SURESH GYAN VIHAR UNIVERSITY
Accredited by NAAC with 'A+' Grade

Bachelor of Computer Application

(B.C.A.)

C Programming

Semester-II

Author- Poonam Ponde

SURESH GYAN VIHAR UNIVERSITY

Centre for Distance and Online Education
Mahal, Jagatpura, Jaipur-302025

Published by:

S. B. Prakashan Pvt. Ltd.

WZ-6, Lajwanti Garden, New Delhi: 110046

Tel.: (011) 28520627 | Ph.: 9205476295

Email: info@sbprakashan.com | Web.: www.sbprakashan.com

Designed & Graphic by : S. B. Prakashan Pvt. Ltd.

Printed at :

Syllabus

C Programming

Learning Objectives

1. Write algorithms, flowcharts and programs.
2. Implement different programming constructs and decomposition of problems into functions.
3. Use and implement data structures like arrays and structures to obtain solutions.
4. Define and use of pointers with simple applications.

Unit 1

Introduction to Computing: Introduction, Art of Programming through Algorithms and Flowcharts. Overview of C: History and importance of C, Basic structure of C program, executing a C program. Constants, Variable and Data Types: Introduction, Character Set, C Tokens, Keywords and Identifiers, Constants, Variables, Data Types, Declaration of Variables, Assigning Values to Variables, Defining Symbolic Constants. Managing Input and Output Operations: Reading a Character, Writing a Character, Formatted Input, Formatted Output. Operators and Expressions: Introduction, Arithmetic Operators, Relational Operators, Logical Operators, Assignment Operators, Increment and Decrement Operators, Conditional Operator, Bitwise Operators, Special Operators, Arithmetic Expressions, Evaluation of Expressions, Precedence of Arithmetic Operators, Type Conversions in Expressions, Operator Precedence and Associativity.

Unit 2

Decision Making and Branching: Introduction, Decision Making with IF Statement, Simple IF Statement, the IF-ELSE Statement, Nesting of IF-ELSE Statements, The ELSE IF Ladder, The Switch statement, The?: Operator, The goto statement. Decision Making and Looping: Introduction, The while Statement, The do statement, The for statement, Jumps in LOOPS.

Unit 3

Arrays: One-dimensional Arrays, Declaration of One-dimensional Arrays, Initialization of One-dimensional Arrays, Example programs- Bubble sort, Selection sort, Linear search, Binary search, Two-dimensional Arrays, Declaration of Two-dimensional Arrays, Initialization of Two-dimensional Arrays, Example programs-Matrix Multiplication, Transpose of a matrix. Character Arrays and Strings: Declaring and Initializing String Variables, Reading Strings from Terminal, Writing Strings to Screen, Arithmetic Operations on Characters, String-handling Functions, Example Programs (with and without using built-in string functions)

Unit 4

User-defined Functions: Need for functions, Elements of User-defined Functions, Definition of Functions, Return Values and their Types, Function Calls, Function Declaration, Category of

Functions, No Arguments and no Return Values, Arguments but no Return values, Arguments with Return Values, No Arguments but Returns a Value, Passing Arrays to Functions, Recursion, The Scope, Visibility and Lifetime of variables. Pointers: Introduction, Declaring Pointer Variables, Initialization of Pointer variables, accessing a Variable through its Pointer, Pointer Expressions, Pointer Increments and Scale Factor.

Unit 5

Structures: Introduction, defining a structure, declaring structure variables, accessing structure members, structure initialization, array of structures. File Management in C: Introduction, Defining and opening a file, closing a file, Input/output and Error Handling on Files.

Reference

- E. Balaguruswamy, “Programming in ANSI C”, 8th Edition, 2019, McGraw Hill Education, ISBN: 978-93-5316-513-0.
- Pradip Dey, Manas Ghosh, “Programming in C”, 2nd Edition, 2018, Oxford University Press, ISBN: 978-01-9949-147-6.
- Kernighan B.W and Dennis M. Ritchie, “The C Programming Language”, 2nd Edition, 2015, Pearson Education India, ISBN: 978-93-3254-944-9.
- Yashavant P. Kanetkar, “Let Us C”, 16th Edition, 2019, BPB Publications, ISBN: 978-938728-449-4.
- Jacqueline A Jones and Keith Harrow, “Problem Solving with C”, Pearson Education. ISBN: 978-93-325-3800-9.
- Dr. Guruprasad Nagraj, “C Programming for Problem Solving”, Himalaya Publishing House. ISBN-978-93-5299-361-1. Weblinks and Video Lectures (e-Resources):
- http://elearning.vtu.ac.in/econtent/courses/video/BS/14CPL16.html
- https://nptel.ac.in/courses/106/105/106105171/

Contents

* * *

1 An Overview Of C

1.1 HISTORY OF C

The development of C language was a result of the evolution of several languages, which can be called 'the ancestors of C'. These were Algol 60, CPL, BCPL and B.

In the 1960s many computer languages, each for a specific purpose, were developed, *for example*, COBOL and FORTRAN. The need was felt for a general purpose language that would suit a variety of applications. An international committee set for this purpose, designed Algol 60, which eventually led to the development of C.

i. **Algol 60** was a modular and structured language but it did not succeed because it was found to be too abstract and too general.

ii. The **Combined Programming Language (CPL)** developed at Cambridge University and University of London in 1963 was a successor of Algol 60.

    However it was hard to learn and difficult to implement.

iii. The **Basic Combined Programming Language (BCPL)** was very close to CPL and developed by Martin Richards at Cambridge University in 1967. BCPL was too less powerful and too specific and hence it failed.

iv. The father of C language was the B language developed by Ken Thompson of Bell Laboratories in 1970. It was designed for an early implementation of UNIX. However, it was machine dependent and a 'type-less language'. For this reason, Dennis Ritchie began work on a new language as a successor to B.

v. The 'C' programming language by Dennis Ritchie came into existence in 1972 at Bell Laboratories. The early development and use of C was closely linked with UNIX for which it was developed. For many years, the only reference available on C was the published informal description in Kernighan and Ritchie's book.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a formal comprehensive definition of 'C'. This ANSI standard known as "ANSI C" was completed in 1988.

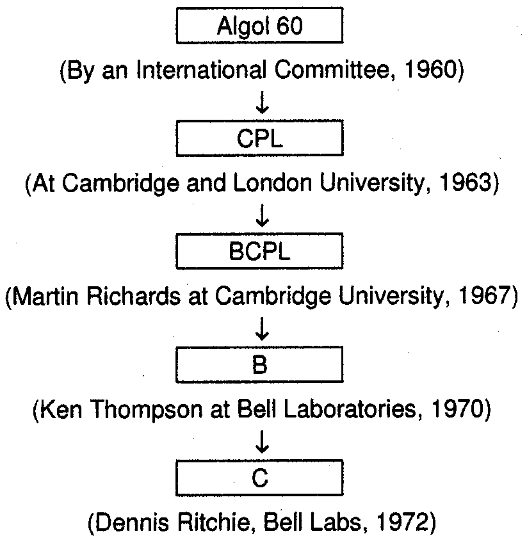

Algol 60
(By an International Committee, 1960)
CPL
(At Cambridge and London University, 1963)
BCPL
(Martin Richards at Cambridge University, 1967)
B
(Ken Thompson at Bell Laboratories, 1970)
C
(Dennis Ritchie, Bell Labs, 1972)

Figure 1.1: Development of C

1.2 COMPUTER LANGUAGES

Computer languages have evolved over the years from the earliest machine language to the recent natural languages.

All the programming languages are divided into 3 levels:

i. Low Level Language
ii. High Level Language
iii. Middle Level Language

i. Low Level Languages

These languages were the earliest languages developed. Under this category, we have Machine and Assembly languages.

Features of Low Level Languages

a. These languages are greatly hardware dependent, i.e., the code had to be written for specific hardware.

b. Programs written on one machine will not run on another (non-portable).

c. Programmers are required to have knowledge about the hardware as well.

1. **Machine Language:** Since the computer is made up of electronic circuits, they can only understand binary logic (0's and 1's). Hence in order to communicate with the computer, the user has to give instructions in term of 0's and 1's. This was called **machine language** and it was one of the earliest computer languages (1940's).

   Advantage

   Since the computer circuits can directly interpret 0 and 1, execution of programs is very fast.

   Disadvantages

   - Writing programs in binary is very difficult.
   - It is very easy to make errors during writing or data entry.
   - Debugging is very difficult.
   - There is no distinction between the instruction and operands or data.
   - It is difficult to understand the program logic by looking at the program.

2. **Symbolic / Assembly Language:** These were developed in the 1950's to remove the disadvantage of Machine Language. In these languages, small English like words, called **mnemonics** were used for instructions (*For example*: ADD, SUB, etc) and hexadecimal codes were used for data.

   Example: 8085, 8086 languages.

   Advantages

   - Writing of programs became easier.
   - Errors are minimized.
   - Identification of errors is easy.
   - There is a distinction between instructions and data.
   - Programs can be easily understood.

   Disadvantages

   1. Because a computer does not understand symbolic language, it has to be translated to machine language.
   2. A special software called **Assembler** is needed to translate assembly code to machine code.
   3. Execution becomes slower.

ii. High Level Languages

High level languages were developed to

a. Improve programming efficiency.
b. Shift focus from the computer to problem solving.
c. Develop portable applications.

Features of high level languages

1. Use of English - like words for instructions.
2. Support to multiple data-types like characters, integers, real numbers etc.
3. Hardware independent instruction set (Portability).
4. Programs have to be converted from high-level languages to machine languages.
5. Conversion is done by special Software (Compiler or Interpreter).

   Example: Pascal, FORTRAN, COBOL, BASIC, etc.

iii. Middle Level Language

C is thought of as a middle level language because it combines elements of high-level language with the functionalism of assembly language. C allows manipulation of bits, bytes and addresses - the basic elements with which the computer functions. Also, C code is very portable, that is software written on one type of computer can be adapted to work on another type. Although C has five basic built-in data types, it is not strongly typed language as compared to high level languages, C permits almost all data type conversions.

It allows direct manipulation of bits, bytes, words, and pointers. Thus, it is ideal for system level-programming.

1.2.1 'C' - Structured Language

The term block structured language does not apply strictly to C. Technically, a block-structured language permits procedures and function to be declared inside other procedures or functions. C does not allow creation of functions, within functions, and therefore cannot formally be called a block-structured language. However, it is referred to as a structured language because it is similar in many ways to other structured languages like ALGOL, Pascal and the likes.

C allows compartmentalization of code and data. This is a distinguishing feature of any structured language. It refers to the ability of a language to section off and hide all information and instructions necessary to perform a specific task from the rest of the program. Code can be compartmentalized in C using functions or code blocks. Functions are used to define and code separately, special tasks required in a program.

This allows programs to be modular. Code block is a logically connected group of program statements that is treated like a unit. A code block is created by placing a sequence of statements between opening and closing curly braces.

1.3 WHERE 'C' STANDS?

The 'C' programming languages is a very powerful and flexible language.

It provides the programmer a facility to write low-level programs as well as high-level programs. Thus, it is designed to have both–good programming efficiency and good machine efficiency.

For these reasons, C is called a **Middle Level Language.** It permits machine independent programs to be written as well as permits close interaction with the hardware.

1.3.1 Application Areas

'C' is a general purpose programming language and not designed for specific application areas like COBOL (business applications) or FORTRAN (scientific and engineering applications).

'C' is well suited for business as well as scientific applications because it has various features (rich set of operators, control structures, bit manipulation, etc.) required for these applications.

However it is better suited and widely used for system software like operating systems, compilers, interpreters, etc. characteristics.

1.3.2 Features of 'C'

In the current scenario there are several languages to choose from. Most are well suited for a variety of tasks. However, there are several reasons why 'C' is a popular programming language.

Reasons why C is a popular programming language:
i. Flexibility
ii. Powerful
iii. Small size
iv. Modular design
v. Portability
vi. High level structured language features
vii. Low level features
viii. Bit engineering
ix. Use of pointers
x. Efficiency

i. **Flexibility:** 'C' is a general purpose language. It can be used for diverse applications. The language itself places no constraints on the programmer.

ii. **Powerful:** It provides a variety of data types, control-flow instructions for structured programs and other built-in features.

iii. **Small size:** 'C' language provides no input/output facilities or file access. These mechanisms are provided by functions. This helps in keeping the language small. 'C' has only 32 keywords, which can be described in a small space and learned quickly.

iv. **Modular design:** The 'C' code has to be written in functions, which can be linked with or called in other programs or applications. C also allows user defined functions to be stored in library files and linked to other programs.

v. **Portability:** A 'C' program written for one computer system can be compiled and run on another with little or no modification. The use of compiler directives to the preprocessor makes it possible to write a single program that can be used on different types of computers.

vi. **High level structured language features:** This allows the programmer to concentrate on the logic flow of the code rather than worry about the hardware instructions.

vii. **Low level features:** 'C' has a close relationship with the assembly language making it easier to write assembly language code in a 'C' program.

viii. **Bit Engineering:** 'C' provides bit manipulation operators, which are a great advantage over other languages.

ix. **Use of pointers:** This provides for machine independent address arithmetic.

x. **Efficiency:** A program written in 'C' has development efficiency as well as machine efficient (i.e., faster to execute).

1.3.3 Limitations of 'C'

The 'C' language, however, does have its limitations:

i. It is not suitable for programming of numerical algorithms since it does not provide suitable data structures.

ii. 'C' does not perform bound checking on arrays. This results in unpredictable errors, which are difficult to locate.

iii. The order of evaluation of function arguments is not specified by the language.

Example: In the function call, f (i,++i); it is not defined whether the evaluation is left to right or right to left.

iv. The order in which operators are evaluated is not specified in some cases.

Example: In a[i] = b [i + +], the value of 'i' could be incremented after the assignment or it could be incremented after b [i] is fetched but before assignment.

The order of evaluation of operands of an operator is also not specified.

Example: Sum = (++a ,– – a). Here it is left to the compiler as to which it evaluates first.

v. 'C' is not a strongly typed language, which means that the compiler does not strictly check and indicate errors for those statements that attempt a mismatch of data types.

This can cause unintentional errors, which are difficult to trace.

1.4 PROGRAM DEVELOPMENT CYCLE

The program development cycle is completed in four steps:

i. Creating the 'C' source code

ii. Compiling the source code

iii. Linking the compiled code

iv. Running the executable file

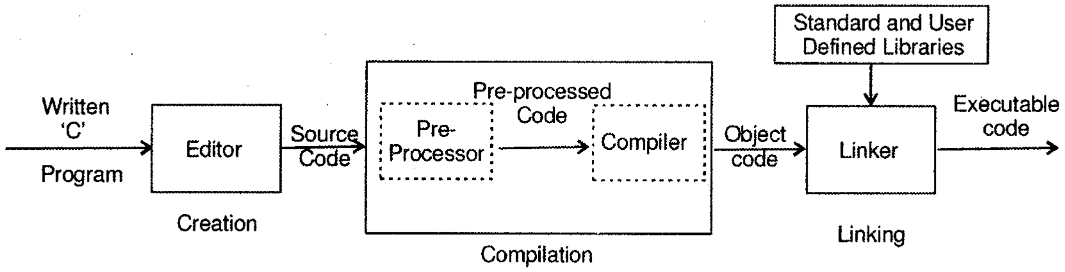


Figure 1.2

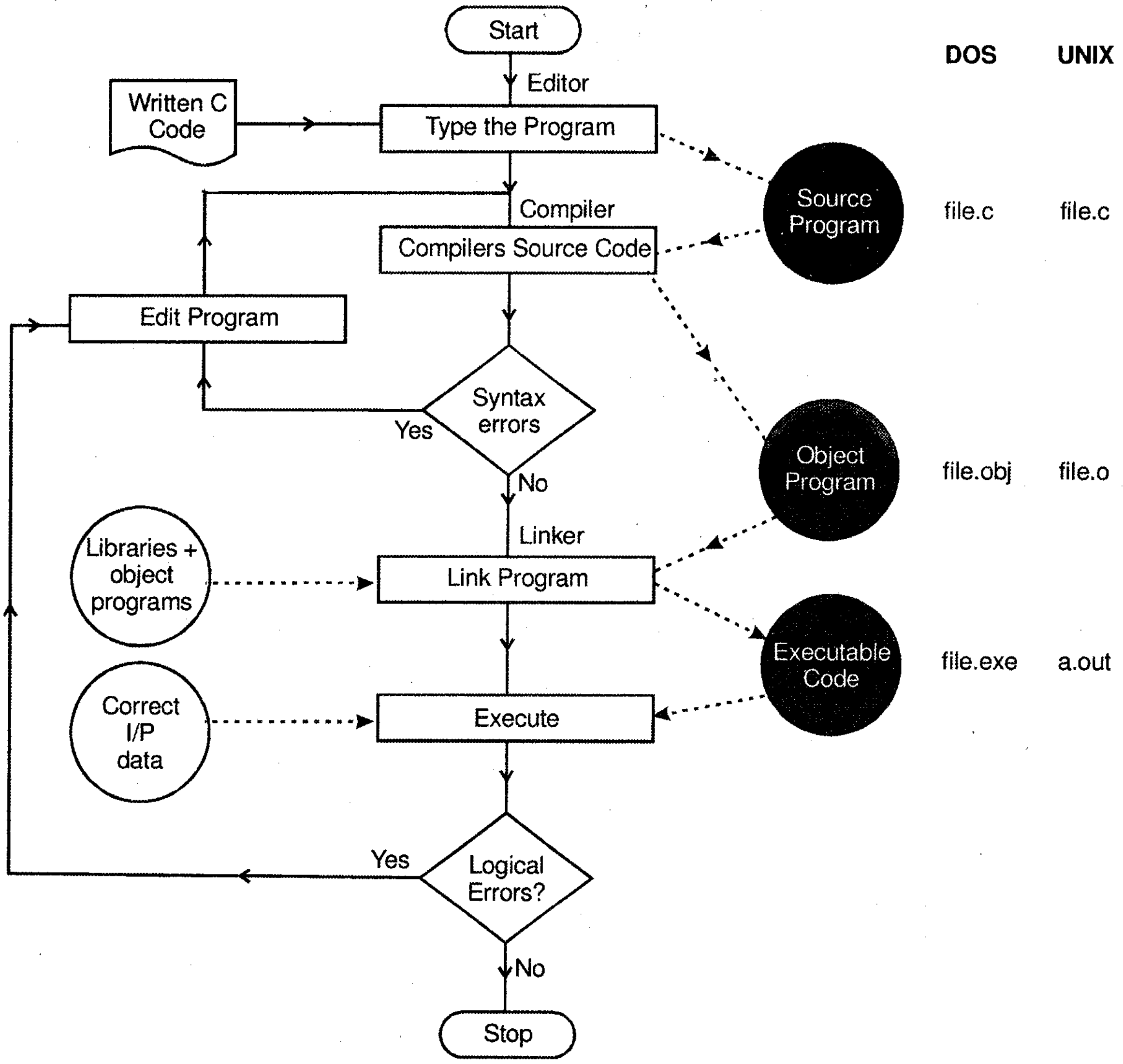


Figure 1.3

i. **Creating the source code:** Any editor or word processor can be used to create the source code. The file containing the source code has to be a 'text' file with an extension .C most compilers come with a built in editor. On UNIX, the editors like vi, emacs, etc. can be used.

ii. **Compiling the source code:** The pre-processing is the first step in the compilation. The source code is given to the pre-processor (Pre-processor is a system program that modifies a C program prior to its compilation) which checks for special instructions (preprocessor directives) in C program (line beginning with # provides an instruction to the preprocessor) and performs other tasks to give the pre-processed code.

   The compiler then converts this code to binary code (object code). On UNIX systems, the object code has an extension .O and on others it is .obj.

   Several compilers have been developed for C. Some of the commonly used ones are: Microsoft C, Borland C, Turbo C, GNU C. Programs can also be compiled on UNIX by the CC compiler.

iii. **Linking the object code to create an executable code:** The object code of the program has to be linked with the object code of precompiled routines from libraries. The linker creates a file with .exe extension.

iv. **Executing the program:** Once the executable file is created, you can run it by typing its name at the DOS command prompt or through the option provided by the compiler software. If the desired results are not achieved, changes may have to be made to the source code. When the source code is changed, it has to be recompiled and linked to create the correct executable code.

1.5 THE FORM OF A C PROGRAM

All C programs will consist of at least one function, but it is usual (when your experience grows) to write a C program that comprises several functions. The only function that has to be present is the function called **main**.

For more advanced programs the **main** function will act as a controlling function calling other functions in their turn to do the dirty work! The **main** function is the first function that is called when your program executes.

C makes use of only 32 keywords which combine with the formal syntax to form the C programming language.

Note that all keywords are written in lower case - C, like UNIX, uses upper and lowercase text to mean different things. If you are not sure what to use then always use lowercase text in writing your C programs. A keyword may not be used for any other purposes. For example, you cannot have a variable called **auto**.

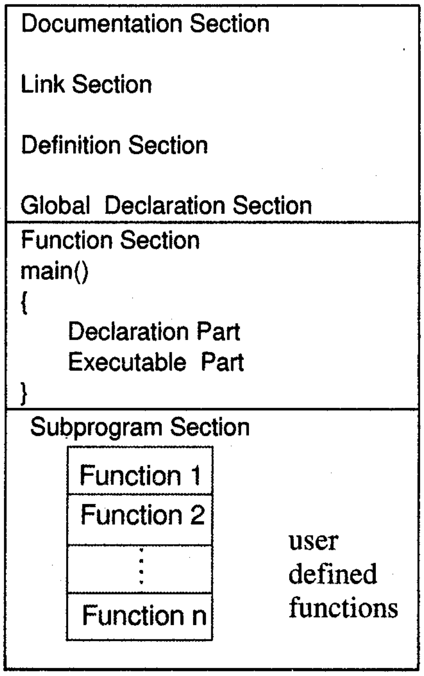
1.6 STRUCTURE OF A 'C' PROGRAM

The basic building block of every C program is **Function.**

A function is nothing but a module or a subprogram, which performs some task. It may accept some information and may return a single output.

The function main

- Every C program consists of one or more functions one of which is the function called **main.**
- Program execution begins from this function and ends when the instructions in the main function have been executed.
- The basic structure of a 'C' program is as shown below:



- The documentation section consists of comment lines (enclosed in /* and */), which are used to convey program information and other details.

  Note: Comments can be put anywhere within the program.
- The link section gives instructions to the compiler to link library files and other user files.
- The definition section defines all symbolic constants.
- Some variables need to be used in all functions. Such variables are declared in the global declaration section.
- Every C program must have one main( ) function. It consists of local declaration (information used only within main) and "C" statements. All statements end with a semicolon.

- The sub-program section contains all user-defined functions that are called in the main function.

  The subprogram section may also appear before main( ) although it is normally placed immediately after main( ).

Sample 'C' Program

To display the following message on the screen.

Hello!

Welcome to C

```
1.  /* My First C Program */
2.
3.  #include<stdio.h>
4.  main()
5.  {
6.     printf("Hello!\n Welcome to C");
7.  }
```

Output

```
Hello!

Welcome to C
```

Explanation

i. Line 1 is a 'C' comment. A comment is used to give additional information about the program. It has to be enclosed in /* and */. Comments are ignored by compiler.

   Comments can be written anywhere in the program and are used for documentation. They cannot be written inside one another (nesting).

   Example: /* First comment /* Second Comment */ */ is invalid.

ii. Line 2 is a blank line. A program can contain any number of blank lines. This improves readability of the program.

iii. Line 3 is the link section and it tells the compiler to include information about the specified file, i.e., Standard Input - Output functions. The #include directive gives the program access to a library. A library is a collection of useful functions and symbols that may be accessed by a program.

   The ANSI (American National Standards Institute) standard for C requires that certain standard libraries be provided for every ANSI C implementation. A C system may expand the number of operations available by supplying additional libraries; an individual programmer can also create libraries of functions. Each library has a standard header file whose name ends with the symbols.

   The #include directive causes the preprocessor to insert definitions from a standard header file into a program before compilation.

The directive

#include <stdio.h> /* printf, scanf definitions */ notifies the preprocesor that some names used in the program (such as prinft, scanf) are found in the standard header file <stdio.h>.

iv. Line 4 is the beginning of the main( ) function. It is the only compulsory and the most important function of any C program.

v. Lines 5 and 7 are the opening and closing braces of main. These braces contain the instructions to be executed (statements).

vi. Line 6 is the only statement in the function. It is a call to another function called printf, which is an output function. Its job is to display the provided information on the screen. The definition of this function is in the standard input output library stdio.h. Hence we have included that file in the program.

vii. The sequence of characters enclosed in " " is called a string which is displayed on the screen as it is.

viii. \n is a special character (although it is composed of two characters) called the **newline** character. This character advances the output to the next line.

printf does not supply a new line automatically. Hence multiple printf( ) statements are used. So, the following printf statements:

```
printf("Welcome");
printf("to");
printf("C");
```

Will give the following **output**

```
Welcome to C
```

We can introduce the new-line character in the string at the appropriate position. The printf statements will now look like.

```
printf("Welcome to \n C");
```

This is analogous to writing

```
printf("Welcome to \n");
printf("C");
```

1.7 COMPILERS AND INTERPRETERS

Programs written in a high level language have to be converted into machine code in order to be executed. The software which does this translation is called a Compiler or Interpreter. Some high level languages use a compiler whereas some use an interpreter.

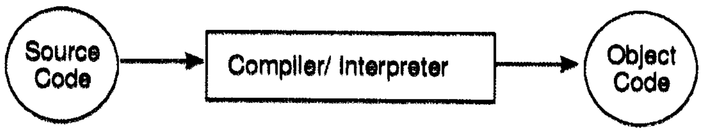


Figure 1.4

Difference between Compiler and Interpreter

	Compiler	Interpreter
1.	A compiler takes the entire program and generates the object code for the program.	An interpreter takes a single instruction of the program, converts it to object code and executes it.
2.	An intermediate object code file is created.	No intermediate file is created.
3.	Once the object code is created, the program need not be compiled every time before execution.	Every time a program is executed, conversion from high level to machine code has to be performed.
4.	A compiled program executes faster especially if the program contains loops.	An interpreter is slower than a compiler.
5.	The compiler is not involved in the execution of the program.	An interpreter also executes the instruction.
6.	There is more memory requirement since object files are created.	Memory requirement is less.
7.	A list of errors is generated after the entire program is checked.	Errors are displayed for every instruction interpreted. ∴ Debugging is easier.
8.	PASCAL, C use compilers.	BASIC has an interpreter.

1.8 EXECUTING A 'C' PROGRAM

Executing a program written in C involves a series of steps. *These are:*

i. Creating the program

ii. Compiling the program

iii. Linking the program with functions that are needed from the C library and

iv. Executing the program

Figure 1.4 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the operating system, system commands for implementing the steps and conventions for naming files may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channelled through the operating system. The operating system which is an interface between the hardware and the user, handle the execution of user programs.

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (For microcomputers). We shall discuss briefly the procedure to be followed in executing C programs under both these operating systems in the following section.

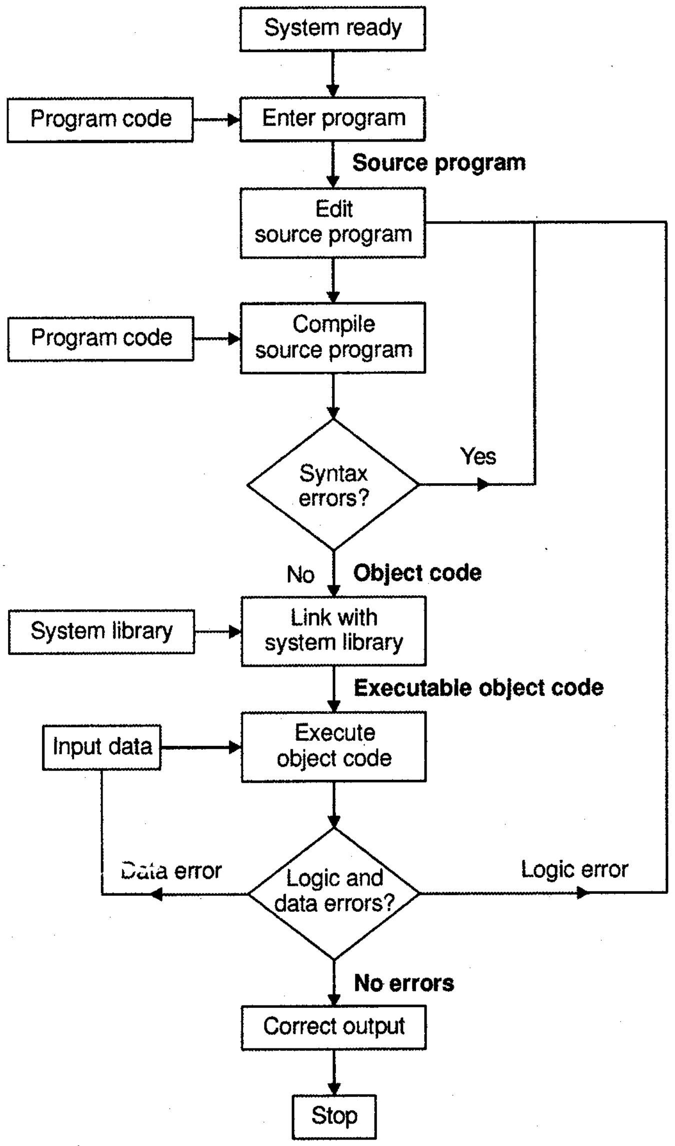


Figure 1.5

Execution of C Program on UNIX and DOS

Unix System

Creating the Program

Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter c. Examples of valid file names are

```
hello.c
program.c
ebg1.c
```

The file is created with the help of text editor, either ed or vi. The command for calling the editor and creating the file is

```
ed filename
```

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor.

When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the source program, since it represents the original form of the program.

Compiling and Linking

Let us assume that the source program has been created in a file named ebg1.c. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

```
cc ebg1.c
```

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name ebg1.o. this program is known as object code.

Linking is the process of putting together other program files and functions that are required by the program. *For Example*, if the program is using exp() function, then the object code of this function should be brought from the math library of the system and linked to the main program. Under UNIX, the linking is automatically done if no errors are detected when the cc command is used.

If any mistakes in the syntax and semantics of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the executable object code and is stored automatically in another file named a.out.

Note that some systems use different compilation command for linking mathematical functions.

```
cc filename-lm
```

is the command under UNIPLUS SYSTEM V operating system.

Executing the Program

Execution is a simple task. The command

```
a.out
```

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program logic or data. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating your own Executable File

Note that the linker always assigns the same name a.out. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

```
mv a.out name
```

We may also achieve this by specifying an option in the cc command as follows:

```
cc-o-name source-file
```

This will store the executable object code in the file name and prevent the old file a.out from being destroyed.

Multiple Source Files

To compile and link multiple source program files, we must append all the names to the cc command.

```
cc filename-1.c...... filename-n.c
```

These files will be separately compiled into object files called

```
Filename-i.o
```

And then linked to produce an executable programs file a.out as shown in *figure 1.5*.

It is also possible to compile each file separately and link them later.

For example, the command

```
cc-c mod1.c
cc-c-mod2.c
```

We may also combine the source files and object files as follows:

```
cc mod1.c mod2.o
```

Only mod1.c is compiles and then linked with the object file mod2.o. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object files is to be used along with the program to be compiled.

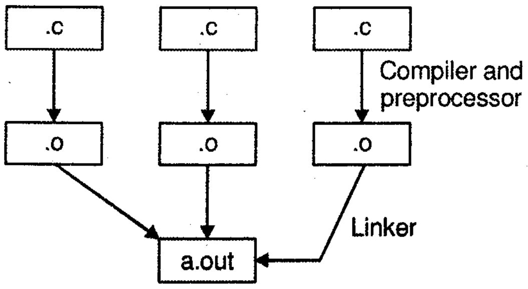


Figure 1.6

MS-DOS System

The program can be created using any word processing software in non-document mode. The file name should end with the character ".c" like program.c, pay.c, etc. Then the command

```
MSC pay.c
```

Under MS-DOS operating system would load the program stored in the file pay.c and generate the object code. This code is stored in another file under name pay.obj. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again. The linking is done by command

```
LINK pay.obj
```

which generate the executable code with the filename pay.exe. Now the command

```
pay
```

would execute the program and give the results.

EXERCISE

1. 'C' is middle level language. Comment.
2. What are the features of C programming?
3. Describe the process of creating and executing a C program under UNIX system.

VISION

2 Variables, Data Types, Operator And Expression

2.1 INTRODUCTION

A programming language is designed to help process certain kinds of data consisting of numbers, characters and strings and to provide useful output known as information. The task of processing of data is accomplished by executing a sequence of precise instructions called a program. These instructions are formed using certain symbols and words according to some rigid rules known as syntax rule (or Grammars). Every program instruction must confirm precisely to the syntax rules of the language.

In this chapter, we will discuss the concepts of constants and variables and their types as they related to C programming language.

2.2 CHARACTER SET

The C character set consists of upper and lowercase alphabets, digits, special characters and white spaces. The alphabets and digits are together called the alphanumeric characters.

i. **Alphabets**

A B C ....Z
a b c ....z

ii. Digits

0 1 2 3 4 5 6 7 8 9

iii. Special characters

, . ; : # ' " ! | ~ < > { } ( ) – _ $ % & ^ * + [ ] / \

iv. White space characters

Blank space, new-line (\n), carriage return (\r), form feed (\f), horizontal tab (\t), vertical tab (\v).

2.3 C TOKENS

The smallest individual units in a C program are called tokens as shown below.

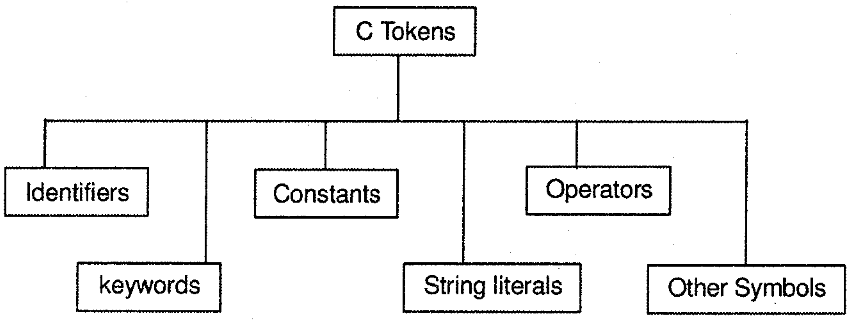


Figure 2.1

We shall be studying each of these in the sections to come.

2.3.1 Identifiers and keywords

Every C word is classified either as an identifier or a keyword.

Identifier

An identifier is a user-defined name given to a program element-variable, function and symbolic constants.

There are certain rules, which should be followed while naming an identifier. They are:

i. Identifier names must be a sequence of alphabets and digits and must begin with an alphabet or an underscore (_).

ii. No special symbols, except an underscore (_) are allowed. An underscore is treated as a letter.

iii. Reserved words (keywords) should not be used as an identifier.

iv. C is case sensitive, i.e., C treats uppercase and lowercase letters differently. It is a general practice to use lower (or mixed) case for variables and function names and uppercase for symbolic constants.

v. For any internal identifier name (an identifier declared in the same file) at least the first 31 characters are significant in any ANSI C compiler.

Examples of valid identifiers: Rate _of_ interest, add _ matrix, Sum, PI,
Month _of _Year, a123

Keywords

Keywords are reserved words and are predefined by the language. They cannot be used by the programmer in any way other than that specified by the syntax. ANSI C language has only 32 keywords. They are:

ANSI C Standard Keywords

auto	double	Int	struct
break	else	Long	switch
case	enum	register	typedef
char	extern	return	union
const	float	Short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

The following are additional keywords in Turbo C.

asm	_es	Far	near
_cs	_ss	Huge	pascal
_ds	cdecl	interrupt	

2.3.2 Constants

Constants refer to fixed values that do not change during program execution. They can be classified as:

i. Integer constants
ii. Floating point constants
iii. Characters constants
iv. String literals
v. Enumeration constants

i. Integer Constants

An integer constant refers to whole numbers. *It can be specified in three ways:*

a. Ordinary **Decimal** number (base 10)

b. **Octal** number (base 8)

c. **Hexadecimal** number (base 16)

An integer constant has to follow the following rules:

1. It contains a sequence of digits from 0 to 9. (Octal contains digits from 0 to 7; Hexadecimal constant contains digits from 0 to 9 and letters A–F).
2. An octal constant is preceded with '0' and hexadecimal constant with 0X or 0x.
3. No commas, spaces or other symbols are allowed in between.
4. The integer can be either positive or negative. It may or may not be prefixed by a + sign.
5. A size or sign qualifier can be appended at the end of the constant.

U or u for unsigned

S or s for short

L or l for long

Examples

123	56789U (unsigned integer)
–31000	7689909L (long integer)
0170	0x34ADL (long hexadecimal)
0x2A	6578890994UL (unsigned long integer)
–100 s	120US (unsigned short)

Note: The ANSI C standard supports a + sign before the positive integer corresponding to the – for a negative integer although it is rarely used.

ii. Floating Point Constants

These are real numbers having a decimal point or an exponential or both. The rules governing the floating point representation are:

a. They have a decimal point and digits from 0 to 9.

b. No embedded spaces, commas and other symbols are allowed.

c. They may or may not be prefixed by a – sign.

d. It is possible to omit digits before or after the decimal point.

Examples: 0.246 975.64 – .54 +5.

Exponential notation

This is used to represent real numbers whose magnitude is very large or very small.

The format is: mantissa e exponent

Or

mantissa E exponent

1. The **mantissa** can be a floating point number or an integer.
2. It can be positive or negative.
3. The **exponent** has to be an integer with optional plus or minus sign.

Example

The number 231.78 can be written as 0.23178e3 representing 0.23178×10^3.

75000000000 can be written as 75e9 or 0.75e11. 0.0000045 can be written as 0.45e – 5.

iii. Character Constant

A character constant is any single character from the C character set enclosed within single quotes.

Example: 'a' '#' '2'

The value of the character constant is the numeric value of the character.

Example: The character constant '0' has ASCII value 48, which is unrelated to numeric digit 0.

Escape sequences

C supports some special character constants used in output functions. They are also called backslash character constants because they contain a backslash and a character.

Although they look like two characters, they represent only one.

Complete set of escape sequence is:

Character	Meaning
\a	alert (bell)
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\0	null character
\\	backslash
\?	question mark
\'	single quote
\"	double quote
\0	octal number
\xN	hexadecimal constant (where N is hexadecimal constant)
\N	octal constant (where N is an octal constant)

iv. String Literals

A string constant or string literal is a sequence of zero or more characters enclosed in double quotes.

Example: "Welcome to C"

"First Line \n Second Line"

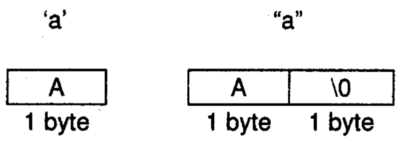
The double quotes are not a part of the string but only act as **delimiters.** If the backslash or double quote is required to be a part of the string, they must be preceded by a backslash (\).

Example:
```
printf("He said \"Hello \" "); //will display
He said "Hello"
printf("\\ is a backslash"); //displays
       \ is a backslash
```

Technically, the internal representation of a string has a null character ('\0') at the end. Therefore the physical storage required is one more than the number of characters in the string.

Difference between 'a' and "a"

'a' is a character constant and stored as the numeric value of a. "a" is a string literal and consists of the characters, a and ' \0'.



v. Enumeration Constant

An enumeration is a list of constant values - each can be represented by an integer.

It is a user defined data type with values ranging over a finite set of identifiers called enumeration constants.

Example: `enum color{red, blue, green};`

Red, blue and green are constants, which represent the integer values of 0, 1 and 2 respectively.

Values can be explicitly specified for the identifiers.

```
enum color{red = 10, blue, green = 30};
```

Here, blue is assigned number 11. If no value is specified for green it will assume the value 12.

Enumerations provide a convenient way to associate constant values with names. It also makes the program easy to read and understand.

2.4 DATA TYPES IN C

Programs work by processing data. A programming language must give you a way of storing the data. Associated with the data is its type.

When a variable is used, you have to specify what type of data it can contain.

The C programming language supports the following data types:

```
int     float     double    char    void
```

They are called basic or fundamentals data types. In addition, C also supports the enumerated data type specified by the keyword enum.

2.4.1 Fundamental Data Types

Data types	Description	Size (in bytes)	Range
char	A single character	1	-128 to 127
int	An integer number	2 (16 bit machine)	-32768 to 32767
		4 (32 bit machine)	–2,147,483,648 2,147,483,647
float	A single precision floating point number (6 precision digits)	4	3.4 e-38 to 3.4 e + 38
double	A double precision floating point number (14 precision digits)	8	1.7e-308 to 1.7e +308
void	Empty data type	0	valueless

The size allocated for an integer depends upon the compiler. The size of a data type can be obtained by using the sizeof( ) operator which gives the size of the specified data type in bytes.

Usage: `sizeof(data_type)`

Example: `printf("%d",sizeof(char));`

2.4.2 Qualifiers

A qualifier, when applied to a data type alters its size or sign.

Qualifiers	
Size Qualifier	Sign Qualifier
short	signed
long	unsigned

Normally, short and long cannot be applied to char and float and signed and unsigned cannot be applied to float, double and long double.

ANSI C has the following rules:

short int < = int < = long int

float < = double < = long double

The data types, sizes and their ranges are as shown in the following table:

All possible Data types in C (basic and qualified)

	Type	Size (in bytes)	Range
char	char	1	-128 to 127
	signed char	1	-128 to 127
	unsigned char	1	0 to 255
int	int	2(16 bit m/c)	-32768 to 32767
		4(32 bit m/c)	–2147483648 to 2147483647
	short int	2	-32768 to +32767
	long int	4	-2147483648 to 2147483647
	unsigned int	2(16 bit m/c)	0 to 65535
		4(32 bit m/c)	0 to 4294967295
	unsigned short int	2	0 to 65535
	unsigned long int	4	0 to 4294967295
float	float	4	3.4E – 38 to 3.4E + 308
double	double	8	1.7 E – 38 to 3.4 E + 38
	long double	10	3.4 E – 4932 to 1.1E 4932

Note: The exact size allocated and the ranges for these data types can be obtained from constants defined in header files <limits.h>, <float.h> and <values.h>.

2.4.3 Enumerated Data type

A user defined data type along with its set of identifiers can be created by the following declaration:

```
enum data_type_name  {constt1, constt2, ..........};
```

Example: `enum daysofweek {Sun, Mon, Tue, Wed, Thu, Fri, Sat};`

2.4.4 void Data Type

void is an empty data type defined by the keyword **void.** It is used with functions.

When used as a function return type, it means that the function does not return anything.

Example: `void calculate_and_display(int a)`

When used in place of the parameter list, it indicates that the function does not accept any information.

Example: `int random_number(void);`

We shall be dealing more with void data type in the book.

2.4.5 Creating New Data-types Names

C provides a facility called **typedef** for creating new data type names.

The syntax of typedef is `typedef data_type synonym`

For example, the statement, `typedef unsigned long ulong;`

declares ulong as a new data type equivalent to unsigned long. It can be used in exactly the same way as the type unsigned long can be.

Example: `typedef int length;`

makes the name 'length' a synonym for int.

- It is important to understand that a typedef statement does not create a new type in any sense; it merely adds a new name for some existing type.
- Use of typedef enhances program readability.

2.5 VARIABLES

A **variable** name is an identifier or symbolic name assigned to the memory location where data is stored. In other words, it is the data name that refers to the stored value. A variable can have only one value assigned to it at any given time during program execution. Its value may change during the execution of the program.

Rules regarding naming variables:

i. Since the variable name is an identifier, the same rules apply.

ii. Meaningful names should be given so as to reflect value it is representing.

student_name	rank 1
basic_sal	amount
roll_num	No_of_years

2.6 DATA DECLARATIONS AND DEFINITIONS

Programs operate on data. The data items, which a program manipulates, can be divided into two classes:

i. Constants

ii. Variables

While variables take different values at different points in time as the program executes, constants have fixed values. These must be declared before they are used.

2.6.1 Declaring Variables

All variables used in the program must be declared at the beginning.

A variable can be used to store data of any data type irrespective of what the variable name is. A variable is declared by the following **syntax**:

```
Storage class Data_type   var1,    var2,........,varn
```

where var1 to varn are variable names separated by commas. We shall study about storage classes in later.

