7] again. The run bar will move to the next program line. The run bar appears on the line about to be executed. You can execute each line of the program in turn by pressing [F7]. Eventually you'll reach the first **if** statement:

```
if (num < 100 )
```

This statement is true (since **number** is –50); so, as we would expect the run bar moves to the second **if** statement:

```
if( num > 0)
```

This is false. Because there's no **else** matched with the second **if**, we would expect the run bar to the **printf( )** statement. But it doesn't! It goes to the line

```
answer  = 0;
```

Now that we see where the program actually goes, the source of the bug should become clear. The **else** goes with the last **if**, not the first **if** as the indenting would lead us to believe. So, the **else** is executed when the second **if** statement is false, which leads to erroneous results. We need to put braces around the second **if**, or rewrite the program in some other way.

Resetting the Debugger
Suppose you've single stepped part way through a program, and want to start over at the beginning. How do you place the run bar at the top of the listing? You can reset the debugging process and initialize the run bar by selecting the Program Reset option from the Run menu.

Watches
Single stepping is usually used with other features of the debugger. The most useful of these is the watch (or watch expression). This lets you see how the value of variable changes as the program runs. To add a watch expression, press [Ctrl+F7] and type the expression.

EXERCISES

1. Fill out all the entities in table by their corresponding values by inserting watches and single stepping the program.

i. int num=*your_roll_no*, sqr ;

   sqr= num*num ;

Before Execution		After Execution	
num	sqr	num	sqr

ii. char ch='First letter of your Name';

   ch ++;

Before Execution	After Execution
ch	ch

iii.
```
int x=your_roll_no,y=your_roll_no+50;
float avg;
avg=(x+y)/2;
```

Before Execution			After Execution		
x	y	avg	x	y	avg

Lab Session 04

OBJECTIVE

Installing Breakpoints for Debugging

Breakpoints

It often happens that you've debugged part of your program, but must deal with a bug in another section, and you don't want to single-step through all the statements in the first part to get to the section with the bug. Or you may have a loop with many iterations that would be tedious to step through. The way to do this is with a breakpoint. A breakpoint marks a statement where the program will stop. If you start the program with [Ctrl][F9], it will execute all the statements up to the breakpoint, then stop. You can now examine the state of the variables at that point using the watch window.

Installing breakpoints

To set a breakpoint, first position the cursor on the appropriate line. Then select Toggle Breakpoint from the Debug menu (or press [Ctrl][F8]). The line with the breakpoint will be highlighted. You can install as many breakpoints as you want. This is useful if the program can take several different paths, depending on the result of if statements or other branching constructs.

Removing Breakpoints

You can remove a single breakpoint by positioning the cursor on the line with the breakpoint and selecting Toggle breakpoint from the Debug menu or pressing the [Ctrl][F8] combination (just as you did to install the breakpoint). The breakpoint highlight will vanish.

You can all set **Conditional Breakpoints** that would break at the specified value only.

EXERCISES

1. The programs given below contain some syntax and/or logical error(s). By debugging and installing breakpoints, mention the error(s) along with their categorization into syntactical or logical error. Also, write the correct program statements.

   i. // To check whether the number is divisible by 2 or not:

```
   int num;
printf("enter any number")
scanf("%f",num);
if(num%2=0)
  printf("Number is divisible by 2");
else
  printf("number is not divisible by 2");
```

____

____

____

ii. // To print your batch

```
int x =Your_batch;

if (x=2010)
    printf("Your batch is 2010")
else
    printf("Your batch is %d",x);
```

iii. // To calculate the number of characters entered by the user. Exit on Esc or Enter.

```
int count=0;
char ch;
while(ch=getche( )!=27)
{
if(Ch=='\r');
              break;
count++;
}
printf("Count =%d",count);
```

Lab Session 05

OBJECTIVE

Decision Making in Programming (If, If-else)

THEORY

Normally, your program flows along line by line in the order in which it appears in your source code. But, it is sometimes required to execute a particular portion of code only if certain condition is true; or false i.e. you have to make decision in your program. There are three major decision making structures. Four decision making structures:

1. *If* statement
2. *If-else* statement
3. *Switch* case
4. Conditional Operator (Rarely used)

The if statement
The ***if*** statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.
The simplest form of an ***if*** statement is:

```
if (expression)
    statement;
```

The expression may consist of logical or relational operators like (> >= < <= && ||)

An understanding of if statement is demonstrated with the following example:

```
            void main(void)
{
 int var;
 printf("Enter any number;");
 scanf("%d",&var);
 if(var==10)
 printf("The user entered number is Ten");
}
```

The if-else statement
Often your program will want to take one branch if your condition is true, another if it is false. The keyword ***else*** can be used to perform this functionality:

```
if (expression)
    statement;
else
    statement;
```

Note: To execute multiple statements when a condition is true or false, parentheses are used. Consider the following example that checks whether the input character is an upper case or lower case:

```
            void main(void)
{
 char ch;
 printf("Enter any character");
 ch=getche();
 if(ch>='A'&&ch<='Z')
    printf("%c is an upper case character",ch);
 else
    printf("%c is a lower case character",ch);
            getch();
}
```

Typecasting

Typecasting allow a variable of one type to act like another for a single operation. In C typecasting is performed by placing, in front of the value, the type name in parentheses.

EXERCISES

1. Write a program that takes a number as input from user and checks whether the number is even or odd using if-else.

____

____

____

____

____

____

____

____

____

2. Mention the output for the following program :

```
          #include<stdio.h>
void main()
{
  int a=100;
  if(a>10)
    printf("Shahid Afridi");
  else if(a>20)
```

```
      printf("Shoaib Akhtar");
    else if(a>30)
      printf("Kamran Akmal");
}
```

Lab Session 06

OBJECTIVE

Decision Making in Programming (Switch Case, Ternary Operator)

THEORY

The switch Statement

Unlike ***if***, which evaluates one value, ***switch*** statements allow you to branch on any of a number of different values. The general form of the ***switch*** statement is:

```
switch (expression)
{
case valueOne: statement;
          break;
case valueTwo: statement;
          break;
....
case valueN:   statement;
          break;
default:      statement;
}
```

An Example:

```
        void main(void)
{
       clrscr();
 char grade;
       printf("\n Enter your Grade: ");
 grade=getche();
 switch(grade)
  {
   case 'A':
          case 'a':
      printf("\n Your percentage is 80 or above 80 ");
      break;

   case 'B':
   case 'b':
      printf("\n Your percentage is in 70-80 ");
      break;

   default:
      printf("\n Your percentage is below 70 ");

  }
     getch();
}
```

Conditional (Ternary) Operator

The conditional operator (? :) is C's only ternary operator; that is, it is the only operator to take three terms.

The conditional operator takes three expressions and returns a value:
(expression1) ? (expression2) : (expression3);

This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable.

An Example:

```
            void main(void)
    {
             clrscr();
     float per;
     printf("\n Enter your percentage;");
     scanf("%f",&per);
             printf("\n you are");
             printf("%s", per >= 60 ?"Passed":"Failed");
             getch();
            }
```

EXERCISES

1. Write a program that takes a number as input from user and checks whether the number is even or odd.
   a) Switch case:

b) Using conditional operator:

2. Write a program that declares and initializes two numbers with *your_roll_no* and *your_friend_roll_no* and displays the greater of the two. Use ternary operator.