Example:

```
int marks, age;
float amount;
```

Declaration does two things:

i. It informs the compiler the name of the variable.

ii. It specifies what type of data the variable will hold.

There are three basic places where variables will be declared:

a. Inside functions: Local variables

b. In the definition of function parameters: Formal parameters

c. Outside all functions: Global variables.

a. Local variables

These variables are also called automatic variables (keyword 'auto' may be used to declare them). They can be used only within the block where they are declared. A local variable is created upon entry into the block and destroyed upon exit.

Example: Consider two functions as shown

```
func1()
{ int x;
     x = 20;
}
func2()
{ int x;
     x = 100;
}
```

Here, x has been declared twice but the variable x in func1( ) is not related to the variable x in func2( ). Both are independent and exist only within their respective functions.

b. Formal parameters

If a function is to accept data, it must use arguments and declare them to accept values. They behave like any other local variable inside the function.

Example:
```
sum(int a, int b)
{
      ≡ } function body
}
```

Here, sum is a function which accepts two integer values in variables a and b. It could also be written as follows:

```
sum(a,b)
int a;
int b;
{
   function body
}
```

We shall be studying formal parameters in detail in the Chapter 'Functions'.

c. Global Variables

Unlike local variables, global variables exist and can be used anywhere in a program. They may be accessed by any expression regardless of what function the expression is in.

They are created by declaring them outside any function.

Example:
```
int count;            /* count is global */
main()
{ count = 200;
   func1();
}
func1( )
{ count = 300 ;
}
```

Initializing Variables

Assigning values to variables during declaration is called initialization.

Example: `int  i = 5;`

This statement not only declares the variable i but also assigns the value 5 to this variable.

Multiple variables can also be initialized.

Example: `int sum = 0, i = 10;`

2.6.2 Defining Constants

A constant can be declared in C by two methods:

i. Using const qualifier

ii. Using the #define preprocessor directive.

const is a qualifier that can be applied to a data item of any data type. The contents of this data item cannot be changed during program execution only assigned at the time of declaration (initialized).

Syntax: `cons data_type constant name = value;`

Example:
```
const float pi = 3.142;
const char quit = 'q';
```

Another method of defining constants is by using a pre-processor directive #define. The #define directive works as follows:

```
#define  CONSTNAME  literal
```

This creates a constant named CONSTNAME, which represents the constant value of the literal. By convention, the constant name is written in uppercase.

Example:
```
#define PI 3.142
#define TRUE 1
```

Any occurrence of PI in the program is replaced by the literal 3.142.

2.7 USER DEFINED TYPE DECLARATION

In C language a user can define an identifier that represents an existing data type. The user defined datatype identifier can later be used to declare variables. The general **syntax** is

```
typedef type identifier;
```

Here type represents existing data type and 'identifier' refers to the 'row' name given to the data type.

Example:
```
typedef int salary;
typedef float average;
```

Here salary symbolizes int and average symbolizes float. They can be later used to declare variables as follows:

```
Salary dept1, dept2;
Average section1, section2;
```

Therefore dept1 and dept2 are indirectly declared as integer datatype and section1 and section2 are indirectly float data type.

The second type of user defined datatype is enumerated data type which is defined as follows:

```
Enum identifier {value1, value2 …. valuen};
```

The identifier is a user defined enumerated datatype which can be used to declare variables that have one of the values enclosed within the braces. After the definition we can declare variables to be of this 'new' type as below.

```
enum identifier V1, V2, V3, ……… Vn
```

The enumerated variables V1, V2, ... Vn can have only one of the values value1, value2 ..... valuen.

Example:
```
enum day {Monday, Tuesday, …. Sunday};
enum day week_st, week_end;
week_st = Monday;
week_end = Friday;
if(week_st == Tuesday)
week_end = Saturday;
```

2.8 OPERATORS AND EXPRESSIONS

An **operator** is a symbol that represents an operation. It instructs the compiler to perform some action on one or more operands.

Example: The symbol + represents addition.

An **expression** is a combination of variables, constants and operators written according to the syntax of the language. In C, every expression evaluates to a value, i.e., every expression results in some value of a certain type that can be then assigned.

Examples of expressions:
```
a + b
P I * r * r
(x + y) - z.
```

An operator can be **unary**, **binary** or **ternary** depending on whether it operates on one, two, or three operands respectively.

Operators can be classified according to the nature of operation they perform. The different categories are:

i. Arithmetic operators
ii. Relational operators
iii. Logical operators
iv. Increment and decrement operators
v. Bitwise operators
vi. Assignment operator
vii. Conditional operators
viii. Other operators

Operator Precedence Hierarchy and Associativity

If an expression contains more than one operator, the important question is what is the order of evaluation? Some rules are needed to specify the order in which operations are performed. These rules are called Operator Precedence or Hierarchy rules.

Precedence states the relative importance or priority of operators with respect to other operators.

Another possibility is that an expression may contain more than one operator having the same priority. Here, the associativity specifies the order of evaluation of operators having the same precedence or at the same hierarchy level.

2.8.1 Arithmetic Operators

These perform arithmetic operations. C provides five arithmetic operators.

Operator	Meaning	Remark
+	Addition	Can also be used as unary plus
–	Subtraction	Also used as unary minus
*	Multiplication	
/	Division	
%	Modulo Division	Can be used only on integer data type

Note: C has no operator for exponentiation. (The function pow(x,y) in math.h can be used to calculate x^y).

- The unary minus operator has the effect of multiplying the operand by –1.
- The unary plus, which was added later, gives the value of the operand.
- Arithmetic operations performed on integers (integer arithmetic) yields an integer values.

  Example: 16 + 5 = 21
  16 – 5 = 11
  16 * 5 = 80
  16 / 5 = 3
  5 / 2 = 2
  16 % 5 = 1
  –16 % 5 = –1 (remainder after division and the sign is of the first operand)

- Arithmetic operations performed on float operands (float arithmetic) yield a float result, which is rounded off to the number of significant digits permissible.

  Example: 5. 0 + 2.0 = 7.0
  5.0 / 2.0 = 2.5
  –2.0 / 3.0 = – 0.666667

- When the operands are of different data types (mixed mode arithmetic), the result is promoted to the 'higher' data type. (char < int < float). Thus if one operand is an integer and the other float, the result will be of float type.

  Example: 5.0 / 2 = 2.5

Hierarchy of Arithmetic Operators

Operators	Associativity
* / %	L → R
+ −	L → R

Example: Consider the integer expression

5/2 +4 – 6 * 2 + 25 / 5 –3 / 4

The order of evaluation is as shown:

$\underline{5/2} + 4 - 6 * 2 + 25 / 5 - 3 / 4$
$2 + 4 - \underline{6 * 2} + 25 / 5 - 3 / 4$
$2 + 4 - 12 + \underline{25 / 5} - 3 / 4$
$2 + 4 - 12 + 5 - \underline{3 / 4}$
$\underline{2 + 4} - 12 + 5 - 0$
$\underline{6 - 12} + 5 - 0$
$- \underline{6 + 5} - 0$
$- \underline{1 - 0}$
$- 1$

Note: In order to override the operator precedence rules, parenthesis can be used. Since parenthesis have higher priority over operators.

Example: In the expression (4+5) * 6, the addition will be done first even though * has higher precedence since the addition operation is parenthesized.

2.8.2 Relational Operators

Relational operators are used to compare expressions. An expression containing a relational operator evaluates to either True (1) or False (0).

Any non-zero value is considered 'True' in C and 0 is false. Thus, even negative values are True!

The six relational operators are

Operator	Meaning
<	Less than
< =	Less than or equal to
>	Greater than
> =	Greater than or equal to
= =	Equal to (equality)
! =	Not equal to (Inequality)

These operators are mainly used in decision–making statements to decide the course of action in a program. These operators are lower in precedence than arithmetic operators. Among themselves, the precedence is

Operators	Associativity
< <= > >=	L → R
= = !=	L → R

Examples

25 < 30	True
2.5<= 2.5	True
'a'== 97	True
'b' < 'a'	False
(a+b) != (x+y)	True if the sum of values of a and b is not equal to the sum of values of x and y

2.8.3 Logical Operators

Sometimes, we need to test more than one condition at a time and make a decision depending upon the result.

The logical operators are used to combine two or more expressions (usually relational). The entire expression is called logical expression which evaluates to True (1) or False (0). The three logical operators in C are:

Operator	Meaning	Remarks
& &	Logical AND	Binary operators
||	Logical OR	
!	Logical NOT	Unary operator

Evaluation of a logical expression stops as soon as a true or false result is known.

The results of logical AND (&&) and OR (||) operators for different combinations of the two operands is given in the following truth table:

Op1	Op2	Op1 && Op2	Op1 || Op2
False	False	0	0
False	True	0	1
True	False	0	1
True	True	1	1

Examples: (marks > = 60) && (marks < 70)
age > 60 || salary > 10000

The logical NOT (!) operator takes a single expression and reverses the value of the expression, i.e., if the expression is True, the ! operator evaluates to false and vice-versa.

Example: !(5 < 10) evaluate to 0 since 5 < 10 is True.

Precedence and Associativity of Logical Operators

Operators	Associativity
!	R → L
&&	L → R
||	L → R

Note: ! has higher priority than arithmetic and relational operators, but && and || have lower priority than both.

2.8.4 Increment and Decrement operators

C provides two useful unary operators not generally found in other languages; increment and decrement operators. They are:

Operators	Meaning
++	Increment
– –	Decrement

They can be used in 2 ways:

i. **Prefix:** The operator is written before the operand. The increment or decrement is done before the value of operand is used in an expression.

Example: ++n, – – x

ii. **Postfix:** The operator is written after the operand. The increment or decrement is done after the operand value is used in an expression.

Example: n++, x – –

Note: When used independently, the prefix and postfix forms make no difference but they behave differently when used in expressions on the right hand of an assignment statements.

Example: If n is 5, then the statements ++n; and n++; both increment the values of n by 1 and are equivalent to n= n+1; However, in the statement,

y=n++;n increments after its value has been assigned to y, i.e., y is given the value 5 and then n becomes 6. Whereas y=++n first increments n to 6 and 6 is then assigned to y. The same logic applies to the decrement operator.

Examples

1. If x=0 and y=0

   z=++x || ++y;

   will result in z =1, x=1, y=0.

   Since ++x increments x to 1, the result of OR is True. Hence ++y will not be evaluated.

2. z=x++ && ++y

   Result: z=0, x=1, y=0

   x++ is post increment. The old value of x, i.e., 0 will be used. Hence the result of && is 0. Thus, ++y will not take place.

3. x++ &&++y ||z++

   If values of x, y and z are 0,1 and 0 respectively, the expression evaluates to 0 and values of x, y and z becomes 1,1 and 1 respectively.

   The && operation is performed before ||. For the && the initial value of x, i.e., 0 is used. ++ y will not be evaluated since the result of the && operation is known to be 0. For the || operation, one operand is 0 and so the other operand is evaluated. The old value of z (i.e., 0) is used since it is post- increment. ∴ 0||0 yields 0 and z then increments to 1.

2.8.5 Bitwise Operators

C has a distinction of providing six operators for manipulation of data at bit level. They are applied only to integral operands, i.e., char, short int and long whether signed or unsigned.

Operator	Operation
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
~	One's complement (Unary)

Except for ~ the others are binary operators and operate on corresponding bits of the two operands.

The bitwise XOR (exclusive OR) operator sets one in each bit position where its operands have different bits and zero where they are the same.

Example: Assume that a and b are integers with values 13 and 7 respectively. Assuming that an integer occupies 2 bytes,

a in binary	=	0000	0000	0000	1101
b in binary	=	0000	0000	0000	0111
a & b	=	0000	0000	0000	0101
a | b	=	0000	0000	0000	1111
a ^ b	=	0000	0000	0000	1010

Shift Operators

The bit pattern of the data can be shifted by a specified number of positions to the left or right using the left shift (<<) and right shift (>>) operators respectively. The shift operators perform shift of their left operand.

When the data is shifted left, the trailing empty spaces are filled with zeros.

Similarly, the leading empty spaces are zero filled when data bits are shifted right.

Example: a = 0000 0000 0000 1101

a<<3 = 0000 0000 0000 $1\underbrace{000}_{\text{zero filled spaces}}$

a>>3 = 0000 0000 0000 0001

(The rightmost three bits drop off)

The general **syntax** is `operand1 shift_operator operand2`

Note: Shifting by one position to left is effectively multiplying the operand by two.
Shifting right by one position divides the operand by two.

One's Complement Operator

The ~ operator yields the one's complement of an integer, that is, it inverts each bit of the operand (1 to 0 and vice versa)

Example: If a = 0000 0000 0000 1101
~a = 1111 1111 1111 0010

Precedence

~ is along with other unary operators like ++, -- and ! in hierarchy with R → L associativity.

The shift operators have higher precedence as compared to Bitwise AND, OR and XOR.

2.8.6 Assignment Operator

The assignment operator = is used to assign the value of an expression to a variable. The **syntax** is

```
variable = expression
```

An assignment expression followed by a; becomes an assignment statement.

Example:
```
sum = a + 10;
```
The expression a + 10 is evaluated and its value is assigned to variable sum.
```
c = a << 3;
x = a*3 + b/5;
```

Shorthand Assignment Operators

These are obtained by combining certain operators with the = operator. They have the format

```
variable  operator=  expression;
```

C supports the following shorthand assignment operators:

+= -= /= %= <<= >>= &= |= ^=

Examples:
```
x += y; implies x = x + y;
m /= 3; implies m = m/3;
a += b +1; implies a = a + (b + 1)
```

Precedence

Assignment operators have the lowest priority so far with associativity R → L.

Example: Consider the statement
```
a = b = c;
```
Here, the value of c is assigned first to b which is then assigned to a.
```
i = j += k;
```
is also a valid assignment statement which is the same as
```
i = j = j + k;
```

2.8.7 Conditional Operators

This is the only ternary operator in C. The operator pair ?: is used to construct conditional expression of the form.

$$\boxed{\text{expression1 ? expression2 : expression 3}}$$
$$\longleftarrow \text{Conditional expression} \longrightarrow$$

expression 1 is evaluated first. If it is True (nonzero), then expression2 is evaluated and becomes the resulting value of the conditional expression.

If expression is 0 (False), the value of the entire expression is that of expression3.

Example: Let a = 10 and b = 15,

```
larger = (a>b)? a : b;
```

Here larger will be assigned 15, i.e., the value of b.

This is the same as

```
if(a>b)
  larger = a;
else
  larger = b;
```

2.8.8 Other Operators

Comma Operator

The comma ',' operator is used to separate a set of expressions. A pair of expressions separated by a comma is evaluated left to right and the type and value of the result are the type and value of the right operand.

Example: Consider i = (j = 3 , j + 2) ;

Here, the right hand side contains two expressions j = 3 and j + 2 which are evaluated L →R.

Thus 3 is first assigned to j and the value 3 + 2 is assigned to i.

It could also be used to interchange the values of two variables in a single statement as shown.

```
temp = a, a = b, b = temp;
```

The comma operator has the lowest precedence and associates from L → R.

sizeof Operator

This unary operator gives the size (in bytes) of the data–type or variable. The *usage* is

```
sizeof(data_type)
```

OR

```
sizeof(object)
```

Example: sizeof(char) gives the result as 1.

Example: `printf("%d %d", sizeof(int), sizeof(float));`

typecast Operator

C provides a unary operator for explicit type conversion called cast operator. Its *usage* is

```
(type_name)expression
```

The expression is converted to the specified data type locally only for the purpose of evaluation of the expression.

Example: The ratio of number of males to the number of females in a town can be calculated as:

```
ratio = no_of_males / no_of_females;
```

Since no_of_males and no_of_females will be declared integers, the division of the two yields an integer. So even if ratio is declared as a float, the fractional part is truncated due to integer arithmetic on the right. This can be solved by locally converting one of the operands to a float so that the result of division is a float.

```
ratio = (float)no_of_males / no_of_females;
```

Address (&) and Indirection (*) Operators

C provides two unary operators for manipulating data using pointers.

The & operator when used with a variable yields its address.

The * operator denotes indirection and returns the value of the object located at the address that follow it.

We shall study more about these in later chapters.

Both these operators have a high precedence along with other unary operators.

The • and -> Operators

The • (dot) and -> (arrow) operators are used to refer to individual elements of structures and unions (covered in later chapters). Structures and unions are compound data types that can be referenced under a single name.

[ ] and ( )

Parenthesis () are used to increase the precedence of operators inside them. Square brackets perform array indexing, i.e., given an array, the expression within [ ] provides an index or subscript to the array.

2.9 TYPE CONVERSIONS IN EXPRESSIONS

In C, conversion takes place in two form:

i. Implicit type conversion ii. Explicit type Conversion

2.9.1 Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic type conversion is know as implicit type conversion.

During evaluation it adheres to very strict rules and type conversion. If the operands are of different types the lower type is automatically converted to the higher type before the operation proceeds. The result is of higher type.

i. If one operand is long double, the other will be converted to long double and result will be long double.
ii. If one operand is double, the other will be converted to double and result will be double.
iii. If one operand is float, the other will be converted to float and result will be float.
iv. If one of the operand is unsigned long int, the other will be converted into unsigned long int and result will be unsigned long int.
v. If one operand is long int and other is unsigned int then
   a. If unsigned int can be converted to long int, then unsigned int operand will be converted as such and the result will be long int.
   b. Else both operands will be converted to unsigned long int and the result will be unsigned long int.
vi. If one of the operand is long int, the other will be converted to long int and the result will be long int.
vii. If one operand is unsigned int the other will be converted to unsigned int and the result will be unsigned int.

2.9.2 Explicit Type Conversion

Many times there may arise a situation where we want to force a type conversion in a way that is different from automatic conversion.

Syntax: `(type_name) expression`

Consider for example the calculation of number of female and male students in a class:

```
Ratio =female_students/male_students
```

Since if female_students and male_students are declared as integers, the decimal part will be rounded off and its ratio will represent a wrong figure.

Example: `Ratio = (float) female_students / male_students`

The operator float converts the female_students to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of the result. The process of such a local conversion is known as explicit conversion or casting a value.

2.10 PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

The operators are listed in order of decreasing precedence. The operators grouped together in one level have the same precedence.

Level	Operator	Description	Associativity
1	( )	Function call	L → R
	[ ]	Array element reference	L → R
	–>	Pointer to structure member reference	L → R
	•	Structure member reference	L → R
2	–	Unary Minus	R → L
	+	Unary plus	R → L
	++	Increment	R → L
	—	Decrement	R → L
	!	Logical negation	R → L
	~	One's complement	R → L
	*	Pointer reference (indirection)	R → L
	&	Address	R → L
	sizeof	Size of an object	R → L
	(type)	Type cast	R → L
3	*	Multiplication	L → R
	/	Division	L → R
	%	Modulo division	L → R
4	+	Addition	L → R
	–	Subtraction	L → R
5	<<	Left shift	L → R
	>>	Right shift	L → R
6	<	Less than	L → R
	<=	Less than or equal to	L → R
	>	Greater than	L → R
	>=	Greater than or equal to	L → R
7	==	Equality	L → R
	!=	Inequality	L → R
8	&	Bitwise AND	L → R
9	^	Bitwise XOR	L → R
10	|	Bitwise OR	L → R
11	&&	Logical AND	L → R
12	||	Logical OR	L → R
13	?:	Conditional	L → R
14	= *= /= %= += –= &= ^= |= << = >>=	Assignment	R → L
15	,	Comma	L → R

SOLVED PROGRAMS

1. /* find simple interest */

```
#include<stdio.h>
main()
{
  float principal, rate, time, interest;
  clrscr();
  printf("Enter the principal:");
  scanf("%f", &principal);
  printf("\nEnter the rate of interest:");
  scanf("%f", &rate);
  printf("\nEnter the time in years:");
  scanf("%f", &time);
  /* echo the data */
  printf(\nPrincipal = %2f\n", principal);
  printf("\nRate = %2f\n",rate)
  printf("\nTime = %2f\n", time);
  interest = principal * rate * time/100.0;
  printf("\n\nSimple interest is : %2f\n", interest);
  getch(); /* freeze the monitor*/
}
```

Output

```
Enter the principal: 1000
Enter the rate of interest : 5
Enter the time in years : 4
Principal = 1000.00
Rate = 5.00
Time = 4.00
Simple interest is 200.00
```

2. /* Compute surface area and volume of a cube */

```
#include<stdio.h>
main()
{
  float side, surface_area, volume;
  clrscr();
  printf("Enter the side of cube");
  scanf("%f", &side);
  surface_area = 6*side*side;
  volume=side*side*side;
  printf("\nSurface area of cube is %2f sq. units\n"), surface_area);
  printf("\nVolume of cube is %2f cubic units\n",volume);
  getch(); /* freeze the monitor*/
}
```

Output

```
Enter the side of cube: 3
Surface area of cube is 54.00 sq. units
Volume of cube is 27.00 cubic units
```

3. /* Calculate the sum of average of five numbers */

```
#include<stdio.h>
main()
{
  float a, b, c, d, e, sum, avg;
  clrscr();
  printf("Enter the five numbers\n");
  scanf("%f%f%f%f%f", &a, &b, &c, &d, &e);
  /* echo the data */
  printf("\nEntered numbers are");
  printf(%8.2f%8.2f%8.2f%8.2f%8.2f\n", a, b, c, d, e);
  sum = a+b+c+d+e;
  avg = sum / 5.0;
  printf("\nSum = %2f\n", sum);
  printf("\nAverage=%2f\n", avg);
  getch(); /* freeze the monitor*/
}
```

Output

```
Enter the five numbers
10    25    38    59    13
Entered numbers are 10.00    25.00    38.00    59.00    13.00
Sum = 145.00
Average = 29.00
```

4. /* leap year checking*/

```
#include<stdio.h>
main()
{
  int year;
  clrscr();
  printf("Enter the year:");
```

```
    scanf("%d", &year);
    if(((year%4 ==0) && (year%100!=0))||(year%400==0))
      printf("n%d is a leap year\n", year);
    else
      printf("\n%d is not a leap year\n", year);
    getch(); /* freeze the monitor*/
}
```

Output

```
Enter the year: 2004
2004 is a leap year
Enter the year: 2005
2005 is not a leap year
```

5. What will be the output? Give explanation.

PU
Oct. 2008 – 4 M

i.
```
void main()
{
   const int a = 5;
   a++;
   printf("%d", a);
 }
```

Ans

This program displays one error to the programmer.

Error: Cannot modify a const object. This indicates an illegal operation on an object declared to be const, such as an assignment to the object.

PU
Oct. 2008 – 4 M

ii.
```
void main()
{
   enum colour (green, red = 5, blue, white,
yellow = 10, pink);
   printf("%d%d%d%d%d%d", green, red, blue, white, yellow, pink);
 }
```

Ans

This program displays total seven errors. All errors are same type "Undefined variable".

Error: Undefined Variable green, blue, white, yellow, pink.

Because when you define the enum data type it must be under { } instead of ( ). According to syntax it is wrong. So it displays seven errors.

iii.
```
int main()
{  int p= -13 >> 1;
   printf("%d", p);
}
```

PU
Apr. 2009 – 4 M

Ans

Output is –7

Because

–13 >> 1

= –13 / 2^1

= –7

iv.
```
int main()
{ printf("%d", printf("% * s % * s, "5",
"6","));
}
```

PU
Apr. 2009 – 4 M

Ans

Output is 11.

Printf function returns the sum of 5 and 6. And %*s leaves that many the blank spaces.

v.
```
void main()
{ int x, y, z;
     x = y = z = 1;
     z=++x || ++y && ++z;
     Printf("x = %d y=%d z = %d", x, y, z);
}
```

PU
Apr. 2009 – 4 M

Ans

Output is: 2 1 1

x = 2 y = 1 z = 1

2 || ++1 results true therefore second part of the statement is not checked and value of y will be as it is, i.e., y = 1 and x = 2. And the result of the logical statement is true, i.e., 1 which is again compared with second part of the statement, i.e., 1 && ++1 which is again true so compiler does not check second part of the statement. And the result of the logical statement is true.

vi.
```
void main()
{
   int x = 0x1234;
   int y = 0x5678;
   x = x & 0x5678;
   y = y | 0x1234;
   x = x ^ y;
   printf ("%x\t", x);
   printf ("%x\t", y);
}
```

PU
Oct. 2010 – 4 M

Ans

x = 0×1234

x in binary = 1001000110100

y = 0×5678

y in binary = 101011001111000

x = x & 0×5678

= 1001000110100 & 101011001111000

x = 1001000110000

Value of x in hexa decimal x = 0×1230

y = y | 0×1234

Value of y in binary = 101011001111000

1234 in binary = 1001000110100

y = y| 0×1234

= 101011001111000 | 1001000110100

= 101011001111100

y = 101011001111100

Value of y in hexa decimal = 567C

x = x ^ y

x = 0x1230 ^ 567C

x = 1001000110000 ^ 101011001111100

x = 100010001001100

x = 444C

Therefore x = 444C and y = 567C

Output = 444C 567C

EXERCISES

A. Answer the following:

1. Write equivalent C expressions for the following equations:

i. $\frac{a+b}{c+d} - \frac{a-b}{c-d}$

ii. $\left[\frac{3x^2y}{x+y} - \frac{x}{(x+y)\,(x-y)}\right]$

iii. $S = ut + \frac{1}{2}at^2$

iv. $f = \frac{9c}{5} + 32$

2. Evaluate the following C expressions:

i. 25 /4 + 3 – 7% 3 +2

ii. 6.5 + (float) 5/2 –3 % 8 – 6.5

iii. (–13 % 2) % (8*2) – 7

iv. (18–3*3) % (99–2 *10) / (2.5–1.5)

v. 2* ((18/5) + (6* (1.5 +1) % (10–2–1)

3. Given that initially i = 0, j = 2 and k = 3, find x and the new values of i, j and k for each of the following expressions:

   i. x = i ++ || ++j && k++;
   ii. x = ((i<j) | | j++)&& k++;
   iii. k *= (i + j) % k;
   iv. x = (j = = 2) ? k: i
   v. x = (i > j) ? ++i : (k>j) ? j : i
   vi. x = i++ ? j – – : k – –,
   vii. k% = j = (i =4)% (j=3)
   viii. x = j >k ? k > i ? 12 : k >j ? 13 : 14 : 15
   ix. k+ = i ++ + ++j * 3

4. What will be the output of the following?

```
main()
{ printf(); }
main()
{ const int i = 10;
i = 20;
}
```

```
main()
{ printf("H\re\rll\O");
main()
    { printf("Hello\n");
main();
}
```

B. Review questions

1. What symbol terminates every ‘C’ statement?
2. What delimiters are used to specify the beginning and end of a string in ‘C’?
3. What is the newline character?
4. Will the following statements written in a program give any errors? If yes, what are they?If no, specify the output.
   i. printf (“Welcome” ,“to” ,”C”);
   ii. printf (“Welcome” “to” “C”);
   iii. printf (“Welcome to C”);
5. Explain the four basic data types in C.
6. Explain the types of constants in C.
7. State the different categories of operators. Explain the arithmetic operators.
8. What are variables? State the rules for naming a variable.
9. What is an escape sequence?
10. What are the two methods for declaring constants?
11. Explain the use of sizeof ( ) and type cast operator.
12. Explain precedence and associativity of operators.
13. What are the different types of C statements?
14. What is the difference between a statement and a block?
15. Are negative numbers considered true or false by C?
16. What happens if a float constant is assigned to an integer variable?
17. What happens if a negative number is put into an unsigned variable?
18. Discuss logical operators of C.
19. Explain bitwise operators of C.
20. Discuss various forms of increment and decrement operators.

Collection of Questions asked in Previous Exams PU

Find and explain the output of following programme:

1.
```
void main()
{
    const int a = 5;
    a++;
    printf("%d", a);
}
```
[Oct. 2008 – *5 M*]

2.
```
void main()
{
    enum colour (green, red = 5, blue, white, yellow = 10, pink);
    printf("%d%d%d%d%d%d", green, red, blue, white, yellow, pink);
 }
```
[Oct. 2008 – *5 M*]

3.
```
int main()
{  int p= -13 >> 1;
       printf ("%d", p);
}
```
[Apr. 2009 – *5 M*]

4.
```
int main()
{   printf("%d", printf("% * s % * s", 5 ," ", 6," "));  }
```
[Apr. 2009 – *5 M*]

5.
```
void main()
{  int x, y, z;
    x = y = z = 1;
    z=++x || ++y && ++z;
    Printf("x = %d y=%d z = %d", x , y , z);
}
```
[Apr. 2009 – *5 M*]

6.
```
void main()
{
   int x = 0×1234;
   int y = 0×5678;
   x = x & 0×5678;
   y = y | 0×1234;
   x = x ^ y;
   printf("%x\t", x);
   printf("%x\t", y);
}
```
[Oct. 2010 – *5 M*]

3 Built In I/O Functions

3.1 INTRODUCTION

All computer programs essentially read, process and display data. Unlike other high level languages, C does not provide built-in input/output statements. All input/output operations have to be carried out by using functions. Many functions for the above purpose have been provided in the C standard input output library (stdio.h). Included in this file are declarations for the I/O functions and definitions of constants (like EOF, NULL, etc.).

All I/O in C is character-oriented. This includes writing and reading not only to and from the console, but also to the disc files as well. Console I/O operations in C are divided into two categories – unformatted console I/O functions and formatted I/O functions. Formatted I/O refers to the fact that these functions may format the information as per user's choice. The standard library provides functions of both categories. These functions are:

i. Unformatted console I/O operations   ii. Formatted console I/O operations

3.2 UNFORMATTED CONSOLE I/O OPERATIONS

These are console Input / Output library functions which deal with one character at a time and string functions for array of characters (String).

String Input / Output Functions

i. getchar( )   ii. putchar( )
iii. gets( )   iv. puts( )

3.2.1 Character Input and Output (getchar and putchar)

The function **getchar** reads and returns an input character from the standard input device.

Usage: `variable_name = getchar();`

The variable is of char or integer type. getchar( ) assigns the character value of the input character to the variable.

Example:
```
char c;
c = getchar();
```

The function **putchar** writes a single character on the standard output device.

Usage:
```
putchar(variable_name);
```
OR
```
putchar(character);
```

Examples

1.
```
char c=getchar();
putchar(c) ;
```
2.
```
char ans = 'y';
putchar(ans);
```
3.
```
putchar('\ n'); /* positions the cursor to the beginning of the next
                   line. */
```

Character Test and Conversion Functions

The header file ctype.h contains declarations of several functions, which are used to test or convert a character.

Function	Description
isalnum(c)	Returns true if c is an alphanumeric character.
isalpha(c)	Returns true if c is an alphabet.
isdigit(c)	Returns true if c is a digit.
islower(c)	Returns true if c is a lowercase alphabet.
isupper(c)	Returns true if c is an uppercase character.
ispunct(c)	Returns true if c is a punctuation mark.
isspace(c)	Returns true if c is white space characters.
toupper(c)	Converts c to uppercase if it is a lowercase letter otherwise keeps c unchanged.
tolower(c)	Returns c converted to lowercase it if is uppercase and unchanged otherwise.

Example:
```
char ch = 'a';
putchar(toupper(ch));
```
will display A on screen.

Character test functions are used with control structures like if, while, etc. However the following program illustrates how they can be used.

Program: /* Illustrates character input-output, test and conversion functions */

```
#include<stdio.h>
#include<ctype.h>
main()
{
  char ch;
  printf("Enter a character: ");
  ch=getchar() ;
  if(isalpha(ch))
  { printf("It is an alphabet");
    if(isupper(ch))
    { printf("\n It is in uppercase \n");
    putchar(tolower(ch)); /* convert to lowercase */
    }
  else
    { printf("\n It is in lower case \n");
      putchar(toupper(ch));
    }
  }
  else
    printf("\n Not an alphabet");
}
```

Output a

```
Enter a character : *
Not an alphabet
```

Output b

```
Enter a character : b
It is an alphabet
It is in lowercase
B
```

Note: getch( ) and **getche( )** can also be used to read a single character as getchar( ). They are defined in <conio.h>. The difference between the two is that **getche( )** accepts an input character and echoes (i.e., displays) it on screen also whereas **getch( )** does not echo it on screen. **getch( )** can be used to accept passwords.

3.2.2 String Input and Output [gets() & puts()]

Two functions **gets( )** and **puts( )** in the standard input / output library are used for string input and output respectively.

gets( ) accepts a string from stdin (Standard input device). **gets( )** continues reading the string, character by character until the 'Enter' key is pressed. The newline is replaced by a NULL character (\0) at the end of the string. Spaces and tabs are allowed within the string.

Usage: `gets(name_of_string);`

puts( ) outputs a string to the standard output device. It also appends a new-line character at the end.

Usage:
```
puts(name_of_string);
          OR
puts(string literal);
```

Program: /* Illustrates string input-output */

```
#include<stdio.h>
main()
{  char str[80];
   printf("Type a string less than 80 characters : ") ;
   gets(str) ;
   printf("You typed : ");
   puts(str) ;
}
```

Output

```
Type a string less than 80 characters: C is easy!
You typed: C is easy!
```

3.3 FORMATTED CONSOLE I/O OPERATIONS

These functions are used to input data from a standard input unit such as keyboard and get the result on standard output unit such as monitor. As the name suggests, these functions follow a basic fix format.

Formatted Console I/O Operations

i. printf( )
ii. sprintf( )
iii. scanf( )
iv. sscanf( )

3.3.1 Formatted Output (printf)

The putchar( ) and puts( ) functions can be used only with character and string respectively.

A versatile output function is **printf** which can handle any built-in data types and you can specify the format in which the data must be displayed, i.e., printf displays formatted output to the standard output device. It returns the number of characters actually printed.

Syntax: `int printf("control string" , arg1, arg2, ……argn);`

Control string consists of

i. Ordinary characters that are printed on screen as they appear.

ii. Format specifiers or conversion specifiers, which define the output format of each argument.

iii. Escape sequences like \n, \b,\r, etc.

Format Specifier

i. There must be exactly the same number of arguments as there are format specifier in the same order.

ii. Each format specifier begins with a % and ends with a conversion character.

iii. Between the % and the character, there may optionally be,

a. A minus sign for left justification of the argument.

b. A number that specifies minimum field width. If * is given, it implies to take next argument as field width.

c. A period, which separates field width from the precision.

d. A number, specifying precision, i.e., the number of characters to be printed from a string, or the number of digits after the decimal point of a float value, or the minimum numbers of digits for an integer. * means take next argument size.

e. h if integer is to be printed as short, l for long and L for long double.

printf conversion character and meaning

Character	Argument type	Printed As
% c	int or char	Single character
%i,%d	int	Signed decimal integer
%x,%X	int	Unsigned hexadecimal number using a......f or A....F.
%O	int	Unsigned octal number
%f	float, double	Floating point numbers 6 decimal places by default
%e,%E	float, double	Floating point numbers in exponential format
%g,%G	float, double	Uses %e or % f whichever is shorter.
%p	void *	Pointer
% %	no argument	Prints a %
%u	unsigned int	Unsigned decimal number
%s	????????	Prints a string

printf conversion character for qualified data types

Format specifier	Argument type	Output
%ld, li	Long	Decimal long integer
%Lu	Unsigned long	Unsigned long integer
%hd, hi	Short	Decimal short integer
%hu	Unsigned short	Decimal unsigned short
%le, lf, lg	Double	Signed double
%le,lf, lg	Long double	Signed long double
%lo	Long	Octal long integer
%lx	Long	Hexadecimal long

Examples

1. `printf("This is a string");`
2. `printf("   ");`
3. `printf("\n");`
4. `printf("The value of x is %d" , x );`
5. `printf("radius %f, area = %f" , rad, area);`
6. `printf("Hi %d %c %s" , 2, 'U', "Welcome !");`
   outputs Hi 2 U Welcome !
7. The following statements illustrate the output of number 1234 in different formats.

`printf("%d" , 1234);`	1	2	3	4		
`printf("%2d", 1234);`	1	2	3	4		
`printf("%6d", 1234);`			1	2	3	4
`printf("%-6d", 1234);`	1	2	3	4		
`printf("%06d" , 1234);`	0	0	1	2	3	4

8. Displaying a float value in various formats

`printf("%f", 12.3456);`	1	2	•	3	5	4	6	0	0		
`printf("%8.2f", 12.3456);`				1	2	•	3	5			
`printf("%10.2e", 12.3456);`				1	•	2	e	+	0	1	
`printf("%-10.2e", 12.3456);`	1	•	2	e	+	0	1				
`printf ("%E", 12.3456);`	1	•	2	3	4	5	6	E	+	0	1

9. Displaying a string "Learn, Write" with different formats.

%s	L	e	a	r	n	,		W	r	i	t	e			
%10s	L	e	a	r	n	,		W	r	i	t	e			
% .10s	L	e	a	r	n	,		W	r	i	.				
%15 s				L	e	a	r	n	,		W	r	i	t	e
% - 15s	L	e	a	r	n	,		W	r	i	t	e			
%15.10s						L	e	a	r	n	,		W	r	i
% -15.10s	L	e	a	r	n	,		W	r	i					
%*.*s,15,2														L	e

10. `printf("%d", printf("Hello"));` will produce the following output : Hello5
11. What will be the output? Give explanation.

```
void main()
{ printf(5 + " Fascimile");
}
```

1 PU Apr. 2009 – 4 M

Ans

Output is mile.

First 5 characters of the string "Fascimile" are truncated.

3.3.2 Formatted Input (SCANF)

The general purpose input function is scanf. It reads characters from the standard input, interprets them according to the format specifics and stores them in the corresponding arguments. It returns a number equal to the number of fields that were successfully assigned values.

Syntax: `int scanf("control string" ,&var1, &var2, ..........&varn);`

The argument, each of which is an address, specifies the location where the corresponding converted input should be stored.

The control string may contain

i. White space characters.

ii. Conversion specifications which consists of a % sign, an optional suppression character *, an optional number specifying a maximum field width, an optional h (for short int), l (for long int or double), L (for Long double) and a conversion character.

iii. A non-white character which causes scanf to discard the matching character.

scanf Conversion Characters

Character	Data read as
%d	Decimal integer
%c	Single character
%i	Integer (may be in octal with leading 0 or hexadecimal with leading Ox or ox)
%o	Octal integer
%u	Unsigned decimal integer
%s	Character string
%e,f,g	Floating point number
%x	Hexadecimal number
%[....]	Search sets, which are a sequence of characters. Scanf stops reading a string as soon as a character not in the set is encountered. If the first character in the set is a ^, scanf reads all characters till the first matching character from the set is read from the input. Search sets are case sensitive.

Examples

1. `scanf("%f", &radius);`
2. `scanf("%d %f", &roll_num, &marks);`
3. `scanf("%d%s", &age, fname);`
   (an & is not given with fname since fname will be defined as a string and the name of the string denotes its address)
4. `scanf("%u",&n);`
   The value of n can be given upto 65535.
5. `scanf("%[abcdef]", address);`
   This will read the input characters as long as the input characters are in the search set, abcdef.
6. `scanf("%[abc]", address);`
   If the input given is Mumbai, only Mum will be stored in address since b is in the search set.

7. `scanf("%d%[/-] %d%[/-] %d", &date, &separator, &month, &separator, &year);`

   If the date is entered as : 31-12/2000 , the values assigned are:

date	31
separator	-
month	12
separator	/
year	2000

8. `scanf("%d * [/-] %d % * [/-] %d", &date, &month, &year)`

   Here, the suppression character * is used which will skip a / or – (i.e., not assign them to any argument).

9. `printf("%d", scanf("%d%s", &a, str));`

   If the values given are 10 and Hello, the output is 2.

3.3.3 sprintf and sscanf

- **sprintf** is the same as printf except that the output is written into a string rather than displayed on the output device. The return value is equal to the number of characters actually placed into the array.

  The string is terminated with '\0' and it must be long enough to hold the result.

  Prototype: `int sprintf(char * buf, char * format, arg_list)`

  Example:
```
char s[80];
sprintf(str, "%s %d %f", "Hello", 2, 5.0);
```

  This will result in the data. Hello 2 5.0 to be put into the string str.

- **sscanf** is identical to scanf except that data is read from the string pointed to by buf rather than stdin. The return value is equal to the number of fields that were actually assigned values.

  Prototype: `int sscanf(char *buf,char *format,arg_list)`

  Example:
```
char s[20];
int n1 , n2;
sscanf("hello 10  20", "%s %d %d", s, &n1, &n2);
```

  This assigns hello to string s and 10 and 20 to n1 and n2 respectively.

SOLVED PROGRAMS

1. Write a program to accept temperature in °C and convert it to °Fahrenheit using formula temp- in- °F = $\frac{9}{5}$ * temp-in- °C +32.

/*This program converts temperature in degree Centigrade to Fahrenheit*/

```
#include<stdio.h>
main()
{
 float centigrade, fahr,; /* declarations*/
 printf("Enter the temperature in Centigrade :");
 scanf("%f", &Centigrade); /* accept input */;
 fahr = (9.0/5) * centigrade + 32;
 printf("\n temperature in Centigrade = %f", centigrade);
 printf("\n temperature in Fahrenheit %f", fahr);
}
```

Output

```
Enter the temperature in centigrade: 37
Temperature in centigrade = 37
Temperature in Fahrenheit = 98.599998
```

2. Write a program to calculate the distance between two points, using formula.

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

/* To calculate the distance between two points whose coordinates are (x1,y1) and (x2,y2) */

```
#include<stdio.h>
#include<math.h>
main()
{ int x1, x2, y1, y2;
 float d;
 printf("Enter the coordinates of the first point:");
    scanf("%d%d",&x1,&y1);
    printf("Enter the coordinates of the second point:");
    scanf("%d%d",&x2,&y2);
    d = sqrt((y2-y1)*(y2-y1)+(x2-x1)*(x2-x1));
    printf("The distance is %lf",d);
}
```

Output

```
Enter the coordinates of the first point:10 0
Enter the coordinates of the second point: 0 10
The distance is 14.142136
```

3. Program to convert time in seconds to equivalent hours, minutes, and seconds.

/* This program converts seconds to hours, minutes and seconds*/

```
#include<stdio.h>
/* Define constants */
#define SECONDS_PER_MIN  60
#define MINS_PER_HOUR  60
main()
(unsigned int seconds, minutes, hours, seconds_left, mins_left;
printf("Enter the number of seconds :");
scanf("%u", &seconds);
hours = seconds / (SECONDS_PER_MIN * MINS_PER_HOUR);
minutes = seconds / SECONDS_PER_MIN;
mins_left = minutes % SECONDS_PER_MIN;
seconds_left = seconds % SECONDS_PER_MIN;
printf("%u Seconds are equivalent to : " seconds);
printf("%u hrs %u mins %u seconds",hours, mins_left, seconds_left);
}
```

Output a

```
Enter the number of seconds : 60
60 seconds are equivalent to : 0 hrs 1 mins 0 seconds
```

Output b

```
Enter the number of seconds : 20000
20000 seconds are equivalent to: 5 hrs 33 mins 20 seconds
```

4. Write a program to accept integer numbers till user enters '0' and display how many non-zero integers are entered.

1 **PU Oct. 2010 – 5 *M***

Ans

```
#include<conio.h>
#include<stdio.h>
void main()
{
  int k=0;
  char c;
  printf("Enter the element is");
  while((c=getchar())!=48)
  {
    if((!isalpha(c))&&(!isspace(c))&&(c!='\n'))/*character is not
alphabet. User can also check whether accepted character is space or not.*/
    {
      k++;
    }
  }
  printf("number of non zero numbers are=%d",k);
}
```

EXERCISES

A. Select appropriate answer

i.
```
main()
{ int i;
    printf("%d", i);
}
```
a. error b. garbage c. 0 d. 32767

ii.
```
main()
    { int i = 10;
    float j = 20;
    printf("%d", sizeof(i+j));
    }
```
a. 2 b. 4 c. 1 d. 30

iii.
```
main()
    { int i,
    i = 0x10 + 010+10;
    printf("%d", i);
    }
```
a. 0 b. error c. 34 d. garbage

iv.
```
main()
    {  char ch = 'ABC';
    printf("%c", ch);
    }
```
a. error b. ABC c. A d. ch

v.
```
main()
    { printf("\nH\ne\nll\ro");
    }
```
a.
```
H
e
ll
o
```
b.
```
H
e
01
```
c. Hello d. O

vi.
```
main()
    { int i = 10, a;
      a = i++ / ++i;
      printf("%d ..%d", a, i);
```
a. 1....12 b. 10....10 c. error d. compiler dependent

vii.
```
main()
    {  int i = 10, j = 20;
       i ^=j ; j^=i; i^=j;
       printf("%d, , %d", i, j);
    }
```
a. 10....10 b. 10....20 c. 20....10 d. 20....20

viii.
```
enum colors{BLACK, BLUE, GREEN};
    main()
    { printf("%d..%d..%d", BLUE, GREEN, BLACK);
    }
```
a. error b. Blue, Green Black c. 0...,1...,2 d. 1...,2...,0

ix. "The stock's value is decreased by 10%"

Which of the following exactly reproduces the above message?

a. `printf("The stock's value decreased by 10 %");`

b. `printf("\"The stock\'s value decreased by %d \ % \.\"\n", 10);`

c. `printf("\"The stock\'s value decreased by % d %%.\ "\n" , 10);`

d. None of the above.