Lab Session 07

OBJECTIVE

Study of Loops (For loop)

THEORY

Types of Loops

There are three types of Loops:

- for Loop
- while Loop
    - do - while Loop

Nesting may extend these loops.

The for Loop

```
for(initialize;condition;increment)
{
    Do this;
}
```

This loop runs as long as the condition in the parenthesis is true. Note that there is no semicolon after the "*for*" statement. If there is only one statement in the ***"for"*** loop then the braces may be removed.

An Example: A program that prints a list of odd numbers from 1 to 100

```
                void main(void)
    {
                    clrscr();
      for(int i=1;i<100;i+=2)
      printf("%d\t",i);
    }
```

EXERCISES

1. Write necessary statements using ***for*** loop for the following:

    i. To print your name *Your_roll_no* times, using ***for*** loop.

ii. To print all the odd numbers between *Your_roll_no* and *Your_roll_no* + 100.

iii. To print the square of number(s) repeatedly till the **1** is entered by user..

2. Write a program to print the average of two numbers input by user.

3. Write a program to generate the following output using ***for*** loop.

```
1 10 2 9 3 8 4 7 5 6 6 5 7 4 8 3 9 2 10 1
```

Lab Session 08

OBJECTIVE

Study of Loops (While, do-while loop)

The while Loop

```
while(condition is true)
 {
   Do this;
 }
```

This loop runs as long as the condition in the parenthesis is true. Note that there is no semicolon after the ***"while"*** statement. If there is only one statement in the ***"while"*** loop then the braces may be removed.

An Example: A program that prints all numbers from 1 to 100 those are divisible by 5

```
                 void main(void)
{
                   clrscr();
 int i;
 while(i<=100)
 {
   if(i%5==0)
   printf("\t\n %d ",i);
   i++;

  }//while ends
}//main ends
```

The do-while Loop

```
do
{
  this;
}
while(condition is true);
```

This loop runs as long as the condition in the parenthesis is true. Note that there is a semicolon after the ***"while"*** statement. The difference between the ***"while"*** and the ***"do-while"*** statements is that in the ***"while"*** loop the test condition is evaluated before the loop is executed, while in the ***"do"*** loop the test condition is evaluated after the loop is executed. This implies that statements in a ***"do"*** loop are executed at least once. However, the statements in the ***"while"*** loop are not necessarily executed.

An Example: A program that prints a string 10 times

```
                 void main(void)
{
 clrscr();
```

```
                      int i=0;
  do
  {
    printf("\n\t Your Name");
    i++;
  }
                      while(i<10);
}//main ends
```

EXERCISES

1. Write necessary statements using ***for*** loop for the following:

   i. To print your name *Your_roll_no* times, using ***while*** loop.

   ii. To print all the odd numbers between *Your_roll_no* and *Your_roll_no* + 100.

   iv. To print the square of number(s) repeatedly till the **1** is entered by user..

3. Write a program to print the average of two numbers input by user using ***while*** loop.

3. Write a program to generate the following output using ***do-while*** loop.

1 10 2 9 3 8 4 7 5 6 6 5 7 4 8 3 9 2 10 1

Lab Session 09

OBJECTIVE

Using Break and Continue within loops

THEORY

Two keywords that are very important to looping are break and continue. The break command will exit the most immediately surrounding loop regardless of what the conditions of the loop are. Break is useful if we want to exit a loop under special circumstances. For example, let's say the program we're working on is a two-person checkers game. The basic structure of the program might look like this:

```
while (true)
{
take_turn(player1);
   take_turn(player2);
}
```

This will make the game alternate between having player 1 and player 2 take turns. The only problem with this logic is that there's no way to exit the game; the loop will run forever. Following example shows a different approach

```
while(true)
{
if (someone_has_won() || someone_wants_to_quit() == TRUE)
{
break;
}
take_turn(player1);
if (someone_has_won() || someone_wants_to_quit() == TRUE)
{
break;
}
   take_turn(player2);
}
```

This code accomplishes what we want--the primary loop of the game will continue under normal circumstances, but under a special condition (winning or exiting) the flow will stop and our program will do something else.

Continue is another keyword that controls the flow of loops. If you are executing a loop and hit a continue statement, the loop will stop its current iteration, update itself (in the case of for loops) and begin to execute again from the top. Essentially, the continue statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me." Let's say we're implementing a game of Monopoly. Like above, we want to use a loop to control whose turn it is, but controlling turns is a bit more complicated in Monopoly than in checkers. The basic structure of our code might then look something like this:

```
for (player = 1; someone_has_won == FALSE; player++)
   {
      if (player > total_number_of_players)
      {player = 1;}
      if (is_bankrupt(player))
      {continue;}
      take_turn(player);
   }
```

This way, if one player can't take her turn, the game doesn't stop for everybody; we just skip her and keep going with the next player's turn.

Exercise:

Write a program to calculate sum of maximum of 10 numbers. Negative numbers should be skipped from calculation.

Lab Session 10

OBJECTIVE

Study of Functions

THEORY

Types of functions in C programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming
There are two types of functions in C programming:

- Standard library functions
- User defined functions

Standard library functions

The standard library functions are in-built functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.These functions are defined in the header file. When you include the header file, these functions are available for use. For example:

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in"stdio.h" header file. There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

User-defined functions

As mentioned earlier, C language allows programmer to define functions. Such functions created by the user are called user-defined functions.Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want. Following is the structure of a program containing functions.

```
#include <stdio.h>
void functionName()
{
   ... .. ...
   ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

The execution of a C program begins from the main() function.When the compiler encounters functionName(); inside the main function, control of the program jumps to

void functionName()

And, the compiler starts executing the codes inside the user-defined function.The control of the program jumps to statement next to functionName();once all the codes inside the function definition are executed.Before defining a function, it is required to declare the function i.e. to specify the function prototype. A function declaration is followed by a semicolon '**;**'. Unlike the function definition only data type are to be mentioned for arguments in the function declaration. The function call is made as follows:

return_type = function_name(arguments);

There are four types of functions depending on the return type and arguments:

- Functions that take nothing as argument and return nothing.
- Functions that take arguments but return nothing.
- Functions that do not take arguments but return something.
- Functions that take arguments and return something.

Consider a simple example of function declaration, definition and call.

```
void function1(void);
void function2(void)
{
  printf("Writing in Function2\n");
}
void main(void)
{
  printf("Writing in main\n");
  function1( );
}
void function1(void)
{
  printf("Writing in Function1\n");
  function2( );
}
```

Consider another example that adds two numbers using a function **sum()** .

```
               void sum(void);
               void main(void)
       {
        printf("\nProgram to print sum of two numbers\n");
        sum(void);
        }
       void sum(void)
      {
        int num1,num2,sum;
        printf("Enter 1st number:");
        scanf("%d",&num1);
        printf("Enter 2nd number:");
        scanf("%d",&num2);
        sum=num1+num2;
        printf("Sum of %d+%d=%d",num1,num2,sum);
      }
```

EXERCISES

1. Using function, write a complete program that prints your name 10 times. The function can take no arguments and should not return any value.

2. Write function definition that takes two complex numbers as argument and prints their sum.

3. Using a function, swap the values of two variables. The function takes two values of Variables as arguments and returns the swapped values:

4. Identify the errors (if any) in the following code:
   a)
```
func(int a,int b)
  {
    int a;
    a=20;
    return a;
   }
```

b) #include<stdio.h>
```
int main()
{
 int myfunc(int);
 int b;
 b=myfunc(20);
 printf("%d",b);
 return 0;
}
int myfunc(int a)
{
        a > 20? return(10): return(20);
}
```

Lab Session 11

OBJECTIVE

Recursion

Theory

Recursion is an ability of a function to call itself.

An example: A program that calculates the following series using recursion.

n + (n-1) + (n-2) + ……….. + 3 +2 + 1

```
 int add(int);
          void main(void)
{
  int num,ans;
  printf("Enter any number:");
  scanf("%d",&num);
  ans=add(num);
  printf("Answer=%d",ans);
          getch();
}
 int add(int n)
{
  int result;
  if(n==1)
    return 1;
  result=add(n-1) + n;
  return result;
 }
```

Built-in Functions
There are various header files which contain built-in functions. The programmer can include those header files in any program and then use the built-in function by just calling them.

EXERCISES

1. Using recursion, write a program that takes a number as input and print its binary equivalent.

2. main( ) is a function. Write a function which calls main( ). What is the output of this program?

Lab Session 12

OBJECTIVE

Study of Arrays

THEORY

An array is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an element of the array. You declare an array by writing the type, followed by the array name and the subscript. The subscript is the number of elements in the array, surrounded by square brackets. For example,

long LongArray[25];

declares an array of 25 long integers, named LongArray. When the compiler sees this declaration, it sets aside enough memory to hold all 25 elements. Because each long integer requires 4 bytes, this declaration sets aside 100 contiguous bytes of memory.

Consider the following program that take 10 numbers as input in an array and then print that array.

```
#include<stdio.h>

          void main(void)

{

 clrscr();

 int arr[10];

            for(int i=0;i<10;i++)

  {

                  printf("\n\tEnter element %d:",i+1);

       scanf("%d",&arr[i]);

  }

            clrscr();
```

```
    for(int j=0;j<10;j++)

    printf("\n\n\t Element %d is %d",j+1,arr[j]);

                 getch();

}
```

EXERCISES

1. Write a program that store first ***n*** Fibonacci numbers in an array, where ***n*** is equal to *your_roll_no*.

2. Implement the built-in function **atoi( )** present in "stdlib.h"**.**

3. Declare an array of length equal to *Your_roll_no*. Take input in the array using a function. Use another function to find the smallest element in that array.

6. Write a program to search a number in an array of 10 elements.

Lab Session 13

OBJECTIVE

String Arrays

THEORY

*A string is an especial type of array of type **char***. Strings are the form of data used in programming languages for storing and manipulating text. A string is a one dimensional array of characters. Following are some examples of string
initializations

char str1[]={ N , E , D , \0 };
char str2[]={ NED };
char str3[]= NED ;

Each character in the string occupies one byte of memory and the last character is always a NULL i.e. \0 , which indicates that the string has terminated. Note that in the second and third statements of initialization \0 is not necessary. C inserts the NULL character automatically.

An example
Let us consider an example in which a user provides a string (character by character) and then the stored string is displayed on the screen.

```
/* Strings*/
void main(void)
{
clrscr();
char str[20];
char ch;
int i=0;
printf( \nEnter a string (20-characters max): );
while((ch=getche())!= \r ) /*Input characters until
return key is hit*/
{
str[i]=ch;
i++;
}
str[i] = \0 ;
printf( \nThe stored string is %s ,str);
getch();
}
```

It is necessary to provide \0 character in the end. For instance if you make that statement a comment, you will observe erroneous results on the screen.

Library Functions for Strings

There are many library functions for string handling in C. Some of the most common are listed below. In order to use these library functions you have to include header file named **string.h**.

Functions Use

strlen	:Finds length of the string
strlwr	:Converts a string to lowercase
strupr	:Converts a string to uppercase
strcpy	:Copies a string into another
strcmp	:Compares two strings
strrev	: Reverses string
gets	Input string from keyboard
puts	:Output string on the screen

EXERCISES

1. Implement the built-in function **strcmpi()** present in header file “string.h”

2. Write a program that takes five names as input and sort them by their lengths. Use a separate function for sorting.

Lab Session 14

OBJECTIVE

Study of Structures

THEORY

If we want a group of same data type we use an array. If we want a group of elements of different data types we use structures. For Example: To store the names, prices and number of pages of a book you can declare three variables. To store this information for more than one book three separate arrays may be declared. Another option is to make a structure. No memory is allocated when a structure is declared. It simply defines the “form” of the structure. When a variable is made then memory is allocated. This is equivalent to saying that there is no memory for **“int”**, but when we declare an integer i.e. **int var;** only then memory is allocated.

Consider the following example of a structure:

```
struct personnel
{
  char name[50];
  int agentno;
 };
           void main(void)
{

  struct personnel agent1={“Mustafa”,35};

  printf(“%s”,agent1.name);

  printf(“%d”,agent1.agentno);

                 getch();

  }
```

EXERCISES

1. Declare a structure named **employee** that stores the *employee id, salary* and *department*.

2. Define a structure to represent a complex number in rectangular format i.e. *real* + **i***imag*. Name it rect. Define another structure called polar that stores a complex number as polar format i.e. mag /angle . Write a function called convert that takes a complex number as input in rectangular format and returns the complex number converted in Polar form.

Lab Session 15

OBJECTIVE

Unions

THEORY

Unions are also used to group a number of different variables together like a structure. But, unlike structures, union enables us to treat the same space in memory as a number of different variables. That is, a union is a way for a section of memory to be treated as a variable of one type on one occasion, and as a different variable, of a different type, on another occasion.

In a structure, all the members are individual objects allocated contiguously. In a union, all the members refer to the same object, and are allocated at the same address. All the members of a union are equivalent, and the size of a union will be the size of its largest member. An example is shows as

```
union
{ char a; int b; long c; }
 u = 1;
```

field a is initialized with the value 1 on a char. It is then more convenient to define the largest field first to be sure to initialize all the union byte.

EXERCISES

1. Declare an array of 40 employees for the structure defined in question1. Also write statements to assign the following values to the employee [6].
   Employee id = "Your_roll_no" *salary* = 30,000 and *department* = "IT dept"

2. Write a function that prints the highest salaried person amongst the employees defined in question 1.

4. How much memory is allocated for obj1 in the following code?