B. Predict the outputs

i.
```
main()
  {   int a = 300 * 300, b;
     b = a/2;
     printf("%d %d", a, b);}
```

ii.
```
main()
  {  char ch = 'A';
     int i = 2 ;
     float f = ++ch+i;}
     printf("%f%d%c", f, ch, ch);
```

iii.
```
main()
  {  int x = 12, y;
     y = x--;
     y - =--x;
     printf("%d%d", x, y);
  }
```

iv.
```
main()
{  int a = 5, b = 10;
   printf("%d\n", a++ + b++ + ++a + ++b);
   a = 5; b = 10;
   printf("% d \n", ++a * ++b);
   a = 5 ; b = 10 ;
   printf("%d\n", a = ++a * ++ b);
}
```

v.
```
main()
   { int Float = 2, pi = 3.14;
     printf("%f%f", Float, pi);
   }
```
vi.
```
main()
   { int i,
     i = 32000 + 1536 + 10 * 0;
     printf("%d", i);
   }
```
vii.
```
main()
   { int x,y,z;
     x = y = z = -1;
     z = ++x && ++y || ++z;
     printf("x = %d, y = %d, z = %d", x,y,z);
   }
```
viii.
```
main()
   { char c = 'z',ch;
     c = c +'a'-'A';
     ch = c -'a'+'A';
     printf("%c",ch);
   }
```
ix.
```
main()
     { int i = 10,5;
       printf("%d",i);
     }
```
x.
```
main()
{ const int x;
  x = 130;
  printf("%d",x);
}
```

C. Programming exercise

i. Find the roots of a quadratic equation using the formula, $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Accept values such that $b^2 > 4ac$.

ii. Accept the basic salary of an employee and calculate and display the following:

Dearness Allowance (DA) = 150% of basic
Income Tax (IT) = 30% of basic
Provident Fund (PF) = 8.33% of basic
Net Salary = Basic + DA – (IT + PF)

iii. Accept two numbers and interchange their values.

iv. Given the three sides of a triangle, calculate its area using the formula $\sqrt{s(s-a)(s-b)(s-c)}$ where a, b and c are the three sides and s is the perimeter.

v. The frequency of an electrical circuit is

$F = \sqrt{\frac{1}{LC} - \frac{R^2}{4C^2}}$ Write a program that accepts Inductance (L), Capacitance (C) and Resistance (R) of the circuit and calculate its frequency.

vi. Write a program to accept a character from the keyboard and check if it is an alphabet, digit or special symbol.

If it is an alphabet, check if it is uppercase or lowercase. If uppercase, convert it to lowercase & vice-versa.

D. Review questions

i. Explain the functions getchar and putchar with examples.

ii. Explain the format specifiers used with the printf functions.

iii. Explain search sets in the scanf function with examples.

iv. Is there a difference between:

```
printf("Hello"); printf("World"); and
puts("Hello"); puts("World");
```

v. What is the difference between getch( ) and getche( ) ?

vi. What format specifiers are used with scanf?

Collection of Questions asked in Previous Exams PU

1. What will be the output? Give explanation. **[Apr. 2009 – 4 M]**

```
void main()
{ printf(5 + "Fascimile");
}
```

2. Write a program to accept integer numbers till user enters '0' and display how many non-zero integers entered. **[Oct. 2010 – 5 M]**

4 Control Statements

4.1 INTRODUCTION

In the previous chapters, we have studied some basic input output functions. We have also seen the different types of C statements. In this chapter, we shall be studying the program control statements, which specify the order in which instructions are executed.

Sometimes, it is necessary to alter the sequence of execution of statements based on certain conditions or we may require some statements to be executed repeatedly until some condition is met. This involves decision-making and looping. In addition we shall also be studying the jump statements, which allow breaking out of decision and loop control statements.

4.2 SELECTION / DECISION MAKING STATEMENTS

Many programs require testing of some conditions at some point in the program and selecting one of the alternative paths depending upon the result of the condition.

C provides three mechanisms to check for conditions and execute or skip certain parts of the program. *The three decision-making statements are:*

i. if statement

ii. if-else statement

iii. switch statement

4.2.1 if Statement

This is the simplest form of decision-making statement in C. It allows decisions to be made by evaluating an expression. Depending upon the result (True or False), the program execution proceeds in one direction or another. Basically it is a two-way decision statement.

The simplest form is:

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

Note: Here, statement could be either a single statement or a block of statements (enclosed in braces) as shown below. Henceforth, we shall use **Statement** to imply both

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

for single statement.

```
if(expression)
{.....
    statements;
... ..
}
```

for more than one statement.

The keyword **if** must be followed by a set of parenthesis containing a **single expression** to be tested. The statement is executed only if the expression is true (i.e., non-zero). If the condition evaluates to false, the statement is skipped.

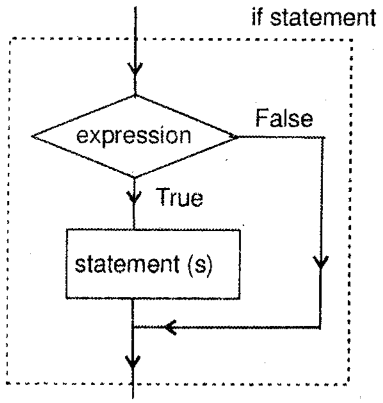


Figure 4.1

Example

```
1    if(n<0)
       printf("The number is negative");
```

2.
```
if(age<30 && salary>10000)
   printf("You are young and rich !!");
```

3.
```
if((n%3 == 0) && (n%5 == 0))
   printf("The number is divisible by 3 and 5");
```

4.
```
if(basic_sal > 10000)
{
   it = 30.0 * basic_sal / 100;
   da = 200.0 * basic_ al / 100;
   hra = 800.0;
}
```

4.2.2 if .... else Statement

The 'if' statement will execute the statement if the expression is true otherwise it will be skipped. However, in many cases we require an alternate statement to be executed if the expression evaluates to false. This is possible using an if ....else statement. The general form is,

```
if(expression)
   statement1;
else
   statement2;
```

Here, the expression is evaluated. If it is true, **statement1** is executed and if it is false, **statement2** is executed. Thus, either **statement1** or **statement2** will be executed; never both.

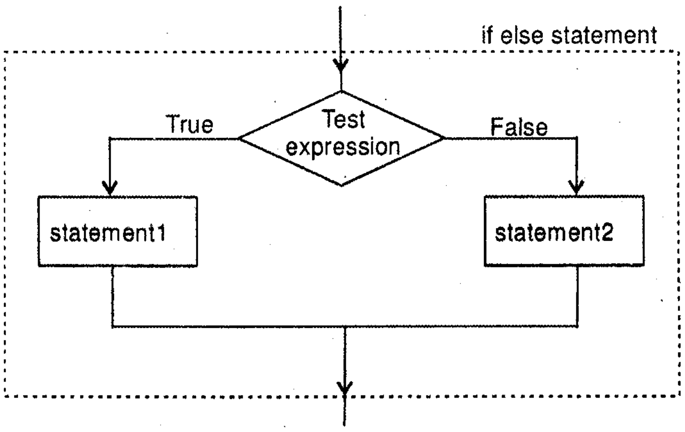


Figure 4.2

Examples

1.
```
if(a>b)
  printf("a is larger");
else
  printf("b is larger");
```

2.
```
if(year%4 ==  0 && year%100 != 0 || year%400 == 0)
  printf("%d is a leap year", year);
else
  printf("%d is not a leap year", year);
```

This can also be written using the conditional operator?

```
(year%4 == 0 && year%100 != 0 || year%400 == 0)?
printf("leap"): printf("Not Leap");
```

3.
```
if(number%2 == 0)
  printf("The number is even");
else
  printf("The number is odd");
```

4.
```
if(basic_sal < 10000)
{
   it = 20 * basic_sal / 100;
   da = 150 * basic_sal / 100;
   hra = 500;
}
else
  {it = 30 * basic_sal/100;
  da = 200 * basic_sal / 100;
  hra = 800;
}
```

Nested if ...else Statement

As seen earlier, the if clause and the else part may contain a compound statement.

Moreover, either or both may contain another if or if ….else statement. This is called as nesting of if ….else statements.

This provides a programmer with a lot of flexibility in programming. Nesting could take one of several forms as illustrated below.

i.
```
if(expression1)
  statement1
else
  if(expression2)
    statement2
```

ii.
```
if(expression1)
  if(expression2)
      statement1
else
      if(expression3)
           statement2
```

iii.
```
if(expression1)
     if(expression2)
          statement1
     else
          statement2
     else
    statement3
```

iv.
```
if(expression1)
      statement1
else
     if(expression2)
        statement2
     else
          statement3
```

v.
```
if(expression1)
  if(expression2)
       statement1
   else
       statement2
else
     if(expression3)
         statement3
     else
         statement4
```

Examples

1.
```
if(a>b)
   if(a>c)
     printf("a is largest");
   else
     printf("c is largest");
 else
   if(b>c)
     printf("b is largest");
   else
     printf("c is largest");
```

2.
```
if((ch >= 'a' && ch <= 'z') || (ch > 'A' && ch <='Z'))
   printf("%c is an alphabet", ch);
else
   if(ch >= '0'&& ch< ='9')
      printf("%c is a digit", ch);
   else
      printf("%c is a special symbol", ch);
```

Note: It is a good idea to enclose each of the 'if' and 'else' blocks in braces if the logic is complex.

Example

A recruitment agency recruits candidates satisfying the following conditions:

i. If the candidate is male, between 25 and 30 years of age, height above 160 cm.

ii. If the candidate is female, between 20 and 25 years of age with height above 155 cm.

The if-else construct for the above can be written as follows:

```
if(sex == 'M')
{
   if(age >= 25 && age <= 30)
      if(height > 160)
         printf("Candidate is recruited");
}
else    /* Candidate is Female */
{
   if(age >= 20 && age <= 25)
      if(height > 155)
         printf("Candidate is recruited");
}
```

Note: else always gets associated with the nearest if statement. Hence { } should be used to associate the else with the correct if.

The else – if ladder

If there is an if else statement nested in each else of an if- else construct, it is called an else – if ladder as depicted below.

```
if(expr1)
     statement1;
else
    if(expr2)
        statement2;
    else
       if(expr3)
          statement3;
       else
          statement4;
```

This can also be written as

```
if(expr1)
      statement1;
else  if(expr2)
       statement2;
else  if(expr3)
      statement3;
else
      statement4;
```

The conditions (expressions) are evaluated from the top downward. As soon as a true expression is found the statement associated with it is executed and the rest of the ladder is bypassed.

If none of the expressions are true, the final else is executed. The last else often acts as a default condition, i.e., if all other tests fail, the last else statement is executed.

If it is not present, no action takes place if all other conditions are false.

Examples

1. **To check whether a character entered from the keyboard is an alphabet, digit, a special symbol or punctuation mark.**

```
if(isalpha(ch) /* ch is the character variable storing the character */
  printf("%C is an alphabet",ch);
  else
    if(isdigit(ch))
      printf("%c is a digit", ch);
else
 if(ispunct(ch))
   printf("%c is a punctuation mark", ch);
 else
   printf("%c is a special symbol", ch);
```

2. **To find class of a student from the marks.**

```
if(marks>=70)
    printf("Distinction");
else if(marks>=60)
    printf("First class");
else if(marks>=50)
    printf("second class");
else if(marks>=40)
    printf("Pass class");
```

4.2.3 The switch Statement

Whenever one of many alternatives is to be selected, nested if - else statements can be used. However, the structure becomes very complicated and the code becomes difficult to read and trace.

For these reasons C has a built-in multiple-branch decision statement called **switch.** This statement tests whether an expression matches one of a number of constant integer values and branches accordingly.

The format is

```
switch(expression)
{
   case const-expr1 : statement;
   case const-expr2 : statement;
   case const-expr3 : statement;
                .
                .
                .
                .
                .
   default : statement;
}
```

As mentioned before **statement** implies a single statement or a compound statement.

- The expression enclosed within parenthesis (integer expression) is successively compared against the constant expression (or values) in each case. They are called case labels and must end with a colon (:).
- The statement in each case may contain zero or more statements. If there are multiple statements for a case they need not be enclosed in braces.
- All case expressions must be different.
- The case labeled default is executed if none of the other cases match. The default case is optional and if not included, no action takes place at all if none other match.
- Cases and the default case can occur in any order.
- More than one case value may be associated with a particular statement.

Example: **/* Use of switch statement */**

```
#include<stdio.h>
main()
{ int number ;
  printf("Enter a number between 1 and 3 :");
  scanf("%d",&number);
  switch(number)
  {
     case 1 : puts("You entered 1\n");
    case 2 : puts("You entered 2\n");
    case 3 : puts("You entered 3\n");
    default : puts("Out of range\n");
  }
}
```

Output

```
Enter a number between 1 and 3:2
You entered 2
You entered 3
Out of range
```

However, this is not the required output. The output is like this because when a match occurs, not only the statement associated with the matching case is executed but those of all the remaining cases are also executed. Using a break statement can solve this problem.

Use of break Statement

The break statement is used to exit a control structure. As soon as a break statement is encountered, program control is transferred to the first statement outside the structure to which the break belongs.

In the above program, if a break statement is included in every case, as soon as a match is found, the statement(s) of the matching case will be executed and the break statement will take control outside the switch statement as illustrated below.

The default case need not have a break statement since it will be the last case executed if no others match.

Example: **Illustration of switch using break.**

```
#include<stdio.h>
main()
{int number;
 printf("Enter any number between 1 and 3 :");
  scanf("%d", &number);
  switch(number)
  {
        case 1 : puts("You entered 1\n");
                 break;
        case 2 : puts("You entered 2\n");
                 break;
        case 3 : puts("You entered 3\n");
                 break;
        default : puts("Out of range.\n");
   }
}
```

Output a

```
Enter any number between 1 and 3:2
You entered 2
```

Output b

```
Enter any number between 1 and 3:10
Out of range.
```

Note: To associate more than one case value with a particular statement, you have to simply list the multiple case values before the common statement (s) that are to be executed. This is called-falling through cases.

Examples

i.
```
switch(operator)
{
  ...
  ...
  case '*' :
  case 'X' : result = value1 * value2;
             printf("%f", result);
             break;
  ...
  ...}
```

ii.
```
switch(c)
{ case '0' : case '1' use: case '2' : case '3':
case '4' : case '5' : case '6' : case '7':
case '8' : case '9':  digit++; break;
case ' ' : case \n : case '\t' : white_space++; break;}
```

Nested 'switch' Statement

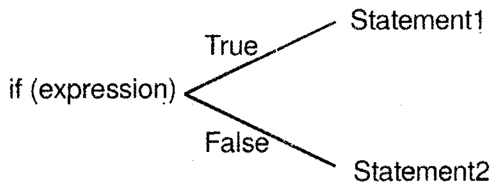
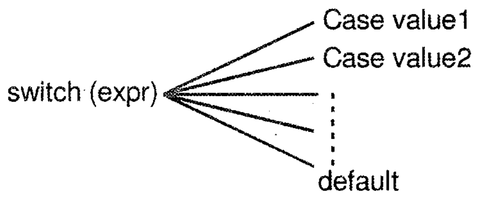
It is possible to have a switch statement as a part of a statement in another switch statement. Even if the case constants of the inner and outer switch contain common values there is no conflict.

Example

```
switch(x)
{ case 0 : printf("Invalid value");
           break;
  case 1 : switch (y)
          { case 0 :  printf("values are 1 and 0");
                      break;
            case 1 :  printf("values are 1 and 1");
                      break; }
          break;
  case 3 :
          :
          : }
```

Comparing if-else and switch Statements

Although both these statements can be used for multi-way decision-making, there are some differences between the two, which are crucial for the selection of one of these in a program.

	If-else structure	Switch statement
i.	The if-else structure allows only two-way branching from a single expression.  if (expression) True Statement1 False Statement2	Switch allows multi-way branching from a single expression. 
ii.	The nested if-else structure is non-elegant and complicated.	Switch statement is very elegant and easier to write.
iii.	If multiple alternatives exist, the nesting can go to many levels and it becomes difficult to match the else part to its corresponding if.	No such problem occurs using a switch statement.
iv.	Debugging becomes difficult.	Tracing of errors and debugging is easy.
v.	The test expression can be a constant expression or an expression involving relational or logical operators. Float and double are also allowed.	Only constant integer expressions and values are allowed.
vi.	Multiple statements within if or else have to be enclosed in braces.	The statements belonging to a case need not be enclosed in braces.

4.2.4 Conditional Operators

The ternary operator?: can also be used for decision-making. We have already seen how this operator works. The general form is

```
expr1?  expr2: expr3;
```

If expr1 is true, the entire expression takes the value of expr2 else it takes the value of expr3.

Examples

i.
```
char ch;
ch = getchar();
x = (ch >= 65 && ch <= 90)? 1: 0;
x? puts("Uppercase alphabet"):puts("Other character");
```

This piece of code checks if character ch is an uppercase alphabet.

ii. The following statement assigns the largest of three numbers (a,b,c) to x.

```
x = (a > b)? (a > c)? a : c  : (b > c)? b : c;
```

4.3 ITERATIVE STATEMENTS (LOOP CONTROL STRUCTURE)

A segment of program code that is executed repeatedly is called a loop. The repetition is done until some condition for termination of the loop is satisfied.

A loop structure essentially contains

i. a test condition

ii. loop statement(s)

The test condition determines the number of times the loop body is executed. It involves evaluating a loop control variable(s), whose value has to change within the loop body so that the loop execution can terminate.

The iteration procedure takes place in four steps:

a. Initializing the loop control variable.

b. Execution of loop statements.

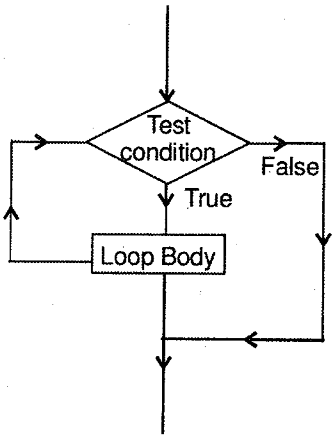
c. Changing the value of the control variable.

d. Testing the condition.

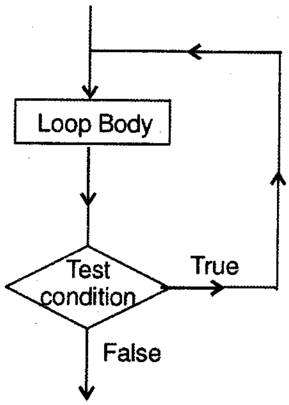
Depending upon when the loop condition is tested, loops can be of two types:

1. Top-tested loop (entry controlled loop)
2. Bottom tested loop (exit controlled loop)

In an entry-controlled loop, the condition is evaluated before the loop body is executed. In the bottom tested or exit controlled loop, the condition is tested after the loop body is executed.



Top-Tested or Entry controlled loop



Bottom Tested or Exit controlled loop

Figure 4.3

The C language provides three loop structures for use in programs.

i. while statement

ii. do…while statement

iii. for statement

4.3.1 The while Statement

The while loop is the simplest loop structure. It is often used when the number of times the loop is to be executed is not known in advance but depends on the test condition.

It is an entry-controlled loop, i.e., the condition is tested before the loop body is executed.

The **syntax** of the loop is:

```
while(expression)
      statement;
```

The expression is the test condition and can be any valid C expression.

The statement can be a single or compound statement.

How it Works?

- The expression is evaluated and the statement (loop body) is executed as long as the expression is TRUE (non-zero).
- As soon as the expression evaluates to false, the execution of the loop body is stopped and control is transferred to the first statement outside the loop body.
- Since it is an entry-controlled loop, if the expression evaluates to false the first time itself, the loop body will not be executed even once.

Examples

1. **Program displaying all even numbers below 50.**

   /* Demonstration of a simple while loop */

```
#include<stdio.h>
main()
{ int even_number = 0 ; /* Initialization */
 while(even_number < 50) /* Loop condition */
 {
    printf("%d \n", even_number);/*Display */
    even_number = even_number+2;/*change value of loop variable */
 }
}
```

Points to Remember

- The loop control variable(s) must be initialized (i.e., given some value before the condition is tested).
- The loop body must contain a statement to alter the value of the control variable.

2. Calculate the sum of numbers from 1 to n (user specified), i.e., 1+2+3+..........+n

*** Illustrates while loop */**

```
#include<stdio.h>
main()
{int sum = 0, n, loop_var =1; /* Initialization*/
 printf("enter the value of n : ");
 scanf("%d",&n);
  while(loop_var < = n)
  {
   sum = sum + loop_var;
   loop_var++;
  }
 printf("\n The sum of numbers from 1 to %d is %d", n, sum);
}
```

3. To accept characters from the keyboard till the user enters * and count the total number of alphabets entered.

```
#include<stdio.h>
main()
{ char ch;
  int counter = 0;
  ch = getchar(); /* Get the first character */
  while(ch !='*')
  { if(isalpha(ch)) /* check if ch is an alphabet */
    counter++ ;
    ch = getchar(); /* alter value of loop variable */
  }
  printf("Number of alphabets are %d",counter);
}
```

The loop can be written in another way as shown:

```
while((ch= getchar()) !='*')
{
 if(isalpha(ch))
     counter++;
}
```

Here, ch = getchar( ) is enclosed in ( ) because != has higher precedence over =. The character has to be read first and then compared. Hence the ( ).

4. To reverse a number

/* Program to reverse a number, i.e., if user enters 324, the output should be 423 */

```
#include<stdio.h>
main()
{ int num , rev_num = 0;
  printf("Enter the number to be reversed");
  scanf("%d", &num);
  while(num>0)
  {
   rev_num = rev_num * 10 + (num % 10);
   num = num /10;
  }
   printf("In The reversed number is %d" rev_num);
}
```

Output

```
Enter the number to be reversed 5678
The reversed number is 8765.
```

Nested 'while' statement

Just like the 'if' statement, while statements can also be nested. Nesting of loops means a loop that is contained within another loop.

```
while(expr1)
{
  while(expr2)
  {
  loop body of while(expr2);
  }
}
```

Nesting can be done upto any levels. However the inner loop has to be completely enclosed in the outer loop. No overlapping of loops is allowed.

Nesting of loops is required in many programming exercise like multidimensional arrays etc.

Examples

1. To display the following structure

```
1
1 2
1 2 3
1 2 3 4
```

i.e., 1 to n rows and numbers from 1 to n in the nth row.

/* program to display triangle of numbers */

```
#include<stdio.h>
main()
{ int n , line_number , number;
  printf("How many lines:");
  scanf("%d",&n);
  line_number = 1 ; /* Initialize line number */
  while(line_number<=n) /* line number goes from 1 to n */
  { number = 1 ; /* display begins from 1 */
    while(number<=line_number)
    {  printf("%d\t", number);
       number++; /* next number */
    }
    printf("\n");
    line=number++ ; /* next line*/
  }
}
```

In the above program, the outer while loop is for the lines from 1 to n. For each line, we have to print numbers from 1 to the line numbers. This is done by the inner loop, i.e., for every value of line-numbers, number takes values from 1 to line_number.

1
PU
Oct. 2009 – *10 M*

2. Write a program which accept a string and count number of lines, characters, spaces, numbers and special characters in a string.

```
#include<stdio.h>
#include<conio.h>
#define MAX 80
void main()
{
  char str[MAX];
  int i, ln, sp, num, spch = 0;
  clrscr();
  printf("\n Enter the String for the Keyboard");
  gets(str);
  while(str[i] != '\0')
  {
    if(str[i] == '\n')
      ln++;
    if(str[i]  == ' ')
      sp++;
    if((str[i]==0)||(str[i]==1)||(str[i]==2)
                  ||(str[i]==3)||(str[i]==4)||(str[i]==5)
                  ||(str[i]==6)||(str[i]==7)||(str[i]==8)
||(str[i]==9)
      num++;
    if((str[i]=='.')||(str[i]=='_')||(str[i]=='@')||(str[i]
=='$')||(str[i]=='@'))
       spch++;
  }
  printf("\n The Total Number of Lines=%d",ln);
  printf("\n The Total Number of Characters=%d",sp);
  printf("\n The Total Number of Numbers=%d",num);
  printf("\n The Total Number of Special Characters=%d",spch);
  getch();
}
```

4.3.2 The do-while Loop

The second iteration statement provided by C is the **do-while** statement.

The while loop seen earlier is top-tested, i.e., it evaluates the condition before executing any of the statements in its body. The do-while loop, on the other hand, is a bottom-tested or exit controlled loop, i.e., it evaluates the condition after the execution of statements in its construct. This means that the statement within the loop are executed at-least once.

The **syntax** is

```
do
  { statement}
    while(expression);
```

The statement (single or compound) is executed as long as the expression is true.

Note: The ; following the while.

The sequence of events is:

i. The statement(s) in statement are executed.

ii. Expression is evaluated. If it is true, execution returns to step 1. If it is false, execution of the loop terminates.

Example

```
do
{ printf("\n 1 - Add a record");
  printf("\n 2 - Delete a record");
  printf("\n 3 - View Records");
  printf("\n 4 - Quit");
  printf("\n Enter your choice:");
  scanf("%d" &choice);
  switch(choice)
  {
  case 1 : add();
           break;
```

```
   case 2 : delete();
            break ;
   case 3 : view();
            break;
   case 4 : printf("Bye");
   }
}while(choice!=4);
```

The above program code shows a do while loop, which displays a menu and accepts a choice.

In this case, we want the menu to be displayed and choice to be accepted at least once and so a do_while loop is preferred.

4.3.3 The for Loop

The for loop is very flexible, powerful and most commonly used loop in C. It is useful when the number of repetitions is known in advance.

This is a top-tested loop similar to the while loop but the advantage is that it combines the initialization test condition and loop variable alteration statement in a single statement. The **syntax** is:

```
for(expr1; expr2; expr3)
         statement
```

where, expr1 is the initialization expression

expr2 is the test condition

expr3 is the update expression

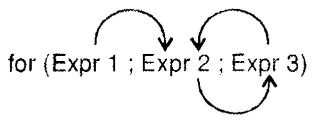
These three expressions have to be separated by semicolon (;).

The above loop is equivalent to

```
expr1;
while(expr2)
{ statement;
  expr3;
}
```

Execution of a for Loop

- expr1 is evaluated only once, i.e., at the beginning. This expression performs initialization of the loop control variable (Multiple initializations can also be done as seen later).
- expr2 is the test expression, which is evaluated before execution of statements in the loop. The statements are executed only if the test expression is true. If it is false, the loop execution terminates. Note that there can be only a **single** test expression.
- expr3 is the update expression, which alters the value of the loop control variable.



Execution of for

Example

```
for(i=1; i<=100; i++)
        printf("%d \n" ,i);
```

i = 1 → initialization

i <= 100 → test expression

i++ → update expression

Different Forms of the 'for' Loop

i.
```
for(i= 0; i < 25; i++)
  statement;                  → single statement
```

ii.
```
for(i = 0; i < 25; i++)
{ statement;
  .......
  statement;                  → compound statement
}
```

iii.
```
for(i = 0; i < 25; i++)
;
or
for(i = 0; i < 25; i++);  → loop with no body.
```

iv.
```
for(i = 0, j = 0; i < 25; i++, j++)
  statement                   → Multiple initialization and multiple
                                updates separated by comma
```

v.
```
for(; i < 25, i++)            → Initialization expression not used.
```

vi.
```
for(; i < 25;)                → Initialization and update expression
                                not used
```

vii.
```
for(;;)                       → All three not used.
  printf("Forever \n");
```

Examples

1.
```
for(i=1,j=50;i<=20||j>=10;i++ j--)
  printf("\n %d %d"i,j);
```

2.
```
for(temp=0;temp<=50;temp=temp+5)
{
      fahr = (9*temp) /5 + 32);
      printf("\n centigrade = %f Fahrenheit = %f",temp, fahr);
}
```

3.
```
/* Accepts values from user till 99 is entered */
int num = 0;
for(;num!=99;)
    scanf("%d",&num);
```

4.
```
for(i = 0; ++i<10;)
  printf("%d \n",i)
```

Example

1. **Calculation of factorial of a number. We know that n! = n × (n-1) × (n-2) ......×1. Thus we have to repeatedly decrement n by 1 till 1 and multiply each value to the previous product.**

   Note: We can also increment from 1 to n and perform multiplication.

   /* Calculation of factorial */

```
#include<stdio.h>
main()
{ int num, product ;
  printf("Enter the number:");
  scanf("%d",&num);
  for(product = 1; num >= 1; num--)
       product = product * num;
  printf("\n the factorial is %d",product);
}
```

Output

```
Enter the number : 5
The factorial is 120
```

Note: The for loop could also have been written as :

```
for ( i= 1, product = 1; i < = num ; i++)
    product = product * i ;
```

2. **To calculate x^y where x is a float and y is an integer.**

   /* Calculation of x^y */

```
#include<stdio.h>
main()
{
   float x, power = 1, i;
```

```
  int y;
  printf("Enter the base and power :")
  scanf("%f %d",&x, &y);
  for(i=1;i<=y;i++)
       power *= x;
  printf("\n %f raised to %d is %f",x,y,power);
}
```

Output a

```
Enter the base and power : 2 3
2.000000 raised to 3 is 8.000000
```

Output b

```
Enter the base and power : 2.5  2
2.500000 raised to 2 is 6.250000
```

Nesting for Statements

One for statement can be written within another for statement. This is called nesting of for statements as illustrated below.

```
for(i=1;i=25;i++)                        ┐
{                                        │
...                                      │
 for(j=1;j<=10;j++)┐                     │
 {                 │ Inner for loop      │Outer loop
 }                 ┘                     │
...                                      │
}                                        ┘
```

Here, for every of i, the inner loop will be executed ten times.

Note: We had earlier written a program to display a triangle of numbers using the while loop. Another triangle is now illustrated using a for loop. The following triangle is called the "Floyd's triangle".

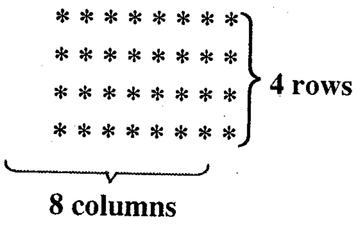
```
1
23
456
78910
```

Program: /* To draw a Floyds triangle using nested for loops */

```
#include<stdio.h>
main()
{
  int n,line_number,number,count;
```

```
 printf("How many lines?");
 scanf("%d",&n);
 number = 1;
 for(line_number=1; line_number<=n, line_number++)
 {
 for(count=1;count<=line_number;count++)
 printf("%d\t",number++);
 printf("\n");
 }
}
```

2. To display a rectangle of n rows and m columns filled with the character '*'.



```
#include<stdio.h>
main()
{ int n_rows, mcols i,j,;
  printf("Enter the number of rows:");
  scanf("%d",&nrows);
  printf("\n Enter the number of columns:");
  scanf("%d",&mcols);
  for(i=1;i<=nrows;i++)
  { for(j =1;j<=mcols;j++)
          printf("*");
   printf("\n");     /* Go to the next line after each row */
  }
}
```

3. To display multiplication tables 2 to 9 (n multiples each). The required display is:

$2 \times 1 = 2$ $3 \times 1 = 3$ ....$9 \times 1 = 9$

$2 \times 2 = 4$ $3 \times 2 = 6$.............$9 \times 2 = 18$

If the multiples do not fit on a single screen, display each screen after a pause (about 24 multiples will fit on a screen)

/* Multiplication Tables */

```
#include<stdio.h>
main()
{ int table_of, multiplier, n, ,count = 1;
  printf("\n How many multiples ? : ");
  scanf("%d",&n);
  for(multiplier=1, multiplier<=n; multiplier++, count++)
  {
    for(table_of=2; table_of<=9; table_of++)
    printf(%2dx%2d=%3d\t", table_of,multiplier, table of * multiplier);
    printf("\n");
    if(count %24==0)    /* Screen full */
    { printf("Press any key to continue...");
      getch(); clrscr();
    }
  }
}
```

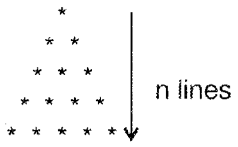
This program, for each value of multiplier, table_of varies from 2 to 9 thereby giving each row.

4. Write a program to accept a positive integer and find the factors. e.g. 8 = 2×2×2

1 PU Oct. 2008 – *5 M*

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c;
    clrscr();
  printf("\n Enter the Integer Number");
  scanf("%d", &a);
  for(int i = 0; i<= a; i++)
{
      c = a % i;
      if(c== 0)
      {
        printf("%d", i);
      }
  }
  getch();
}
```

5. To display 'n' lines of the structure from the center of the first line on screen.



/* Triangle using the * character */

```
#include<stdio.h>
main()
{
  int spaces = 39, n, no_of_stars, line_no, s;
  printf("Enter the number of lines:");
  scanf("%d \n",&n);
  for(line_no = 1; line_no<=n; line_no++)
  {
    for(s=1;s<=spaces; s++)
      printf(" ");          /* display spaces*/
    for(no_of_stars = 1; no_of_stars<=line_no; no_of_stars++)
      printf("*");
    printf("\n");
  spaces--;   /* reduce number of spaces by 1*/
  }
}
```

Note: Instead of using a loop to display spaces, we can use a single printf statement as:

`printf("%*S", spaces, " ");`

6. Write a program to print the following pattern:

PU
Oct. 2009 – *6 M*

```
A B C D   C   B   A
A B C      C  B    A
A B           B   A
A                A
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char ch;
  int n, l, count, spaces = 39;
  clrscr();
  for(l=1;l<=4;l++)
    {
      printf("%*/s, ", spaces, "     ");
      spaces = spaces-2;
      for(count =1, ch ='A'; count <= l; count++, ch ++)
      printf("%c", ch);
      ch = ch - 2;
      for(count = 1; count <= l-1; count ++, ch--)
      printf("%c", ch);
      printf("\n");
    }
  getch();
}
```

4.4 JUMP STATEMENTS

4.4.1 Break and Continue

We have already seen the use of the break statement in the switch-case statement. It also has one more use.

Sometimes, it is required to exit a loop as soon as a certain condition is met, i.e., to force immediate termination of a loop bypassing the normal loop condition test.

When the break statement is encountered inside a loop, the loop is immediately terminated.

Subsequent statements in the loop are skipped and program control resumes at the next statement following the loop.

Break Statement

Format: `break;`

Examples

1. **The following program checks whether a number is prime or not. To check a prime number, we successively divide it by 2 to number –1. If it is divisible the number is not prime. Thus, as soon as we get a 0 remainder, we have to break out of the loop.**

```
#include<stdio.h>
main()
{ int number, i, prime = 1;
  printf("enter the number:");
  scanf("%d", &number);
  for(i=2;i<number; i++)
  {
    if(number%i == 0)
      { prime = 0;
        break;
      }
  }
  if(prime==0)
    printf("\n The number is not prime");
  else
    printf("\n The number is prime");
  }
```

Note: If there are nested loops, the break statement will cause exit only from the innermost loop.

2.
```
count =1;
for(i=1;i<=5;,i++)
{ for(j=1;j<=5;j++)
   {
      printf("Enter a number:");
      scanf("%d",&n);
      if(n<0)
         break;
    }
   count++;
}
```

Here, if the user enters a negative number, the block statement will take control to the statement count++, in the outer loop.

Continue Statement

The continue statement is somewhat similar to the break statement except that it does not cause the loop to terminate. It bypasses the remaining statements and it forces the next iteration of the loop to take place as usual.

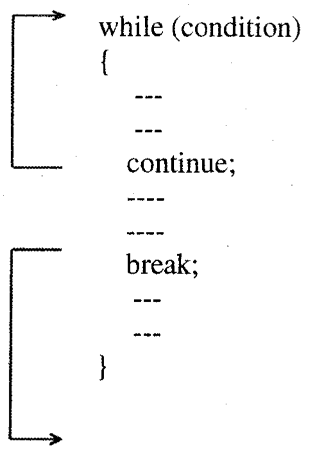
Format: `continue;`

Example

```
do
{  printf("Enter a number :");
   scanf("%d",&n);
   if(n<0)
      continue;
   sum = sum + n;
} while(n! = 999);
```

This code accepts integers and calculates the sum of only positive numbers. The loop terminates after the user enters 999.

In the case of for loop, first the increment part of the loop is performed, next the condition is tested and finally the loop continues.



```
while (condition)
{
   ---
   ---
   continue;
   ----
   ----
   break;
   ---
   ---
}
```

Examples

1.
```
int i=5;
while(i)
{ i--
  if(i == 3)
     break;
   printf("%d",i);
 }
```
o/p 4

2.
```
int i=5,
while(i)
{ i-- ;
  if(i== 3)
     continue;
printf("%d",i);
}
```
o/p 4210

4.4.2 goto and label

The goto statement is an unconditional jump statement. The goto statement (although not used frequently) is used to alter the normal sequence of program execution by unconditionally transferring control to some other part of the program.

Format: `goto label;`

The statement where control has to be transferred is identified by the label.

i. A label is a valid C identifier.
ii. A label is followed by a colon.
iii. The label can be attached to any statement in the same function as the goto.
iv. The label does not have to be declared like other identifiers.

Example

```
X=1;
loop:
  X++;
    if(X<100)
      goto loop;
```

One good use for the goto statement is to come out of several layers of nesting.

Example

```
for(...)
  { for(...)
```

```
    { while(...)
      { ...
        if(error)
            goto out;
        ...
      }
    }
  }
out:
... ...
```

Note: Control cannot be transferred from outside to within a loop using the goto statement.

4.4.3 Using exit( ) Function

The exit ( ) function causes immediate termination of the entire program.

The exit ( ) function is called with an argument 0 to indicate that termination is normal. Other arguments are used to indicate some sort of error.

A common use of exit ( ) occurs when some mandatory condition for program execution is not satisfied. Invalid password entered, absence of color graphics card for running computer games, negative or invalid input entered, etc.

Example

```
main()
{
  int code;
  printf("Enter the security code:");
  scanf("%d",&code);
  if(!valid(code))
      exit(0);
  ...
  ...
  ...
}
```

In this example, a user-defined function valid (code) accepts the code and validates it. If invalid, it returns 0 and 1 if valid. If the code is not valid, the program execution is terminated.

Another use could be in the switch case statement as shown to stop program execution if user enters 4.

```
do
{ ch = getchar();
  switch(ch)
  { case '1'  : add_record();
                break;
    case '2'  : delete_record();
```

```
                    break;
      case '3'  :   view_records();
                    break;
      case '4'  :   exit(0)
    }
} while (ch!='4');
```

4.5 COMPOUND STATEMENT

A compound statement (also called a "block") typically appears as the body of another statement, such as the **if** statement. Declarations and Types describe the form and meaning of the declarations that can appear at the head of a compound statement.

Syntax

```
compound-statement:
  { declaration-list opt statement-list opt }
  declaration-list:
       declaration
    declaration-list declaration
  statement-list:
    statement
    statement-list statement
```

If there are declarations, they must come before any statements. The scope of each identifier declared at the beginning of a compound statement extends from its declaration point to the end of the block. It is visible throughout the block unless a declaration of the same identifier exists in an inner block.

Identifiers in a compound statement are presumed **auto** unless explicitly declared otherwise with **register**, **static**, or **extern**, except functions, which can only be **extern**. You can leave off the **extern** specifier in function declarations and the function will still be **extern**.

Storage is not allocated and initialization is not permitted if a variable or function is declared in a compound statement with storage class **extern**. The declaration refers to an external variable or function defined elsewhere.

Variables declared in a block with the **auto** or **register** keyword are reallocated and, if necessary, initialized each time the compound statement is entered. These variables are not defined after the compound statement is exited. If a variable declared inside a block has the **static** attribute, the variable is initialized when program execution begins and keeps its value throughout the program.

This example illustrates a compound statement:

```
if(i > 0)
{
    line[i] = x;
    x++;
    i--;
}
```

In this example, if `i` is greater than 0, all statements inside the compound statement are executed in order.

4.6 NULL STATEMENT

A "null statement" is a statement containing only a semicolon; it can appear wherever a statement is expected. Nothing happens when a null statement is executed. The correct way to code a null statement is:

```
;
if(1);
for(i=0;i<10;i++);
while(i++<10)
{
}
```

Note: Statements such as **do**, **for**, **if**, and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body.

As with any other C statement, you can include a label before a null statement. To label an item that is not a statement, such as the closing brace of a compound statement, you can label a null statement and insert it immediately before the item to get the same effect.

This example illustrates the null statement:

```
for(i = 0; i < 10; line[i++] = 0)
     ;
```

In this example, the loop expression of the **for** statement line[i++] = 0 initializes the first 10 elements of line to 0. The statement body is a null statement, since no further statements are necessary.

SOLVED PROGRAMS

PU
Apr. 2010 – *5 M*
1

1. **Write a C program to display the following pattern:**

```
         A
      b     c
   D     E     F
g     h     i     j
```

```
#include<conio.h>
#include<stdio.h>
main()
{
  int a=65;
  int k,j,n,i,c;
  printf("\nEnter how many lines:");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    c=0;
    for(j=n-1;j>=i;j--)
    {
      printf(" ");
    }
      for(k=0;k<=i;k++)
      {
        if(c==0)
        {
                if(i%2==0)
                {
                  if(i!=0)
                  {
                    a=a-32;
                    c=1;
                  }
                }
           else
           {
             a=a+32;
             c=1;
           }
       }
    printf("%c ",a);
    a=a+1;
   }
     printf("\n");
   }
getch();
}
```

2. /* First n prime numbers, use of nested loops */

```
#include<stdio.h>
#define PRIME 1
#define NOTPRIME 0
main()
{ int n, divisor, flag = PRIME, number,count =1;
  printf("\n How many prime numbers ? :");
  scanf("%d",&n);
  printf("\n The first %d prime numbers are : \n");
  printf("2\t");
  number = 3;
  while(count<=n)
  {  /* check if number is prime */
    for(divisor =2; divisor<=n-1;divisor++)
    { if(n% divisor == 0)
        { flag = NOTPRIME;
          break;
        }
    }
    if(flag == PRIME) /* if number is prime */
    { count ++ ; printf("%d \t", number) };
        flag = PRIME; /* reset flag */
        number++; /* check if next number is prime */
  } /* end of while */
}/* end of main */
```

Output

```
How many prime numbers? : 5
The first 5 prime numbers are : 2  3  5  7  11
```

3. Write a program to compute the real roots of quadratic equation $px^2 + qx + r = 0$. The roots are given by equation
$x_1 = -q + \text{sqrt}(q^2 - 4pr)/2p$ and
$x_2 = -q - \text{sqrt}(q^2 - 4pr)/2p$.

1 PU Oct. 2010 – *5 M*

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<process.h>
void main()
{
  float p,q,r,x1,x2,disc;
  clrscr();
  printf("Enter the co-efficients\n");
  scanf("%f%f%f",&p,&q,&r);
  disc=q*q-4*p*r;              /*to find discriminant*/
  if(disc>0)                 /*distinct roots*/
  {
    x1=(-q+sqrt(disc))/(2*p);
    x2=(-q-sqrt(disc))/(2*p);
    printf("The roots are distinct\n");
    exit(0);
  }
  if(disc==0)                /*Equal roots*/
  {
    x1=x2=-q/(2*p);
    printf("The roots are equal\n");
    printf("x1=%f\nx2=%f\n",x1,x2);
    exit(0);
  }
  x1=-q/(2*p);               /*complex roots*/
  x2=sqrt(fabs(disc))/(2*p);
  printf("The roots are complex\n");
  printf("The first root=%f+i%f\n",x1,x2);
  printf("The second root=%f-i%f\n",x1,x2);
getch();
}
```

4. Write a program to display the following pattern:

```
A1
B2,  C3
D4,  E5,  F6
G7,  H8,  I9,  J10.
```

1 PU Oct. 2010 – *5 M*

```
#include<stdio.h>
#include<conio.h>
main()
{
  charch='A';
  int i,j,n,k=1;
  printf("\n How many number of lines:");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    for(j=0;j<=i;j++)
```

```
        { printf("%c%d\t",ch,k++);
          ch++;
        }
        printf("\n");
    }
}
```

5. What will be the output

i.
```
switch (i)
   {
   case "hello";
   case "goodbye";
      printf("Greetings");
      break;
   case default;
      printf("Boring");
   }
```

PU
Oct. 2009 – *4 M*

Ans

In the program it displays seven syntax errors on the first two case statements. Because the syntax of case is wrong. The case statement take the column first and then expression, but in the program expression is first and then column is mentions.

ii.
```
void main( )   {
   int i;
   for (i = 1; i < 4; i + +)
      switch (i)
      case 1 : printf ("%d", i);
               break;
   {
      case 2 : printf("%d", i);
               break;
      case 3 : printf("%d", i);
               break;
   }
      switch (i)
}
```

PU
Apr. 2010 – *4 M*

Find and explain the output of following program.