```
union x
{
  int i[(int)ceil(your_roll_number/2)];    //declare an array
                                    //having as many elements as your //half of your roll
                                    number
  char c;
  float f;
} obj1;
```

Lab Session 16

OBJECTIVE

Graphics in C

THEORY

There are two ways to view the display screen in Turbo C graphics model:

- The Text Mode
- The Graphics Mode.

The Text Mode
In the Text Mode, the entire screen is viewed as a grid of cells, usually 50 rows by 80 columns. Each cell can hold a character with certain foreground and background colors (if the monitor is capable of displaying colors). In text modes, a location on the screen is expressed in terms of rows and columns with the upper left corner corresponding to (1,1), the column numbers increasing from left to right and the row numbers increasing vertically downwards.

The Graphics Mode
In the Graphics Mode, the screen is seen as a matrix of pixels, each capable of displaying one or more color. The Turbo C Graphics coordinate system has its origin at the upper left hand corner of the physical screen with the x-axis positive to the right and the y-axis positive going downwards.

The ANSI Standard Codes
The ANSI – American National Standards Institute provides a standardized set of codes for cursor control. For this purpose, a file named ANSI.sys is to be installed each time you turn on your computer. Using the config.sys file, this job is automated, so that once you've got your system set up, you don't need to worry about it again. To automate the loading of ANSI.sys follow these steps:

1. Find the file ANSI.sys in your system. Note the path.
2. Find the config.sys file. Open this file and type the following:
   DEVICE = path_of_ANSI.sys
3. Restart your computer.

All the ANSI codes start by the character \x1B[after which, we mention codes specific to certain operation. Using the #define directive will make the programs easier to write and understand.

LIBRARY FUNCTIONS

initgraph():

This function initializes the graphics system by loading a graphics driver from disk (or

validating a registered driver) then putting the system into graphics mode. initgraph also resets all graphics settings (color, palette, current position, viewport, etc.) to their defaults, then resets graphresult to 0.
Declaration:
void far initgraph(int far *graphdriver, int far *graphmode, char far *pathtodriver);

***graphdriver:**
Integer that specifies the graphics driver to be used. *graphmode : Integer that specifies the initial graphics mode (unless *graphdriver =DETECT).
If *graphdriver = DETECT, initgraph sets *graphmode to the highest resolution available for the detected driver.

pathtodriver : Specifies the directory path where initgraph looks for graphics drivers *pathtodriver. Full pathname of directory, where the driver files reside. If the driver is not found in the specified path, the function will search the current directory for the .BGI files.

closegraph():
This function switches back the screen from graphcs mode to text mode. It clears the screen also. A graphics program should have a closegraph function at the end of graphics. Otherwise DOS screen will not go to text mode after running the program. Here, closegraph() is called after getch() since screen should not clear until user hits a key.

EXERCISES

1. Write down program statements to initialize the graphics mode of operation.

____

____

2. Which header file is required to be included while working in (a) text mode (b) graphics mode?

____

____

____

____

3. Name the functions used to clear the screen in (a) text mode (b) graphics mode

____

____

Lab Session 17

OBJECTIVE

Executing Different Functions in Graphics

THEORY

outtextxy():

Function outtextxy() displays a string in graphical mode. You can use different fonts, text sizes, alignments, colors and directions of the text. Parameters passed are x and y coordinates of the position on the screen where text is to be displayed.
Declaration:
void far outtextxy(int x, int y, char *text);

circle():

circle() function takes x & y coordinates of the center of the circle with respect to left top of the screen and radius of the circle in terms of pixels as arguments.
Declaration:
void far circle(int x, int y, int radius);
(x,y): Center point circle. radius: Radius of circle.

rectangle() & drawpoly():

To draw a border, rectangle and square use rectangle() in the current drawing color, line style and thickness. To draw polygon with n sides specifying n+1 points, the first and the last point being the same.
Declaration:
void far rectangle(int left, int top, int right, int bottom);
void far drawpoly(int numpoints, int far *polypoints);
(left,top) is the upper left corner of the rectangle, and (right,bottom) is its lower right corner.
numpoints: Specifies number of point.s

olypoints:

Points to a sequence of (numpoints x 2) integers. Each pair of integers gives the x and y coordinates of a point on the polygon. To draw a closed polygon with N points, numpoints should be N+1 and the array polypoints[] should contain 2(N+1) integers with first 2 integers equal to last 2 integers.
Following is an example that displays different shapes in graphics mode (Circle, Rectangle and Line etc.)

```
void main(void)
```

```
{
int gm,gd=DETECT;
initgraph(&gd,&gm,"path");
circle(100,100,50);
outtextxy(75,170,"Circle");
rectangle(200,50,350,150);
outtextxy(240,170," Rectangle");
line(100,250,540,250);
outtextxy(300,260,"Line");
ellipse(500,100,0,360,100,50);
outtextxy(480,170,"Ellipse");
getch();
closegraph();
}
```

EXERCISES

1. Write a program to draw a circle whose diameter should be equivalent to the width of the screen and an ellipse whose center should be at center of screen.

2. Write a program to simulate a countdown timer from 5 to 0 in center of screen. The desired font color is CYAN, font style is TRIPLEX_FONT and font size is 10. Hint: You may have to use itoa( ) function.

3. Identify the error(if any) in the following code:

```
    #include<conio.h>
void main()
{
int gd=DETECT, gm;
initgraph(&gd, &gm, "c:\\turboc3\\bgi " );
circle(200,100,150);
getch();
closegraph();
}
```

4. Write a program that draws a circle using **putpixel( )** function.

Lab Session 18

OBJECTIVE

Study of Pointer Variables

THEORY

A pointer provides a way of accessing a variable without referring to the variable directly. The address of the variable is used.
The declaration of the pointer ip,

```
int *ip;
```

means that the expression *ip is an int. This definition set aside two bytes in which to store the address of an integer variable and gives this storage space the name ip. If instead of int we declare

```
char * ip;
```

again, in this case 2 bytes will be occupied, but this time the address stored will be pointing to a single byte.

EXERCISES

1. Give the function definition for the following function declarations:

   i. void sort (char **x ,int no_of_strings);
   // Sorts the strings in alphabetical order

ii. char* strstr(char *s1, char *s2);
//Returns the pointer to the element in s1 where s2 begins.

iii. int strlen (char *str);
// Determines length of string

iv. void swap (int *x, int *y);
// You can NOT declare any variable in the function definition

Lab Session 19

OBJECTIVE

Pointers and Arrays

THEORY

There is an inherent relationship between arrays and pointers; in fact, the compiler translates array notations into pointer notations when compiling the code, since the internal architecture of the microprocessor does not understand arrays.An array name can be thought of as a constant pointer. Pointer can be used to do any operation involving array subscript. Let us look at a simple example.

An example:

```
/*Pointers and Arrays*/
void main(void)
{
int arr[4]={1,2,3,4}; /*Initializing 4-element
integer type array*/
for(int indx=0;indx<4;indx++)
printf( \n%d ,arr[indx]);
for(int indx=0;indx<4;indx++)
printf( \n\t%d ,*(arr+indx));/*arr is a constant pointer
referring to 1st element*/
int *ptr=arr; /*ptr is a pointer variable, storing
base address of array*/
for(int i=0;i<4;i++)
printf( \n\t\t%d ,*ptr++);/*ptr will be incremented(by 2-
byte) on the bases of its type*/
getch();
}
```

EXERCISES

1. Write pointer notation equivalent to the following array notations:

   a. arr[10] : ________________________________

   b. arr2D[5][6] : ________________________________

2. Using dynamic memory allocation, declare an array of the length user wants. Take input in that array and then print all those numbers, input by the user, which are even. The verification of whether a number is even or not should be done via macro.

Lab Session 20

OBJECTIVE

Pointers as Function Arguments

THEORY

One of the best things about pointers is that they allow functions to alter variables outside of their own scope. By passing a pointer to a function you can allow that function to read *and write* to the data stored in that variable. Say you want to write a function that swaps the values of two variables. Without pointers this would be practically impossible, following is an example showing its application.

```
#include <stdio.h>

int swap_ints(int *first_number, int *second_number);

int
main()
{
  int a = 4, b = 7;

  printf("pre-swap values are: a == %d, b == %d\n", a, b)

  swap_ints(&a, &b);

  printf("post-swap values are: a == %d, b == %d\n", a, b)

  return 0;
}

int
swap_ints(int *first_number, int *second_number)
{
  int temp;

  /* temp = "what is pointed to by" first_number; etc... */
  temp = *first_number;
  *first_number = *second_number;
  *second_number = temp;

  return 0;
}
```

EXERCISES

1. Using pointers, write a program that takes a string as input from user and calculates the number of vowels in it.

Lab Session 21

OBJECTIVE

Working with Files

THEORY

A majority of programs need to read and write data to disk-based storage systems. This is to be done to provide permanent storage of data. Disk I/O operations are performed on entities called **files** A file is a collection of bytes that is given a name. The various ways file I/O can be performed in C form a number of overlapping categories, which include Standard I/O and System I/O.

When accessing files through C, the first necessity is to have a way to access the files. For C File I/O you need to use a FILE pointer, which will let the program keep track of the file being accessed. For Example:

```
FILE *fp;
```

To open a file you need to use the **fopen** function, which returns a FILE pointer. Once you've opened a file, you can use the FILE pointer to let the compiler perform input and output functions on the file.

```
FILE *fopen(const char *filename, const char *mode);
```

Here filename is string literal which you will use to name your file and mode can have one of the following values

w - open for writing (file need not exist)

a - open for appending (file need not exist)

r+ - open for reading and writing, start at beginning

w+ - open for reading and writing (overwrite file)

a+ - open for reading and writing (append if file exists)

To close a function you can use the function:

```
int fclose(FILE *a_file);
```

fclose returns zero if the file is closed successfully.

Data can be written to and read from the files in different ways using different functions. These functions include:

Character IO
int getc(FILE *stream);
getc returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

int putc(int c, FILE *stream);
putc outputs the character given by ***c*** to the stream given by stream.

String IO
char *fgets(char *s, int n, FILE *stream);
fgets reads characters from stream into the string s. It stops when it reads either ***n – 1*** characters or a newline character, whichever comes first. fgets retains the newline character at the end of ***s*** and appends a null byte to ***s*** to mark the end of the string.

int fputs(const char *s, FILE *stream);
fputs copies the null-terminated string ***s*** to the given output stream. It does not append a newline character, and the terminating null character is not copied.

Formatted IO
int fscanf (FILE *stream,const char *format [,address,...]);
Scans and formats input from a stream

int fprintf (FILE *stream, const char* format[,argument,...]);
fprintf sends formatted output to a stream

Record IO
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
fread reads a specified number of equal-sized data items from an input stream into a block.

size_t fwrite(void *ptr, size_t size,size_t n, FILE* stream);
fwrite appends a specified number of equal-sized data items to an output file.

EXERCISES

1. Write a program that takes input the names of the subjects from the user and write them on to the file **Subject.txt**.

2. Using formatted IO; write the information about a book onto a file which might be taken as input from user. Book information includes its *ISBN number, title* and *price*.

3. Write down necessary statements to open a file **document.txt** for reading and appending in binary mode.

4. Write down a single statement to write the records of 15 employees as declared in *question 2* of *Lab Session 09* on to the file **document.txt**.

Lab Session 22

OBJECTIVE

Linked List

THEORY

Form the term *dynamic memory allocation,* we mean that, the size of memory allocated for holding data can grow as well as shrink, on the time of execution. This will effectively help in managing memory since a predefined memory chunk can either be too large or insufficient. Linked list is one of the way of obtaining *dynamic data structures* in which collection of data items (structures) are lined up in a row and the insertion and deletion can be made any where in the list. Each member of the linked list (*node*) is connected to another by a pointer which holds the address of the next node. The pointer of the last node points to NULL, which means end of list. With the help of linked list we can implement a complex data base system. following are some important building blocks and functions used in dynamic memory allocation.

malloc( )

It takes as an argument the number of bytes to be allocated and on call, returns a pointer of type *void* to the allocated memory. This pointer can be assigned to a variable of any pointer type. It is normally used with sizeof operator.

ptr = malloc (sizeof(struct node));
The sizeof() operator determines the size in byte of the structure element *node*. If no memory is available malloc( ) returns NULL.

free( )

The function free() deallocates memory. It takes as an argument the pointer to the allocated memory which has to be freed.

free(ptr);

Let s explore some of the basic features of linked list from the following example. Here we have implemented a small record holding program. It asks from the user about the employee s name and his identification number. For that we have declared a stricture *emp* capable of holding above mentioned fields along with the address of next node. The program has two user defined functions creat() and lstal() to perform the functions of growing nodes and listing the data of all nodes respectively.

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>
#include<dos.h>
struct emp
{
int perno;
char name[20];
struct emp *ptnxt;
};
struct emp *ptcrt,*ptfst,*ptlst; // declaring pointer to
current, first and last nodes
void creat(void); // Creates a node
void lstal(void); // List the all data
void main(void)
{
clrscr();
ptfst=(struct emp*)NULL; // Initializing 1st pointer in
the beginning
char ch;
while(1)
{
printf("\ n\ nEnter Option\ nc to creat \ nd to display all
entries\ nt to terminate");
printf("\ n\ t\ t\ t Your Option:");
ch=getche();
switch(ch)
{
case 'c':
creat();
break;
case 'd':
lstal();
break;
case 't':
printf("\ n\ nThanks");
getch();
exit(0);
}
}
}
void creat(void)
{
struct emp *pttemp;
pttemp=(struct emp*)malloc(sizeof(struct emp));//
Generating a node
char temp[20];
if(ptfst == (struct emp*)NULL)
{
ptfst = pttemp;
ptlst = pttemp;
}
ptlst->ptnxt=pttemp;
fflush(stdin);
printf("\ n\ nENTER NAME:");
gets(pttemp->name);
fflush(stdin);
printf("ID NO:");
gets(temp);
pttemp->perno=atoi(temp);
pttemp->ptnxt=(struct emp*)NULL;
ptlst=pttemp;
}
void lstal(void)
{
ptcrt=ptfst; // Starting from the first node
while(ptcrt!=(struct emp*)NULL)
{
printf("\ n\ n%s",ptcrt->name);
printf("'s id is %d",ptcrt->perno);
ptcrt=ptcrt->ptnxt; // Updating the address for next node
}
}
```

EXERCISES

1. Write a program that could delete any node form the linked list. Also make use of *free( )* function.

Lab Session 23

OBJECTIVE

File and Directory Manipulation Functions

THEORY

File Manipulation is a key task to be performed by C programs, because all applications and data reside in files. If you develop an application, it is likely to use files for storage of its data and results so that they can be reused at a later date. File manipulation covers routines that enable us to determine the status of a file and to perform certain operations to keep them in order. These functions are included in the header file <io.h>.