Ans

Correct code is

```
int i;
for(i=1;i<4;i++)
switch(i)
{
   case1:printf("%d",i);
   break;//misplaced break
   case2:printf("%d",i);//case outside switch
   break;
   case3:printf("%d",i);
   break;
}
switch(i)
case4:printf("%d",i);
}
```

i = 1	i < 4	switch(i)	i ++
1	1 < 4 (true)	switch(1) so case 1 will be executed 1 will be printed	2
2	2 < 4 (true)	switch(2) so case 2 will be executed 2 will be printed	3
3	3 < 4 (true)	switch(3) so case 3 will be executed 3 will be printed	4
4	4 < 4 (false)	Loop over	

Last switch will be executed i = 4 so 4 will be printed

Output = 1 2 3 4

EXERCISES

A. Predict the output of the following:

i.
```
main()
  { int x = 1;
    switch(x)
    { case 0 :x= 1;
      case 1 :x= 3;
      case 2 :x+= 4;
      case 3 :x = 2;
      default:x+= 2;
    }
  printf("%d" x);
  }
```

ii.
```
main()
{ int x=5,y=50,z=(x+y)*10;
while(x<=5)
     x=y/x;
}
How many times will the loop execute?
```

iii.
```
int l = 4;
switch(l)
{ default : printf("A");
  case 1  : printf("B");
  case 4  : printf("C");
}
```

iv.
```
main()
{ int i,j,k;
for(j=1;j<=4;j++)
if(j*4==12)
  goto there;
else
  printf("here\n");
for(i=1,i<=5 i++)
{ k = ixi;
  there : printf("there\n");
}
}
```

v.
```
main()
{ int c=97;
switch(c);
{ case 'a':
      if(c>3)
      case 'b':
         c=10;
         printf("%d",c);
}
}
```

B. Programming exercises

1. Write a program to display all Armstrong numbers below 1000.
   (An Armstrong is a number whose sum of cubes of digits is the number itself.
   e.g., $153 = 1^3 + 5^3 + 3^3$)
2. Display all perfect numbers below 500.
   (A perfect number is a number, such that the sum of its factors is equal to the number itself. 6 = 1 + 2 + 3)
3. Find the sum of first 'n' terms of the following series
   i. $1+3+5+\ldots.$
   ii. $x + x^3 + x^5 + \ldots.$
   iii. $\frac{1}{1!} + \frac{2}{2!} + \frac{3}{3!} + \ldots.$
   iv. $x - \frac{x^2}{2!} + \frac{x^3}{3!} - \ldots\ldots$
4. Accept two integers a and b and display a*b , a/b and a%b without using *, / and % operators.
5. Accept characters from the keyboard till the user enters EOF. Count the number of uppercase, lowercase alphabets and vowels in the text.
6. Write a program to display digits of an integer separated by tabs
   Example: 1009 → 1 0 0 9
   2000 → 2 0 0 0
7. Accept data from the keyboard and check if it is valid or invalid.
8. Accept lines of text from the user and find the length of the longest line.

C. Review questions

1. What are the different forms of the if statement?
2. Explain the switch-case statement with examples.
3. Differentiate between if-else and switch-case.
4. Explain else-if ladder with an example.
5. How does a do-while loop differ from a while loop?
6. Explain different ways to terminate loop execution.
7. Explain the for loop with examples.
8. Distinguish between break and continue.
9. Write a note on goto and labels.
10. Illustrate the use of the break statement in the switch –case statement.
11. Discuss the working of if-else and switch statement.

Collection of Questions asked in Previous Exams PU

1. Write a program to accept a positive integer and find the factors. e.g. 8= 2×2×2 **[Oct. 2008 – 5 M]**
2. Find and explain the output of following programme: **[Oct. 2009 – 4 M]**

```
switch(i)
  {
  case "hello";
  case "goodbye";
    printf("Greetings");
    break;
  case default;
    printf("Boring");
  }
```

3. Write a program to print the following pattern. **[Oct. 2009 – 6 M]**

```
A B C D   C     B     A
A B C      C   B     A
A B            B    A
A                  A
```

4. Write a program which accept a string and count number of lines, characters, spaces, numbers and special characters in a string. **[Oct. 2009 – 10 M]**
5. Find and explain the output of following programme: **[Apr. 2010 – 4 M]**

```
void main()
{
  int i;
  for(i = 1; i < 4; i + +)
    switch (i)
    case 1 : printf ("%d", i);
             break;
  {
    case 2 : printf("%d", i);
             break;
    case 3 : printf("%d", i);
             break;
  }
    switch (i)
}
```

6. Write a C program to display the following pattern: **[Apr. 2010 – 5 M]**

```
      A
    b   c
  D   E   F
g   h   i   j
```

7. Write a program to compute the real roots of quadratic equation $px^2 + qx + r = 0$. The roots are given by equation
$x_1 = -q + \text{sqrt}(q^2 - 4pr)/2p$ and $x_2 = -q - \text{sqrt}(q^2 - 4pr)/2p$. **[Oct. 2010 – 5 M]**
8. Write a program to display the following pattern:

```
A1
B2, C3
D4, E5, F6
G7, H8, I9, J10.
```

5 Array And String

5.1 INTRODUCTION

So far we have used variables to store a single data item in memory. However in many applications we need to store a large amount of data. Thus, we would have to declare and use a large number of variables, which is very inconvenient.

Moreover, these variables are independent and unrelated to each other. Many applications require multiple data items to be grouped so that it becomes easy to manipulate them. This can be done using an array.

Definition

An array is a collection of data items of the same data type referred to by a common name. Individual data items can be accessed by the name of the array and an integer called the 'index' or 'subscript'. These items occupy contiguous or consecutive memory locations.

An array is also called a "subscripted variable".

Single and Multidimensional Arrays

An array having only a single subscript is referred to as a single subscripted, linear or one-dimensional array.

An array whose elements are specified by two subscripts is a two-dimensional array (also called a matrix).

Conceptually, an array can have any number of dimensions, limited only by the available memory.

5.2 ARRAY DECLARATION

An array has to be declared before it is used in C program. The declaration tells the compiler,

i. the type of the array,

ii. the name of the array,

iii. the number of dimensions,

iv. number of elements in each dimension.

Syntax: `data_type array_name[size1][size2]............[sizen];`

- data_type specifies the data type of each element of the array.
- array_name is a valid C identifier.
- [size1]..........[sizen] are the n dimensions of the array. size1.....sizen are positive integers indicating the maximum number of elements in each dimensions.

Note: For each dimension, an array subscript begins from 0 and has a maximum of size-1.

Example:
```
int a[10];
char name[80];
```

In addition, a storage class can also be specified for the array (default is auto).

Example: `static int x[20];`

5.3 ONE DIMENSIONAL ARRAY

A one dimensional array is declared as follows:

```
data_type array_name[size]
```

Example: `int n[10];`

This is a declaration of an array n of 10 integers. When an array is declared, the compiler reserves or allocates a block of memory large enough to store the entire array.

The total number of bytes allocated is:

Total bytes = length of array * sizeof (data_type)

Thus, for the above declaration, 20 bytes will be allocated (considering that an integer requires 2 bytes).

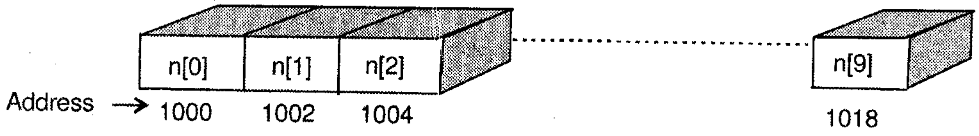


Figure 5.1: One dimensional array

5.3.1 Initializing an Array

Just like ordinary variables, an array can be initialized when it is declared. The entire array or a part of it can be initialized. An array can be initialized by the declaration followed by an=sign and a list of values enclosed in braces and separated by commas.

The values are assigned in order, to array elements from subscript 0.

Example

1. `int num[5]={10, 15, 25, 90, 100};`

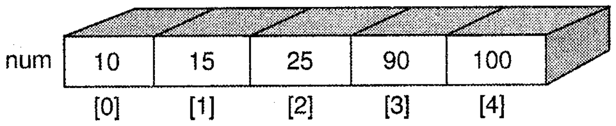


Figure 5.2: Array subscripts

a. During initialization, it is not necessary to specify the array size. The compiler allocates memory to hold the initialization values.

```
int num[]={10, 15, 25, 90, 100};
```

b. If less number of initialization values are specified, the remaining are initialized to 0.

```
int a[10]={1, 2, 3};
```

Here a[0], a[1] and a[2] are initialized to the specified values and the rest contain 0.

c. If more initializers than the specified number of array elements are specified, the compiler gives an error.

2. `char c[5]={'a', 'b', 'c', 'd', 'e'};`

5.3.2 Accessing Array Elements

To access a particular array element we have to specify the name of the array followed by the index in square braces. The index indicates the particular element we want to access.

Syntax: `array_name[integer_expression];`

Example

```
n[0] refers to the element at position 0, i.e., the first element.
n[2] refers to the element at position 2 which is the third element.
```

An integral expression can also be used as a subscript. *Example* is as follows:

```
n[5-2]
n[i++]
n[i-2]
n[--i]
n[i+j]
```

are all valid.

Assigning Values to Array Elements

Values can be assigned to individual elements by using the assignment operator (=).

Syntax: `array_name[index]=value;`

Example:
```
n[0] = 20;
n[2] = 35;
```

Entering Data into an Array

In most of the cases, the values are not known in advance. In such cases, we can accept the data from the standard input device (stdin) and store it in the array. This can be done in following manner.

The following code accepts ten numbers from the user and places them into the array.

```
for(i=0; i<10;i++)
{ printf("\n Enter the value for position %d", i);
  scanf("%d",&n[i]); }
```

The value of i goes from 0 to 9. Initially i=0 and the scanf statement will cause the integer read from the keyboard to be stored at the location (address) of n[0]. This process will be repeated for the entire array.

Warning

C does not perform bound checking for an array, i.e., it does not check for the validity of the subscript. This responsibility is of the programmer. Hence the programmer should ensure that the array length is not exceeded. Otherwise some other data may be overwritten.

Example:
```
int n[5];

for(i=0;i<10;i++)
  scanf("%d",&n[i]);
```

This code is perfectly valid in a C program.

Reading Data from an Array

All the array elements can be read (accessed) from the array using a for loop as shown:

```
for(i=0;i<10;i++)
printf("\n The value at position %d is %d", i, n[i]);
```

Examples

1. **/* Program to read 10 integers in an array, display them and calculate their average */**

```
#include<stdio.h>
main()
{ int n[10], i, sum = 0;
  float average;
/* Accept data and calculate sum */
  for(i = 0; i<10;i++){
    printf("\n Enter element %d",i);
    scanf("%d",&n[i]);
    sum  =  sum + n[i];}
  /* Display data */
  for(i = 0; i<10; i++)
    printf("\n Element %d is %d",i,n[i]);
/* calculate average */
  average = (float)sum /10;
  printf("\n The average is %f",average);
}
```

Note: An operation cannot be performed on a numeric array as a whole. The operation has to be performed on individual elements.

An array cannot be directly copied into another by using the assignment operator. Individual elements of the array have to be copied one-by-one.

2. Write a program which accept the array of an integer and find GCD and LCM.

1 PU Oct. 2009 – *5 M*

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int a, b, prod, lcm, arr[100];
  clrscr();
  prod =0;
  printf("\n Enter the value of array");
  for(a=0;a<100;a++)
  {
    scanf("%d", &arr[a]);
    prod = prod * a;
  }
  while(arr[a] != prod)
  {
    if(arr[a] > prod)
      arr[a]=arr[a] - prod;
    else
      prod = prod - arr[a];
  }
  printf("\n The GCD is %d", arr[a]);
  printf("\n The LCM is %d", prod/arr[a]);
  getch();
}
```

5.4 MULTIDIMENSIONAL ARRAYS

Multidimensional arrays have more than one subscript. Most often, we require these for storing and manipulating data structures such as matrices and tables. Here, a two-dimensional array is used. One subscript denotes the row and the other, the column.

Examples

1. `int m[3][2];`

   m is declared as a two dimensional array (matrix) having 3 rows (numbered 0 to 2) and 2 columns (numbered 0 to 1). The first element is m[0] [0] and the last is m [2][1].

2. `int arr[3][4][2];`

   arr is a three dimensional array which can be thought of 3 two-dimensional arrays having 4 rows and 2 columns each.

5.4.1 Initializing the Array

A multidimensional array can be initialized in two ways as illustrated in the example below.

```
int m[3][2] = {
                    {1,2}
                    {3,4}
                    {5,6}
                 };
or
  int m[3][2] = {1,2,3,4,5,6};
```

Note: While initialization, the row dimension (first subscript) is optional.

Example

1. `int m[][2] = {1,2,3,4,5,6};`

   All or only some elements could be initialized.

2.
```
int [4][3] = {
                   {0},
                   {1,2},
                   {3,4,5},
                   {6,7,8},
                 };
```

A three dimensional array can be initialized as shown.

```
int num[3][4][2] = {
                       {
                         {1,2},
                         {3,4},
                         {5,6},
                         {7,8}
                       },
                       {
                         {9,10},
                {11,12},
                {13,14},
                {15,16},
            },
            {
                {17,18},
                {19,20},
                {21,22},
                {23,24},
            }
        };
```

5.4.2 Memory Representation

The arrangement of elements of array m in the previous example can be shown as

	Col 0	Col 1
Row 0	1	2
Row 1	3	4
Row 2	5	6

These elements are stored in contiguous memory locations row-wise, as illustrated below.

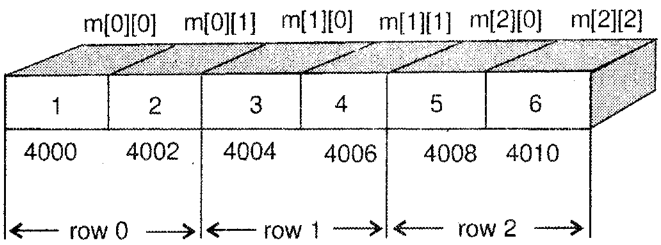


Figure 5.3

The three dimensional array num in the previous example can be represented as

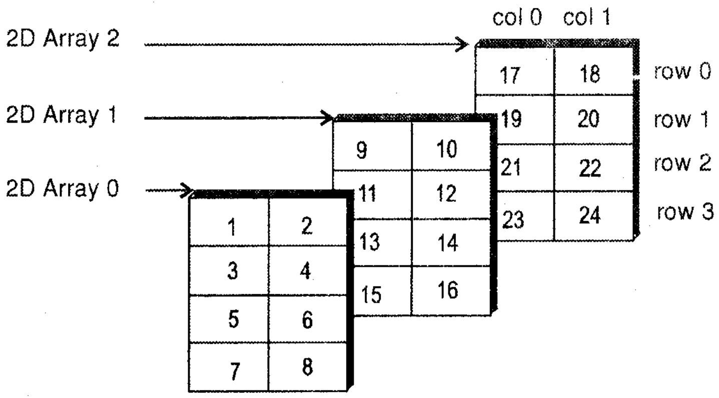


Figure 5.4

Memory Map

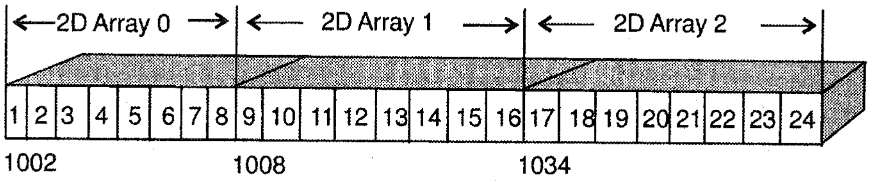


Figure 5.5

5.4.3 Accessing Array Elements

The elements of a two dimensional array can be accessed by the following expression:

array_name[i] [j]

where i refers to the row number and j, the column number.

Example: `m[1][0]refers to the number 3`

For the 3D array, three subscripts will be required.

Example: `num[2][1][0] refers to 19`

We have already seen how data for one-dimensional array can be accepted from the user. A similar method is used for a two-dimensional array except that we will now have two loops, one for the row subscript and the other for the column.

For every value of row subscript, the column subscript has to increment from 0 to number of-columns-1.

Examples

The following program illustrates matrix additions.

1. /* Program to add two matrices */

```
#include<stdio.h>
main()
{
  int mat1[10][10], mat2[10][10], mat3[10][10],
      r1,c1,r2,c2,i,j;
  printf("How many rows and columns in matrix 1?:");
  scanf("%d%d",&r1, &c1);
  printf("How many rows and columns in matrix 2?:");
  scanf("%d%d",&r2, &c2);

  if((r1==r2)&&(c1==c2))  /* check if they can be added */
  {
  printf("\n Addition possible \n");
  printf("\n Input Matrix 1 : \n");
  for(i=0;i<r1;i++)        /* row subscript */
```

```
    for(j=0;j<c1;j++)       /* column subscript */
      scanf("%d", &mat1[i][j]);
    printf("\n Matrix 2 : \n);
    for(i=0; i<r2; i++)
      for(j=0; j<c2; i++)
      {
        scanf("%d", &mat2[i][j]);
        /*  add corresponding elements of both matrices */
        mat3[i][j]  = mat1[i][j] + mat2 [i][j];
      }
  /* Display result */
    printf("\n The sum is : \n");
    for(i=0;i<r1;i++)
    {
      for(j=0;j<c1;j++)
        printf("%5d",mat3[i][j]);
      printf("\n");
    }
  }/* end if */
  else
      printf("\n Addition not possible \n");
} /* end main */
```

Output a

```
How many rows and columns in matrix 1?  :    2    3
How many rows and columns in matrix 2?  :    3    3
Addition not possible.
```

Output b

```
How many rows and columns in matrix 1?:
             2    2
How many rows and columns in matrix 2?:
             2    2
Addition possible
Input Matrix 1
1    2
3    4
Input Matrix 2
1    1
1    1
The sum is
2    3
4    5
```

2. Write a program to read m × n size matrix and print its transpose.

PU
Oct. 2008 – *5 M*

```
#include<stdio.h>
#include<conio.h>
#define m 100
#define n 100
void main()
{
   int a[m][n],b,c,i,j;
   clrscr();
   printf("\n Please Enter Limits of Matrix");
   printf("\n No of Rows");
   scanf("%d", &b);
   printf("\n No of Columns");
   scanf("%d", &c);
   printf("\n Enter the Value of Matrix");
   for(i=0;i<b;i++)
   {
      for(j=0;j<c;j++)
      {
         scanf("%d", &a[i][j]);
      }
   }
  printf("\n Matrix Value after Transpose");
  for(i=0;i<c;i++)
  {
        printf("\n");
        for(j=0;j<b;j++)
        {
           printf("%d", a[j][i]);
        }
  }
  getch();
}
```

3. Write a C program to display the 2D matrix in a circular way. (e.g., A[3][3] = {1,2,3,4,5,6,7,8,9} then your output is 1 2 3 6 9 8 7 4 5).

PU
Apr.09, Oct.09 – *4 M*

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int a[10][10],i,j,n;
  clrscr();
  printf("\nEnter the order of matrix");
  scanf("%d",&n);
```

```
 printf("\nEnter the element");
 for(i=0;i<n;i++)
 {
   for(j=0;j<n;j++)
   {
     scanf("%d",&a[i][j]);
   }
 }
 for(i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
   {
     printf("%d",a[i][j]);
     printf("\t\t");
   }
 printf("\n\n");
 }
  printf("\n matrix in circular order\n\n");

  for(i=0,j=0;j<n;j++)
  {
   printf("%d\t",a[i][j]);
  }

   j=j-1;
  for(i=1;i<n;i++)
  {
     printf("%d\t",a[i][j]);
   }
    i=i-1;
    for(j=1;j>-1;j--)
    {
      printf("%d\t",a[i][j]);
    }
    j=j+1;
   for(i=1;i>=1;i--)
   {
   printf("%d",a[i][j]);
   }
   i=1;
   for(j=1;j<n-1;j++)
   {
   printf("\t%d",a[i][j]);
   }
 getch();
}
```

4. **Accept 5 × 5 matrix from the user and display the sum of each column.**

1 PU Apr. 2010 – *5 M*

```
#include<stdio.h>
void main()
{
   int sum=0,a[5][5],n=5,i,m=5,j;
   for(i=0;i<6;i++]
   {
     for(j=0;j<6;j++)
     {
       printf("Enter the elements");
       scanf("%d",&a[i][j]);//accepting matrix elements
     }
   }
   for(j=0;j<n;j++)
   {
     sum=0;
     for(i=0;i<m;i++)
     {
       sum=sum+a[i][j];
       a[m][j]=sum; //making sum of column
     }
     printf("\n");
   }
   printf("\n printing matrix and sum of their columns\n");
   for(i=0;i<=m;i++)
   {
     for(j=0;j<n;j++)
     {
       printf("\t%d",a[i][j]);
     }
       printf("\n");
   }
}
```

5.4.4 Limitations of an Array

i. The compiler uses static memory allocation for an array, i.e., we have to specify the array size in advance. It is not possible to increase or decrease the array size at runtime.

ii. Elements cannot be inserted into an array.

iii. We cannot delete elements into an array.

iv. If the number of elements to be stored is not known in advance, there may be memory wastage if an array of large size is specified.

v. If a small array size is specified, there may not be enough memory to place all elements.

vi. C does not perform bound checking on an array, i.e., it does not check for the validity of the array subscript. Hence, if the array range is exceeded, some other data may get overwritten.

4. Accept 5 × 5 matrix from the user and display the sum of each column.

1
PU
Apr. 2010 – *5 M*

```
#include<stdio.h>
void main()
{
  int sum=0,a[5][5],n=5,i,m=5,j;
  for(i=0;i<6;i++]
  {
    for(j=0;j<6;j++)
    {
      printf("Enter the elements");
      scanf("%d",&a[i][j]);//accepting matrix elements
    }
  }
  for(j=0;j<n;j++)
  {
    sum=0;
    for(i=0;i<m;i++)
    {
      sum=sum+a[i][j];
      a[m][j]=sum; //making sum of column
    }
    printf("\n");
  }
  printf("\n printing matrix and sum of their columns\n");
  for(i=0;i<=m;i++)
  {
    for(j=0;j<n;j++)
    {
      printf("\t%d",a[i][j]);
    }
      printf("\n");
  }
}
```

5.4.4 Limitations of an Array

i. The compiler uses static memory allocation for an array, i.e., we have to specify the array size in advance. It is not possible to increase or decrease the array size at runtime.

ii. Elements cannot be inserted into an array.

iii. We cannot delete elements into an array.

iv. If the number of elements to be stored is not known in advance, there may be memory wastage if an array of large size is specified.

v. If a small array size is specified, there may not be enough memory to place all elements.

vi. C does not perform bound checking on an array, i.e., it does not check for the validity of the array subscript. Hence, if the array range is exceeded, some other data may get overwritten.

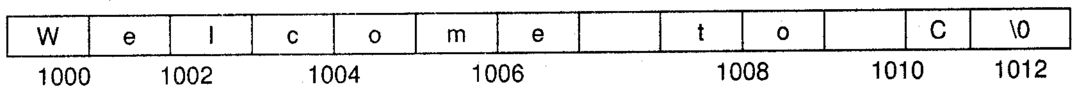
```
 printf("\nEnter the element");
 for(i=0;i<n;i++)
 {
   for(j=0;j<n;j++)
   {
     scanf("%d",&a[i][j]);
   }
 }
 for(i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
   {
     printf("%d",a[i][j]);
     printf("\t\t");
   }
 printf("\n\n");
 }
  printf("\n matrix in circular order\n\n");

  for(i=0,j=0;j<n;j++)
  {
   printf("%d\t",a[i][j]);
  }

   j=j-1;
  for(i=1;i<n;i++)
  {
     printf("%d\t",a[i][j]);
   }
    i=i-1;
    for(j=1;j>-1;j--)
    {
      printf("%d\t",a[i][j]);
    }
    j=j+1;
   for(i=1;i>=1;i--)
   {
   printf("%d",a[i][j]);
   }
   i=1;
   for(j=1;j<n-1;j++)
   {
   printf("\t%d",a[i][j]);
   }
 getch();
}
```

5.5 STRINGS

A string is an array of characters terminated by a special character called NULL character ('\0').

Strings in C are enclosed within double quotes.

Example: "Welcome to C" is a string and it is stored in memory as:



Each character is stored in 1 byte as its ASCII code. Since the string is stored as an array, it is possible to manipulate individual characters using either subscript or pointer notation.

5.5.1 Declaring and Initializing Strings

Since a string is a character array, it is declared as follows:

char string_name[length];

The length determines the maximum number of characters in the string.

Examples:
```
char city[10];
char name[20];
char message[80];
```

There are two ways to initialize strings.

i.
```
char city[] = {'p','u','n','e','\0'};
```
However, C offers a better way to initialize strings.

ii.
```
char city[] = "Pune";
```
The compiler automatically stores the null character at the end of the string.

Consider the following two declarations. Both are valid, however, there is a distinction between the two.

```
char amessg[] = "c programming language";  /* array */
```
amessg is an array big enough to hold the sequence of characters and '\0'.

Even if the characters are later changed, amessg will always refer to the same storage.

amessg
C programming language \0

5.5.2 String Input/Output

The functions printf and scanf can also be used with the format specifier %s. The scanf function does not allow a string with embedded spaces. gets() allows a string to contain spaces.

Example: Accept the name of a person and display a greeting.

```
char name[80];
printf("Enter your name")·;
gets(name);
printf("Good Morning %s", name);
```

5.5.3 String Manipulation Functions

C language provides a large number of functions in the header file **string.h** for the handling of strings.

The most commonly used functions are:

i. **strlen():** This function returns an integer corresponding to the number of characters in the specified string.

Syntax: `size_t strlen(char * s)`

Example:

```
char str[20];
gets(str);
printf("% d", strlen(str));
```

If the user enters **C language,** the output will be 10.

ii. **strcat():** This function is used to concatenate (join) two strings. It concatenates a copy of the second string to the first and returns the first. The second remains unchanged.

Syntax: `char * strcat (char * s1, char* s2)`

Example:

```
char s1[20] = "Pune" , s2[20] = "Mumbai";
strcat(s1,s2);
puts(s1);
puts(s2);
```

The **output** is:

```
PuneMumbai
Mumbai
```

iii. **strcmp( ):** This function is used to compare two strings. It returns an integer, which is

–ve if string 1 < string 2

0 if string 1 is equal to string 2

+ve if string 1 > string 2.

Syntax: `int strcmp(char * s1, char * s2);`

Example:

```
char s1[10] = "ABC", s2[10] = "abc";
printf("%d", strcmp(s1,s2));
```

The output will be –ve. Since "ABC" is less than "abc" because the ASCII value of 'A' is < ASCII value of 'a'.

Note: The function **strcmpi** is used to compare two strings ignoring the case.

iv. **strcpy:** This function copies the contents of string 2 to string 1 and returns string 1. The original contents of string are lost.

Syntax: `char * strcpy(char * s1, char * s2)`

Example:
```
char s1[20] = "Pune", s2[20] = "Mumbai";
strcpy(s1,s2);
puts(s1);
puts(s2);
```
The **output** will be Mumbai
Mumbai

Name	Prototype	Description
strlen	`size_t strlen(char * s)`	Returns the length of the string (numbers of characters excluding the NULL character)
strcpy	`char * strcpy(char *d, char *s)`	Copies the contents of string s to d and returns pointer to d
strcat	`char * strcat (char * s1,char *s2)`	Concatenates a copy of s2 to s1 and terminates s1 with a null. Returns s1.
strcmp	`int strcmp(char *s1,char *s2)`	Compares s1 and s2 and returns, -ve if s1 is less than s2 0 if s1 is equal to s2 +ve if s1 is greater than s2
strcmpi	`int strcmpi(char *s1, char * s2)`	Compares s1 and s2 ignoring the case and returns similar results as strcmp.
strlwr	`char *strlwr(char *s)`	Converts a string pointed to by s to lowercase.
strupr	`char *strupr(char *s)`	Converts a string pointed to by s to uppercase.
strncat	`char * strncat (char *s1,char *s2, int n)`	Concatenates first n characters of s2 to s1 and returns s1. s2 is unchanged.
strrev	`char *strrev(char *s)`	Reverses the string s and returns the reversed string.
strchr	`char *strchr(char *s,char ch)`	Returns a pointer to the first occurrence of character ch in string s.
strstr	`char *strstr(chars*1,char *s2)`	Returns a pointer to the first occurrence of s2 in s1. Returns null if no match is found.
strset	`char * strset(char *s,char ch)`	Sets all characters in pointed to by s to the value of ch.
strsnet	`char * strnset (char *s,char ch, int n)`	Sets the first n characters of string s to the value of ch
atoi	`int atoi(char *s)`	Converts a string pointed to by s into an integer, returning the result. Similarly there is atol and atof.
strncmp	`int strncmp (char*s1,char*s2,int n)`	Compares first n characters of s1 and s2. Returns <0 if s1 is less than s2,0 if they are the same, >0 if s1 is greater than s2.
strdup	`char *strdup(chars *s)`	Duplicates a string at another location and returns NULL if space could not be allocated or returns a pointer to the storage location containing duplicated string.
strchr	`char * strchr(char *s, int c)`	Returns a pointer to the last occurrence of character c in string s, NULL if not found.
strtok	`char *strtok(char *s1,char *s2)`	Searches s1 for tokens that are separated by delimiters specified in s2. Returns the pointers to the first character of first token in s1.

Sample Programs

1. **Write a program to accept a string and check that string is palindrome or not.**

PU Oct. 2009 – *5 M*

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
   char str[45];
   int i,j, len, flag = 1;
   clrscr();
   printf("\n Enter the String \n");
   gets(str);
   len=strlen(str);
   for(i=0,j=len-1;i<=len/2;i++,j--)
   {
      if(str[i] != str[j])
      {
         flag = 0;
         break;
      }
   }
   if(flag)
      printf("\n String is a Palindrome");
   else
      printf("\n String is not a Palindrome");
   getch();
}
```

2. **Program to reverse a string**

```
#include<stdio.h>
#include<string.h>
main()
{  char s1[30], rev[30];
   int i,j,len;
   printf("Enter a string:")
   gets(s1);
   len = strlen(s1);
   for(j = 0, i = len-1; i>= 0; i--, j++)
         rev[j] = s1[i];
   rev[j] = '\0';   /* terminate string*/
   printf("The reversed string is: ";
   puts(rev);
}
```

Output

```
Enter a string : Pune
The reversed string is enuP
```

3. Substring of a string (return n characters of a string from location m)

```
#include<stdio.h>
#include<string.h>
main()
{  char str[30]
   int i,j,m,n;
   printf("Enter the string");
   gets(str);
   printf("Enter the number of characters and position");
   scanf("%d%d",&n, &m);
   printf("\n The substring is : \n");
   for(i= m-1 , j=1; (j<=n)&& (str[i]! ='\0'); i++, j++)
          printf("%c", str[i]);
}
```

5.5.5 Array of Strings

An array of strings is a two dimensional array. This is often required in applications dealing with a list of names, etc.

An array of strings can be initialized.

Example: `char cities[4][10] = {"Pune", "Mumbai", "Delhi", "Chennai"};`

They are stored as:

	0	1	2	3	4	5	6	7	8	9
cities [0]	P	u	n	e	\0					
cities [1]	M	u	m	b	a	i	\0			
cities [2]	D	e	l	h	i	\0				
cities [3]	C	h	e	n	n	a	i	\0		

In the following program, we will accept 'n' names and sort them alphabetically.

/* Illustrates an array of strings */

```
#include<stdio.h>
main()
{
  char names[20][30],temp[30],   /*list of 20 names */
  int i,j,n;
  printf("How many names?:");
  scanf("%d", &n);
  /*Accept names */
  for(i=0;i<n; i++)
  { printf("Enter name %d",i+1);
    gets(names[i]);     /*list[i] is the name of the ith string */
    }
    /*Sorting*/
    for(i=0; i<(n-1); i++)
    for(j = i+1; j<n; j++);
    {
      if(strcmp(names[i],names[j])> 0)
    {
      strcpy(temp,names[i]);
      strcpy(names[i],names[j]);
      strcpy(names[j],temp;
    }
    }
/* Display Sorted List */
  puts("The sorted names are");
  for(i=0; i<n ; i++)
       puts(names[i]);
}
```

SOLVED PROGRAMS

What will be the output of the following segments of program code?

1.
```
int arr[12];
printf("%d",sizeof(arr));
```
Output: 24

arr is declared as an array of 12 integers; each requiring 2 bytes. Thus the size of arr is 12 * sizeof (int) = 24 bytes.

2.
```
char city[20] = "Pune";
printf("%d", sizeof(city));
```
Output: 20

city is declared as an array of 20 characters each requiring 1 byte.

3.
```
char names[][10] = {"Pune", "Delhi", "Bangalore"};
printf("%d\t%d",sizeof(names), sizeof(names[2]));
```
Output: 30 10

names is an array of 3 strings each of length 10 characters. Thus, the size of the array is 30 characters. Each string is of length 10 characters.

4.
```
void main()
{ char message[]="This is extremely long prompt\n"
            "How long is it?\n"
Printf("%s\n %s", message,message+5);
}
```

PU Apr. 2009 – 4 M

Output is

```
This is extremely long prompt
How long is it?
is extremely long prompt
How long is it?
```

Because, message → will print the content of the array.

message +5 → truncate first 5 character from the string and print remaining message.

5.
```
#define kMaxArraySize 100
int main(void)
{
  char myArray[kMaxArraySize];
  int i;
  for(i = 0; i<kMaxArraySize;i++)
  myArray[i] = 0;
  return 0;
}
```

PU Oct. 2009 – 4 M

The above program runs successfully, but there is no output because in the program no printf() statement is mention.

EXERCISES

A. Predict the output

1.
```
main()
   {  int x[25];
      x[0] = 100;
      x[24]= 400;
      printf("\n%d%d",*x,*(x+24)+*(x+0));
   }
```
2.
```
main()
{  char a[5* 2/2] = {'a','b','x','y','z'};
   printf("%c\n",a[3]);
}
```
3.
```
main()
{  int p[6]  = {1,2,3,4,5,6}
   int *q = p;
   printf("%d%d%d", *p+6, 2[q], p[1]);
}
```
4.
```
main()
{  static int num[10] = {1,0,0,0,0,0,0,0,0,0};
   int i,j;
   for(j=0;j<10; ++j)
      for(i=0;i<j;++i)
         num[j] = num[j]+num[i];
      for(i=0;i<10;i++)
         printf("%d\n",num[i]);
}
```

B. Programming exercises

1. Convert a decimal number to its binary, hexadecimal and octal equivalents.
2. Accept 'n' integers in an array. Accept an integer and check whether it is present in the array. If it is, display its position.
3. Accept a matrix and check if it is symmetric.
4. Accept a string and display it in the following forms.
   If string is ABCDE, the output should be
   ABCDE BCDEA CDEAB DEABC EABCD ABCDE
5. Read a string and rewrite it in alphabetical order.
6. Accept a matrix and find the largest and smallest number from the matrix.
7. Shift all zeroes in the series of digits to the end of the series.
   Example: I/p = 001054, O/p = 154000

C. Review questions

1. What is an array?
2. How can an array be initialized?
3. What are multidimensional arrays? How are they initialized?
4. What is the significance of the name of an array?
5. How can an array be passed to a function? Give examples.
6. What are strings?
7. Explain the function.
   i. strcpy ii. strcmp iii. strlen iv. strcat

Collection of Questions asked in Previous Exams PU

1. Write a program to read m × n size matrix and print its transpose. **[Oct. 2008 – 5 M]**
2. What will be the output? Give explanation. **[Apr. 2009 – 4 M]**

```
void main()
{ char message[]="This is extremely long prompt\n"
          "How long is it?\n"
Printf("%s\n %s", message,message+5);
}
```

3. Write a C program to display the 2D matrix in a circular way. **[Apr. 2009 – 4 M]**
(e.g., A[3][3] = {1,2,3,4,5,6,7,8,9} then your output is 1 2 3 6 9 8 7 4 5).
4. What will be the output? Give explanation.
i.

```
#define kMaxArraySize 100
int main(void)
{
  char myArray[kMaxArraySize];
  int i;
  for(i = 0; i<kMaxArraySize;i++)
  myArray[i] = 0;
  return 0;
}
```

[Oct. 2009 – 4 M]

ii.

```
char s[20];
  strcpy (s, "Hello");
  if(strcmp(s,"Hello"))
  printf("The string are the same!");
```

[Oct. 2009 – 4 M]

5. Write a program to accept a string and check that string is palindrome or not. **[Oct. 2009 – 5 M]**
6. Write a program which accept the array of an integer and find GCD and LCM. **[Oct. 2009 – 5 M]**
7. Write a program that accept N*N matrix. Print the matrix in a circular form. e.g., A[3] [3] = {1,2,3,4,5,6,7,8,9} then your o/p is 1,2,3,6,9,8,7,4,5. **[Oct. 2009 – 10 M]**
8. Write a C program to accept a string from the user and reverse each word in the given string.
(e.g: Input is: My Name is Computer
Output should be: yM emaN si retupmoC). **[Apr. 2010 – 10 M]**
9. Accept 5 × 5 matrix from the user and display the sum of each column. **[Apr. 2010 – 5 M]**
10. Write a program to accept five strings from user and display all those strings in descending order. **[Oct. 2010 – 10 M]**
11. Write a user defined function * strcmp (char * t_1, char * t_2) which compare two strings t_1 and t_2. Do not us library functions. **[Oct. 2010 – 5 M]**

6 Pointers

6.1 INTRODUCTION

Pointers are an important part of C language, which provide a powerful and flexible way to manipulate data. They should, however, be used correctly and carefully. Before we go into the details of pointers, it is essential to know some concepts about the organization of memory and how the variables are stored there.

6.2 MEMORY ORGANIZATION

The computer's main memory (RAM), consists of a large number of sequential storage locations, each capable of storing one word of data (usually 1 byte) and identified by an unique address. Typically, the addresses are numbered sequentially from 0 to some maximum depending upon the memory size, i.e., they are positive integer values. When the system is running, the operating system, uses some part of the memory. When we are running a program, the program code and program data also occupy some of the system's memory. In this chapter, we will deal with the memory storage for program data.

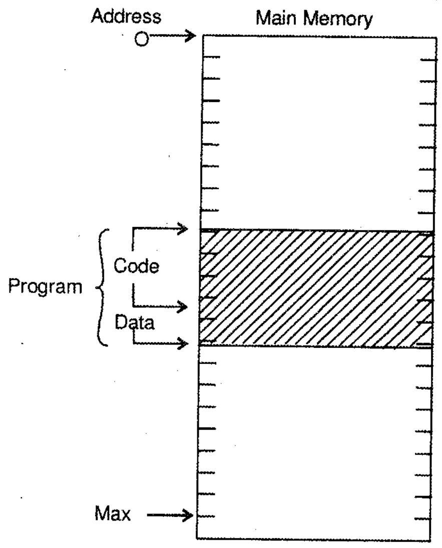


Figure 6.1: System memory and addresses

When we use a variable in a program, the compiler sets aside (allocates) a memory location for that variable. It associates the location's address (which is unique) with the variable name. Whenever the program uses the variable, the compiler automatically translates the name into address.

Example

Consider the following statement:

```
int n = 100;
```

When this statement executes, the compiler,

i. reserves space in memory to hold an integer value.

ii. associates the name 'n' with this memory location.

iii. stores value 100 at this location.

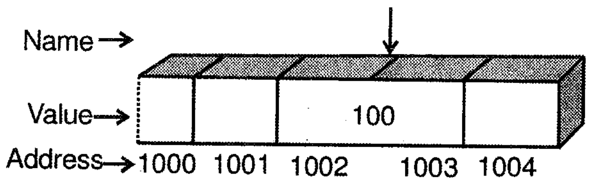


Figure 6.2: Storage of variable

In *figure 6.2*, 2 bytes of memory from location 1002 have been allotted to variable 'n' assuming that an integer requires 2 bytes of storage.

6.3 BASICS OF POINTERS

In order to manipulate the value stored at a particular memory location, a user is allowed to access the address of the variable by using the '&' operator.

The Address operator (&)

When used as a prefix to a variable name, the '&' operator gives the address of that variable.

For the above example, & n will yield 1002.

Note: '&' can be used only with single variable or array elements.

&75, & (a+b), & ('x') are all invalid.

1. /* Demonstration of value and address of a variable */

```
#include<stdio.h>
main()
{
 int n = 20;
 printf("\n The value of n is %d",n);
 printf("\n The address of n is %u",&n);
}
```

Output

```
The value of n is 20
The address of n is 1002
```

Since addresses are integers, it is possible to assign an address to another variable, which will also be stored in memory just like any other variable.

The assignment can be done by the = operator as shown.

```
ptr_n = &n;
```

This statement assigns the address of 'n' to a variable ptr_n.

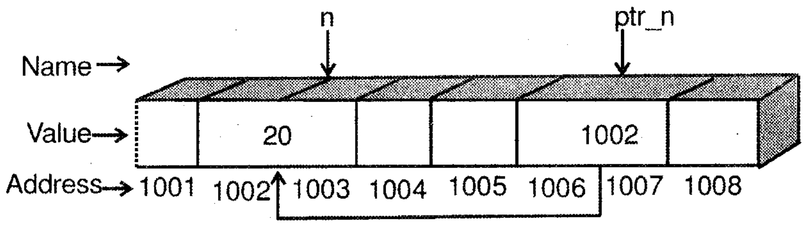


Figure 6.3

What is a Pointer?

A pointer is a variable that stores the memory address of another variable.

Since it is a variable, the pointer variable itself will be stored at some other memory location.

In the above example, ptr_n is a pointer variable because it stores the address of another variable n and hence it is called a pointer to n.

A pointer provides a method for accessing a variable indirectly.

6.4 APPLICATIONS OF POINTERS

i. Pointers can be used to simulate passing parameters by reference, i.e., the arguments can be modified.

ii. They provide an alternate method to access array elements.

iii. They are used for passing arrays and strings to functions.

iv. They are more efficient in handling complex data structures like linked lists, trees, graphs etc.

v. One of the most important use of pointer is in dynamic memory allocation where memory is allocated and released for a variable during run-time.

6.5 USING POINTERS

6.5.1 Declaring a Pointer

Since a pointer is a variable like any other, it has to be declared before it can be used.

Syntax: `data_type * pointer_name;`

i. data_type is any C data type and it indicates the type of the variable that the pointer points to.

ii. The asterisk(*) is the indirection operator and it indicates that `pointer_name` is a pointer variable and that it stores the address of a variable of the specified data type.

iii. pointer_name is a valid C identifier.

Examples

```
char *p_ch1,*p_ch2; /* p_ch1 and p_ch2 are pointers to type char */
int num , *ptr_num ; /* num is an integer variable and ptr_num is a pointer
to type integer */
```

6.5.2 Initializing Pointers

Until a pointer holds the address of a variable it is not useful. The address of a variable has to be specifically put into a pointer variable by using the address-of operator (&).

A pointer can be initialized by a statement of the form

```
pointer = &variable;
```

Example: `ptr_n = &n ;   /* assign address of n to ptr_n */`

A pointer variable can also be initialized as , char *p = "ABCD";

The pointer p points to a string ABCD which is stored somewhere in memory.

6.5.3 De-referencing Pointer

After having declared and initialized pointers, we come to the main part, i.e., how to use them.

The * (indirection) operator is used along with pointer variables. It is also called the value-at operator. When used with a pointer variable, it refers to the variable being pointed to. This is **de-referencing** of pointers.

Syntax: `*pointer_name`

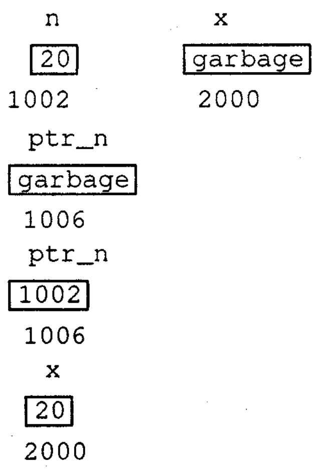
Thus, if ptr_n is a pointer, * ptr_n implies value at address stored in ptr_n or variable whose address is in ptr_n.

De-referencing is the operation performed to access or manipulate data contained in the memory location pointed to by a pointer.

Note: Any operation performed on the de-referenced pointer directly affects the value of the variable it points to.

The following example illustrates these concepts.