The **Directory Manipulation** routines in Turbo C++ provide the basic tools necessary to create, modify and remove directories from the C program. The header file <dir.h> includes the functions for directory manipulation.

int chdir(const char *path);
chdir causes the directory specified by path to become the current working directory. Path must specify an existing directory.

int setdisk(int drive);
drive designating the new drive to which the current drive will be set.

int getdisk(void);
getdisk returns the current drive number (0 = A, 1 = B, 2 = C, etc.)

int mkdir(const char *path);
mkdir creates a new directory from the given path.

int rmdir(const char *path);
rmdir deletes the directory whose path is given by path.

char *getcwd(char *buf, int buflen);
getcwd gets the current working directory.

void fnmerge char *path, char *drive, char *dir, char *name, char *ext);
fnmerge make a full path name from components.

int fnsplit(char *path, char *drive, char *dir, char *name, char *ext);
fnsplit take a file's full path name as a string, split the name into its four components, then store those components.

fnmerge and fnsplit are invertible. If you split a given path with fnsplit, then merge the resultant components with fnmerge, you end up with the path.

long filelength(int handle);

filelength returns the length (in bytes) of the file associated with handle.
The file handle may be determined by using the macro fileno.

int fileno(FILE *stream);
fileno is a macro that returns the file handle for the stream.

int remove(const char *filename);
remove deletes the file specified by filename.

int rename(const char *oldname, const char *newname);
rename changes the name of a file from oldname to newname. If a drive specifier is given in newname, the specifier must be the same as that given in oldname. Directories in oldname and newname do not need to be the same, so rename can be used to move a file from one directory to another.

EXERCISES

1. Name the header files that contain file and directory manipulation functions.

____

____

____

2. Write down a program that asks the user for a file name and then displays the length of that file in bytes.

____

____

____

____

____

____

____

3. Write a program to create a directory xxx (where xxx is your first name). Copy the noname00 file from tc\bin\ to xxx. Now delete this noname00 from xxx and then also delete the directory xxx.

Lab Session 24

OBJECTIVE

Command Line Arguments

THEORY

main( ) is a function that can accept arguments just like other functions. It can also return a value. main( ) receives arguments from the command line. Use of arguments in the command line is a useful feature.

Consider the following program statement:

void main(int argc, char *argv[])

main( ) always receives two arguments. The first argument 'argc' is the number of command line arguments and the second argument 'argv' is an array of pointers to the individual arguments. Any of the element may be accessed by referring to them as *(argv + 1), *(argv +2), and so on (or in the array notation, argv[1], argv[2]). The first string, *(argv +0), is the full path name of the program itself.

The names **argc** (for Argument Count) and **argv** (Argument Value) are traditionally used in these roles, but any other name could be used instead.

To pass arguments to the main function you first need to create the exe file of the program. Consider, for example, your program named First.cpp, that would print the total number of arguments passed and their values. To create the exe file for this program, go to the Compile Menu and click on Make, or directly press F9. To execute this program, go to the command prompt and write down the following command:

```
c:\tc\bin>  First          ↵
```

This will simply pass one argument to the main. Hence the value of argc will be 1 and *(argv + 0) = "c:\tc\bin\First.exe".

The following statement will pass 4 arguments to the main function:

```
c:\tc\bin> First 5 2.3 "Programming Languages"
```

where: argc = 4
*(argv + 0) = c:\tc\bin\First.exe
*(argv + 1) = 5
*(argv + 2) = 2.3
*(argv + 3) = Programming Languages

Note that all these arguments are strings. They are not treated as their original data type. To print these values simply execute a loop argc times and each time print the required contents using *(argv + i) or arg[i] with the format specifier being %s.

In order to return a value from main, it is required to specify the data type instead of 'void' before the function name. Finally, a return statement is to be used. The following program returns an integer.

```
int main(void)
{
int num;
printf("Enter a number");
scanf("%d", &num);
return num;
}
```

EXERCISES

1. When a program is executed, how many arguments are passed to the main by default?

____

____

2. What is the value of the argument passed to main by the operating system?

____

____

3. Write a small program that uses the alternate main() declaration syntax given above. In the body of your main function, just print the value of argc. Compile your program and run it with various numbers of arguments.

____

____

____

____

____

____

____

4. Write down the command to pass the string “C is my favorite programming lanuage” as argument to the program **comline,** whose exe file is present in the folder c:\tc\bin.

Lab Session 25

OBJECTIVE

Variable Length Arguments

THEORY

Usually, when we write a function, we know beforehand the number and type of arguments that it will take. In certain cases it is not known what would be the arguments to the functions. Functions like printf, scanf etc can take a variable number of arguments. The parameter-passing conventions in C help us to access a variable number of arguments. Upon entry to the function the first argument appears on the stack (a special data structure that is used to hold the arguments and addresses of functions) just above the return address (meaning it has the next higher address). Additional arguments have successively higher addresses. If you could get to the first argument on the stack and you knew the size of all other arguments you could retrieve the arguments one by one. This is done by the va_start and va_arg macros. While the macros help us access the arguments on the stack, they cannot tell us when the argument list ends. This is done by adopting a convention. If each argument were a pointer, for example, you could mark the end of the argument list with a NULL value.

Suppose, we are writing a function **findmax** to accept a variable number of integers and return the largest of the arguments. Since, we want positive numbers only, we assume that a value of –9999 indicates the end of argument list. Here is one way to implement the **findmax** function.

```
int findmax(int firstint, ...)
{
 int maxval = -9999, x = 0;
 va_list argp;
 // Get the first optional parameter using “va_start”
 va_start(argp, firstint);
 x = firstint;
 while(x != -9999)     //-9999 marks the end of arguments
 {
```

```
            if(maxval<x) maxval = x;

            x = va_arg(argp, int);

     }

     return (maxval);

    }
```

The variable argp of type va_list is used to point to arguments. The first step in accessing the arguments is to use va_start to initialize argp. The ANSI standard requires that a function accepting a variable number of arguments must have at least one argument. The va_start macro uses the address of this compulsory first argument to set up argp. Once this is done, you can get subsequent arguments by repeatedly using the va_arg macro.

Consider the following implementation of the C **printf** function.

```
#include<stdio.h>

#include<conio.h>

#include<stdarg.h>

#include<stdlib.h>


void print(char *ptr, ...);

void putstring(char *ptr);


void main(void)

{

clrscr();

print("%c",'H');

getch();

}
```

```
void print(char *ptr, ...)

{

char * string;

va_list arg_ptr;

int integer;

char buffer[20], ch, *p;

double d_value;

int x,y;


string = ptr;

va_start(arg_ptr,ptr); //initialize the list

while(*string!='\0')

{
  if(*string!='%' && *string!='\n' &&
*string!='\t')
                                    /*if not a
format specifier or escape character*/

   putch(*string++);  //then print the
character as it is

  else

  if(*string == '%')     //if a format
specifier

  {
                                    //then
                                    extract the
                                    next
                                    character to
                                    find its data
                                    type
string++;

            switch(*string)
                                    //different
                                    data types
                                    are to be
                                    dealt
                                    differently

   {

    case 'd':              //integer

    integer=va_arg(arg_ptr,int); //retrieve
an integer from the list

//to print this integer change it to ASCII

    itoa(integer,buffer,10);

    putstring(buffer);

                                         //puts
                                    gives
                                    linefeed so
                                    we make
                                    our own
                                    function

         break;


         case 'x':              //Hex

         integer=va_arg(arg_ptr,int); //retrieve
an integer from the list

//To print this integer change it to
ASCII

         itoa(integer,buffer,16);   //16 for Hex

         putstring(buffer);

                                    // puts gives
                                    linefeed so
                                    we make
                                    our own
                                    function

              break;


                  case 'f':              //float


d_value=va_arg(arg_ptr,double);

                  gcvt(d_value,10,buffer);

                  putstring(buffer);

                  break;


                  case 'c':              //character

         ch=va_arg(arg_ptr,char);

         putch(ch);

         break;
```

```
      case '%':              //user wants to print the % sign
      putch('%');
      break;

      case's':               //string
      p=va_arg(arg_ptr,char*);
      putstring(p);
            break;

      default:
      return;
    }                              //switch ends
    string++;
   }                                   //if ends
  else if(*string=='\n') //An Escape Sequence: \n
  {
    putch('\n');
    putch('\r');
    string++;
  }
  else if(*string=='\t') //An Escape Sequence: \t
  {
    x=wherex();
    y=wherey();
    x+=8;
    x%=80;
    if(x<8)
      y++;
    if(y>25)
    {
      y=25;
      putch('\n');
      putch('\r');
    }
    gotoxy(x,y);
    string++;
   }                                               //else ends
 }
          //while ends

}
          //print ends


void putstring(char *ptr)
{
  while(*ptr!='\0')
    putch(*ptr++);
}
```

EXERCISES

1. Show output of the following program.

```
    void main(void)
    {
      clrscr();
      char ch='H';
      int i=5;
      char string[]="Programming Languages";
   print("Character:\t%c\nInteger:\t%d\nString:\t%s\n
                    Percentage:\t%f%%",ch,i,string,83.21);
      getch();
    }
```

2. Implement the scanf( ) function using Variable Length Arguments.

Lab Session 26

OBJECTIVE

Hardware Interfacing through User-developed Programs

THEORY

Hardware interfacing is a very important feature of C language. There are various approaches to interact with the hardware. These include:

- Using high level language functions
- Using ROM-BIOS functions (routines)
- Using DOS functions (routines) stored in the file IO.sys and MSDOD.sys
- Directly programming the hardware.

You may decide to employ any one of these approaches in your programs, but the one, which directly programs the hardware, would run fastest. At the same time this is the one, which is the most unreliable. This is because, first, the programmer must have a detailed knowledge of the hardware he is trying to program. Secondly, the programs that we write for one type of hardware may not be portable to another computer, which has a different hardware. The programs, which use High Level Language functions to interact with the hardware are no doubt the most reliable, but work very slowly. Moreover, you are limited by what the functions have been designed to do. So, it is preferred to use the other two approaches i.e. either the ROM BIOS or the DOS functions.

To receive a **byte** of data from a particular particular the macro inp or the function inportb may be used. Similarly to receive a **word** of data from a particular port the macro inpw or the function inport may be used.

int inp(unsigned portid);
reads a byte from a hardware port

unsigned inpw(unsigned portid);
reads a word from a hardware port

unsigned char inportb(int portid);
inportb reads a byte from a hardware port

int inport(int portid);
reads a word from a hardware port

int outp(unsigned portid, int value);
outputs a byte to a hardware port

unsigned outpw(unsigned portid, unsigned value);
outputs a word to a hardware port

void outportb(int portid, unsigned char value);
outputs a byte to a hardware port

void outport(int portid, int value);
outputs a word to a hardware port

The int86( ) Function

The function used to make a software interrupt occur and thereby invoke a ROM-BIOS function is a standard library function called **int86( ).** The 'int' stands for 'interrupt' and the '86' refers to the 8086 family of microprocessors. The function needs three arguments:

- Interrupt number corresponding to the ROM-BIOS function to be invoked
- Two union variables

The first union variable represents values being sent to the ROM-BIOS routine, and the second represents the values being returned from the ROM-BIOS routines to the calling C program. The values are passed to and returned from ROM-BIOS routines through CPU registers.

Consider the following functions to access the color palette:

```
void set_color(int color, char red, char green, char blue)
{
        union REGS r;
        r.h.ah = 0x10;
        r.h.al = 0x10;
        r.x.bx = color;//assign values to be sent to the ROM-BIOS
        r.h.ch = green;
        r.h.cl = blue;
        r.h.dh = red;
        int86(0x10,&r,&r);     //call service 10 hex
}


void get_color(int color, char *red, char *green, char *blue)
{
        union REGS r;
        r.h.ah = 0x10;
        r.h.al = 0x15;
        r.x.bx = color;
        int86(0x10,&r,&r);     //call service 10 hex
        *green = r.h.ah;
        *blue = r.h.cl;
        *red  = r.h.dh;
}
```

These functions can broaden the range of colors from 16 to 256. Now we use the above functions in a program as:

```
void main(void)
{
        char r,g,b;
        int color= RED;
        int *driver = DETECT, *mode;
        initgraph(driver, mode, "c:\\tc\\bgi");
        get_color(color, &r, &g, &b);//get components of
                                        //original color
        set_color(color,r+100, g-200,b);    //set new color values
        setbkcolor(color);              //use the C built-in function to
                                    //set the background to your //desired color

        getch();
        closegraph();
}
```

EXERCISES

1. Implement the functionality of gotoxy() which is used to place the cursor at the desired location on the screen by specifying the x and y coordinates. For this first select the active page, for that generate interrupt with interrupt number 0x10, service number is 5 hex and Page number (in this case insert 1) in register al. Then use following data to move the cursor:

   Interrupt Number: 0x10 (Interrupt for Video routines)
   Input registers: ah (Service number) = 2 Hex.
   dh = x- coordinate value
   dl = y- coordinate value
   Output registers: None

2. Write a C program that would send a **byte** of data to the port address 1f Hex.

Lab Session 27

OBJECTIVE

Parallel Port Interfacing

THEORY

Parallel Port Description:

Parallel port interfacing is a simple and inexpensive tool for building computer controlled devices and projects. The simplicity and ease of programming makes parallel port popular in electronics hobbyist world. You can see the parallel port connector in the rear panel of your PC. It is a 25 pin female (DB25) connector (to which printer is connected). On almost all the PCs only one parallel port is present, but you can add more by buying and inserting ISA/PCI parallel port cards.In computers, ports are used mainly for two reasons: Device control and communication. We can program PC's Parallel ports for both purposes. In PC there is always a D-25 type of female connector having 25 pins, the function of each pins are listed below.

Parallel ports are easy to program and faster compared to the serial ports. But main disadvantage is it needs more number of transmission lines. Because of this reason parallel ports are not used in long distance communications. The Pins having a bar over them means that the signal is inverted by the parallel port's hardware. If a 1 were to appear on the 11 pin [S7], the PC would see a 0. Only the Data Port will be covered in this Lab.