```
int n = 20,x ;

int *ptr_n;/*Uninitialized
pointer*/

ptr_n = &n;/*stores address of n
        in ptr_n*/

x= *ptr_n; /*put value at ptr_n
          in x */
```



In the last statement, x = *ptr_n, the right hand side, i.e., *ptr_n gets the value stored at address in ptr_n.

$\therefore$ *(ptr_n) $\Rightarrow$ value_at (ptr_n)

$\Rightarrow$ value_at (1002)

$\Rightarrow$ 20

1. /* Illustrate basic pointer use */

```
#include<stdio.h>
main()
{
  /* Declarations */
  int n = 20 , *ptr_n;
  ptr_n = &n ;
  /* Displaying Value */
  printf("\n Direct access, value = %d", n);
  printf("\n Indirect access, value = %d" *ptr_n);
  /* Displaying address */
  printf("\n\n Direct access,address = %u",&n)
  printf("\n Indirect access, address = %u", ptr_n);
  /* Modifying value */
  n = 30 ;  /* Direct modification */
  printf("n\n Direct modification, value = %d%d",n,*ptr_n);
  *ptr_n = 50;       /* Indirect modification */
  printf("\n Indirect modification , value = %d%d,"n,*ptr_n);
}
```

Output

```
Direct access, value    = 20
Indirect access, value =   20
Direct access, address =  1002
Indirect access, address   =  1002
Direct modification, value = 30     30
Indirect modification, value =  50     50
```

6.6 POINTER EXPRESSION

Like other variables, pointer variables can be used in expressions.

For example: If p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y = *p1 * p2; same as (*p1) *(*p2)
sum = sum + *p1;
z = 7* - * p2 / *p1; same as (7(-(*p2)))/(*p1)
*p2 = *p2 + 10;
```

Note that there is a blank space between / and * in the item 3 above. The following is wrong.

```
z = 7* - * p2 /*p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

6.6.1 VOID Pointer

Pointers defined to point to a specific data type cannot hold the address of any other type of variable.

Example

The following code is invalid.

```
float *ptr;
int x;
ptr = &x;
```

C supports a general purpose pointer type called the void pointer. A void pointer does not have any data type associated with it and can contain the address of any type of variable. They can be declared as:

```
void * pointer_name;
```

Example:
```
void * v_ptr;       /* declare v_ptr as void pointer */
char ch;
int i;
float fvar;

v_ptr = &ch;        /* valid */
v_ptr = &i;         /* valid */
v_ptr = &fvar;      /* valid */
```

6.6.2 De-referencing Void Pointer

Pointers to void cannot be directly de-referenced like other pointer variables by using the * operator.

Before de-referencing, the pointer has to be typecast to the required data type.

Any pointer can be typecast to a pointer of another type by

```
(data_type *)pointer_name;
```

Example:
```
char ch;
void *v_ptr = &ch;
```

In the above code, if the pointer v_ptr has to be used to refer to the character ch, it can be typecast using

```
(char *)v_ptr
```

Program: Illustrate the use of void pointers

```
#include<stdio.h>
main()
{ int n = 20;
  float m = 12.5;
  void *v_ptr;
  v_ptr = &n;
  printf("Value of n is %d %d", n,*((int*)v_ptr));
  v_ptr = &m;
  printf("\n Value of m is %f %f", m ,*((float *)v_ptr));
  /* Typecast using(float *) */
}
```

Output

```
Value of n is 20      20
Value of m is 12.5    12.5
```

6.6.3 Pointer Arithmetic

When the * operator is used with a pointer, the number of bytes accessed from the memory will depend upon the data type to which the pointer points.

For example, when de-referenced,

- A pointer to an int accesses 2 bytes of memory.
- A pointer to a char accesses 1 byte of memory.
- A pointer to a float accesses 4 bytes of memory.
- A pointer to a double accesses 8 bytes of memory.

The C language allows five arithmetic operations to be performed on pointers

i. Increment ++
ii. Decrement - -
iii. Addition +
iv. Subtraction -
v. Differencing

i. Increment and decrement

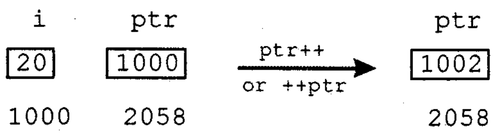
When a pointer to some data type, (where data-type may be int, char, float, etc.) is incremented by an integral value, i.e., the new value will be,

```
(current address in pointer) + i * sizeof(data_type)
```

Example: Incrementing a pointer to an int will cause its value to be incremented by 2 if an input occupies 2 bytes.

Similarly, a pointer to a float will be incremented by 4 and not 1.

Example:
```
int i = 20;
int *ptr = &i;
```



The same concept applies for decrementing a pointer, i.e., if a pointer is decremented; it is decreased by the size of the data item it points to.

ii. Addition and subtraction

C allows integers to be added to or subtracted from pointers.

Example:
```
int *ptr1, n;
ptr1    = &n;
ptr1    = ptr1+3;
```

This code will increment the content of ptr1 by 6 since ptr1 points to an integer (2 bytes) and we are incrementing it by 3 (i.e., 3 * sizeof (int)).

We cannot add two pointers, i.e., P1+P2 is illegal.

Subtraction is performed in the same way.

iii. Differencing

The only other pointer arithmetic operation allowed is called differencing which is the subtraction of two pointers.

The subtraction of the two pointers indicates how far apart they are. The result is of a type called size_t, which is an unsigned integer. It gives the number of elements between two pointers.

Example:
```
int n, *p,*q;
p = &n;
q = p+2;
printf("%d", q-p);
```

This code will yield a value of 2 even though numerically q and p differ by 4. This is because, both point to the integer data type and the difference between them is 2 objects.

iv. Comparison of two pointers

Pointer comparison is valid only between pointers that point to the same array. In such a case, all relational operators can be used.

However the comparison operators == and != can be used to compare pointers of the same type, void pointers and any other pointer, and any pointer and NULL.

Operations on pointers

Operations on pointers are

Assignment	:	The value assigned should be an address.
Indirection	:	Getting the value stored at a location.
Address of	:	The & operator used with a pointer gives its address.
Increment	:	Adding an integer to a pointer increments it by the bytes required for those many data objects.
Decrement	:	Subtracting an integer from a pointer reduces it by the specified number of data-objects * sizeof (data-object).
Differencing	:	Subtraction of two pointers.

Comparison : Equality and inequality can be used with all. The other relational operators can be used only with pointers pointing to the same array.

Pointers cannot be multiplied or divided, i.e., expressions such as p_1/p_2 or p_1*p_2 or $p_1/3$ are not allowed.

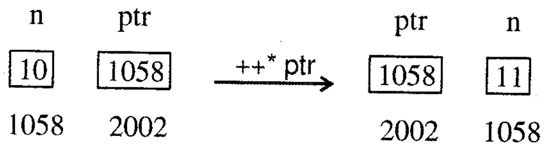
6.7 PRECEDENCE OF & AND * OPERATORS

Both are unary operators and have precedence equal to other unary operators. They associate from right to left.

Example

1.
```
int n = 10, *ptr;
ptr = &n;
printf("%d", ++*ptr);
```

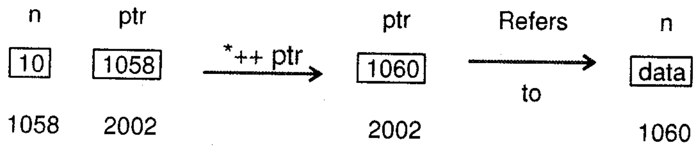
Let us consider the expression ++ * ptr . There are two operators, ++ and *, both unary with an associativity R → L. Thus, the * operation is performed first. It will fetch the value being pointed to by ptr, i.e., 10. Next, the ++ operator will increment it to 11 as illustrated below.



i.e., ++* ptr ⇒ ++(*ptr) ⇒ ++ (* 1058) ⇒ ++ (10) = 11

2.
```
int n = 10, *ptr ;
ptr = &n;
printf("%d",*++ptr);
```

In the expression *++ptr , ++ will be done first and then the value pointed to by the changed ptr will be fetched and displayed.



i.e., *++ ptr ⇒ * (++ptr) ⇒ * (++1058)

⇒ *(1060)

⇒ data_at_1060

3. *ptr ++ means first fetch the value pointed to and then increment ptr (since it is post increment).

6.8 POINTER TO POINTER

The concept of a pointer can be further extended. Since, a pointer variable contains the address of another variable, we could have a variable, which will contain the address of the pointer variable. Thus, we have a pointer to a pointer.

```
int i = 10;
int *ptr;
int **ptr_to_ptr:
ptr = &i;
ptr_to_ptr  = &ptr;
```

$$\begin{array}{ccc} \texttt{i} & \texttt{ptr} & \texttt{ptr\_to\_ptr} \\ \boxed{10} & \boxed{1058} & \boxed{2065} \\ 1058 & 2065 & 1002 \end{array}$$

Here,

$$\left.\begin{array}{l} \texttt{i} \\ \texttt{*ptr} \\ \texttt{*ptr\_to\_ptr} \end{array}\right\} \Rightarrow \texttt{value of i}$$

$$\left.\begin{array}{l} \texttt{\&i} \\ \texttt{ptr} \\ \texttt{*ptr\_to\_ptr} \end{array}\right\} \Rightarrow \texttt{Address of i}$$

The declaration

```
int *ptr;
```

implies that ptr is a pointer to an integer whereas,

```
int **ptr_to_ptr:
```

implies that ptr_to_ptr is a pointer to a pointer.

The double ** indicate that ptr_to_ptr contains the address of a pointer variable. We can also have a pointer to a pointer to a pointer. Conceptually, there is no limit on how much we can extend the pointers. However in practice, we rarely use more than two levels of pointers, i.e., pointer to a pointer.

Program: /* Illustrates pointer to pointer */

```
#include<stdio.h>
main()
{
  int i = 10, *p1, **p2,***p3;
    p1 =  &i;
    p2 =  &p1;
    p3 =  &p2;
  printf("\nThe value of i is %d %d %d %d", i,*p1,**p2,***p3);
  printf("\nThe address of i is %u %u %u %u", &i,p1,*p2,**p3);
}
```

Output

```
The value of i is 10 10 10 10
The address of i is 5498      5498      5498      5498
```

Pointers to pointers are often used in handling of strings, multidimensional arrays, linked lists, trees and graphs.

6.9 POINTERS TO CONSTANT OBJECTS

A pointer to a constant object can be declared as

```
const data_type *pointer_name;
```

Example: `const int *ptr;`

i.e., ptr is a pointer to a constant integer.

Consider the following code,

```
int i = 10 ;
const int *ptr;
ptr = &i;
```

i.e., ptr is a pointer to i. Such a declaration of ptr indicates that the contents pointed to by ptr cannot be changed, i.e.,

`*ptr = 20;` // is invalid

However, ptr itself can be changed, i.e.,

`ptr ++;` // is valid.

6.10 CONSTANT POINTER

A constant pointer cannot be modified; however the data item to which it points can be modified.

Example:

```
int i = 10;
const int *ptr;   /*  declares a constant pointer ptr */
ptr = &i;
*ptr = 20;       /* valid */
ptr++;           /* Invalid */
```

A declaration such as

`const int *ptr;` will not allow any modification to be made to ptr nor the integer to which it points to.

6.11 DYNAMIC MEMORY ALLOCATION

Dynamic memory allocation means allocating memory storage space at runtime.

So far, we have explicity allocated memory in the program source code by declaring variables and arrays.

This method is called **static** memory allocation. The programmer has to specify how much amount of memory is required.

For example, when we declare an array, we have to specify its size.

In many cases, a user does not know how many elements are to be put.

In such a case, memory is either wasted if the size specified is very large or enough memory is not allocated if the size specified is smaller than required.

In the Dynamic Allocation method, we can allocate and de-allocate memory whenever required.

C provides memory allocation and de-allocation functions.

	Function	Use
1.	malloc	Allocates requested number of bytes and returns a pointer to the first byte.
2.	calloc	This also allocates memory for a group of objects, initializes them to zero and returns a pointer to the first byte.
3.	realloc	It changes the size (expands or shrinks) of a previously allocated block of memory and returns a pointer to the block.
4.	free	Releases or frees previously allocated space.

6.11.1 Allocating a Block of Memory

The malloc( ) function can be used to allocate memory. The prototype of malloc is

```
void * malloc(size_t num);
```

size_t is defined in stdlib.h as an unsigned int. The malloc( ) function allocates num bytes of storage and returns a pointer to the first byte. It returns NULL if allocation is unsuccessful or if num is 0.

Example: To allocate memory to store n integers

```
int * ptr;          /*pointer to the block */
ptr = (int *) malloc(n * sizeof(int));
```

In this example, explicit type casting is required because by default malloc( ) returns a void pointer.

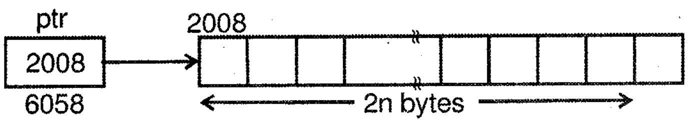


Figure 6.4: Allocated memory

The calloc( ) function is also similar but it allocates a group of objects and initializes the bytes to 0.

Prototype: `void * calloc(size_t num, size_t size);`

num is the number of objects to allocate and size is the size(in bytes) of each object.

Example:

```
int *ptr;
ptr = (int*)calloc(n,sizeof(int));
```

6.11.2 Freeing or De-allocating Memory

The free( ) function is used to release the memory that was allocated by malloc( ) or calloc( ) or realloc( ).

When memory is allocated it is taken from the dynamic memory pool (heap) that is available to the program.

After the program finishes using a particular block of dynamically allocated memory, it should be freed to make memory available for future use.

Prototype: `void free(void *ptr);`

The free( ) function releases the memory pointed to by ptr.

Program: /* Illustrate dynamic allocation and de-allocation */

```
#include<stdio.h>
#include<stdlib.h>
main()
{
  int *ptr, n, i;
  printf("How many elements?:");
  scanf("%d",&n);
  /* allocate memory */
  ptr = (int)* malloc(n*sizeof(int));
  if(ptr == NULL)
  { printf("Memory was not allocated");
    exit(0);
  }
  printf("\n Enter the  elements :\n");
  for(i=0;i<n;i++)
  scanf("%d", ptr + i);
  printf("\n You entered : \n");
  for(i=0;i<n; i++)
  printf("%d\n", ptr[i]);    /* or *(ptr+i)*/
    /* Free allocated memory */
    free(ptr);
}
```

In the above program, we have allocated and de-allocated memory for a 1D array.

We can also apply the same concepts for allocating memory for a 2D array.

6.11.3 Altering the Block Size

The realloc( ) function changes the size of a block of memory that was previously allocated with malloc( ) or calloc( ).

Prototype: `void * realloc(void *ptr, size_t size);`

ptr points to the original block, size is the required new size in bytes.

- If ptr is NULL, realloc acts like malloc and returns a pointer to it.
- If argument size is 0, the memory that ptr points to is freed and function returns NULL.
- If sufficient space exists to expand the memory block, additional memory is allocated and function returns ptr.

- If sufficient space does not exist to expand the current block, a new block of size bytes is allocated, existing data copied into it, old block is freed and a pointer to the new block is returned.
- If memory is insufficient for reallocation (either for expanding or allocating of new one), the function returns NULL and the old block remains unchanged.

The following program illustrates reallocation.

Program: /*Using realloc( )to change memory allocation */

```
#include<stdio.h>
#include<stdlib.h>
main()
{
  int *ptr,n,n1,i;
  printf("\n How many numbers :");
  scanf("%d",&n);
  ptr =(int*)malloc(n *sizeof(int));      /* Allocate */
  printf("\n Enter the numbers : \n ");
  for(i=0; i<n;i++)
  scanf("%d", ptr+i);
  printf("\n How many new numbers?:");
  scanf("%d",&n1);          /* Reallocate */
  ptr =(int *) realloc(ptr,(n+n1) * sizeof(int));
  if(ptr == NULL)
  exit(1);
  printf("\n Enter the remaining numbers \n"):
  for(i=n; i<(n+n1); i++)
  scanf("%d",ptr + i);
  printf("\n The entire list is : \n");
  for(i=0; i<(n+n1); i++)
  printf("%d\t", *(ptr+i));
}
```

Output

```
How many numbers : 3
Enter the numbers :
10
5
25
How many new numbers ? :2
Enter the remaining numbers
50
6
The entire list is :
10    5     25    50    6
```

6.12 POINTERS AND ARRAYS

Pointers and arrays are very closely related. As seen before, an array name without the subscript is a pointer to the first element in the array. The same holds for arrays of two or more dimensions.

Examples

i. `int p[10];`

Here, p and &p[0] are identical.

ii. `char a[10][10];`

Here, a and &a[0] [0] are identical.

Pointers are used very often to access arrays because pointer arithmetic is often a faster process than array indexing, especially when the array elements have to be accessed sequentially.

The reason for this is that, the C compiler internally converts a subscripted notation x[i] to the form *(x+i).

Here, x is the base address of the array.

(*x will give the value of the 0th element. Similarly, *(x+i) gives the value of the ith element.

Thus,

$$\begin{aligned} x[i] &\Rightarrow *(x+i) \\ &\Rightarrow *\,(i+x) \\ &\Rightarrow i\,[x] \end{aligned}$$

The following programs proves this,

1. /* Accessing array elements in different ways */

```
#include<stdio.h>
main()
{
  int x[]= {10,20,30,40,50};
  int *ptr, i;
  ptr = x;    /*   ptr stores the base address of array */
  for(i=0;i<5;i++)
  {
    printf("\n address = %u",&x[i]);
    printf("elements = %d%d%d%d", x[i], *(x+i),i[x], *ptr);
    ptr++; /*Make ptr point to the next element */
  }
}
```

Output

```
address =   6800   elements =   10   10   10   10
address =   6802   elements =   20   20   20   20
address =   6804   elements =   30   30   30   30
address =   6806   elements =   40   40   40   40
address =   6808   elements =   50   50   50   50
```

Similarly, a 2D array can also be accessed using the pointer notation instead of the subscripted one.

$$x[i]\,[j] \Rightarrow *(x\,[i] + j)$$

$$\Rightarrow *\,(*(x + i) + j)$$

This will refer to the element x[i][j].x[i] is the address of the i^th 1-D array.

The following program illustrates passing of an array to a function and the use of pointers.

2. /* Illustrates arrays and pointers */

```
#include<stdio.h>
main()
{
  int a[10],n,i;
  void display(int *x,int n);
  printf("\n How many numbers :");
  scanf("%d",&n);
  for(i=0;i<n;i++)
      scanf("%d",&a[i]);
  printf("\n You entered \n");
  display(a,n);
}
    void display(int *x, int n)
    { int i ;
      for(i = 0; i<n ; i++)
        printf("%d",x[i]); }
```

Output

```
How many numbers : 3
10
20
30
You entered
10    20    30
```

The expression x[i] is treated by the compiler as * (x+i).

The for loop in the above function could also be written as follows:

```
for(i=0;i<n;i++)
{ printf("%d",*x);
  x++; }
```

6.13 POINTERS AND CHARACTER STRING

Strings and pointers are very closely related, since a string is an array, the name of the string is a constant pointer to the string.

In previous chapter, we have seen some functions on strings.

Examples

1. **Find the length of the string.**

```
int length(char *s)
{
  int count = 0;
  while(*s! = '\0')
  { count++;
    s++;
  }
  return count;
}
main()
{
  char str[80];
  gets(str);
  printf("The length is %d", length(str));
}
```

2. **Write a program to accept five strings from user and display all those strings in descending order.**

1 PU Oct. 2010 – *10 M*

```
#include<string.h>
main()
{
  int i,j;
  char*str[5],*temp;
  for(i=0;i<5;i++)
  {
    str[i]=(char*)malloc(10*sizeof(char));//allocate the memory for str
                                                //array
    printf("\nEnter the string:");
    gets(str[i]);
  }
  for(i=0;i<4;i++)
  {
    for(j=i+1;j<5;j++)
    {
      if(strcmp(str[i],str[j])<0)//compares two strings this is bubble
                                   //sort logic
      {
        strcpy(temp,str[i]);
        strcpy(str[i],str[j]);
        strcpy(str[j],temp);
```

```
            }
        }
    }
    printf("Desecending order\n");
    for(i=0;i<5;i++)
    {
        puts(str[i]);
    }
}
```

6.14 ARRAY OF POINTERS

Just like we can have an array of integer, float or char, we can also have an array of pointer variables.

As pointer variables contain address, an array of pointers is a collection of addresses. These elements are stored in memory just like elements of any other array. All rules also apply to this array.

Syntax: `data_type * arrayname[size];`

Example: `int *p[5];`

p is an array of 5 integer pointers.

The following program illustrates their use.

/* Array of pointers */

```
#include<stdio.h>
main()
{
    int arr[3] = {1,2,3};
    int i, *ptr[3];
    for(i=0;i<3;i++)
        ptr[i] = (arr + i);   /*Assigning addresses of elements */
    for(i=0;i<3; i++)
    printf("\n address = %u value = %d", ptr[i], *ptr[i]);
}
```

Output

```
address = 5804    value = 1
address = 5806    value = 2
address = 5808    value = 3
```

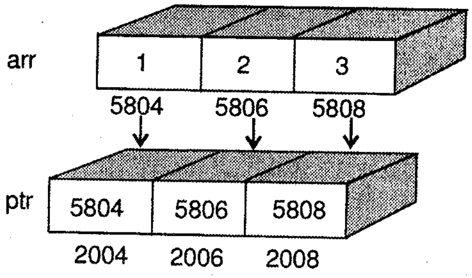


Figure 6.5: Array of pointers

Initializing Array of Pointers

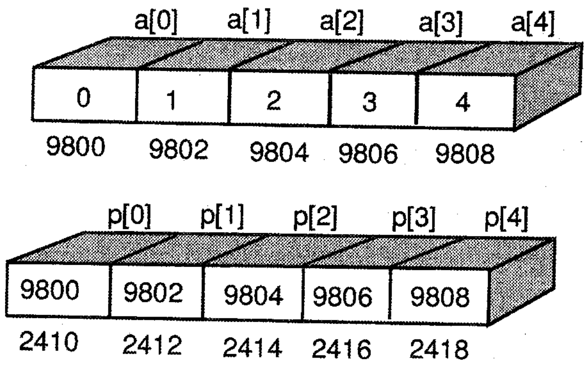
An array of pointers can be initialized during declaration as illustrated by the following examples:

Examples

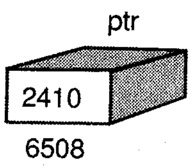
1. Array of pointers to integers.

```
static int  a[] = {0,1,2,3,4};
static int *p[] = {a,a+1,a+2,a+3,a+4};
int **ptr = p;
```

These three lines of code can be pictorially represented as shown.



(a) Array of pointers



(b) Pointers to pointer

Figure 6.6

2. Array of pointers to strings.

```
char * message[6] = {"Pointers","are","interesting","but","need", "practice"};
for(i=0; i<6; i++)
  printf("%s", message[i]);
```

The array of pointers is represented as illustrated below.

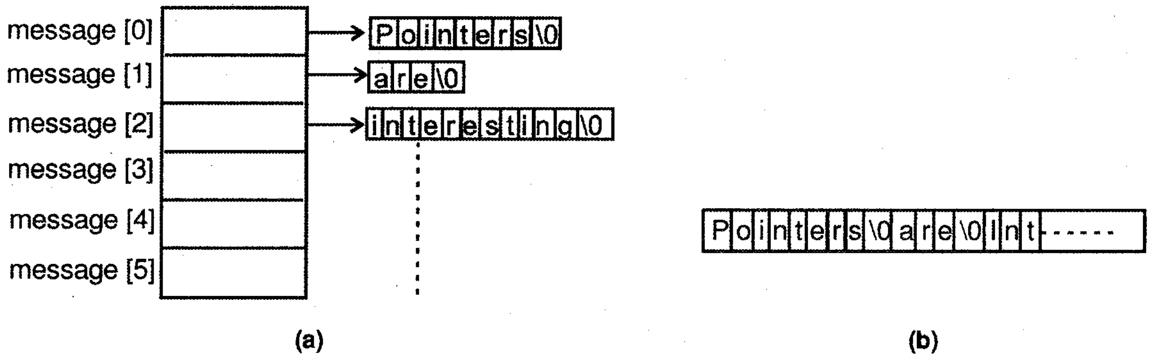


Figure 6.7: Array of pointers to strings

These strings are stored in memory consecutively.

This array can be passed to a function as shown below:

```
print_words(message,6);         /* Call to function */
void print_words(char *p[],int n)
{  int i;
   for(i = 0; i<n;i++)
      printf("%s",p[i]);
}
```

SOLVED PROGRAMS

PU
Oct. 2008 – *5 M*

1

1. **Using pointers accept two strings and store concatenation of these two strings without using library functions.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char *s1;
   char *s2;
   char *t1;
   int i,j;
   clrscr();
   printf("\n Enter the First String");
   scanf("%s", &s1);
   printf("\n Enter the Second String");
   scanf("%s", &s2);
   for(i = 0; s1[i] != '\0';i++)
   t1[i] = s1(i];
   t1[i] = '    ';
   for(j = 0; s2[j] != '\0';j++)
   t1[i+j+1] = s2[j];
   t1[i+j+2] = '\0';
   printf("\n\n\n\n");
   printf("%s", t1);
   getch();
}
```

2. **What will be the output? Give explanation.**

i.
```
main()
{  int a = 8;
   int *p = &a;
   int i = a / *p;
   printf("%d", i);
}
```

PU
Apr. 2009 – 4 M

Ans

Output is 1

Because, value of a = 8 and p points to a so *p = 8. Therefore 8/8 = 1.

ii.
```
int main(void)
    {
    int num, i;
    num = 5;
      for (i = 0; i < 20; i ++)
      {
        AddOne(&num);
        printf("Final value is %d", num),
        return 0;
       }
      void AddOne(int*myVar)
      {
        (*myVar)++;
      }
    }
```

PU
Oct. 2009 – 4 M

Ans

In the statement Addone (&num) display the one error: - Function Addone should have a prototype, because the function prototype is not specify in the program. Also this program display one warning: Function should return a value, because main () function is specify with int return data type.

iii.
```
main()
{ static int a[]= {10,20,30};
  static int * mess [] = {a,a+1,a+2};
  printf("%d%d%d", sizeof(a), sizeof(mess), sizeof(mess[1]));
}
```

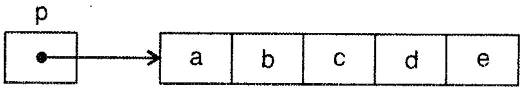
Ans

Output: 6 6 2

1. Here a is an array of 3 integers....sizeof(a) = 6 bytes (Assuming 2 bytes for int)
2. mess is an array of 3 pointers. Each pointer occupies 2 bytes

   ∴ sizeof (mess) = 6
3. mess[1] is the second pointer in the array. Since it stores a single address its size = 2 bytes.

v.
```
main()
      { char *p = "abcd";
        printf("%c",*p++);
        printf("\t%c",*p);
}
```



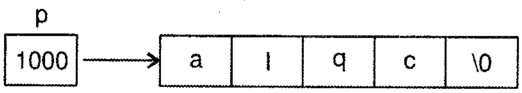
Ans

Output: a b

Consider the expression *p++. There are two unary operators * and ++ which have a R→L associativity. Hence p++ will be done first. But since it is post increment, the old value of p will be used to evaluate * and then p will increment.

p currently points to the first character of the string, i.e., a. After incrementing, it will point to the next character, i.e., b. Hence the second printf will yield b.

vi.
```
main()
    {char *p ="alqc";
    printf("%c...", ++*(++p));
    printf("%c", *++p);
}
```



Output: m...q

p points to beginning of the string, i.e., ++p will increment p to point to next character, i.e., l .p now contains address 1001.

Value at 1001 = l; next ++ operator will increment the value of l.

This value is incremented giving m.

In the second printf, p is incremented to point to the next character, i.e., q.

∴ q is displayed.

vii.
```
void main()
{
   int a = 10;
   void *p = &a;
    int *ptr = p;
    clrscr();
    printf("%u", *ptr);
    getch();
}
```

1
PU
Oct. 2008 – 4 M

Ans

This program display one error and one warning.

Error: Cannot convert void * to int *. That means an assignment, initialization or expression requires the specified type conversion to be performed, but the conversion is not legal.

Warning: p is assigned a value that is never used. That means the variable appeared in an assignment, but is never used anywhere else in the function just ending.

viii.
```
Void main()
{
    int a[3][3][3] = {1, 2, 3, 4, 5, 6};
    printf("%u %u %u %d", a, *a, **a, ***a);
}
```

1 PU Oct. 2008 – 4 M

Ans

This program displays the following output:

65472, 65472, 65472, 1

In this program, we create an array namely a with multidimension and assign only rows value of one dimension. At the print time we are using %u as a control string which is basically used to display the address of memory value. So in this program display the address of first, second and third index of array.

ix.
```
void main()
{
  static int b[6] = {10, 20, 30};
  int i, *k;
  k = &b[3] - 3;
  for (i = 0, i < = 5; i + +)
  {
    printf ("%d", *k);
    k + +;
  }
}
```

1 PU Apr. 2010 – 4 M

Ans

b is integer array which is defined using static storage class so all elements with values zero will be initialized. 3 elements value is given. Others are zero so array b will look like

Subscript	b[0]	b[1]	b[2]	b[3]	b[4]	b[5]
Value	10	20	30	0	0	0
Assumed address	1001	1003	1005	1007	1009	1011

k=&b[3] – 3;

k=1007-3*sizeof(int) = 1007 – 6 =1001(pointer arithmetic, whatever number we add or subtract we have to multiply size of pointer)

so k will point to first element of an array.

i = 0	i < = 5	printf ("%d", *k)	k++	i + +
0	0 < 5 (true)	*k is value at k Value at(1001) 10 will be printed	k=k+1 k=1001+1(sizeof(int)) =1001+2=1003	1
1	1 < 5	*k is value at k Value at(1003) 20 will be printed	k=k+1 k=1003+1(sizeof(int)) =1003+2=1005	2
2	2 < 5	*k is value at k Value at(1005) 30 will be printed	k=k+1 k=1005+1(sizeof(int)) =1005+2=1007	3
3	3 < 5	*k is value at k Value at(1007) 0 will be printed	k=k+1 k=1007+1(sizeof(int)) =1007+2=1009	4
4	4 < 5	*k is value at k Value at(1009) 0 will be printed	k=k+1 k=1009+1(sizeof(int)) =1009+2=1011	5
5	5 = 5	*k is value at k Value at(1011) 0 will be printed	k=k+1 k=1011+1(sizeof(int)) =10011+2=1013	6
6	6 < 5 (false) 6 = 5 (false)	Loop will end		

Output: 102030000

x.
```
void main()
{
char *p = "PROGRAM";
printf("%c\t", *(+ + p));
p - = 1;
printf("%c\t", *(p + +));
}
```

PU Apr. 2010 – 4 M

1

Ans

Subscript	p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]	p[7]
String characters	P	R	O	G	R	A	M	\0
Assumed address	1000	1001	1002	1003	1004	1005	1006	1007

p will point to first letter of string PROGRAM

(++p) will be p = p+1

= 1000+1*(sizeof(char))

= 1000+1*(1) = 1001

*(1001) will be = value at(1001) = R. So R will be printed. After that tab will be printed, cursor moves by 4 spaces.

p – = 1

p = p – 1

p = 1001 – 1*(sizeof(char))

p = 1000

*(1000) will be value at (1000) = P so P will be printed.

Output: = R_ _ _ _P

(_) means blank spaces.

xi.

```
void main()
{
char *s[] = {"Dharma", "Norton", "Simens",
                "ibm"};
char **p;
     p = s;
printf("%s", ++ *p);
printf("%s", *p ++);
printf("%s", ++ *p);
}
```

1

PU
Oct. 2010 – 4 M

Ans

s = {"Dharma", "Norton", "Simens", "ibm"};

D	h	a	r	m	a	\0	N	o	r	t	o	n	\0	S	i	m	e	n	s	\0	i	b	m	\0
1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024

p = s

p will contain starting address of s. *for example,*

s = 1000

p = 1000 therefore p will point to string "Dharma"

printf ("%s", ++*p);

++*p

= ++(value at p)

= ++(1000)

= 1000 + 1(sizeof (char)

= 1001 + 1 = 1001

Therefore this printf will print string "harma".

printf ("%s", *p ++);

*p++

= value at p ++

= 1001++

= 1001+1(sizeof(datatype))// here data type means p is a pointer to pointer. So size 6 will be considered.

= 1001+6 = 1007

The statement in printf is post increment so it will first print the value pointed by p then it will increment the value. So the string first pointed by p will be 'harma' so it will be printed then p will be incremented.

Therefore "harma" will be printed.

Now p is pointing to character 'N'.

printf ("%s", ++ *p);

++*p

= ++(value at p)

= ++(1007)

= 1007+1*(size of(char))

= 1007+1*1 = 1008

= p will be pointed to character 'O'

So string 'orton' will be printed.

Output will be

```
harma
harma
orton
```

EXERCISES

A. Predict the output

1.
```
main()
{
   char *c;
   int *i;
   float *f;
   double *d;
   printf("\n c = %u c+1 = %u", c, c+1);
   printf("\n i = %u i+1 = %u", i, i+1);
   printf("\n f = %u f+1 = %u", f, f+1);
   printf("\n d = %u d+1 =  %u", d, d+1);
}
```

2. main()
```
   {
      char *ptr = "computer";
      printf("%s", p++);
      printf("%c",++*++p);
      printf("%s", ++p);
   }
```

3. main()
```
   { int i = 10, *ptri = &i;
      char *ptrc= "abcd";
      printf("%d\t%d",sizeof(ptri), sizeof(ptrc));
   }
```

4. main()
```
   { char *str = "Pointers";
      int i;
      for(i = 0; i<3; i++)
         printf("%c",*(str+i));
      for(i=6; i>=3; i--)
         printf(("%c", *(str+i));
   }
```

B. Review questions

1. What is a pointer? What is its use?
2. How does the declaration of a pointer variable differ from an ordinary one?
3. Explain address of (&) and indirection operator (*).
4. What are the different operations that can be performed on pointers?
5. How can a pointer to pointer be declared?

Collection of Questions asked in Previous Exams PU

1. What will be the output of the following programs? Give the explanation.

[illegible]

[Oct. 2008 – 4 M]

ii.
```
void main()
{
    int a[3][3][3] = {1, 2, 3, 4, 5, 6};
    printf("%u %u %u %d", a, *a, **a, ***a);
}
```
[Oct. 2008 – 4 M]

2. Using pointers accept two strings and store concatenation of these two strings without using library functions. [Oct. 2008 – 5 M]

3. What will be the output? Give explanation. [Apr. 2009 – 4 M]
```
main()
{  int a = 8;
   int *p = &a;
   int i = a / *p;
  printf("%d", i);
}
```

4. Using pointer write following string manipulation functions: [Apr. 2009 – 10 M]

i. String concatenation ii. String comparison iii. String subtraction

5. What will be the output? Give explanation. [illegible]
```
int main(void)
{
  int num i;
  num = 5;
    for(i = 0; i < 20; i ++)
    {
      AddOne(&num);
      printf("Final value is %d", num),
      return 0;
    }
  void AddOne(int*myVar)
  {
    (*myVar)++;
  }
}
```

6. Find and explain the output of following program:

i.
```
void main() {
    static int b[6] = {10, 20, 30};
    int i, *k;
    k = &b[3] - 3;
    for(i = 0, i < = 5; i + +)
    {
      printf ("%d", *k);
      k++;
    }
}
```
[Oct. 2009 – 5 M]

ii.
```
void main()
{  char *p = "PROGRAM";
   printf("%c\t", *(++p));
   p - = 1;
   printf("%c\t", *(p++));
}
```
[Apr. 2010 – *5 M*]

7. Find and explain the output of following program:

i.
```
void main()
{     char *s[] = {"Dharma", "Norton", "Simens", "ibm"};
      char **p;
             p = s;
      printf("%s", ++*p);
      printf("%s", *p++);
      printf("%s", ++*p);
}
```
[Oct. 2010 – *5 M*]

ii.
```
void main()
{
char *s[] = {"Dharma", "Norton", "Simens", "ibm"};
char **p;
      p = s;
printf("%s", ++ *p);
printf("%s", *p ++);
printf("%s", ++ *p);
}
```
[Oct. 2010 – *5 M*]

8. Write a program to accept five strings from user and display all those strings in descending order.
[Oct. 2010 – *10 M*]

7 Function

7.1 INTRODUCTION

Functions are the building blocks of C and are central to C programming and to the philosophy of C program design.

main() is the function where execution begins. The other functions are executed when they are called directly or indirectly by main.

It is mandatory to have a single main() function in every program. In the following sections, we shall be studying more about main and other functions.

7.2 WHAT IS A FUNCTION?

The program development cycle includes problem analysis, problem definition, design and coding. The code is a set of instructions in a logical sequence, which performs the specified task. 'Real world' applications programs are large and complex. Therefore it is more logical and convenient to break-up the task into smaller, compact and more manageable modules, called functions.

Definition

A function is a named, independent or self-contained block of statements that performs a specific, well defined task and may return a value to the calling program.

- A function is named. Each function is identified by an unique name and is invoked (or called) using this name.
- A function is independent. It can perform the task on its own. It can contain its own variables and constants to be used only within the function.
- It performs a specific task. A function is given a discrete job to perform as a part of the overall program. The task has to be well defined.
- It can return a value to the calling program. The function can perform execution and optionally returns information to the calling program.

7.3 FUNCTIONS AND STRUCTURED PROGRAMMING

Functions and structured programming are closely related. In structured programming, independent section of program code performs program tasks.

Advantages of Functions

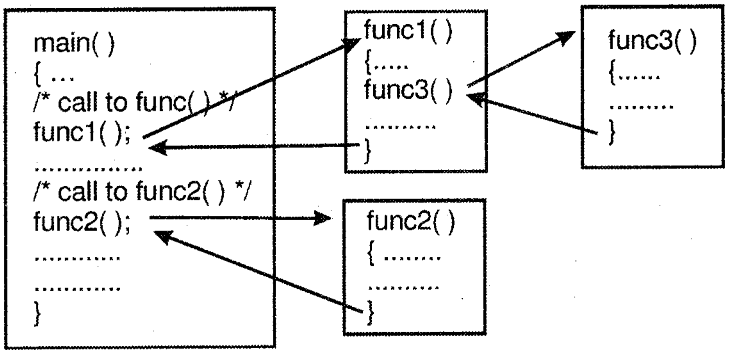
1. Modular or structured programming can be done by the use of functions.
2. By following the top-down approach, the main function can be kept very small and all the tasks can be designated to various functions.
3. Troubleshooting and debugging becomes easier in structured programs.
4. Individual functions can be easily built and tested.
5. Program development becomes very easy.
6. It is easier to understand the program logic.
7. Multiple functions can be developed and tested simultaneously thereby reducing the program development cycle time.
8. A repetitive task can be put into a function that can be called whenever required. This reduces the size of the program.
9. Frequently used functions can be put together in a customized library.
10. A function can call other functions. It may even call itself. This technique called recursion is very useful in solving complex problems and in writing a compact code.

7.4 HOW A FUNCTION WORKS?

A C program does not execute the statements in a function until the function is invoked or called. When the function is called, control passes to the function and returns back to the calling part after the execution of function is over.

The calling program can send information to the functions in the form of argument. An argument stores data needed by the function to perform its task. Functions can send back information to the program in the form of a return value.

Function calls and returns can be illustrated by the following example:



main( ) calls func1( ) and func2( ); func1( ) calls func3( )

Figure 7.1

Note: A function can be called as many times as needed and can be written and called in any order.

7.5 LIBRARY AND USER DEFINED FUNCTIONS

In a C program, functions are of two types:

i. Pre-defined functions or library functions

ii. User defined functions

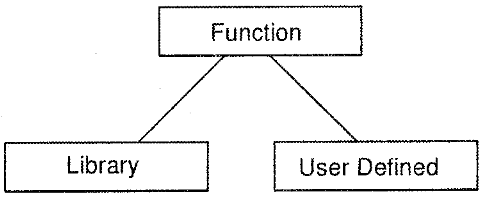


Figure 7.2

The pre-defined or library functions are pre-written, compiled and placed in libraries. They come along with the compiler.

User defined functions are written by the user and the user has the freedom to choose the name, arguments (number and type) and return data type of the function.

One of the greatest feature of C is that there is no conceptual difference between the user defined functions and library functions. A user can write functions, collect them and put them into a library, which can be used by anyone.

In this chapter, we shall be mainly studying user defined functions.

Standard Library Functions

Some commonly used library functions are given in the table below. We shall be using some of them in the later chapters. To use a library function in a program, its corresponding header file must be included in the program.

i. stdio.h

Function	Prototype	Purpose
getchar	`int getchar(void)`	gets a character from stdin
putchar	`int putchar(int c)`	writes a character to stdout
gets	`char *gets(char *)`	gets a string from stdio
puts	`int puts(const char *)`	outputs a string to stdout
printf	`int printf(const char* format, [arg, …]);`	writes a character to stdout
scanf	`int scanf(const char * format,[address, ….]);`	scans and formats an input from stdin
sprintf	`int sprintf(char* buffer, char * format, [argument , ….]);`	writes formatted output to a string
sscanf	`int sscanf(const char * buffer, const char * format , [address, …]);`	scans and formats input from a string
fflush	`int fflush(file *);`	flushes a stream

ii. math.h

Function	Prototype	Purpose
abs	`int abs(int x)`	Returns the absolute value of x
cos	`double cos(double x)`	Returns cosine of x (x is in radians)
exp	`double exp(double x)`	Calculates e^x
floor	`double floor(double x)`	Returns the largest integer < = x
log	`double log(double x)`	Returns natural log of x
pow	`double pow(double x, double y)`	Calculates x^y
sin	`double sin(double x)`	Calculates sine of x
sqrt	`double sqrt(double x)`	Calculates square root of x

iii. conio.h

Function	Prototype	Purpose
clrscr	`void clrscr(void)`	Clears the text mode window
clreof	`void clreof(void`	Clears to end of line in text window
getch	`int getch(void)`	Gets a character from console. No echoing
getche	`int getche(void)`	Same as getch but echoes to screen. No buffering is done
kbhit	`int kbhit(void)`	Returns an integer corresponding to a keystroke
putch	`int putch(int ch)`	Outputs a character to the text window on screen

iv. stdlib.h

Function	Prototype	Purpose
atof	`double atof(const char *s)`	Converts a string to float
atoi	`double atoi(const char *s)`	Converts a string to int
atol	`double atol(const char *s)`	Converts a string to long
random	`int random(int num)`	Returns an integer between 0 and (num-1)
randomize	`void randomize (void)`	Initialize the random number generator with a random value
system	`int system(const char * command)`	Used to execute an MS-DOS command

7.6 FUNCTION DECLARATION AND DEFINITION

Just as variables used within a program have to be declared, so as the functions. The function declaration is called the **function prototype** and it provides the following information to the compiler:

- The name of the function
- The return data type (optional, default is integer)
- The number and type of arguments that will be passed to the function

(The argument name need not be specified).

A prototype should always end with a semicolon.

Syntax: `return_type function_name(type arg1, type arg2 ...);`

Examples

1. `int sum(int a, int b, int c);     OR int sum(int, int, int);`
2. `void display(void);`
3. `double square(double number);`

Function Definition

The function definition is the actual function. The definition contains the code that will be executed. The first line of the definition called the **function header** should be identical to the function prototype with the exception of the semicolon. The argument names have to be specified here.

7.7 WRITING A FUNCTION

Each function definition has the following form:

```
return_type function_name(parameter list)
{
  declarations;
  statements;
}
```

7.7.1 The function header

The first line of every function is the function header, which has three components.

i. **The function return type:** This specifies the data type that the function returns to the calling program. If the function does not return a value, the return data type of void is used.

Examples:
```
int  func1(....)    /* Returns an integer value */
float func2(....)   /* Returns a type float */
void func3(....)    /* Returns nothing */
```

ii. **The function name:** The function name can be any valid C identifier. The function name has to be unique and it should be preferably named so as to reflect the purpose of the function.

iii. **The parameter list:** Function parameters are the means of communication between the calling and the called functions. They can be classified as:

- Formal parameters (or parameters), which are given in the function header.
- Actual parameters (or arguments) which are specified in the function call.

Each function has to declare the type and name of the parameter. Commas separate multiple parameters. For each argument passed in the function call there has to be corresponding parameter in the parameter list in the function headers with the same data type and the order in which arguments are sent.

Examples

1.
```
main()
{ int x,y, result;
result = sum(x,y);}      /* function call */   }
int sum(int a, int b)    /* function definition */
{return a + b};
```

In this example, sum is a function accepting two integers and returning an integer. x and y are the actual parameters. a and b are the formal or dummy parameters.

2.
```
float area(float radius)
```
area is a function returning a float and accepts one float argument.

3.
```
int max(int a, int b, int c)
```
max is a function accepting three integers and returning an integer.

4.
```
int random(void)
```
This function returns an integer but takes no arguments.

7.7.2 The function body

The function body is enclosed in braces and immediately follows the function header. It consists of,

i. **Declarations:** You can declare and initialize variables within a function. These are called local variables, which means that they can be used only within that function.

iv. stdlib.h

Function	Prototype	Purpose
atof	`double atof(const char *s)`	Converts a string to float
atoi	`double atoi(const char *s)`	Converts a string to int
atol	`double atol(const char *s)`	Converts a string to long
random	`int random(int num)`	Returns an integer between 0 and (num-1)
randomize	`void randomize (void)`	Initialize the random number generator with a random value
system	`int system(const char * command)`	Used to execute an MS-DOS command

7.6 FUNCTION DECLARATION AND DEFINITION

Just as variables used within a program have to be declared, so as the functions. The function declaration is called the **function prototype** and it provides the following information to the compiler:

- The name of the function
- The return data type (optional, default is integer)
- The number and type of arguments that will be passed to the function

(The argument name need not be specified).

A prototype should always end with a semicolon.

Syntax: `return_type function_name(type arg1, type arg2 ...);`

Examples

1. `int sum(int a, int b, int c);` OR `int sum(int, int, int);`
2. `void display(void);`
3. `double square(double number);`

Function Definition

The function definition is the actual function. The definition contains the code that will be executed. The first line of the definition called the **function header** should be identical to the function prototype with the exception of the semicolon. The argument names have to be specified here.

7.7 WRITING A FUNCTION

Each function definition has the following form:

```
return_type function_name(parameter list)
{
  declarations;
  statements;
}
```

7.7.1 The function header

The first line of every function is the function header, which has three components.

i. **The function return type:** This specifies the data type that the function returns to the calling program. If the function does not return a value, the return data type of void is used.

Examples:

```
int  func1(....)    /* Returns an integer value */
float func2(....)   /* Returns a type float */
void func3(....)    /* Returns nothing */
```

ii. **The function name:** The function name can be any valid C identifier. The function name has to be unique and it should be preferably named so as to reflect the purpose of the function.

iii. **The parameter list:** Function parameters are the means of communication between the calling and the called functions. They can be classified as:

- Formal parameters (or parameters), which are given in the function header.
- Actual parameters (or arguments) which are specified in the function call.

Each function has to declare the type and name of the parameter. Commas separate multiple parameters. For each argument passed in the function call there has to be corresponding parameter in the parameter list in the function headers with the same data type and the order in which arguments are sent.

Examples

1.
```
main()
{ int x,y, result;
result = sum(x,y);}      /* function call */   }
int sum(int a, int b)    /* function definition */
{return a + b};
```

In this example, sum is a function accepting two integers and returning an integer. x and y are the actual parameters. a and b are the formal or dummy parameters.

2.
```
float area(float radius)
```
area is a function returning a float and accepts one float argument.

3.
```
int max(int a, int b, int c)
```
max is a function accepting three integers and returning an integer.

4.
```
int random(void)
```
This function returns an integer but takes no arguments.

7.7.2 The function body

The function body is enclosed in braces and immediately follows the function header. It consists of,

i. **Declarations:** You can declare and initialize variables within a function. These are called local variables, which means that they can be used only within that function.

Example:
```
float area(float radius)
{ float result;
  const float pi = 3.142;
  ...... /* function code */
  ......    }
```

ii. **Function statements:** These statements perform the specified task. There is no limitation on the statements that can be included within a function.

However, another function cannot be defined in a user-defined function.

iii. **The return statement:** The keyword **return** is used to terminate the execution of the function and return program control to the calling program.

Syntax: `return;`

Example:
```
if(n<0)
return;
```

It is also used to return a value to the calling program. (A function can accept any number of values but can send back only one).

Syntax: `return(expression);`

OR

`return expression;`

Example:
```
return(0);
return(a+b);
return ++i;
```

A return statement at the end is optional for functions not returning a value. There may be multiple return statements within a function but only the first return statement encountered during control flow will be executed.

Example:
```
int max(int a, int b)
{    if(a>b)
     return a;
     else
     return b; }
```

7.8 CALLING A FUNCTION

A function can be called by two ways:

i. Any function can be called by simply using its name and arguments alone in a statement as shown. If the function has a return value, it is discarded.

Example:
```
disp_message();
display_value(x);
```

ii. The second method can be used only with functions that return a value. Since they return a value, they can be used anywhere. A C expression can be used in a printf statement, on the right side of an assignment operator, etc.

Here are some examples.

i. `printf(“Square of %d is % d”,x, square(x));`

ii. `area = calculate_area(radius);`

iii. `Sum_of_all  = sum(a,b) + sum(c,d);`

iv.
```
if(sum(a,b)>100)
{
  /* statements */ }
```

v. `maximum  = max(a,b);`

vi. `max_of_three  =  max(c, max(a,b));`

7.9 TYPES OF FUNCTIONS

7.9.1 Functions with No Arguments and No Return Values

Types of Functions:
i. Functions with no arguments and no return values
ii. Functions with arguments and no return value
iii. Function accepting arguments and returning a value

These functions do not take any information from the calling function nor do they pass back any value. Such functions are commonly used to display messages.

Examples

1.
```
#include<stdio.h>
main()
{ void greet(void);   /* function prototype */
  greet();            /* function call */
}
void greet(void)      /* function definition */
{ printf("\n Hello and welcome to C"};
```

2.
```
#include<stdio.h>
main()
{ int n ;
  void error_msg(void);
  printf("Enter the value of n : ");
  scanf("%d",&n);
  if(n<0)
  { error_msg();
    exit();
  }
  .....
  .....
}
void error_msg(void)
{ printf("Error! Negative value");
}
```

7.9.2 Functions with Arguments and No Return Value

Here, the function accepts arguments but does not return any value back to the calling program. It is a one way communication, i.e., calling program to function.

In such functions, the result of operations on the arguments may be displayed from the function itself.

Program: /* Demonstrate functions */

/* Calculate and display the area of a circle */

```
#include<stdio.h>
main()
{  float radius;
   void area(float);  /* function prototype */
   printf("Enter the radius : ");
   scanf("%f", &radius);
   area(radius);
}
   void area(float r)
   {  float result ;
      const float pi = 3.142;
      result = pi*r*r;
      printf("The area is %f", result);
   }
```

7.9.3 Function accepting Arguments and Returning a Value

Such a function accepts information and also returns back a value to the calling program. Thus, there is a two way communication between the two.

Example

We shall modify the above program such that the function area now returns the calculated value back to main.

/* Illustrate function returning a value */

```
#include<stdio.h>
main()
{  float radius, a;
   float area(float);
   printf("Enter the radius : ");
   scanf("%f", &radius);
   a = area(radius);
   printf("\n The area is %f", a);
}
float area(float r)
  {  const float pi = 3.142 ;
     return(pi*r*r);
  }
```

7.10 METHODS OF PASSING ARGUMENTS

There are two mechanisms to pass arguments to a function.

i. Call by value

ii. Call by reference

In C all function arguments are passed by value. **C language does not support call by reference.**

7.10.1 Call by value

- In this method, the value of the actual parameters gets copied into the corresponding formal parameters.
- Any changes made to the formal parameters will not affect the actual parameters.

In order to illustrate call by value, let us write a function which interchange two numbers.

Program: Interchanging two numbers

```
void main()
{
  int a=10, b=20;
  void swap(int, int);    /* declaration */
  printf("Before interchange a=%d, b=%d", a,b);
    swap(a,b);            /* call */
  printf("\n After interchange a=%d, b=%d", a,b);
}
void swap(int x, int y)
{ int temp;
  temp=x; x=y; y=temp;
  printf("In function x=%d, y=%d", x, y);
}
```

Output

```
Before interchange a=10, b=20
In function x=20, y=10
After interchange   a=10, b=20
```

In this program, the values of the actual parameters, a and b get copied into two different variables x and y (formal parameters). These formal variables exist only in the function swap. Hence, any changes made to these formal parameters will not be made to the actual parameters.

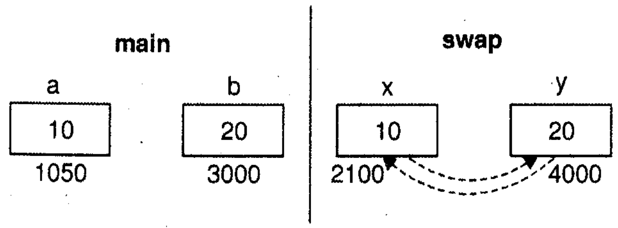


Figure 7.3

7.10.2 Call by reference

In this method of passing arguments, the called function has access to the original argument, not the local copy. Languages like Pascal and Fortran allow this method.

Although C language allows passing of arguments only by value, the call by reference method can be simulated by the use of addresses and pointers. This allows the function to directly access the original variables and modify their values.

We will rewrite the program to interchange two numbers but in a slightly different way.

Program: Simulating call by reference to swap two numbers

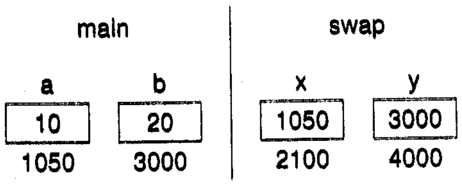
```
#include<stdio.h>
void main()
{  int a=10, b=20;
   void swap(int *x, int *y);
   printf("Before swapping a=%d b=%d", a,b);
   swap(&a, &b);
   printf("\n After swapping a=%d b=%d", a,b);
}
void swap(int *x, int *y)
{  int temp;
   temp=*x;  *x=*y;  *y=temp;
}
```

Output

```
Before swapping a=10    b=20
After swapping  a=20  b=10
```

In call by value method, the function cannot access the original variables. It can only access duplicate or copied variables. In order to make changes to the original variables, the function must get access to the actual variables and not their copies. This is possible if we send the **address** of the variable to the function rather than its value. Since the address of any variable is unique, the function will access the original variable.

The addresses of a and b are obtained using the & operator (address of operator). The addresses are stored in 2 special variables x and y which are declared as 'pointer' variables and they store the addresses of a and b respectively. *x and *y are the values in a and b respectively. Hence, any changes made to *x and *y will change a and b as well.



7.11 FUNCTIONS WITH VARIABLE ARGUMENTS

It is possible to declare functions with variable numbers of arguments. Such functions are called "Variable" functions. Some standard library functions can accept a variable list of arguments (such as printf).

A function is also defined as variable using an ellipsis ('...') in the argument list. The function is called by passing fixed arguments followed by the additional variable arguments.

Example:
```
int func1(int x, …)
{
}
```

Here, func1 is a function with one fixed argument and the ellipsis indicates variable arguments.

Accessing Variable Arguments

Since variable arguments have no names, they must be accessed sequentially using special macros from "stdarg.h". These macros are:

i. va_list
ii. va_start
iii. va_end

Example

```
int addnos(int count,…)
{ va_list ab;
  int i, sum;
  va_start(ab, count); /*Initialize the argument list*/
  sum = 0 ;
  for(i = 0; i < count; i++)
     sum = + va_arg(ab,int); /*Get next argument*/
  va_end(ab);    /*clean up*/
  return sum;
}
main()
{ printf("%d\n", addnos(3,5,5,6));
  /* This prints 16 */
  printf("% d\n",addnos(5,10,20,30,40,50)); /* This prints 150*/
}
```

7.12 ARRAYS AND FUNCTIONS

Arrays can be passed to a function in two ways.

i. Element by element
ii. Passing the entire array

When the elements are passed to the function, their values are copied into the corresponding function parameters and cannot be modified by the function.

An entire array can be passed as an argument to the function. The function gets complete access to the original array rather than a copied array.

Before we see how it can be done, it is important to understand the importance of the array name.

7.12.1 Name of the Array

Consider the declaration

```
int n[10];
```

The name of the array is n. The name n without any subscript refers to the starting address or base address of the array in memory, i.e., it refers to the memory location of the 0th element of the array.

Thus, n is equivalent to &n[0]. It is a **constant pointer** and cannot be changed (i.e., incremented / decremented, etc.).

When we want to pass the array n to a function sum_of_elements, the call will be given as

```
sum_of_elements(n);
```

This makes the complete array available to the function and the function can make modifications to the original array.

This is so because by specifying only the array name, we are in effect sending the base address of the array to the function.

Thus, the name of the array can be used effectively as any other pointer to the array in accessing the array elements. This is illustrated below.

Program: /* Illustrate array name as a pointer to the array */

```
main()
{ int a1[5] = {10,20,30,40,50};
  int i;
  for(i=0;i<5;i++)
    printf("%u contains %d\n",(a1+i),*(a1+i));}
```

Output

```
65492contains 10
65494contains 20
65496contains 30
65498contains 40
65500contains 50
```

Since a1 contains the base address, a1+0 points to the 0th element and *(a1+0) gives the value of the 0th element.

7.12.2 Passing the Array Element by Element

The array elements can be passed one-by-one to the function. The function thus gets access only to one element at a time and cannot modify this value. Example as follows:

```
main()
{    int n[5] = {10,20,30,40,50};
     void display(int);   /* function prototype */
     int i;
     for(i=0;i<5;i++)
     display(n[i]); }
  void display(int x)
  { printf("%d",x);  }
```

7.12.3 Passing the Entire Array

In order to pass the entire array, we just have to send the name of the array to the function. Since the name contains the base address, we are effectively sending the entire array. The function gets access to the original array (since it has the base address). It can modify the array contents. Example is as follows:

```
main()
{  int n[5] = {10,20,30,40,50};
   void modify(int b[5]);       /*Function declaration */
   modify(n);  }
void modify(int b[5])
{  int i;
   for(i=0;i<5;i++)
   b[i] = b[i] * 2; }
```

In this example, the function modifies the contents of the array n, they are changed to 20,40,60,80 and 100.

Note: b is not a new array created but it is the same as n because b stores the base address of n.

7.12.4 Passing Dimensional Array to Function

The following program illustrates passing of one dimensional array to a function to find the largest number from the array.

1. /* Passing an array to a function */

```
#include<stdio.h>
main()
{  int array[50],i,n;
   int largest(int x[],int n);  /*function prototype */
   printf("\n How many elements?:");
   scanf(%d",&n);
   /*Accept values */
   printf("\n Input values : \n");
   for(i=0;i<n; i++)
     scanf("%d", &array[i]);
   /*Function call and display result */
   print("\n Largest number = %d", largest(array,n));  }
   int largest(int x[], int n)
{  int j, large = x[0];     /* initialize */
     for(j=1;j<n; j++)       /* compare remaining elements */
     { if(x[j] > large)
          large = x[j]; }
   return(large);}
```

Output

```
How many elements? : 5
Input values:
100
0
-90
2000
48
Largest number = 2000
```

Note: The function header could also be written as

```
int largest(int *x, int n)
```

In both the cases, i.e., int x[ ] and int *x, x means a pointer to an int. The loop inside the function could also be written as

```
for(i=1;j<n;j++)
  {  if(*(x+j)>large)
     large = *(x+j);
  }
```

In the above programs, we have passed a 1D array to the function. In the same way, we can pass a 2D array to the function as shown below. Program to accept, multiply and display matrices using functions.

2. /* Illustration of 2D Arrays and functions */

```
# include<stdio.h>
main()
{  int verify(int x,int y);
   void readmat(int x[10][10], int r, int c );
   void multmat(int x[10][10], int y[10][10], int z[10][10]; int r1, int c1,
                int c2);
   void dispmat(int m[10][10], int r, int c);
   int a[10][10],b[10][10], c[10],[10], r1, r2, c1, c2;
   printf("\n Number of rows and columns in matrix A:");
   scanf("%d%d", &r1, &c1);
   printf("\n Number of rows and columns in matrix B:");
   scanf("%d%d", &r2, &c2);
   if(verify(c1,r2)==1)
   {  printf("\n Multiplication Possible \n");
      printf("\n Input Matrix A \n");
      readmat(a,r1,c1);
      printf("\n Input Matrix B \n");
      readmat(b,r2,c2);
      /*Multiply A and B giving C */
      multmat(a,b,c,r1,c1,c2);
      /* display result */
      printf("\n The resultant matrix is \n");
      dispmat(c, r1, c2);  } /*  end if */
      else
{printf("Columns of A must be equal to rows in B\n");
 printf("\n Multiplication not possible"); }  }
  void readmat(int x[10][10], int r, int c)
  { int i, j;
    for(i=0; i<r; i++)
      for(j=0; j<c; j++)
        scanf("%d", &x[i][j]);    }
  int verify(int x, int y)
  { return(x==y);    }
  void dispmat(int m[10][10], int r, int c)
  { int i, j;
    for(i=0; i<r; i++)
    { for(j=0; j<c; j++)
      printf("%5d",m[i][j]);
      printf("\n"); }    }
  void multmat(int x[10][10], int y[10][10],
    int z[10][10], int r1,int c1, int c2)
  { int i,j,k;
    for(i=0;i<r1,i++)
    for(j=0;j<c2;j++)
      { z[i][j]= 0;
        for(k=0; k<c1;k++)
          z[i][j] = z[i][j] + x[i][k] * y[k][j];  }  }
```

Output a

```
Number of rows and columns is matrix A : 2     3
Number of rows and columns in matrix B :  2     2
Columns of A must be equal to rows in B
Multiplication not possible.
```

Output b

```
Number of rows and columns in matrix A :  2     2
Number of rows and columns in matrix B : 2     3
Multiplication possible
Input Matrix A
1  1
1  2
Input Matrix B
2  3  1
4  5  1
The resultant matrix is
6   8  2
10  133
```

7.13 POINTERS AND FUNCTIONS

7.13.1 Pointer as a Function Argument

When we pass a variable to a function, its value is passed to the function which gets copied into another variable. Any changes made in the function will be made to the function variable and not the original variable.

To modify the value of a variable in the function, we have to pass its address to the function. When the address of a variable is passed to a function, it has to be stored in a pointer variable. This pointer allows access to the original variable from the function.

The following program shows how to modify the values of two variables using a function:

Program: Pointer as function argument

```
#include<stdio.h>
void main()
{
   void modify(int *, int *);   /* prototype */
   int n1=10, n2=20;
   printf("Enter the two numbers");
   scanf("%d %d",&n1,&n2);
   modify(&n1, &n2);
   printf("\n The modified values are %d %d", n1, n2);
}
void modify(int *ptrn1 ,int *ptrn2)
{
   *ptrn1=50;
   *ptrn2=100;
}
```

Function to Calculate Area and Circumference of Circle and Display the Results in Main

In order to do this, we will have to pass the addresses of variables to store area and circumference from main to the function.

```
void calculate(float r, float *area_ptr, float *circum_ptr)
{
    *area_ptr=3.142*r*r;
    *circum_ptr=2*3.142*r;
}
```

In main, the function will be called as :

```
calculate(radius, &area, &circum);
```

7.13.2 Function Returning a Pointer

A function can return a pointer to the calling function. The function header has to be declared as

```
pointer_datatype * function_name(parameter list)
```

Example

i. `int *f1(int);`

   f1 is a function accepting an integer and returning pointer to an integer.

ii. `char *f2(int *, int *);`

f2 is a function returning a pointer to datatype char and accepting the addresses of two integer arguments in two integer pointers.

Program: /* This program accepts the addresses of two integer variables and returns the address of the larger variable to main */

```
#include<stdio.h>
main()
{
  int *larger(int *, int *);  /* prototype */
  int n1, n2,*max;
  printf("Enter the two numbers:");
  scanf("%d %d", &n1,&n2);
  max = larger(&n1, &n2);
  printf("\n The larger value is %d", *max);
}
int *larger(int *ptrn1,int *ptrn2)
{
  if(*ptrn1 > *ptrn2)
     return(ptrn1);
  else
     return(ptrn2);
}
```

Output

```
Enter the two numbers: 10     20
The larger value is 20.
```

7.13.3 Pointers to Functions

A confusing yet powerful feature of C is the function pointer.

Even though a function is not a variable, it still has a physical location in memory.

A function's address is the starting address of the code of the function in memory. This address assigned to a pointer is the entry point of the function. The pointer can then be used in place of the function name. It also allows function to be passed as arguments to other functions.

- **Function Name:** It is a constant pointer pointing to the block of memory where it is stored.
- **Declaring Pointer to a function:** The **syntax** for declaring a pointer to a function is:
  `return_type(* pointer_variable)(function's argument_list);`

  The * along with the pointer_name acts as the function name.

  Example

  ```
  int(*ptr) (int,int);/* ptr is a pointer to a function accepting two
  integers values and returning an integer */
  ```

- **Assigning function address to a pointer:** The address of a function can be obtained by only specifying its name without parenthesis.

The following program illustrates the concepts discussed above:

1. /* Illustrates pointers to functions */

```
#include<stdio.h>
int f1(int x,int y)
{
  return((x>y)?x :y);
}
void f2(float f)
{
  printf("\n The value is %f", f);
}
main()
{
  int(*ptr1)(int,int);  /*declaring pointers*/
  void(*ptr2)(float);   /*to functions*/
  int n1,n2 ;
  float f=50.752;
  ptr1=f1;        /*assign address of function*/
  ptr2=f2;        /*to pointer*/
  printf("Enter the two numbers:");
  scanf("%d%d", &n1,&n2);
  printf("\n The larger is %d",ptr1(n1,n2)); /*function call*/
  ptr2(f); /*call to function f2 using pointer*/
}
```

Output

```
Enter the two numbers : 10    20
The larger is 20
The value is 50.752000
```

In the following program, we shall calculate the factorial of a number using a pointer to function.

2. /* Illustrates pointer to function */

```
#include<stdio.h>
unsigned int fact(int n)
{
  if(n<=1)
    return(1);
  else
    return(n*fact(n-1));
}
  main()
{ unsigned int(*ptr)(int);
/*pointer to function with prototype of fact*/
  unsigned int ans;
  int n;
  ptr  = fact;  /*assign address of fact to ptr*/
  printf("Enter the number whose factorial is required : ");
  scanf("%d",&n);
  ans=ptr(n); /*call to function fact using ptr*/
  printf("\n The result is %u",ans);
}
```

Output

```
Enter the number whose factorial is required : 5
The result is 120
```

7.14 RECURSION

Recursion is a process by which a function calls itself either directly or indirectly. It is called circular definition. Direct recursion is when a statement in the body of the function calls itself. Indirect recursion occurs when the function calls another function, which in turn makes a call to the first one. They are commonly used in applications in which the solution to a problem can be expressed in terms of successively applying the same solution to subset of the problem. *Two important conditions should be satisfied by any recursive function.*

i. Each time the function is called recursively it must be closer to the solution.
ii. There must be some terminating condition, which will stop recursion.

There are many examples of recursion. One of the most common example is the calculation of the factorial of a number. *The factorial can be stated as*:

a. The factorial of 0 is 1 and the factorial of any positive integer is the product of all integers from 1 to n.

b. The factorial of 0 is 1 and the factorial of any positive integer n is the product of n and the factorial of number n-1.

The first definition is iterative while the second is recursive and represented as

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

and so on. It continues till n becomes 1. This is where the recursion terminates.

1. /* Using a recursive function to calculate factorial */

```
#include<stdio.h>
main()
{ unsigned int num;
  unsigned int factorial(int n);
  printf("\n Enter the value of the number:");
  scanf(("%d", &num);
  printf("\n The factorial of %d is %u", num, factorial(num));
}
unsigned int factorial(unsigned int n)
{
  if(n==0 || n==1)
      return(1);
  else
      return(n* factorial(n-1));
}
```

Output

```
Enter the value of the number: 3
The factorial of 3 is 6.
```

The function calls are depicted below:

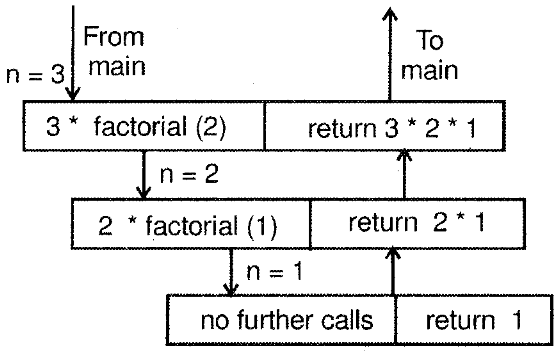


Figure 7.4

i.e., 3! = 3 * factorial (2)
= 3 * 2 * factorial (1)
= 3 * 2 * 1
= 6

Advantage

Recursive code is much more compact and often much easier to write and understand than the non-recursive equivalent.

Disadvantages

Recursive functions may not provide saving in storage since a stack of values being processed has to be maintained by the system.

It will not be faster than iterative functions because function calls and returns take longer.

More Examples of Recursion

1. Computation of Fibonacci series

   0, 1, 1, 2, 3, 5, 8, ……

   Each element in this sequence is the sum of the two preceding elements. The series can be defined by the relations.

   fib (n) = n if n == 0 or n==1

   fib (n) = fib (n-2) + fib (n-1) if n>=2

The following program displays the first 'n' fibonacci numbers using a recursive function to calculate the n^th^ fibonacci number.

2. **/* Fibonacci series */**

```
#include<stdio.h>
main()
{ int num, i;
  unsigned int fib(int);       */ function prototype */
  printf("How many numbers: ");
  scanf("%d",&num);
  printf("\n The first %d, fibonacci numbers are : \n" num);

  /* display the n numbers */
  for(i=0; i<num, i++)
    printf("%u\t",fib(i));
  }
  unsigned int fib(int n)
  { if(n<=1)
      return(n);
    return(fib(n-2) + fib(n-1));
  }
```

Output

```
How many numbers   :  5
The first 5 fibonacci numbers are:
0  1  1  2  3
```

The recursion tree in the calculation of the fifth fibonacci number is:

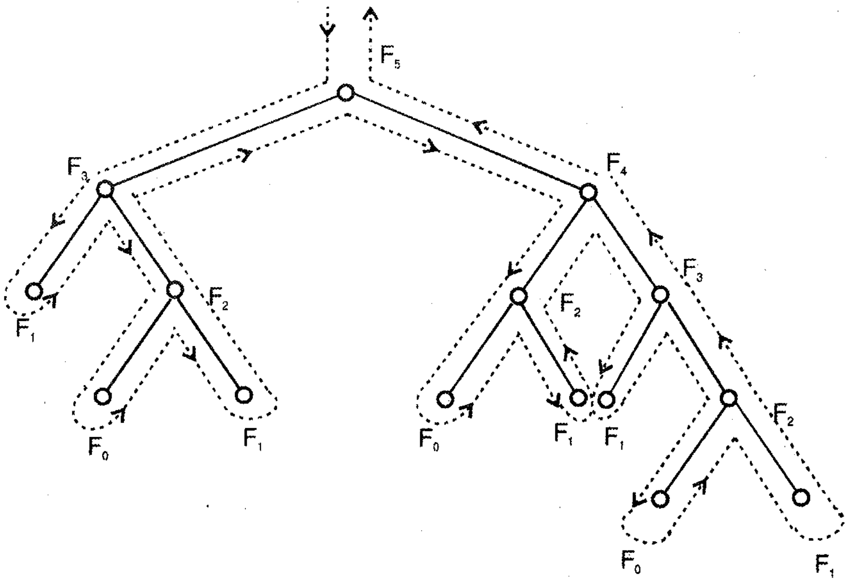


Figure 7.5: Recursion tree

3. The recursive relation can define calculation of greatest common divisor (GCD) of two positive integers.

   gcd (x,y) = x, if y ==0

   gcd (x,y) = gcd (y,x%y), otherwise

   The recursive function can be written as:

```
int gcd(int x, int y)
{
  if(y==0)
    return(x)
  else
  return(gcd(y,x%y));
}
```

   and it can be used in main as

4. /* Calculation of GCD using above function */

```
#include<math.h>
#include<stdio.h>
main()
{ int a, b;
  printf("Enter two numbers:");
  scanf("%d %d", &a, &b);
  a = abs(a); /* if a is negative, convert it to positive */
  b = abs(b);
  printf("\n The gcd of %d and %d is %d", a,b,gcd(a,b));
}
```

Output

```
Enter two numbers    :  25  20
The gcd of 25 and 20 is 5
```

Note: abs is a function which returns the absolute value of its argument.

5. Write a program to accept a positive integer and displays its equivalent binary number using recursion.

1 PU Oct. 2008 – *5 M*

```
#include<stdio.h>
#include<conio.h>
int convert(int);
int n,sum=0,a,b=1;
void main()
{
  clrscr();
  printf("\n Please enter the number");
  scanf("%d", &n);
  int temp = convert(int n);
  printf("\n The Binary Number = %d", temp);
  getch();
}
int convert(int n)
{
  while(n > 0)
  {
    a = n % 2;
    n = n /2;
    sum = sum + a * b;
    b = b * 10;
  }
  return sum;
}
```

6. Write a recursive function to find the sum of digits of a given integer number in a single digit.

1
PU
Apr. 2009 – *6 M*

```
#include"stdio.h"
#include"conio.h"
int fun(int);
void main()
{
  int num,result;
  clrscr();
  printf("Enter the number : ");
  scanf("%d",&num);
  result=fun(num);
  printf("\nThe sum of a number in single digit :   %d",result);
  getch();
}
int fun(int n)
{
  int r,sum=0,num;
  while(n!=0)
  {
    r=n%10;
    n=n/10;
    sum=sum+r;
  }
  if(sum<=9)
    return(sum);
  else
    num=fun(sum);
  return(num);
}
```

7. Write a C program to accept a number from the user and find out the product of digits of that number using recursion.

1
PU
Apr. 2010 – *5 M*

```
main() [illegible]
{
  int n,z[illegible]
  printf([illegible] number:");
  scanf([illegible]
  z=product(n);
  printf("product of digit using recursion=%d",z);
}
  int product(int x)
  {
    if(x==0)return 1;
    else
    return(x%10 * product(x/10));
  }
```

SOLVED PROGRAMS

1. **Write a program using recursive function to print factorial of a given number.**

PU Oct. 2009 – *5 M*

Ans

```
#include<stdio.h>
#include<conio.h>
int fact(int n);
void main()
{
  int num;
  printf("\n Enter the value of a number");
  scanf("%d", &num);
  printf("\n The Factorial of %d is %d", num, fact(num));
  getch();
}
int fact(int a)
{
  if((n==0)||(n==1))
    return(1);
  else
    return(n * fact(n-1));
}
```

2. **What will be the output? Give explanation.**

PU Oct. 2009 – *4 M*

i.
```
int gNumber;
int main(void)
{
   int i;
   gNumber = 1;
     for(i=1; i<=10; i++)
   {
       gNumber = DoubleIt(gNumber);
       printf("Final value is %d", gNumber);
    }
int DoubleIt(int myVar)
{
  return 2 * myVar;
}
```

Ans

int main (void) statement displays warning "Function should return the value" because no return statement is mentioned in the main function.

In gNumber =Doublet (gNumber) statement display Error "Function should have prototype", because we are declaring function before main.

If we specify the prototype before main () then it display the value, such as

2, 4, 8, 16,32,64,128,
256, 512, 1024, 2048

ii.

```
int f()
main()
{
        f(1);
        f(1,2);
        f(1,2,3);
}
f(int a, int b, int c)
{ printf("%d %d %d", a,b,c);}
```

PU Oct. 2010 – *5 M* 1

Ans

f is a function which accepts 3 integers.

a. Call for function f(1) shows an error too few parameters to call f.

b. Call for function f(1,2) shows an error too few parameters to call f.

c. Call for function f(1,2,3) will be executed and gives output 1 2 3.

Output: 1 2 3

EXERCISES

A. Predict the output

1.
```
main()
{ int i;
  for(i = 1; i<=5; i++)
  { printf("%d",i);
    main();
  }
}
```

2.
```
main()
{ int a = 10, b = 15;
  change(a, &b);
  printf("%d%d", a,b);
}
change(int x, int *y)
{     x  =  20;
      *y =  30;
}
```

3.
```
main()
{ abc(100,200)
}
abc(int n)
{ printf("%d",n);
}
```

4.
```
main()
{ int i = 5, j = 10;
  abc(i,j);
  printf("i = %d", i);
  printf("\n j = %d", j);
}
abc(int i, int j)
{ i = i+j;
  j = i-j;
  i = i-j;
}
```

B. Programming exercises

1. Write a function to calculate the roots of a quadratic equation.
2. Write a function that takes two integer parameters and returns the sum of all integers between them.
3. Write a function power which accepts two integers x and y and returns x^y.
4. Write a function ctoi which accepts a character and returns its integer equivalent if it is a digit and returns –1 otherwise.

   Example: ctoi(ch) should return integer 5 if ch has value '5'.
5. Write a recursive function to calculate and return the sum of digits of a number. *Example:* Sum of digits of 397= 19.
6. Modify the above function such that the sum of digits is a single digit number.

   *Example: S*um of digits of 397 = 1
7. Write a recursive program to find the multiplication of two integers.

C. Review questions

1. Define a function and illustrates how it works.
2. What are the advantages of using functions?
3. What are library and user defined functions?
4. What do you mean by a function prototype?
5. State the different parts of a function? Explain the function header.
6. What are formal and actual parameters?
7. Illustrate with an example function declaration, function definition and function call.
8. What is a local variable? Explain using examples.
9. Explain call by value and call by reference.
10. What is recursion? Explain with examples.
11. What is the meaning of the following declarations?

    i. int f(float, char); ii. void g(int, int , int); iii. double h(void);

Collection of Questions asked in Previous Exams PU

1. Write a program to accept a positive integer and display its equivalent binary number using recursion. **[Oct. 2008 – *5 M*]**
2. Write a recursive function to find the sum of digits of a given integer number in a single digit. **[Apr. 2009 – *6 M*]**
3. What will be the outputs? Give explanation. **[Oct. 2009 – *4 M*]**

```
int gNumber;
int main(void)
{
      int i;
      gNumber = 1;
         for(i=1; i<=10; i++)
      {
         gNumber = DoubleIt(gNumber);
         printf("Final value is %d", gNumber);
      }
   int DoubleIt(int myVar)
   {
         return 2 * myVar;
}
```

4. Write a program using recursive function to print factorial of a given number. **[Oct. 2009 – *5 M*]**
5. Write a C program to accept a number from the user and find out the product of digits of that number using recursion. **[Apr. 2010 – *5 M*]**
6. Find and explain the output of following program. **[Oct. 2010 – *5 M*]**

```
int f()
main()
{
      f(1);
      f(1,2);
      f(1,2,3);
}
f(int a, int b, int c)
{ printf("%d %d %d", a,b,c);}
```

8 Storage Classes And Scope

8.1 MEANING OF TERMS

Every variable in a program has some memory associated with it. Memory for variables is allocated and released at different points in the program.

The **scope** of a variable can be defined as the region or part of the program in which the variable is visible or valid. Visible here also means accessible.

When speaking about scope, the term variable refers to all C data types: simple variables, arrays, structures, pointers, symbolic constants, etc.

Scope also affects a variable's **extent** or **lifetime.**

Extent: This is the period of time during which memory is associated with a variable. In other words, a variable lifetime is how long the variable persists in memory.

Storage class refers to the manner in which memory is allocated by the compiler to variables.

The storage class determines the scope and the lifetime of a variable.

Storage classes are:

i. auto
ii. static
iii. extern
iv. register

We have written a number of programs so far and have not used any of these classes as yet. The reason that the previous programs compile and run is that if no class is mentioned, a default storage class will be assigned depending upon the context in which the variable is used.

8.2 SCOPE

A demonstration of scope

1. /* Illustration variable scope */

```
#include<stdio.h>
main()
{ int n = 5;
  void display(void);       /* function prototype */
  printf("\n %d",n);
  display();
}
void display(void)
{
printf("%d\n",n);
}
```

Output

Compiler error: The variable n is defined within main and is visible only in function main. It cannot be accessed in the function display.

We will now make a small modification to the above program.

2. /* Illustrates variable scope */

```
#include<stdio.h>
int n = 5;
main()
{ void display (void);     /* function prototype */
printf("\n %d",n);
display();
}
void display(void)
{
printf("\n%d",n);
}
```

Output

```
5
5
```

We have made a minor modification in the first program by moving the definition of n outside main ( ). By doing so, we have changed its scope.

In program 1, n is a local variables, i.e., its scope is limited to the block where it is defined.

In program 2, n is a global (external) variable and its scope is the entire program.

Block Scope and File Scope

The scope of an identifier falls under two categories

i. Block scope (or local scope)

ii. File scope

i. **Block Scope:** An identifier is said to have local or block scope if it is defined within a function or a block. It can be used only within that function or block. It cannot be used outside. Such identifiers are called **local identifiers.**

ii. **File Scope:** If an identifier is defined outside a function it can be used in any function in the program, i.e., it has a visibility over the entire file. Such identifiers are called **global identifiers.**

Examples:
```
/* Local and file scope*/

#include<stdio.h>
int n = 20;
main()
{  int m = 10;
   disp_values(); }
void disp_values()
{  printf("%d %d", m,n); }
```

In this program, variable n has **file scope** whereas m has **block** scope. n can be used in any function in the file whereas m can only be used in function main because it has been defined in main.

Advantages of Block Scope

1. Data integrity is preserved since a function cannot access the data of another.
2. Only the necessary data can be passed to a function thus protecting the remaining data.

Advantages of File Scope

1. If some common data is needed by all functions, passing it as parameters will not be feasible. Making it global will be much easier.
2. Any changes made to the global data by a function can be seen and used by other functions.

Disadvantages of File Scope

1. If too many variables are made global, they will remain in memory till program execution is over. Thus, memory will remain allocated even when they are not being used.
2. Any function can modify global data. Hence data cannot be protected.

8.3 STORAGE CLASSES

The storage class of a variable determines

i. where it is stored,

ii. its default initial value,

iii. scope of the variable,

iv. lifetime of the variable,

We shall now study the four storage classes

2 PU Oct.08,Apr.09 – *5M*
* Write short note on Storage classes.

8.3.1 Automatic Storage Class

This is the default storage class of variables that are declared within a function. (All the variables that we have studied in previous chapters belong to this class).

In order to explicity declare a variable which belongs to this class, the keyword auto is used.

Example: `auto int i;`

This variable comes into existence only when the function (where it is defined) is called and ceases to exist after the function is exited; hence termed automatic.

Features

i. Storage : Memory

ii. Scope : Local to the block where it is defined. (Block scope)

iii. Lifetime : It exists as long as control remains in the block where it is defined.

iv. Default initial value : Garbage.

Program: /* Illustrate automatic variables*/

```
#include<stdio.h>
main()
{ auto int i = 10;
  {
    auto int i = 20;
    printf("%d\n",i);
  }
  printf("%d\n",i);
}
```

Output

```
20
10
```

In this program, the two variables i are different variables since they are defined in different blocks.

8.3.2 Extern Storage Class

Variables belonging to this class are also called as global variables or external variables. They are declared outside all functions and are accessible to all the functions in that source code file.

The variable n in (*Refer point 8.2, program 2*) is a global variable. In this program, n is declared outside main( ) which makes it accessible to all the functions in that file.

In some cases, however, the program code may extend over two or more separate files. In such a case, special handling is required for external variables.

Use of Extern Keyword

If the function uses an external variable, it is a good programming practice to declare it again within the function using the extern keyword.

The **syntax** is

```
extern data_type var;
```

Example

/* Illustrates external variables */

```
#include<stdio.h>
int n = 5;          /* definition */
main()
{
  extern int n;     /* declaration */
  void display(void);
  printf("\n %d",n);
  display();
}
void display(void)
{
  extern int n;     /* declaration */
  printf("\n%d",n);
}
```

Note

i. The declaration within the function indicates that the function uses an external variable, which is defined elsewhere.

ii. If both these functions are in the same source code file, the declarations are not required.

iii. If the variable n is to be used in functions written in separate source code files, the declaration using the extern keyword is required.

Features

i.	Storage	:	Memory
ii.	Scope	:	File scope

iii. Lifetime : It exists as long as the program which uses the variable is running. It retains its value between functions.

iv. Default initial value : Zero

Uses of Global Variables

1. Use of global variable simplifies communication, i.e., they need not be passed to functions, (thereby making argument lists shorter) and any function can use them whenever required.
2. Symbolic constants are often declared globally.

Disadvantages

1. By using external variables, the principles of modular programming i.e. data isolation is violated.
2. Even when not required, external variables persist in memory.
3. Variables can be changed in unexpected and inadvertent ways and it is difficult to keep track of the changes made thereby leading to problems.

8.3.3 Static Storage Class

Local variables are automatic by default, which means that every time the function in which they are declared is called, they are created and destroyed when the function ends. They do not retain their value between functions calls.

However, in many cases it is required that a variable retains its value between function calls. This is possible if the variable is declared belonging to the static storage class.

Syntax: `static data_type variable;`

Example:
```
static int x;
static long factorial;
```

Types of Static Variables

i. **Local static variables:** These variables have block or function scope and they retain their value between calls to the function.

ii. **Global static variables:** They are global to the file in which they are defined. Unlike an ordinary external variable, which is visible to all functions in the file and functions in other files, a static external variable is visible only to functions in its own file.

Program: /* Illustration of local static variable and automatic variable */

```
#include<stdio.h>
main()
{ int n ;
  void increment(void);
  for(n=1;h<=5; n++)
```

```
	increment();
}
void increment(void)
{
  int lcount = 0 ;    /* automatic variable */
  static int scount = 0 ;    /* static variable */
  lcount++;
  scount++;
  printf("\n lcount = %d scount = %d", lcount, scount);
}
```

Output

```
lcount = 1   scount = 1
lcount = 1   scount = 2
lcount = 1   scount = 3
lcount = 1   scount = 4
lcount = 1   scount = 5
```

The result shows that every time function increment is called lcount is created and initialized to 0 whereas scount is initialized only once and its value persists between function calls.

Features

i. Storage : Memory
ii. Scope : Block or file scope depending upon where it is declared.
iii. Lifetime : Persists between function calls if scope is block scope.
iv. Default initial value : Zero.

8.3.4 Register Storage Class

The **register** keyword is used to tell the compiler to store the variable in a CPU register rather than in main memory. The register variables have similar features as the automatic storage class except for the storage location.

Register Variables

The CPU has its own limited storage locations, which it uses for actual data operations. These locations are called registers. To manipulate data and perform operations, the CPU moves data back and forth between the memory and registers, which takes a finite amount of time.

Thus, if a particular variable is kept in the register itself, the CPU can access it faster. Hence, variables, which are heavily used, may be declared of this type so that execution is faster.

Syntax: `register data_type variable;`

Example:
```
register int i;
register char ch;
```

Limitations

1. There are only a limited number of registers in the CPU. So, a register may not be available for the variable. In such a case, the variable is treated as an ordinary automatic variable.
2. Most compilers allow this storage class to be used only with integer data type. (int or char)
3. The unary & operator (address of) cannot be used with these variables either explicitly or implicitly.
4. It cannot be used with either static or external storage classes.
5. It cannot be used for structures, arrays or unions.