Sending Commands to the Data Port

Sending commands involves only the data pins [**D0 to D7**].Though it is possible to use the some other pins as input, we'll stick to the basics. The word "Parallel" denotes sending an entire set of 8 bits at once [That's why Parallel Port term]. However we can use the individual pins of the port ; sending either a 1 or a 0 to a peripheral like a motor or LED.

Example:

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
void main(void)
{
outportb(0x378,0xFF);
}
```

Now take an LED and put one terminal at pin2 and the other to pin 18,it would glow.[Use a 2K resistor in series with the LED, otherwise you'll end up ruining your LED, or source too much current from the port pin] To switch it off Use this command outportb(0x378,0x00); Instead of the line outportb(0x378,0xFF);

Explaination of outportb(0x378,0x00) & outportb(0x378,0xFF) cmmands:

0x378 is the parallel port address.For a typical PC, the base address of LPT1 is **0x378** and of PT2 is 0x278. The data register resides at this base address , status register at base address + 1 and the control register is at base address + 2. So once we have the base address , we can calculate the address of each registers in this manner. The table below shows the register addresses of LPT1 and LPT2.

Register LPT1 LPT2

data registar(baseaddress + 0) 0x378 0x278
status register (baseaddress + 1) 0x379 0x279
control register (baseaddress + 2) 0x37a 0x27a
0x00 is the command appearing at the output pins. The Format is in Hexadecimal So if u want to make pin no 2 high, that's the first data pin, send 0x01 to the parallel port. **0x01** which would mean 0000 0001 for the data port. Similarly for other pins.

EXERCISE

1. Construct an interfacing circuit to control the direction of DC motor using the parallel port.

Lab Session 28

OBJECTIVE

Mouse Interfacing using Programs in C-Language

THEORY

Mouse is a hardware device, and it may be interfaced in C programs using the ROM-BIOS routines. Using these routines, the mouse may be initialized and by supplying different values (service numbers) to the AX register and issuing the interrupt number 33h, different mouse functions may be accessed. Consider the following code that displays the current x and y coordinates and also whether a particular mouse button is pressed or not. For this purpose, first the mouse is initialized, its region of operation is specified and the mouse pointer is displayed. Then a function is developed to retrieve the mouse coordinates and the button status:

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <graphics.h>
#include <stdlib.h>

union REGS i,o;
int initmouse(void);    //initializes mouse
void showmouseptr(void); //dispalys mouse pointer
void restrictmptr(int x1,int y1,int x2,int y2); //restricts mouse movement
void getmousepos(int *button, int *x, int *y); //get mouse coordinates & button status

void main (void)
{
 clrscr();
 int *driver=DETECT, *mode;        //graphics driver
 int gm, maxx,maxy,x,y,button;
 char display[30];
 initgraph(driver, mode, "c:\\tc\\bgi");
 maxx=getmaxx();              // Find maximum x screen coordinate
 maxy=getmaxy();              // Find maximum y screen coordinate
 gotoxy(26,1);
 printf("Mouse Demonstration Program");

 if(initmouse()==0)   //initialize mouse
        {
          outtextxy(400,425,"Cannot Initialize Mouse...");
          outtextxy(400,450,"Press any key to exit...");
          getch();
```

```
            exit(0);
           }
   gotoxy(1,2);
   printf("Left Button");
   gotoxy(15,2);
   printf("Right Button");

  //define the region for mouse
  restrictmouseptr(1,57, maxx-1, maxy-1);
  showmouseptr();                 //display mouse

  while(!kbhit())                 //unless the user presses a key
  {
   getmousepos(&button,&x,&y);     //get the current mouse position
   gotoxy(5,3);
   (button&1)==1 ? printf("DOWN"):printf("UP  ");//Left Button Press/Release
   gotoxy(20,3);
  (button&2)==2 ? printf("DOWN"):printf("UP  "); //Right Button Press/Release
   gotoxy(55,2);
   printf("X: %d  Y: %d  ",x,y);     //Current X and Y coordinates
  }

  getche();
  closegraph();
}

int initmouse(void)               //initializes mouse
{
  i.x.ax=0;
  int86(0x33,&i,&o);
  return(o.x.ax);
}

void showmouseptr(void)       //displays mouse pointer
{
  i.x.ax=1;
  int86(0x33,&i,&o);
}

void restrictmouseptr(int x1,int y1, int x2, int y2) //restricts mouse movement
{
  i.x.ax = 7;
  i.x.cx = x1;
  i.x.dx = x2;
  int86(0x33,&i,&o);
  i.x.ax = 8;
  i.x.cx = y1;
  i.x.dx = y2;
  int86(0x33,&i,&o);
}
```

```
void getmousepos(int *button, int *x, int *y)
                                          //get mouse coordinates and button status
{
 i.x.ax = 3;
 int86(0x33,&i,&o);
 *button = o.x.bx;
 *x = o.x.cx;
 *y = o.x.dx;
}
```

EXERCISES

1. Extend the above program to draw a small rectangle (like a popup menu) whenever the user press the right mouse button.

2. Write the code for hiding the Mouse.

Lab Session 29

OBJECTIVE

Type and Storage Classes

THEORY

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope and lifetime of a variable.There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

Local Variable

The variables declared inside the function are automatic or local variables.The local variables exist only inside the function in which it is declared. When the function exits, the local variables are destroyed.

```
int main()
{
    int n; // n is a local varible to main() function
    ... .. ...
}

void func() {
   int n1; // n1 is local to func() fucntion
}
```

Global Variable

Variables that are declared outside of all functions are known as external variables. External or global variables are accessible to any function.

Example #1: External Variable

```
#include <stdio.h>
void display();
```

```
int n = 5;  // global variable

int main()
{
   ++n;     // variable n is not declared in the main() function
   display();
   return 0;
}

void display()
{
   ++n;     // variable n is not declared in the display() function
   printf("n = %d", n);
}
```

Output

```
n = 7
```

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword extern is used in file2 to indicate that the external variable is declared in another file.

Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables. However, modern compilers are very good at code optimization and there is a rare chance that using register variables will make your program faster. Unless you are working on embedded system where you know how to optimize code for the given application, there is no use of register variables.

Static Variable

A static variable is declared by using keyword static. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

Example #2: Static Variable

```
#include <stdio.h>
void display();

int main()
{
   display();
   display();
}
void display()
{
   static int c = 0;
   printf("%d  ",c);
   c += 5;
}
```

Output

```
0  5
```

During the first function call, the value of c is equal to 0. Then, it's value is increased by 5.During the second function call, variable c is not initialized to 0 again. It's because c is a static variable. So, 5 is displayed on the screen.

Lab Session 30

OBJECTIVE

Project

THEORY

Select a project based on the following areas

1) Hardware interfacing(e.g. controlling of home appliances, stepper motor control, virtual instrumentation etc)
2) Database management system (e.g. Hospital management system, Payroll system etc)
3) Graphical interface(e.g. Scientific calculator, Snake game, Brick game etc)
Also prepare a report on your project.