Features

i. Storage : CPU registers

ii. Scope : Block scope

iii. Lifetime : Exists as long as control is within the block where it is defined

iv. Default initial value : Garbage

Summary

The following table summarizes the storage classes, scope and initializations.

Storage Class	Variable is declared	Visibility	Remarks
Static	Outside a function	Anywhere within the file	Are initialized only once, Values retained through function calls, default initial value is zero.
	Inside a function block	Function/Block Scope	
Extern	Outside a function	Anywhere within the file	If they are to be used in multiple files, they have to be declared in each function using the extern keyword. Initialization can be done only once-outside the functions.
Register	Inside a function/block	Function/block scope	Limited number of registers, restriction on the type of variables, cannot use pointers for register variables, no default value.
Auto	Inside a function/block	Function / block scope, i.e., local to the function/block	Variable is initialized each time the function / block is entered, no default value. Does not exist outside function block where declared.

EXERCISES

A. Predict the outputs

1.
```
main()
{ int i ;
  i = abc();
  printf("%d....";i);
  i = abc();
  printf("%d",i);
}
static int abc()
{ int i = 1;
  return i++;
}
```

2.
```
extern int i ;
main()
{ printf("%d",i);
}
```

3.
```
static int i = 100;
main()
{ static int i = 200;
  abc();
  printf("%d",i);
}
abc()
{ printf("%d..", i);
}
```

4.
```
/* File aa.c */
int a = 100;
/* File bb.c */
#include "aa.c"
extern int a;
main()
{ printf("%d",a);
}
```

B. Review Questions

1. What do the following terms mean?
   i. Scope ii. Extent iii. Storage class
2. What do you mean by block scope and file scope? Explain with examples.

3. What is meant by the storage class of a variable? Name the different storage classes in C.
4. What is meant by local variables?
5. Distinguish between local and global variables.
6. What are static variables? What are the two types of static variables?
7. Differentiate between automatic and static storage classes.
8. What is the purpose of the extern keyword?
9. What values does an un-initialized global variable contain?
10. What do you understand by block scope of a variable? How does nested blocks affect its accessibility?
11. What are the advantages and limitations of the register storage class?
12. When is the register storage class most useful?
13. Discuss different storage classes in C.
14. Write two differences between auto and static variables.

Collection of Questions asked in Previous Exams PU

1. Write short note on Storage classes. [Oct. 2008, Apr. 2009 – *5 M*]

VISION

9 Structure, Union, Enumeration And typedef

9.1 STRUCTURES

Structures are also called 'records' in some languages like PASCAL. The use of structures helps organize complicated data, particularly in large programs because they permit a group of related variables to be treated as a unit rather than as separate entities.

Definition

A structure is a composition of variables possibly of different data types, grouped together under a single name. Each variable within the structure is called a 'member'. The name given to the structure is called a 'structure tag'. The data type of the variables could be any of C's data types including arrays, pointers and other structures.

9.1.1 Declaring and Initializing Structure

A structure can be declared in the following way.

Syntax:
```
struct tag
{
  member1;
  member2;
   .
   .
   .
  membern;
};
```

Examples

1.
```
struct student
{
   char name[20];
   int rollno;
   int marks;
};
```

2.
```
struct data
{
   int day;
   int month;
   int year;
};
```

The struct keyword is used to declare structures. The members of the structure are enclosed in { }. A structure declaration as above reserves no storage. It merely describes a **'template'** or shape of a structure.

Memory is allocated only when **'instances'** or variables for the structures are created.

There are two ways to create instances of a structure.

i.
```
struct tag
{
   structure_members;
} instance;
```

ii.
```
struct tag
{
   structure_members;
};
struct tag instance;
```

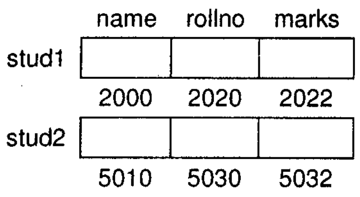
In (i), the instances are declared immediately after the structure template.

In (ii) the instances are declared later using the structure tag.

Examples

1.
```
struct student
{  char name[20];
   int rollno;
   int marks;
}  stud1, stud2;
```

2.
```
struct student
{ char name[20];
   int rollno;
   int marks;
};
struct student stud1, stud2;
```



Initializing a Structure Variable

An instance of a structure can be assigned values during declaration.

i.
```
struct student
{ char name[20];
  int rollno;
  int marks;
} stud1 = {"ABCD",10,95};
```

ii.
```
struct time
{ int hours;
  int minutes;
  int seconds;
} time_of_birth = {10,15,0};
```

If there are fewer initializers than the members, the remaining members are initialized with 0.

9.1.2 Accessing Structure Members

Individual members of the structures can be used just like other variables. Structure members can be accessed using the structure member operator (·) also called the dot operator. This operator is used between the structure name and the member name.

Syntax: `variablename.fieldname`

Example: The individual members of the structure variable stud1 in the previous example, can be accessed as

```
stud1.name
stud1.rollno
stud1.marks
```

Values can also be assigned to those members.

```
strcpy(stud1.name,"xyz");
stud1.rollno = 100;
stud1.marks = 80;
```

They can be read and displayed using the scanf and printf functions.

```
scanf("%s%d%d",stud1.name,&stud1.rollno, &stud1.marks);
printf("%s%d%d",stud1.name,stud1.rollno,stud1.marks);
```

9.1.3 Complex Structures

i. Structure within a Structure

The individual members of a structure can be other structures as well. This can be done in two ways:

a.
```
struct date
{  int day;
   int month;
   int year;
};
struct student
   {  char name[20];
      int rollno;

      struct date birthdate;
      int marks;
   }  stud1;
```

b.
```
struct student
{  char name[20];
   int rollno;
   struct date
   {  int day;
      int month;
      int year;
   }  birthdate;
   int marks;
}  stud1;
```

In (a) date is declared as a separate structure. Thus it can be used as any other structure.

In (b) date is an embedded structure and cannot be used directly elsewhere.

Accessing members of nested structures: The members of variable stud1 will be

```
stud1.name
stud1.rollno
stud1.birthdate.day
stud1.birthdate.month
stud1.birthdate.year
stud1.marks
```

Example

```
struct addition
{  float da;
   float hra;
};
struct deduction
{  float itax;
   float ptax;
};
```

```
struct employee
{  char name[20];
   float bas_sal;
   struct addition add;
   struct deduction deduct;
} e1;
```

The individual members are

```
e1. name
e1.bas_sal
e1.add.da
e1.add.hra
e1.deduct.itax
e1.deduct.ptax
```

Initialization

To initialize variables that belong to a structure containing a nested structure, the initialization values have to be given in order.

Example:
```
struct student stud1 = {"ABCD",10,
                        {10,12,1985}, 95};
```

ii. Structure containing an array

A structure can contain arrays as its members. *For example*, if we wish to store the information about the marks of 3 subjects of a student, the declaration will be

```
struct student
{  char name[20];
   int rollno;
   int marks[3];
   int total;
}  stud1;
```

The members will be

```
stud1.name, stud1.rollno,
stud1.marks[0]
stud1.marks[1]
stud1.marks[2]
```

9.1.4 The dot operator

The dot operator is in the highest precedence group in the precedence table and has a priority over unary, arithmetic, relational, logical and assignment operators.

The expression ++ stud1.rollno is equivalent to ++ (stud1.rollno).

9.1.5 Size of a Structure

The size of a variable of a structure is the sum of sizes required for its individual members.

Example:
```
struct student
{  char name[20];
   int rollno;
   int marks;
}  stud1;
```

Size of stud1 = size of name + size of rollno + size of marks

= 20 + 2 + 2 = 24 bytes

The size of a structure variable can be found using the sizeofoperator.

9.1.6 Operations on a Structure

i. Copying one structure variable's members to another: The values of the members can be assigned to those of another variable by assigning them individually as shown below.

```
struct date date1  = {1,1,1900}
struct date date2;
     date2.day = date1.day;
     date2.month = date1.month;
     date3.year  = date1.year;
```

A better and convenient way is to use the assignment operator directly.

```
date2 = date1;
structure_variable1 =structure_variable2
```

This is a perfectly valid assignment.

ii. The sizeofoperator can be used on a structure.

It will give the number of bytes required for a variable of the structure.

Example: `sizeof(struct date) will give 6`

iii. A structure variable can be passed as a parameter to a function.

iv. The address of a structure can be obtained using the & (address of) operator.

v. A function can accept or return a structure variable or a pointer to a structure.

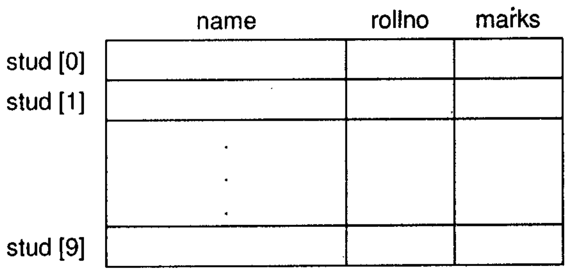
9.1.7 Array of Structures

It is possible to declare an array of structures just like any other array. This array will have individual structures as its elements. The array can be declared when the structure is declared or later using the structure tag. All array elements occupy consecutive memory locations.

Syntax: `struct tag array_name[size];`

Example: `struct student stud[10];`

Accessing Elements of the Array



Individual elements can be accessed as:

```
stud[0].name
stud[0].rollno
stud[0].marks
```

Initializing the Array of Structures

Consider the declaration

```
struct student stud[4];
```

The stud array can be initialized as shown.

```
struct student stud[4]=
{  "ABC",1,89,
   "DEF",2,64,
   "GHI",3,75,
   "JKL",4,90  },
```

stud[0] to stud[3] are stored sequentially in memory.

We shall now write a program to store the data of 10 students, viz., name, rollno, marks in three subjects and percentage. The percentage will be calculated. The display should be of all students scoring 70 percent or more.

We shall be using the concept of structures studied so far.

1. /* Illustrates structures */

```
#include<stdio.h>
struct student
{  char name[20];
   int rollno;
   int marks[3];
   float perc;
}  stud[10];
main();
{  int i, sum j;
   printf("\n Enter the details of the 10 students \n");
   for(i=0;i<10;i++)
   { printf("\n Enter the name and roll number \n");
   scanf("%s%d", stud[i].name, &stud[i].rollno);
   printf("\n Enter marks for three subjects:");
```

```
  sum = 0 ;
  for(j=0;j<3;j++)
  { scanf("%d",&stud[i].marks[j]);
      sum = sum + stud[i].marks[j];  }
    stud[i].perc  =  (float)sum/3;  }
/* Display details of students having percentages > = 70 */
 printf("\n\n Name \t rollno\t Percentage");
 for(i=0;i<10;i++)
 { if(stud[i].perc>=70)
    printf("\n%s\t%d\t%f",stud[i].name,stud[i].rollno,stud[i].perc);  }
```

2. **Create a structure of employee in Software company which contains name of employee, qualification, year of joining, total work experience and salary. Display the information of employee which having maximum experience and minimum salary.**

1 PU Oct. 2009 – *10 M*

```
#include<stdio.h>
#include<conio.h>
struct employee
{
  char name[21];
  char qualification[11];
  char d_o_j[11];
  int exp;
  float salary;
} rec[100];
void getdata(void);
void putdata(void);
void main();
{
  clrscr();
  getdata();
  putdata();
  getch();
}
void getdata(void);
{
  for(int i=0;i<100;i++)
  {
    printf("\n Enter Employee Name");
    scanf("%s", rec[i].name);
    printf("\n Enter the Qualification of Employee");
    scanf("%s",rec[i].qualification);
    printf("\n Enter the Date of Joining of Employee");
    scanf("%s", rec[i].d_o_j);
    printf("\n Enter the Exp of Employee");
    scanf("%d", &rec[i].exp);
    printf("\n Enter the Salary of Employee");
    scanf("%f", &rec[i].salary);
```

```
void putdata(void)
{
  for(int i=0;i<100;i++)
  {
    if((rec[i].exp <= maxexp) && (rec[i].salary <= maxsalary))
    {
      printf("\n Employee Name %s", rec[i].name);
      printf("\n Qualification of Employee %s",
      rec[i].qualification);
      printf("\n Date of Joining of Employee %s", rec[i].d_o_j);
      printf("\n Exp of Employee %d", rec[i].exp);
      printf("\n Salary of Employee %f", rec[i].salary);
    }
  }
}
```

3. Create a structure item having Item Id, Name and Price. Accept the details for 50 records and find out the item having highest price and lowest price. Display the report.

PU
Apr. 2010 – 10 M

```
#include<stdio.h>
#include<conio.h>
struct item
{
  int item_id;char name[10];float price;
} i1[50];
main()
{
  int i,m1=0,m2=0,max,min;
  for(i=0;i<50;i++)
  {
    printf("\nenter the item id, name, price");
    scanf("%d%d%f",&i1[i].item_id,&i1[i].name,&i1[i].price);
  }
  max=i1[0].price;
  min=i1[0].price;
  for(i=0;i<50;i++)
  {
    if(i1[i].price >max)//finding maximum price item
    {
      max=i1[i].price;
      m1=i;
    }
    if(i1[i].price<min)//finding minimum price item
    {
      min=i1[i].price;
      m2=i;
    }}
  printf("\nItem having maximum price\n");
  printf("\nitem id=%d",i1[m1].item_id);
```

```
    printf("\nitem_name=%s", i1[m1].name);
    printf("\nprice=%f",i1[m1].price);
    printf("\nItem having minimum price\n");
    printf("\nitem id=%d",i1[m2].item_id);
    printf("\nitem_name=%s", i1[m2].name);
    printf("\nprice=%f",i1[m2].price);
}
```

9.1.8 Structures and Pointers

i. **A pointer within a structure:** A structure can have a pointer as one of its members. They can be used just like any other pointer variables.

Example:
```
struct data
   { int *amount;
     char *itemname;
     }item;
```

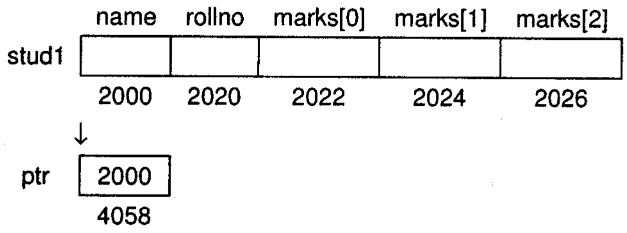
They can be used as illustrated below-

```
item.amount = &cost;
item.itemname = "steel";
```

They can be de-referenced using the * operator. The expression *item.amount evaluates to the value of cost. item.itemname points to the string "steel" stored elsewhere in memory.

ii. **Pointer to a structure:** The address of a structure variable can be obtained by using the & operator. This address can be assigned to a pointer variable, which has to be declared as a pointer to a structure as illustrated below.

```
struct student
{  char name[20];
   int rollno;
   int marks[3];
}  stud1;
struct student *ptr;  /* pointer to struct student */
ptr = &stud1
```



Accessing members: There are three ways by which members of stud1 can be accessed:

a. Using the structure name

```
stud1.name, stud1.rollno
```

b. Using the pointer and indirection operator

```
(*ptr).name, (*ptr).rollno
```

c. Using the pointer and the membership operator

```
ptr->name,  ptr->rollno,  ptr->marks[i]
```

Following example illustrates incrementing a pointer to a structure variable. Incrementing causes the pointer to point to the next structure variable.

iii. **Pointer to an array of structures:** A pointer variable can be made to point to an array of structures by assigning it the base address of the array. The array elements can be accessed by incrementing the pointer. Incrementing causes the pointer to point to the next structure variable.

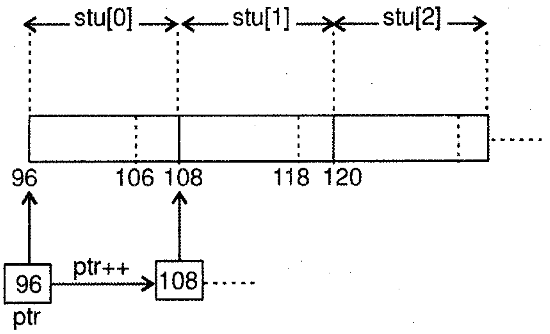
Program: Illustrates pointer to a structure array

```
struct student
{ char name[10];
  int rollno;
} stu[4] = {  "ABC",1,
             "DEF",2,
             "GHI",3,
             "JKL",4};
main()
{ struct student *ptr = stu;
for(i=0;i<4;i++)
  { printf("At address %u: %s %d", ptr, ptr->name, ptr->rollno);
  ptr++; /*Move pointer to next structure element in the array */}}
```

Output

```
At address 96  :  ABC 1
At address 108 :  DEF 2
At address 120 :  GHI 3
At address 132 :  JKL 4
```

This can be depicted pictorially as:

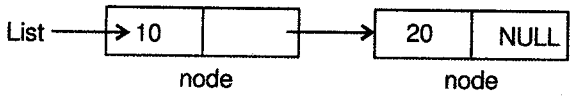


iv. **A pointer to a structure within the same structure (self referential structure):** In this type of structures, the structure contains a pointer to itself.

These type of structures are widely used in data structures like linked lists, trees etc.

```
struct node
{  int data;
   struct node *ptr ; /* pointer to struct node */ };
```

In this example, the structure node contains a pointer to itself. We can see its use in a linked list. A linked list is a collection of nodes each linked to the next using a pointer.



Dynamically allocating memory for each node and then linking the node can create this list.

v. **Array of Pointers to Structures:** In the last chapter we saw an array of pointers to strings. Similarly, we can have an array of pointers to structure elements.

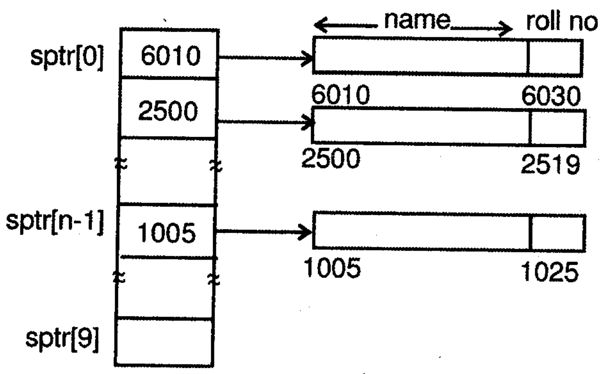
They are very useful in dynamic memory allocation, which saves a lot of memory space.

Example:
```
struct student
{ char name[20];
  int rollno;
} *sptr[10];
```

Here sptr is an array of 10 pointers to struct student. We would use an array of pointers when we do not know how many students there are initially. So declaring an array of a large number of students will cause a lot of memory wastage. Hence, we can dynamically allocate space for 'n' students and store the address in the array as shown below.

```
printf("How many students?");
scanf("%d",&n);
for(i=0;i<n;i++)
{sptr[i] =(struct student *) malloc(sizeof(struct student));
     scanf("%s%d",sptr[i]->name,sptr[i]->rollno); }
/* Displaying data */
for(i=0;i<n;i++)
printf("\n Name:%s Rollno:%d",sptr[i]->name,sptr[i]->rollno);
```

Each student record is stored at different memory locations, since we are allocating memory for individual members. Their addresses are stored in the array of pointers using which we can access individual members.



vi. **Some declarations and their meanings:** The structure operators. and → together with ( ) for function calls and [ ] for subscripts are at the top of the precedence hierarchy.

Example:
```
struct {
    int len ;
    char *str;
} *p;
```

Then

a. ++p –>len; increments len because it is equivalent to ++ (p→len)
b. (++p) –>len; increments p before accessing len
c. (p++)–>len; increments p afterwards
d. *p–>str; fetches whatever str points to
e. *p–>str ++; increments str after fetching whatever str points to
f. (*p–> str)++; increments whatever str points to
g. *p++–>str; increments p after accessing whatever str points to

9.1.9 Using typedef with Structures

The typedef keyword can be used to give a new type name for the structure. The new name can be used to create instances, passing values to functions and declaring pointers, etc.

Examples

1.
```
typedef struct
{ char name[20];
  int rollno;
  int marks;
} student;
```

2.
```
typedef struct studrec
{ char name[20];
  int rollno;
  int marks;
} student;
```

student is the name of the new type. In (2) studrec is the tag name, which is not needed but used for clarity.

Variables of this type can be created by the statement:

```
student stud1, s[100],*ptr;
```

Examples

1.
```
typedef struct
{ int day;
  int month;
  int year;
} date;              /*date is a new data type */
```

2.
```
typedef struct
{  char name[25];
   date birthdate;
   char address[50];
} person;           /* person is a new data type */
person list[10];   /* list is an array of 10 elements of type person */
```

9.2 STRUCTURES AND ENUMERATED DATA TYPE

Enumerated data types can be used as a part of a structure as illustrated in the example below.

Program: Illustrates enumerated data type

```
enum qual{tenth = 1, twelfth, graduate, masters, doctorate};
enum  grade{ worker=1, clerk, manager };
struct emp
{ char name[20];
  enum qual emp_qual;
  enum grade emp_grade;
  basic_sal;  };
main()
{ struct emp e1;
  printf("\nEnter the name of the employee :");
  gets(e1.name);
  printf("\nEnter the qualification of the employee :");
  scanf("%d",&e1.emp_qual);
  switch(e1.emp_qual)
  {  case tenth :
     case twelfth : e1.emp_grade = worker;
                    e1.basic_sal = 5000;
                    break;
     case graduate : e1.emp_grade= clerk;
                     e1.basic_sal = 7000;
                     break;
     case masters :
     case doctorate : e1.emp_grade = manager;
                      e1.basic_sal = 10000;
                      break;  }
  getch();
}
```

Structures and Functions

i. **Passing structures to functions:** It is possible to send an entire structure to a function as an argument in the function call. The following program illustrates this.

/* Demonstrate passing a structure to a function */

```
struct data
{ char item[20];
    float amount;
  } list[10];
```

```
main()
{ void acceptdata(struct data *record);
    void display(struct data *info);
    int i;
    for(i=0;i<9;i++)
    acceptdata(&list[i]);
    /*Display data */
    for(i=0;i<9;i++)
    display(&list[i]);}
void acceptdata(struct data *record)
{ printf(("\n Enter the itemname and amount");
    scanf("%s%f",record->name, &record->amount);}
void display(struct data *info)
{ printf("\n Name = %s, amount = %f", info->name, info->amount);}
```

Similarly, a function can also return a structure.

For example, struct student acceptdata( ); is a function which returns a structure of type student.

ii. **Passing an array of structures to functions:** In the above program, we passed individual structure items to the function. However, we can pass the entire array to the function just like any other array.

Example: `void acceptdata(struct data record[], int n);`

Accepts an array of structures and an integer n. The call to this function can be given as

`acceptdata(list, 10);`

where list is an array of structures as seen in the previous example.

Example: We shall now write a program to store student information of n students and display the information in descending order of total marks. We will use functions.

```
#include<stdio.h>
struct student
{ char name[20];
int rollno;
int marks[3];
int total;};
main()
{ struct student s[20];
void acceptdetails(struct student s[], int n);
void dispdetails(struct student s[], int n);
void sortdetails(struct student s[], int n);
printf("\n How many students?");
scanf("%d", &n);
acceptdetails(s,n);
sortdetails(s,n);
dispdetails(s,n);  }
void acceptdetails(struct student s[20], int n)
{ int i,j;
for(i=0; i<n; i++)
{ printf("\n Enter the name, rollno, and marks");
```

```
      gets(s[i].name);
      scanf("%d",&s[i].rollno);
      s[i].total = 0;
   for(j=0;j<3;j++)
      {scanf("%d", &s[i].marks[j]);
      s[i].total = s[i].total + s[i].marks[j];   } } }
   void sortdetails(struct student s[20], int n)
   { int i,j;
   struct student temp;
   for(i=0;i<n-1;i++)
      for(j=i+1;j<n;j++)
      {  if(s[i].total > s[j].total)
         { temp = s[i];     /*Interchange student records */
           s[i] = s[j];
           s[j] = temp;   }  }  }
   void dispdetails(struct student s[20], int n)
   { int i; printf("\n Name  Rollno Total \n");
   for(i=0; i<n; i++)
   { printf("%s %10d %10d", s[i].name, s[i].rollno, s[i].total);
     printf("\n"); }
   }
```

9.3 UNION

A union is analogous to a variant record in PASCAL. It is very similar to a structure but only one of its members can be used at a time.

Definition

A union is a variable that contains multiple members of possibly different data types grouped together under a single name. However, all of them occupy the same memory area. Hence, only one of the members will be active at a time.

Unions provide a way to manipulate different kinds of data in a single area of storage. There are many applications where we would want to store different data terms at different times. A union provides a facility to do this.

9.3.1 Declaration of a Union

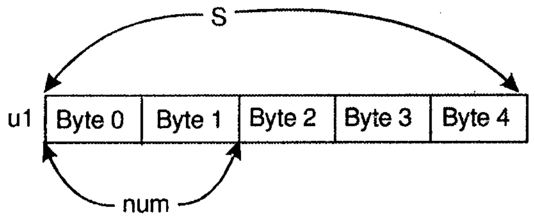
A union is declared in the same way as a structure except that the keyword 'union' is used instead of 'struct'.

Syntax:

```
union tag
{ union_members;
} instance;
```

Example:

```
union u
   { char s[5];
     int num;
   } u1;
```



The variable u1 will be allocated sufficient storage for the variable to accommodate the largest member of the union. In this example, it will be allocated 5 bytes.

Only one member, i.e., either the string or the integer num will be stored and can be accessed at a time. Both do not exist simultaneously.

9.3.2 Accessing Members of the Union

Union members can be accessed using the dot operators, i.e.,

```
union_variable.member
```

If we have a pointer to the union (similar to the pointer to a structure), the members are accessed using the–>operator.

```
union_pointer->member
```

Example: `union u *ptr = &u1;          /* Initialize pointer to union */`

The following expressions are valid:

u1.s	ptr–>s
u1.num	ptr–>num

9.3.3 Initializing a Union

Since only one member of the union can be used at a time, only one can be initialized.

The initializer for the union is either a single expression of the same type or a brace enclosed initializer for the first member of the union, i.e., only the first member can be initialized.

9.3.4 Union within a Structure and Union

Just as it is possible to include one structure within the other, the same can be done with unions.

This can be better understood with an example. Suppose we wish to store employee information viz., name, id and designation.

If the designation is 'M' (for managers) the number of departments he manages should be stored and if his designation is 'W' (for workers), his department name should be stored.

The structure and union declarations will be:

```
union info
{
   int no_of_depts;
```

```
  char deptname[20];
};
    struct empre
  { char name[20];
    int id;
    char desig;
    union info details;
  } emp[100];
```

9.3.5 Structure Assignment

i.
```
strcpy(emp[0].name,"ABC");
emp[0].id = 1015;
emp[0].desig = 'M';
emp[0].details.no_of_depts = 3;
```

ii.
```
strcpy(emp[1].name,"XYZ");
emp[1].id = 2008;
emp[1].desig = 'W';
strcpy(emp[1].details.deptname, "Manufacturing");
```

9.3.6 Operations on a Union

i. A union variable can be assigned to another union variable.
ii. The address of the union variable can be obtained using the address-of operator.
iii. Only the first member of the union can be initialized.
iv. A function can accept and return a union or a pointer to a union.
v. The sizeof operator can be used with a union.
vi. The `typedef` keyword can be used to create a synonym for a union. Instances can then be created using this synonym.

Example:
```
typedef union
{ int no_of_depts;
  char deptname[10];
} info;
info details; /* Instance of info */
```

9.4 DIFFERENCE BETWEEN STRUCTURE AND UNION

Although the syntax for declaring and accessing structure and union variables is the same, there are important differences between them.

i. Memory allocation

Each member of a structure is allocated memory space, i.e., a structure variable occupies the sum of sizes of all members in the variables.

In case of a union, the amount of memory required is the same as its largest member.

ii. Accessing members

All the structure members can be accessed at any given time.

Only one member of the union can be accessed at any given time.

iii. Initialization

All members of a structure variable can be initialized.

Only the first member of a union variable can be initialized.

SOLVED PROGRAMS

1. **Create structure Elect Bill having members consumer_no, name, no_of_units, amt. Write a program to accept 10 records.**

   Calculate amt – using following rate.

   For no_of_units less than 100 – rate Rs. 1.50 per unit.

   no_of_units greater than 100 – rate Rs. 6.50 per unit.

   Display the records of largest and smallest amt.

1 PU Oct. 2010 – *10 M*

```
#include<stdio.h>
#include<conio.h>
struct ebill
{
  int consumer_no;
  char name[10];
  int units;
  float amt;
}e[10];
main()
{
  int maxi,mini,max,min,a,i;
  for(i=0;i<10;i++)
  {
    printf("\nEnter consumer Number:");
    scanf("%d",&e[i].consumer_no);
    printf("\nEnter consumer Name:");
    scanf("%s",e[i].name);
    printf("\nEnter number of units:");
    scanf("%d",&e[i].units);
  }
  for(i=0;i<10;i++)
  {
    if(e[i].units<=100)
    {
      e[i].amt=e[i].units*1.50;
    }
    else
    {
      a=e[i].units-100;
      e[i].amt=(100*1.50)+(a*6.50);
    }
  }
  max=e[0].amt;
  min=e[0].amt;
  for(i=0;i<10;i++)
  {
    if(e[i].amt>max)
    {
      max=e[i].amt;
      maxi=i;
    }
    if(e[i].amt<min)
    {
      min=e[i].amt;
      mini=i;
    }
  }
  printf("\n Maximum Bill amount customer\n");
  printf("\nconsumer no=%d",e[maxi].consumer_no);
  printf("\nconsumer name=%s",e[maxi].name);
  printf("\nunits=%d",e[maxi].units);
  printf("\nBill Amount =%f",e[maxi].amt);

  printf("\n Minimum Bill amount customer\n");
  printf("\nconsumer no=%d",e[mini].consumer_no);
  printf("\nconsumer name=%s",e[mini].name);
  printf("\nunits=%d",e[mini].units);
  printf("\nBill Amount =%f",e[mini].amt);
  getch();
}
```

2. Write a C program that stores the student information (name, dob, admission taken to the course) in a "student.dat" file. Read the file and display all student information along with their age. The list should be age wise.

1 PU Apr. 2009 – *10 M*

```
struct Student
{
  char name[30],dob[12],course[15];
  int age;
}stud[3];
main()
{
  clrscr();
  readinfo();
  displayinfo();
  getch();
}
readinfo()
```

```
{
  int i;
  printf("\n Enter information about students : \n");
  for(i = 0; i < 3; i++)
  {
    printf("\n Enter name, dob(dd/mm/yy), course and age of student #%d
          \n",i + 1);
    gets(stud[i].name);
    flushall();
    scanf("%s %s %d",stud[i].dob,stud[i].course,&stud[i].age);
    flushall();
  }
}
displayinfo()
{
  int i, j, temp_age;
  char temp_name,temp_dob,temp_course;
  for(i = 0; i < 3; i++)
  {
    for(j = 0; j < 3; j++)
    {
      if(stud[i].age > stud[j].age)
      {
        temp_age = stud[j].age;
        stud[j].age = stud[i].age;
        stud[i].age = temp_age;

        strcpy(temp_name,stud[j].name);
        strcpy(stud[j].name,stud[i].name);
        strcpy(stud[i].name,temp_name);

        strcpy(temp_dob,stud[j].dob);
        strcpy(stud[j].dob,stud[i].dob);
        strcpy(stud[i].dob,temp_dob);

        strcpy(temp_course,stud[j].course);
        strcpy(stud[j].course,stud[i].course);
        strcpy(stud[i].course,temp_course);
      }
    }
  }
  for(i = 0; i < 3; i++)
  {
    printf("\n\n Name : %s",stud[i].name);
    printf("\n DOB    : %s",stud[i].dob);
    printf("\n Course : %s",stud[i].course);
    printf("\n Age    : %d",stud[i].age);
  }
}
```

3. **What will be the output of the following program? Give the explanation.**

PU
Oct. 2008 – *4 M*

```
Void main()
{
  struct employee
     {
  unsigned id: 8;
  unsigned sex: 1;
  unsigned age: 7;
     };
Struct employee emp = {203, 1,23};
   clrscr();
printf("%d%d%d", emp.id, emp.sex, emp.age);
   getch();
}
```

Ans

This program displays four errors and one warning.

Error

1. Too many initialization. That means we are initialized many time the program of a particular variable.
2. id is not a member of employee. In this program, we are using the column with every variables when we define the template of structure and when we access the variable that time we do not use the column, so it displays the above message. *Example*: id instead of id:.
3. Sex is not a member of employee. In this program, we are using the column with every variables when we define the template of structure and when we access the variable that time we do not use the column, so it displays the above message. *Example*: sex instead of sex:
4. age is not a member of employee. In this program, we are using the column with every variables when we define the template of structure and when we access the variable that time we do not use the column, so it displays the above message. *Example*: age instead of age:

Warning: emp is assigned a value that is never used.

EXERCISES

A. Predict the output

1.
```
struct student
{  char name[20];
   int rollno;
}  s1,*ptr,s[10];
   printf("\n %d"sizeof(s1));
   printf("\n%d"sizeof(ptr));
   printf("\n %d"sizeof(s));
```

2.
```
main()
{
  struct {
     int i;
     } *ptr;
(&*ptr)→i = 10;
printf("%d",ptr →i );
}
```

3.
```
main()
{ struct a
   { int i;
   };
   struct a  a;
   a.i = 100;
   printf("%d", a.i);
}
```
4.
```
struct abc
{ int i ;
};
main()
{ int abc = 20;
  struct abc m ;
  m.i  =  200;
  printf("%d",m.i);
  printf(("\n%d",abc);
}
```
5.
```
union
{
  union
  { char a;
    char b;
  } car;
  union
  { int j;
    int k ;
  } abc;
  float z;
  } pqr;
  printf("%d",sizeof(par));
```

B. Programming exercises

1. Write a program to accept student data_name, roll numbers and marks of 3 subjects. Calculate the total and arrange these records in descending order of marks.
2. Accept book details of 'n' books viz., book title, author, publisher and cost. Assign an accession numbers to each book in increasing order. Display these details as
   i. Books of a specific author
   ii. Books by a specific publisher
   iii. All books costing Rs. 500 and above
   iv. Information about a particular book (accept the title)
   v. All books

   The above five should be options for the user.
3. Write a program to store information about 'n' employees. The details are :
   name, emp_id, designation(M-Manager, D-Director, W-worker, details (for director-years of experience, for manager-name of the department, for worker-his specializations viz., electrician, mechanic, draftsman, etc.)
   Use a menu to display details of: i. All directors ii. All managers iii. All workers
4. Read cricket player information – Name, Player Type and Score. The score depends on Player type.
   If batsman- store batting average .If bowler store no. of wickets
   If wicketkeeper – store no of stumpings
   Display the name of batsman, bowler and wicketkeeper with best performance .

5. Read names and addresses using a structure and rearrange the data in alphabetical order of names and display.

C. Review questions

1. How is a structure declared and initialized? Give an example.
2. How is a union declared? Can it be initialized? Explain.
3. Define: i. Structure ii. Union
4. What are the differences between a structure and a union? Illustrate with an example.
5. How can an array of structure be declared? Can it be initialized? Give an example.
6. Explain nesting of structures. How can members of nested structures be accessed?

Collection of Questions asked in Previous Exams PU

1. What will be the output of the following program? Give the explanation. **[Oct. 2008 – 4 M]**

```
Void main()
{
   struct employee
     {
unsigned id: 8;
unsigned sex: 1;
unsigned age: 7;
     };
Struct employee emp = {203, 1,23};
  clrscr();
printf("%d%d%d", emp.id, emp.sex, emp.age);
  getch();
}
```

2. Difference between structure and union. **[Oct. 2008 – 5 M]**
3. Write a C program that store the student information (name, dob, admission taken to the course) in a "student.dat" file. Read the file and display all student information along with their age. The list should be age wise. **[Apr. 2009 – 10 M]**
4. Create a structure of employee in Software company which contains name of employee, qualification, year of joining, total work experience and salary. Display the information of employee which having maximum experience and minimum salary. **[Oct. 2009 – 10 M]**
5. Create a structure item having Item Id, Name and Price. Accept the details for 50 records and find out the item having highest price and lowest price. Display the report. **[Apr. 2010 – 10 M]**
6. Create structure Elect Bill having members consumer_no, name, no_of_units, amt. Write a program to accept 10 records.
   Calculate amt – using following rate.
   For no_of_units less than 100 – rate Rs. 1.50 per unit.
   no_of_units greater than 100 – rate Rs. 6.50 per unit.
   Display the records of largest and smallest amt. **[Oct. 2010 – 10 M]**

10 C Preprocessor

10.1 WHAT IS A PREPROCESSOR?

A preprocessor is a program that processes or analyzes the source code file before it is given to the compiler.

It performs the following tasks:

i. Replaces trigraph sequences (not covered in this book) by their equivalents. Trigraph sequences are used to handle non ASCII character sets.

ii. Joins any lines that end with a backslash character into a single line.

iii. Divides the program into a stream of tokens.

iv. Remove comments, replacing them by a single space.

v. Processes preprocessor directives and expands macros.

vi. Replaces escape sequences by their equivalent internal representation.

vii. Concatenates adjacent constant character strings.

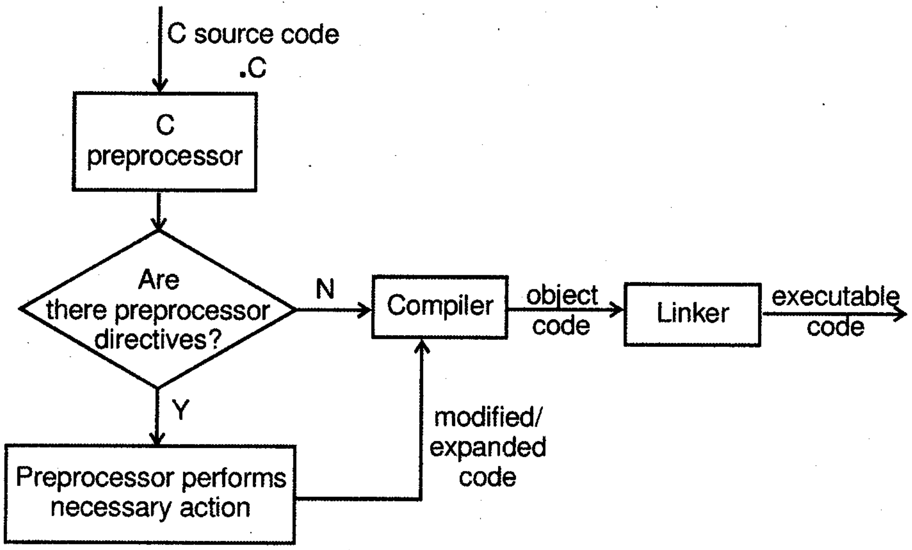


Figure 10.1

10.2 PREPROCESSOR DIRECTIVES

Preprocessor directives are special instructions for the preprocessor.

i. They begin with a # which must be the first non-space character on the line.
ii. They do not end with a semicolon.
iii. Each preprocessing directive must be on its own line.

Preprocessor directives come under three categories

a. Macro substitution directive
b. File inclusion directive
c. Conditional compilation directive

10.2.1 Macro Substitution Directive

A macro is a small subprogram which contains executable code and is similar to a function. Wherever a macro name occurs in a program the preprocessor substitutes the code of the macro at that position (unlike a function). The execution is faster since time is not wasted in function call and return.

i. Simple substitution macro

```
#define macro_id value
```

#define is a preprocessor directive that defines an identifier and a value that is substituted for the identifier each time it is encountered in the source file.

We have already used this directive to define symbolic constants.

The identifier is usually written in uppercase to distinguish it from other variables.

- A second #define for the same identifier is erroneous unless the second value is exactly identical to the first.
- Use of macros enhances readability of the program.

Examples

1. #define PI 3.142
2. #define TRUE 1
3. #define AND &&
4. #define LESSTHAN <
5. #define GREET `printf("Hello");`
6. #define MESSAGE `"Welcome to C"`
7. #define INRANGE (a >= 60 && a<70)

Every occurrence of the macro-id in the program will be replaced by its corresponding value.

Example:

```
int a = 50;
if(INRANGE)
printf("First class");
```

ii. Argumented Macros

An argumented macro is also called a function macro. The macroname can have arguments. Each time the macroname is encountered, the arguments associated with it are replaced by the actual arguments found in the program.

Advantages

a. Their arguments are not type sensitive. Therefore we can pass any numeric variable type to an argumented macro that expects a numeric argument.

b. Argumented macros execute much faster as compared to their corresponding functions.

Example

1.
```
#define HALFOF(x)   ((x)/2)
  result = HALFOF(10);
```

The occurrence of HALFOF is replaced by

Result = ((10 /2));

The reason for enclosing x in ( ) is that the parameter could also be an expression in which case, the expression has to be first evaluated. If it is not enclosed in ( ), it may yield wrong results.

Example: `result = HALFOF(10+2);`

This will be evaluated as

```
result = ((10+2)/2);
```

Thus giving the correct result. If no brackets are used, it would evaluate to

```
result = (10+2/2);
```

thereby giving the wrong result.

2. `#define LARGER(x,y)     ((x)>(y) ?  (x) :  (y))`

3. All the parameters of the macro must be used in the substitution value, i.e.,

   `#define ADD(x,y,z) ((x) + (y))`

   is invalid because Z is not used. The correct macro is

   `#define ADD(x,y,z)    ((x)+(y)+(z))`

4. `#define SQUARE(x)  ((x)*(x))`

5.
```
#define STREQL(s1,s2) (strcmp((s1),(s2)==0)
if(STREQL(str1,str2)
   -
   -
```

iii. Nested Macros

A macro name can be contained within another macro. This is called nesting of macros.

Example:
```
#define CUBE(x)  (SQUARE(x)*(x))
#define MAX(a,b,c) LARGER(LARGER(a,b),c)
```

Macros versus Functions

i. Macros are small and do not usually extend beyond one line. They are used when the code is relatively short.

ii. Since the macro is replaced by its code, if a macro occurs many times, the final program contains the expanded code of all the macros; thereby increasing program size.

   In contrast, a function code appears only once. A function has space advantage over a macro.

iii. When a function is called, a certain amount of processing is required to pass control to the function code and return control back to the calling program. This takes a finite amount of time.

This does not occur for a macro because the macrocode is put into the program. Therefore, a macro has a speed advantage over a function.

10.2.2 File Inclusion Directive

The file inclusion directive is the one that begins with #include. We have already used this directive a number of times.

This directive instructs the compiler to include the specified file, i.e., it replaces the entire contents of the file at that position.

Syntax: `#include<filename>`

OR

`#include"filename"`

- In the first format, the file is searched in standard directories only.
- In the record, the file is first searched in the current directory. If it is not found there, the search continues in the standard directories.
- Any external file-containing user defined functions, macro definitions etc. can be included.
- An included file can include other files.

Example

```
/* group.h */
  #include<stdio.h>
  #include<math.h>
  #include "myfile.c"
  #define PI 3.142
/* mainprog.c     */
#include "group.h"
main()
{
_
_
}
```

10.2.3 Compiler Control Directives / Conditional Compilation

Several directives allow compilation of selective portions of the program's code if certain conditions are met. *These are:*

i. #if

ii. #else

iii. #elif

iv. #endif

They work similar to the if else statement in C. The different formats in which they can be used are as follows:

i.
```
#if expression
  statement_block;
#endif
```

ii.
```
#if expression
  statement_block1;
#else
  statement_block2;
#endif
```

iii.
```
#if expression
   statement_block1;
#elif expression
   statement_block2;
#elif expression
   statement_block3;
#else
   default statement_block;
#endif
```

If the constant expression is true, the statement block is compiled otherwise it is skipped and goes to the #else part (if it exists).

Examples

1.
```
#define MAX 10
main()
{  #if MAX>99
      /* Code for larger array */
#else
      /* Code for smaller array */
#endif
}
```

2.
```
#if BACKGROUND==5
   #define FOREGROUND 1
   #elif BACKGROUND==8
   #define FOREGROUND 0
#endif
```

Another method for conditional compilation is the use of #ifdef, #ifndef.

#ifdef means if defined and #ifndef means if not defined.

In case of a large C program, many macros are defined in various files so it is difficult to remember if a particular macro has been defined or not.

In such a case we can check for its definition using the above two macros.

- Redefining an existing macro is erroneous.
- Un-defining a non existent macro is also erroneous.

So the definition of a macro has to be first checked for.

The **syntax** is

```
#ifdef macro_id
     statement_block;
#endif

#ifndef macro_id
     statement_block;
#endif
```

Example:
```
#include "declare.h"
#ifndef FLAG
  #define FLAG 1
#endif
```

Un-defining a Macro

A macro can be undefined using the # undef directive.

Example:
```
#ifdef FLAG
#undef FLAG
#define FLAG 0
#endif
```

#ifdef and #endif can be used to compile and run debugging code in the program.

Example:
```
#define DEBUG 1
main()
{ _
  _
#ifdef DEBUG
  /* debugging code put here */
#endif
_
_ }
```

Another important use of conditional compilation directive is when a program has to be run on different machines. In such a case, the common part of program can be run and the machine dependent program part can be conditionally compiled as shown below.

```
main()
{ #ifdef IBM-Pc
  { code for IBM-Pc }
    #else
  { code for HP machine}
    #endif }
```

SOLVED PROGRAMS

1. **What will be the output of the following programs? Give the explanation.**

i.

```
#define MAN(x,y)        (x)>(y)? (x):(y)
Void main()
{
    int i =10, j = 9, k = 0;
    K = MAN(i++, ++j);
    printf("%d%d%d", i, j, k);
    getch();
}
```

PU
Oct. 2008 – 4 M

Ans

This program displays the following output:

11 11 11

In this program first we define the macro MAN. This macro takes two arguments. According to the value of argument they update the value of argument.

e.g., we initialized the value of i = 10 and j = 9 and we are checking the values of x and y. If x is greater than y it returns the updated value of x otherwise it return updated value of y.

ii.

```
# define       SQ(x) x * x
# define CB(x)      SQ(x) * x
main( ) {
int a, b, c;
a = 4;
b = SQ(+ + a);
c = CB(b + +);
printf("a = %d b = %d c = %d", a, b, c);
```

PU
Apr. 2010 – 5 M

Ans

a = 4

b = ++a* ++a // macro SQ is executed for b. Value of a will become 5 then 6. Final value 6 will be considered.

b = 6 * 6 = 36

b = 36

c = b++*SQ(b++)// macro within macro concept is used. MacroSQ is called within macro CB.

c = b++*b++*b++ (b=37 then b=38 then b=39 so final value b=39 will be considered)

c = 39*39*39 = 59319

printf will print 6 39 59319

Output = 6 39 59319

iii.

```
#define lee(a,b,c)      avg (a,b,c) <=60
#define des(a,b,c,d)      (d= =1 ? geq(a,b,c) : lee(a,b,c))
void main(void)
{
     int num = 70;
     char ch = '0';
     float f = 2.0;
if des(num, ch, f, 0) puts("lee");
     else puts ("geq");
}
```

PU Oct. 2010 – 5 M

Ans

avg is a function which is not defined here.

So the correct code is

```
int avg(int a,int b,int c)
{
return(a+b+c)/3;
}
#define lee(a,b,c)      avg(a,b,c)<=60
#define des(a,b,c,d)  (d==1 ? geq(a,b,c):lee(a,b,c))
void main(void)
{
int num = 70;
char ch = '0';
float f = 2.0;
if(des(num,ch,f,0))
puts("lee");
else
puts("geq");
}
//des,geq,lee are macros.
```

des is called with parameters

```
num=70,ch='0'(ch=48 ASCII value of zero is taken),f=2.0 and 0(zero)
so
a=70,b=48,c=2.0,d=0.
d!=1 so lee will be executed with parameters(a,b,c)
```

lee will call average function

```
a+b+c/3
=70+48+2.0/3
=120/3=40
```

Lee will give answer 40 < 60. So des will print string lee

Output: lee

EXERCISES

A. Predict the output

1.
```
#define GREAT "xyz"
main()
{  printf(GREAT);
}
```

2.
```
#define GREET HELLO
main()
{  printf(GREET);
}
```

3.
```
main()
{  #include <stdio.h>
}
```

4.
```
#define MAIN main()
#define BEGIN {
#define END    }
#define GREET printf("Hello")
MAIN
BEGIN
  GREET;
END
```

5.
```
#define SQUARE(x)    (x*x)
main()
{ int i = 20,  j=10,k;
  k = SQUARE(i-j)
  printf("%d",k);
}
```

6.
```
#define SQUARE(x)  (x)*(x)
main()
{  int i = 20,j=10,k;
   k = SQUARE(i-j);
   printf("%d",k);
}
```

7.
```
#define FLAG
#ifdef FLAG
  int i = 10;
#endif
main()
{  int i = 5;
   printf("%d",i);
}
```

8.
```
/* File abc.h */
  printf("Hello");
/* File my.c */
main()
{ #include "abc.h"
  printf("C");
}
```

9.
```
/* File xxx.h */
  printf("Hello")
/* File my.c */
main()
{ #include "xxx.h"
  ;
  printf("C");
}
```

B. Review questions

1. Write a note on the C Preprocessor.
2. Explain Macro substitution in brief with examples.
3. When an argumented macro is defined, why should each argument be enclosed in parenthesis?
4. Do header files need to have a .h extension?
5. Illustrate the use of #ifdef and #undef with examples.
6. Explain any four preprocessor directives.

Collection of Questions asked in Previous Exams PU

1. What will be the output of the following programs? Give the explanation. **[Oct. 2008 – 4 M]**
```
#define MAN(x,y) (x)>(y)? (x):(y)
Void main()
{
  int i =10, j = 9, k = 0;
  K = MAN(i++, ++j);
  printf("%d%d%d", i, j, k);
  getch();
}
```
2. Write short note on Preprocessor directive. **[Apr. 2009 – 5 M]**

3. Find and explain the output of following program: **[Apr. 2010 – 5 M]**

```
# define SQ(x) x * x
# define CB(x) SQ(x) * x
main() {
int a, b, c;
a = 4;
b = SQ(+ + a);
c = CB(b + +);
printf("a = %d b = %d c = %d", a, b, c);
```

4. Find and explain the output of following programme: **[Oct. 2010 – 5 M]**

```
#define lee(a,b,c) avg (a,b,c) <=60
#define des(a,b,c,d) (d= =1 ? geq(a,b,c) : lee(a,b,c))
void main(void)
{
   int num = 70;
   char ch = '0';
   float f = 2.0;
if des (num, ch, f, 0) puts ("lee");
   else puts ("geq");
}
```

11 File Handling

11.1 INTRODUCTION

All the input, output functions that we have seen so far are console oriented I/O functions. However, most applications require a large amount of data. If this data has to be entered through the standard input device, it is time consuming and moreover, once the execution is over, the data is lost. We may also require a program's output for later use.

Therefore, data can be stored on the disk and read whenever required. Similarly, the output of a program may also be stored in files.

There are many file I/O functions provided in the C library. But before we go into the details of file handling, it is important to know something about 'Streams in C'.

11.2 STREAMS

- All C input/output is done with streams, no matter where the input is coming from and no matter where it is going to.
- A stream is a sequence of bytes of data. A sequence of bytes flowing into a program is an input stream, and the one flowing out is an output stream.
- The use of streams makes I/O device independent.

Predefined Streams

There are 5 predefined streams, which are automatically opened when a C program starts executing and are closed when the program terminates.

Name	Stream	Meaning	Device
stdin	Standard input	Standard input device (opened for input)	Keyword
stdout	Standard output	Standard output device (opened for output)	Screen
stderr	Standard error	Standard error output device (opened for output)	Screen
stdprn*	Standard printer	Standard printer (opened Printer for output)	Printer (LPT1)
stdaux*	Standard auxiliary	Standard auxiliary device (opened for input and output)	Serial Port (COM1)

* *supported only under DOS*

Example: The output of a function like `printf()` or `puts()` goes to the stream `stdout`. The `scanf()` receives its input from stream `stdin`.

A stream is associated with a file. For every disk file, that the program uses, a stream associated with that file has to be created.

11.3 TYPES OF FILES

Streams are of two types - text and binary. Either type of stream can be associated with a file. Hence, we can have two types of files: Text and Binary.

Difference between Text and Binary Modes

i. Text files consist of a sequence of characters organized into lines and terminated by one or more characters that signal end-of-line. The maximum line length is 255 characters.

In binary files, all data is written and read with no interpretation and separation. All bytes of data are considered the same.

ii. A character translation may take place in text files. Thus, the number of characters read or written may not be the same as those stored on the external device. For example, the C new-line character is converted into a Carriage Return-Line Feed combination and stored in the file. When data is read from the file, the CR-LF combination is translated to a '\n'.

In binary files, there is a one-to-one correspondence between the bytes read and the bytes stored, i.e., no character translation will occur.

iii. In text, files, ASCII 0×1A is considered as end-of file character. No special character indicators are there in binary files. The end-of-file is detected from the number of bytes in the directory entry of the file.

iv. In a text file, characters are stored one byte per character. Even numeric data is stored this way, i.e., one byte per digit of the number.

In binary files, a character occupies 1 byte, integer 2 bytes and float 4 bytes. Thus they occupy same amount of disk space as memory space.

11.4 OPERATIONS ON A FILE

C provides various functions to handle files. These functions are to

i. Open a file
ii. Read data from a file
iii. Write data to a file
iv. Close a file
v. Detect end-of-file

Before we learn more about these functions, it is essential to know about the File Pointer.

The File Pointer

A file pointer is a pointer variable of type FILE, which is defined in stdio.h. The type FILE defines various properties about the file including its name, status and current position.

Basically a file pointer identifies a specific disk file. This pointer is used by the stream associated with it to tell the I/O functions where to perform the operations.

11.4.1 Defining and Opening a File

If we want to store data in a file in the secondary memory, we must specify certain things about the file to the operating system. They include:

i. Filename
ii. Data structure
iii. Purpose

Filename is a string of characters that makes up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension. *For example*: Employee, C, input, data, PROG.C etc.

Data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. FILE is a defined data type. The process of opening of a stream for use and linking a disk file to it is called opening a file.

When we open a file, we must specify what we want to do with file. *For example*: we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

a.
```
FILE *fp;
fp = fopen ("filename", "mode");
```
OR

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

The first statement declares the variable fp as a "pointer to the data type FILE". As stated earlier, FILE is a structure that is defined in the I/O library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer which contains all the information about the file is subsequently used as communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job. Mode can be one of the following:

Mode	Meaning
"r"	Opens a file for reading. If file does not exist fopen( ) returns NULL.
"w"	Creates a file for writing. If specified file does not exist, a new one is created. If it exists, existing contents will be overwritten.
"a"	Opens a file for appending. If specified file does not exist it is created.
"r+"	Opens the file for reading and writing. If specified file does not exist, it is not created. If it exists, new data is not added to the beginning of the file.
"w+"	Creates a file for reading and writing. If the specified file does not exist, it is created. If it exists, it is overwritten.
"a+"	Opens a file for reading and appending. If the specified file does not exist, it is created. If it exists, new data is appended to the end of the file.

Note: The default mode is text. The character t can also be used along with the specified characters to indicate a text file, i.e., "rt", "w + t".

To open a file in binary mode, 'b' has to be appended to the specified modes.

i.e., "rb" "wb"

Example

```
FILE *fptr;          /* declares fptr as a pointer to type FILE */
fptr = fopen("in.txt","r");
```

The file pointer `fptr` is to be used in all subsequent read/write operations on the file in.txt.

11.4.2 Closing a File

After the operations on the file have been performed and it is no longer needed, the file has to be closed using the `fclose()` function.

- It ensures that all information associated with the file is flushed out of buffers.
- Prevents accidental misuse of the file.
- It is necessary to close a file before it can be reopened in a different mode.

Prototype: `int fclose(FILE *fp);`

Example: `fclose(fptr);`

We can also close all open streams (except the predefined ones) by using the fcloseall( ) function.

Prototype: `int fcloseall(void);`

11.4.3 End-of-file

If we know exactly how long a file is, there is no need to detect the end of file. However, in many cases, we do not know how big the file is. So its necessary to detect end of file. This can be done in two ways:

i. In text files, a special character EOF (defined in stdio.h, value-1) denotes the end of file. As soon as this character is read, the end-of-file can be detected.

ii. In binary files, the EOF is not there. Instead we can use the library function feof( ) which returns TRUE if end of file is reached. It can be used for text files as well.

Prototype: `int feof(FILE *fp);`

Example

1.
```
if(feof(fptr)== 1)
printf("File has ended");
```

2.
```
while(!feof(fptr))
{
 -
 -
 -
}
```

3. **Write a program that reads the information of the employee (name, age, city, salary) and store into employee.dat file. Also find the highest salary paid employee (Use structure/union).**

PU
Oct. 2008 – *10 M*

```
#include<conio.h>
#include<stdio.h>
struct employee
{
  char name[21];
  char city[11];
  int age;
  int salary;
};
struct employee e;
void main()
{
  FILE *fs;
  char ans = 'y';
  clrscr();
  fs = fopen("employee.dat", "w");
  if(fs == NULL)
      {
        puts("Cannot Open File");
        exit(1);
      }
        while(ans == 'y');
      {
        printf("\n Enter Ur Name, City, Age, Salary");
        scanf("%s%s%d%d", e.name, e.city, &e.age, &e.salary);
        fprintf(fs, "%s%s%d%d", e.name, e.city, e.age, e.salary);
        printf("\n Do You Want Another Records Y/N");
        fflush(stdin);
        ans = getche();
      }
      fclose(fs);
```

```
     fs = fopen("employee.dat", "a+");
     if(fs == NULL)
     {
       puts("Cannot Open File");
       exit(1);
     }
     while(fscanf(fs, "%s%s%d%d", e.name, e.city, e.age, e.salary) != EOF)
     {
       int temp = e.salary;
       for(int i = 0; i<= e.salary ; i++)
       {
       if(e.salary >= temp)
       {
       printf("%s", e.name);
       printf("%s", e.city);
       printf("%d", e.age);
       printf("%d", e.salary);
       }
   } }
   fclose(fs);
}
```

11.4.4 Reading and Writing File Data

File I/O operations can be performed in three ways:

i. Character/string input/output to read/write characters or line of characters. Although its possible to do this with binary files, these operations are commonly used with text files only. *Functions: fgetc, fputc, fgets, fputs, getw, putw*

ii. Formatted I/O to read/write formatted data. This can only be used with text mode files. *Functions: fprintf, fscanf*

iii. Direct input/output to read or write blocks of data directly. This method is used only for binary files. *Functions: fread, fwrite*

i. Reading and Writing Characters

a. **getc( ) and fgetc( ):** Both are identical (getc is a macro, fgetc is a function) and are used to input (read) a single character from the specified stream.

Prototype: `int getc(FILE *fp);`

They return a single character. When used with the stream stdin, they input a character from the keyboard.

Example:
```
ch = getc(fptr);
ch = getc(stdin);
```

b. **putc( ) and fputc( ):** Both are used to write a single character to the specified stream. If the stream is stdout, they display it in the standard output device.

Prototype: `int putc(int ch, FILE *fp);`

Example:
```
char ch = 'A';
putc(ch,fptr);
fputc(ch,stdin);
```

c. **fgets( )**: It reads a line of characters from a file.

Prototype: `char *fgets(char *str,int n, FILE *fp);`

- str points to the string where the read string has to be stored.
- n is the maximum number of characters to be read. Characters are read until a new line is encountered or (n-1) characters have been read whichever occurs first.

Example:
```
char name[80];
fgets(name,80,fptr);
```

d. **fputs( )**: Writes a line of characters to a stream. It does not add a new-line to the end of the string. If it is required, then the new line has to be explicitly put.

Prototype: `char fputs(char *str, FILE *fp);`

Example:
```
char city[] = "Pune";
fputs(city , fptr);
```

e. **getw:** It reads an integer from a file.

Prototype: `int getw(FILE *fp);`

Example:
```
int n;
n = getw(fptr);
```

f. **putw:** Writes an integer to a stream.

Prototype: `putw(int n , FILE *fp);`

Example:
```
int num = 10;
putw(num,fptr);
```

We shall now see an example using the above functions. The task is to accept characters from the keyboard till user enters EOF and store them in a file. These characters are then read from the file and displayed on the screen. The number of characters is also displayed.

Note: The above two functions are not defined in the ANSI C standard and hence may not be portable.

1. **/* Reads characters from keyboard, stores them in a file and displays from the file */**

```
#include<stdio.h>
main()
{ char c;
  FILE *fp1 ;                      /* declare pointer to FILE */
  int count = 0;
  fp1 = fopen("data.txt", "w");
/*   open txt in write mode */
  if(fp1 == NULL)
  { printf("Error opening file");
```

```
      exit(0); }
    printf("Enter the data ctrl + z to terminate \n");
    while((c = getchar()!= EOF)
      putc(c,fp1);
    fclose(fp1);                        /* close file */
    fp1 = fopen("data.txt", "r");       /* reopen file in read mode */
    if(fp1==NULL)
    { printf("\n Error opening file");
    exit();}
    printf("\n The data in the file is \n");
    while((c = fgetc(fp1))! = EOF)
    { printf("%c",c); count++;}
    printf(("\n Number of characters = %d",count);
    fclose(fp1);}
```

2. Write a C program to read a text file and copy all contents of that file into another file. When you copy the contents the source file content the words "and", "i.e.", and "e.g." are replaced with "&", "that is" and "for example" respectively.

PU
Apr. 2009 – 5 M

```
#include<stdio.h>
#include<string.h>
main()
{
  FILE *fp1, *fp2;
  char source_file_name[15], dest_file_name[15], ch, str[20];
  int i = 0;
  clrscr();
  printf("\n Enter source file name : ");
  gets(source_file_name);
  printf("\n Enter destination file name : ");
  gets(dest_file_name);

  fp1 = fopen(source_file_name,"r");
  if(fp1 == NULL)
  {
    printf("\n Unable to open source file.");
    getch();
    exit();
  }
  fp2 = fopen(dest_file_name,"w");
  if(fp2 == NULL)
  {
    printf("\n Unable to open destination file.");
    getch();
    exit();
  }
  do
  {
    ch = getc(fp1);
```

```
      if(ch != ' ' && ch != EOF && ch != '\n')
      {
        str[i] = ch;
        i++;
      }
      else
      {
        str[i] = '\0';
        i = 0;
        if(strcmpi(str,"and") == 0)
        {
          putc('&',fp2);
          putc(ch,fp2);
        }
        else if(strcmpi(str,"i.e.") == 0)
        {
          fprintf(fp2,"%s","that is");
          putc(ch,fp2);
        }
        else if(strcmpi(str,"e.g.") == 0)
        {
          fprintf(fp2,"%s","for example");
          putc(ch,fp2);
        }
        else
        {
          fprintf(fp2,"%s",str);
          putc(ch,fp2);
        }
      }
    }while(ch != EOF);
    fclose(fp1);
    fclose(fp2);
    fp1 = fopen(source_file_name,"r");
    printf("Contents of source file are : \n");
    do
    {
      ch = getc(fp1);
      printf("%c",ch);
    }while(ch != EOF);
    fp2 = fopen(dest_file_name,"r");
    printf("\n Contents of destination file are : \n");
    do
    {
      ch = getc(fp2);
      printf("%c",ch);
    }while(ch != EOF);
    getch();
  }
```

3. **Write a program to display frequency of each character in a given file.**

PU
Oct. 2010 – *10 M*

```
#include<stdio.h>
#include<conio.h>
main()
{
  char fname[11],ch;
  FILE * fp;int i;
  int ascii[255]={0};//store 0 assuming frequency zero for each
                          //character
  char store[255];//stores ascii character in array for checking
                          //character
  printf("\nEnter the file name:");
  scanf("%s",fname);
  fp=fopen(fname,"r");
  if(fp==NULL)
  { printf("\nFile does not exist...\n");
    exit(0);
  }
  for(i=0;i<255;i++)
  {
    store[i]=i;
  }
  while(1)
  {
    ch=fgetc(fp);
    if(ch==EOF) break;
    for(i=0;i<255;i++)
    {
      if(ch==store[i])
      ascii[i]=ascii[i]+1;//checks for frequency
    }
  }
  printf("\nCharacter occurrences\n");
  for(i=0;i<255;i++)
  {
    if(ascii[i]!=0)
    printf("\n%c\t\t%d",i,ascii[i]);
  }
  fcloseall();
}
```

ii. Formatted File Input / Output Functions

The functions seen above can handle only single data types. In order to deal with multiple data types, formatted file I/O functions are used. `fprintf()` is used for output `and fscanf()` is used for input.

a. **fprintf:** This is similar to `printf` except that a pointer to a file must be specified. Data is written to the file associated with the pointer.

Prototype: `int fprintf(FILE *fp, char *format, argumentlist);`

The format string is the same as used for `printf.`

The argument lists are the names of variables to be output to the specified stream.

Example:
```
char name[ ] = "ABCD",
int age = 20;
float amount = 1005.75;
fprintf(fptr, "%s%d%f", name, age, amount);
```

b. **fscanf:** This is similar to `scanf` except that input comes from a specified stream instead of `stdin.`

```
int fscanf(FILE *fp, char *format, addresslist);
```

The format string is the same as used for scanf. The address list contains the addresses of the variables where fscanf( ) is to assign the values.

Example: `fscanf(fptr,"%s%d%f", name,&age,&amount);`

Program: /* Illustrates fprintf and fscanf */

```
/* This program stores item information in a file_itemname,
Quantity and Price. It reads this info and displays the inventory -
Item, Qty, Price and Amount */
#include<stdio.h>
struct iteminfo
{ char name[10];
  int qty;
  float price;
};   /* Declare a structure to hold information */
main()
{ FILE * f1;
  struct iteminfo item;
  int i, n;
  f1 = fopen("iteminfo.txt","w");
  if(f1 == NULL)
  { printf("Error opening file");
    exit();
  }
  printf("\n How many items?");
  scanf("%d",&n);
  printf("\n Enter details as name, qty and price\n");
  for(i=0;i<n;i++)
  { fflush(stdin);
    fscanf(stdin, "%s%d%f",item.name,&item.qty,&item.price);
    fprintf(f1,"%s %d %f\n",item.name,item.qty,item.price);
  }
```

```
    fclose(f1);
    fprintf(stdout,"\n\n");
    f1 = fopen("iteminfo.txt","r"); /* Open file in read mode */
    fprintf(stdout,"Item Qty  Price  Amount");
    while(!feof(f1))
    {
    fscanf(f1,"%s%d%f", item.name,&item.qty,&item.price);
    fprintf(stdout, "%-10s%6d%10.2f%11.2f\n",item.name,item.qty,
    item.price,item.qty*item.price);
    }
    fclose(f1);
}
```

iii. Direct File Input /Output

This is used only with binary-mode files. It is used to read or write blocks of data.

i. **fwrite:** This function writes a block of data from memory to a binary mode file.

Prototype: `int fwrite(void *buf, int size, int count, FILE *fp);`

- buf is a pointer to the region of memory which holds the data to be written to the file.
- size specifies the size in bytes of individual data items.
- count specifies the number of items to be written.

Example

To write a single float variable x to a file, the statement will be.

```
fwrite(&x,sizeof(float),1,fptr);
```

To write a 100 element integer array n to the file,

```
fwrite(n,sizeof(int),100,fptr);
```

ii. **fread:** It reads a block of data from binary-mode file and assigns it to the region of memory specified. It returns the number of values read.

Prototype: `int fread(void *buf, int size, int count, FILE *fp);`

- buf is the pointer to memory that would receive data read from the file (i.e., it is the address of the variable).
- size specifies the size, in bytes of individual data items being read.
- count specifies the number of items to be read.

Examples

1. `fread(&num,sizeof(int),1,fp);`

This reads an integer from the file and assigns it to num. If we have a structure variable emp and its members have to be read from the file, fread can be used.

```
fread(&emp,sizeof(emp),1,fp);
```

2. **/* Illustrates fread and fwrite to store and read employee information from a file */**

```
#include<stdio.h>
#include<stdlib.h>
struct employee
{ char name[20],e;
  float sal;
};
main()
{
  FILE *fp;
  struct employee emp[20];
  int i,n;
  if((fp=fopen("employee.in","wb"))==NULL)
    { printf("Error opening file");
      exit();
    }
printf("\n How many employees? :");
scanf("%d",&n);
for(i=1;i<n;i++)
{
fprintf(stdout,"\n Enter the name and salary");
scanf("%s%f",e.name,&e.sal);
fwrite(&e,sizeof(e),1,fp);
}
fclose(fp);
fp=fopen("employee.in","rb");  /*reopen file */
if(fp==NULL)
{ fprintf(stderr, "Error opening file");
exit();}
/*Read data of n employees from file into array emp */
if((fread(emp,sizeof(struct employee),n,fp)!=n)
{ fprintf(stderr, "Error reading file");
    exit();}
fclose(fp);
/*Display array emp */
for(i=0;i<n,i++)
printf("\n Name = %s Salary = %f",emp[i].name,emp[i].sal);  }
```

11.4.5 Other Functions

i. **fflush:** This causes the buffer associated with an open output stream to be written to the specified file.

If it is an input stream, its buffer contents are cleared.

Syntax: `fflush(FILE *fp);`

Example: `fflush(stdin);`

ii. **remove:** This deletes the file specified. If it is open, be sure to close it before removing it.

Syntax: `int remove(const char *filename);`

Example: `remove("my.txt");`

iii. **rename:** This function changes the name of an existing disk file.

Syntax: `int rename(const char *oldname, const char *newname);`

Both files must be in the same disk drive.

Example: `rename("c:\my.txt","c:\mynew.txt");`

11.5 ERROR HANDLING DURING I/O OPERATIONS

It is possible that an error may occur during I/O operations on a file. *Typical error situations include:*

i. Trying to read beyond the end-of-file mark.
ii. Device overflow.
iii. Trying to use a file that has not been opened.
iv. Trying to perform an operation on a file, when the file is opened for another type of operation.
v. Opening a file with an invalid filename.
vi. Attempting to write a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of a program or incorrect output. So to handle this situation, we have two status - inquiry library functions. feof (as already seen) and ferror that can help us detect I/O errors in the files.

The feof function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a non-zero integer value if all the data from the specified file has been read, & returns zero otherwise. If fp is a pointer to a file that has just been opened for reading, then the statement,

```
if(feof(fp))
  printf("End of data.\n");
```

would display the message "End of data" on reaching the end of file condition. The ferror function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a non-zero integer if an error has been detected up to that point, during processing. Otherwise, it returns zero. The statement

```
if(ferror(fp)! = 0)
  printf("An error has occurred \n");
```

would print the error message, if the reading is not successful.

We know that whenever a file is opened using fopen function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not. *For example:*

```
if(fp == NULL)
   printf("\n File could not be opened");
```

The following program illustrates the use of the NULL pointer test and feof function. When we input filename as Result, the function call

```
fopen("Result", "r");
```

returns a NULL pointer because the file Result does not exist and therefore the message "Cannot open the file" is printed out.

Similarly the call `feof(fp2)` returns non zero integer when the entire data has been read, and hence the program prints the message "Ran out of data" and terminates further reading.

Program: Illustrates error handling in file operations.

```
#include<stdio.h>
main()
{
   char *filename;
   FILE *fp1, *fp2;
   int i, number;
   fp1 = fopen("Result", "w");
   for(i = 10; i<= 100; i+=10)
   putw(i, fp1);
   fclose(fp1);
   printf("\n Input filename \n");
   open_file:
   scanf("%s", filename);
   if((fp2 = fopen(filename, "r"))== NULL)
   {
      printf("Cannot open the file.\n");
      printf("type filename again.\n");
      goto open_file;
   }
   else
      for(i=1; i<=20; i++)
      {
         number = getw(fp2);
         if(feof(fp2))
         {
            printf("\n Ran out of data.\n");
            break;
         }
         else
         printf("%d\n", number);
      }
      fclose(fp2);
```

11.6 RANDOM ACCESS TO FILES

Every open file has a position pointer or a position indicator associated with it. This indicates the position where read and write operation takes place.

In all earlier programs, we read the file sequentially. However, C provides functions to control the position pointer by means of which data can be read from or written to any position in the file. These functions are:

i. **ftell:** This function is used to determine the current location of the position pointer.

Prototype: `long ftell(FILE *fp);`

It returns a long integer that gives the current pointer position in bytes from the start of the file. (i.e., it gives the offset in bytes the beginning of the file). The beginning of the file is considered at position 0.

Example:
```
fp = fopen("in.txt"), "r");
printf("%ld",ftell(fp));
```

Here the output will be 0, because when a file is opened, the position pointer points to the beginning of the file.

ii. **rewind:** This function sets the position pointer to the beginning of the file.

Prototype: `void rewind(FILE *fp);`

This function can be used if we have read some data from a file and want to start reading from the beginning of the file, without closing and reopening the file.

Example:
```
rewind(fp);
printf("%ld",ftell(fp));
```
This will yield 0 since rewind positions the pointer to the start of the file.

iii. **fseek:** More precise control over the position pointer is possible using fseek. The function fseek allows the pointer to be set to any position in the file.

Prototype: `fseek(FILE *fp, long offset, int origin);`

- offset indicates the distance in bytes that the position pointer has to be moved by
- origin indicates the reference point in the file with respect to which the pointer is moved offset number of bytes.

There can be three values for origin with symbolic constants defined in stdio.h

Constant	Value	Meaning
SEEK_SET	0	Moves the position pointer offset bytes from beginning of file.
SEEK_CUR	1	Moves the position pointer offset bytes from its current position.
SEEK_END	2	Moves the position pointer offset bytes from the end of the file.

Example

1. `fseek(fptr,0,SEEK_END);`

   This positions the pointer to the end of the file.

2.
```
struct emprec
{ _
  _
} rec ;
```
If file contains four records of struct emprec as shown:

fp →	rec1	rec2	rec3	rec4

3.
```
fseek(fp,2*sizeof(struct emprec),SEEK_SET);
```
This positions the pointer to the beginning of rec3.
```
fread(&rec, sizeof(struct emprec),fp);
```
will read the third record.

4.
```
fseek(fp,-sizeof(struct emprec), SEEK_END);
```
will position the pointer to the beginning of the 4th record.
```
fseek(fp,-2*sizeof(struct emprec),SEEK_CUR);
```
will now position it to the start of the second record.

5. ftell can be used with fseek to find the size of the file.
```
fseek(fp,0,SEEK_END);
printf("%ld",ftell(fp));
```
This will display the size of the file in bytes.

Note: 0, 1 and 2 can be used in place of SEEK_SET,SEEK_CUR and SEEK_END respectively. However, the use of symbolic constants makes the program more readable.

SOLVED PROBLEMS

1. **Accept a filename from the user and write a C program to replace all vowels in a given file with '*'.**

1 **PU Apr. 2010 – *10 M***

```
#include<stdio.h>
#include<conio.h>
main()
{
  char ch;
  FILE *fp1,*fp2;
  char fname1[11],fname2[11];//fname1 ,fname2 is a variable declared for
                  //accepting source and destination file name respectively.
  printf("\nEnter the source file name:");
  scanf("%s",fname1);
  fp1=fopen(fname1,"r");
  if(fp1==NULL)
  {
      printf("File does not exist:");
      exit(0);
   }
   printf("\nEnter the destination file name:");
   scanf("%s",fname2);
   fp2=fopen(fname2,"w");
   while(1)
   {
      ch=fgetc(fp1);
      if(ch==EOF) break;
      switch(ch)// switch is used for vowel replacement
      {
         case 'A':
         case 'a':
         case 'e':
         case 'E':
         case 'o':
         case 'O':
         case 'i':
         case 'I':
         case 'u':
         case 'U': fputc('*',fp2); break;// if character is vowel then it is
                                                //replaced
         default: fputc(ch,fp2);// character other than vowel copy it as it
                                            //is
      }
   }
fcloseall();
}
```

2. **Write a program to count no. of sentences in given file.**

1
PU
Oct. 2010 – *5 M*

```
#include<stdio.h>
#include<conio.h>
main()
{
   char fname[11],ch;
   int i;
   FILE *fp;int l=0;//number of sentences zero
   printf("\nEnter the file name:");
   scanf("%s",fname);
   fp=fopen(fname,"r");
   if(fp==NULL)
   {
      printf("\nFile does not exist...\n");
      exit(0);
   }
   while(1)
   {
      ch=fgetc(fp);
```

```
    if(ch==EOF) break;
    if(ch=='.')l++;//if.then sentence is counted
  }
  printf("\n Number of sentences=%d",l);
  fcloseall();
}
```

3. What will be the outputs? Give explanation.

PU Oct. 2009 – 4 *M*

```
char myData = 7;
FILE *fp;
fp = fopen("r" . "My_File");
fscanf("Here's a number %d" & myData);
printf("%d", myData);
```

Ans

In the fp = fopen ("r", "My_File"); statement displays two errors:

i. Cannot convert Char * to FILE*.

ii. Type mismatch in parameters.

Because the syntax of fopen() function is wrong. It takes the file name first and the mode of file, but in program file, mode is mentioned and then file name.

EXERCISES

A. Programming exercises

1. A file named DATA contains integers. Read this file and copy all even numbers into a file named EVEN and all odd numbers in the file named ODD.
2. Accept three file names as command line arguments. First two contain roll numbers and names. The roll numbers are in ascending orders. Merge these two into the third file such that the third file still remains sorted on Roll number.
3. Read a file source.txt which has data in uppercase. Convert it to lowercase and store it into target.txt.
4. Read a file and encrypt the file using command line arguments.

   For example, c:\> ENCRYPT filename
5. Write a menu driven program to manipulate employee data in a file-Employee.dat. The details to be stored are: Employee id, Employee name, Employee designation, age, basic salary. The menu should consist of the following options.

   i. Add a record ii. Delete a record iii. View all record

   iv. Modify a record v. Exit
6. Write a program to accept two filenames as command line arguments and copy the contents of source file to target file.
7. Write a program to read a file and count the number of lines, tabs and characters in the file.

B. Review questions

1. What are streams? List the five streams that are automatically opened when a program runs?
2. Differentiate between text and binary files.
3. Illustrate how a character can be written to and read from a file?
4. Explain the following functions with examples

   i. fprintf ii. fscanf iii. fread iv. fwrite
5. What is the use of the fseek function?
6. What are the different modes in which a file can be opened?
7. Explain ftell and rewind functions.

Collection of Questions asked in Previous Exams PU

1. Write a program that reads the information of the employee (name, age, city, salary) and store into employee.dat file. Also find the highest salary paid employee (Use structure/union). **[Oct. 2008 – *10 M*]**
2. Write a C program to read a text file and copy all contents of that file into another file. When you copy the content the source file content's the words "and", "i.e.", and "e.g." are replaced with "&", "that is" and "for example" respectively. **[Apr. 2009 – *5 M*]**
3. What will be the outputs? Give explanation. **[Oct. 2009 – *4 M*]**

```
char myData = 7;
FILE *fp;
fp = fopen("r" , "My_File");
fscanf("Here's a number %d" & myData);
printf("%d", myData);
```

4. Write a C program to accept the string and filename from the user by using command line argument, display the number of occurrences in a given file for a given string. **[Apr. 2010 – *10 M*]**
5. Accept a filename from the user and write a C program to replace all vowels in a given file with '*'. **[Apr. 2010 – *10 M*]**
6. Write a program to display frequency of each character in a given file. **[Oct. 2010 – *10 M*]**
7. Write a program to count no. of sentences in given file. **[Oct. 2010 – *5 M*]**

12 Bitwise Operators

12.1 INTRODUCTION

The C language offers some bit-wise operators for manipulation of bits. In this chapter, we shall study them in greater detail and see some of the important applications of these operators.

The six bit-wise operators are:

~	One's complement
>>	Right shift
<<	Left Shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

These operators operate on integer and character but not on float and double. Let us revise these operators again in brief.

Bitwise Operator

The bitwise AND, OR and XOR work according to the following rules:

Input bits		&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Examples

1. Assume that a and b are integers with values 13 and 7 respectively. Assuming that an integer occupies 2 bytes,

a in binary	=	0000	0000	0000	1101
b in binary	=	0000	0000	0000	0111
a & b	=	0000	0000	0000	0101
a | b	=	0000	0000	0000	1111
a ^ b	=	0000	0000	0000	1010

2. **Using the bitwise operator add two positive integer numbers (without using + operator).**

1 PU Apr. 2009 – *10 M*

```
#include<stdio.h>
int add(int x, int y)
{
  int a, b;
  do
  {
    a = x & y;
    b = x ^ y;
    x = a << 1;
    y = b;
  } while(a);
  return b;
}
int main(void)
{
  int num1,num2;
  clrscr();
  printf("\n Enter two numbers : ");
  scanf("%d %d",&num1,&num2);
  printf("%d + %d = %d", num1,num2,add(num1,num2));
  getch();
  return 0;
}
```

Shift Operators

The bit pattern of the data can be shifted by a specified number of positions to the left or right using the left shift (<<) and right shift (>>) operators respectively.

When the data is shifted left, the trailing empty spaces are filled with zeros.

Similarly, the leading empty spaces are zero filled when data bits are shifted right.

Example:
```
        a  = 0000  0000  0000  1101
      a<<3 = 0000  0000  0000  1000
                                 zero filled spaces
      a>>3 = 0000  0000  0000  0001
             (The rightmost three bits drop off)
```

The general **syntax** is:

```
operand1 shift_operator operand2;
```

Note: Shifting by one position to left is effectively multiplying the operand by two.
Shifting right by one position divides the operand by two.

One's Complement Operator

The ~ operator yields the one's complement of an integer, that is, it inverts each bit of the operand (1 to 0 and vice versa).

Example:
```
   If  a = 0000  0000  0000  1101
      ~a = 1111  1111  1111  0010
```

Precedence: ~ is along with other unary operators like ++ , - - and ! in hierarchy with R → L associativity.

12.2 APPLICATIONS

1
PU
Apr. 2010 – *5 M*
* Write a short note on applications of bitwise operators.

12.2.1 Masking

Masking is a process by which only the required part of the data is retained and the rest is masked. The masking can be done by using bit-wise operators to extract some portions of the data.

The most common mask used is the AND mask by using the & operator.

1
PU
Oct. 2010 – *5 M*
* Write a short note on Masking using bitwise operator.

Example

1. **An integer is stored as 16 bits. If we require only the last eight bits of the data, the data can be ANDed with the following 'mask':**

   0000 0000 11111111

   Thus, the first 8 bits of the result will be 0 (since the data bits are ANDed with 0) and the last eight bits will be the data bits.

```
            1000  0010  1001  1110
   mask     0000  0000  1111  1111
            ----------------------
            0000  0000  1001  1110
```

Thus, the portion to be retained has to be ANDed with 1 and the portion to be masked off has to be ANDed with 0.

2. **Displaying the binary equivalent of an integer, i.e., display the integer bit-wise.**

   In order to do this, we have to start from the most significant bit (MSB) which is the leftmost bit (Position 15) and check if it is 0 or 1 and display it. Next we have to consider the second MSB and so on till it reach the 0th bit (leaset significant bit).

   In order to do so we will have to use a mask such that its bit at the position under consideration is 1 and the rest are 0. For example, when we are considering the MSB of the data, the MSB of the mask should be set to 1 and rest 0. For the 2nd MSB, the 1 has to be shifted one position to the right.

   Thus, each bit of the data can be considered by successively shifting the mask one position to the right till it becomes 0.

   To find out if the data bit is 1 or 0, ANDing the mask with the data will give 0 if the data bit is 0 and non-zero if the data bit is 1.

3. **/* Displays an integer bit wise */**

```
#include<stdio.h>
main()
{ unsigned int n1, n2;
  printf("Enter the two numbers:");
  scanf("%u%u",&n1,&n2);
  printf("%u in binary is :" ,n1);
  displaybits(n1);
  printf("\n %u in binary is :",n2);
  displaybits(n2);
}
void displaybits(unsigned int n)
{
  unsigned int mask = 32768;
  /* set MSB of mask to 1 */
  while(mask>0)
  {
    if((n & mask)==0)
      printf("0");
    else
      printf("1");
mask = mask >>1; /* shift mask right */
  }
}
```

Output

```
Enter the two numbers :     65535     32768
65535 in binary is :        1111111111111111
32768 in binary is :        1000000000000000
```

4. **Masking can also be used for manipulation of hexadecimal numbers. 4 bits make up a 'nibble' and each nibble is represented in hex by 0-9 or A-F. Individual nibbles can be extracted and manipulated by using the masking method.**

   To extract the second nibble from the left of the hexadecimal number 0xA3B1, the mask used will be 0×0F00

i.e., 0×A3B1	1010	0011	1011	0001
	A	3	B	1
mask 0×0F00	0000	1111	0000	0000
	0	F	0	0
&	0000	0011	0000	0000
	0	3	0	0

5. **/* Illustrates masking with hexadecimal integers. This program interchanges the first and second nibble (from LSB) and masks the rest */**

```
#include<stdio.h>
main()
{ unsigned int hex_num, mask, n1, n2, result;
  printf("\n Enter the hexadecimal number :");
  scanf("%x",&hex_num);

/*get the first nibble and mask the rest */
  mask = 0x000F;
  n1 = hex_num & mask;

  /*get the second nibble and mask the rest */
  mask = 0X00F;
  n2 = hex_num & mask;
  /*shift nibble 1 to the position of nibble 2 */
  n1 = n1<<4;

  /*Shift nibble 2 to the position of nibble 1 */
  n2 = n2>>4;

  /*Get the result by ORing the two */
  result = n1|n2;
  printf("\n The result is %x",result);
}
```

Output

```
Enter the hexadecimal number: 3A2F
The result is F2.
```

12.2.2 Internal Representation of Date

A date is internally stored in memory as an unsigned integer (16 bits) in the following format.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
y	y	y	y	y	y	y	m	m	m	m	d	d	d	d	d
← year →							← month →				← day →				

DOS converts the actual date to this 2-byte form using the conversion.

512 * year + 32 * month + day

For example, if the date is 11-01-81, the converted date will be

512 * 81 + 32 * 1 + 11 = 41515, which will be stored as

1	0	1	0	0	0	1	0	0	0	1	0	1	0	1	1
Year							month				day				

In order to extract information about the year, month and day, left and right shift operators have to be used.

To obtain

i. day : shift data left by 11 bits followed by right shift by 11 bits.
ii. month : shift left by 7 and then shift right by 12
iii. year : shift right by 9.

Program: /* Illustrates conversion of date to bit format */

```
#include<stdio.h>
main()
{
  unsigned int date, day, month,year;
  printf("Enter the date,month and year-dd. mm. yy:");
  scanf("%u%u%u",&date, &month, &year);
  /*Convert to a 2 byte format */
  date = 512 * year + 32 * month + day;
  printf("\n In binary, date = :");displaybits(date);
  /*Extraction */
  day = ((date<<11)>>11);
  month = ((date<<7)>>12);
  year = date>>9;
  printf("\n Day = %u \n", day);
  displaybits(day);
  printf("\n Month = %u\n", month);
  displaybits(month);
  printf("\n Year = %u\n", year);
  displaybits(year);
}
```

Output

```
Enter the date, month and year –dd mm yy : 31 12 89
Day = 31
0000000000011111
Month = 12
0000000000001100
Year = 89
0000000001011001
```

12.3 BIT FIELDS

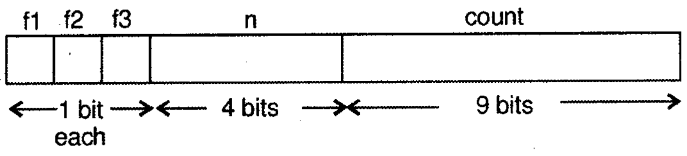
When storage space is limited, it may be necessary to pack several objects into a single memory word.

Some applications require data with very small values. Here too, we can pack information into the bits of a byte or a word if we do not need to use the entire word to represent data.

Example: We use flags to represent Boolean TRUE or FALSE, i.e., 1 or 0. These can be represented in a single bit. However declaring them as a char or int will use 8 bits and 16 bits respectively.

A method to define a structure of packed information is known as a bit field. The syntax of definition and access is based on structures.

Example: In order to store the values of flags, f1, f2, and f3, an integer whose range is from 1 to 12 and an integer_count whose value ranges from 0 to 500, the assignments could be



The bit field declaration will be

```
struct data
{ unsigned int f1 : 1;
  unsigned int f2 : 1;
  unsigned int f3 : 1;
  unsigned int n : 4;
  unsigned int count : 9;
} d1;
```

The individual members can be accessed as d1.f1, d1.f2, etc.

Initialization

The variable d1 can be initialized as

```
struct data
{ _
  _
} d1 = {0,1,1,10,250};
```

This assigns

d1.f1 = 0

d1.f2 = 1

d1.f3 = 1

d1.fn = 10

d1.count = 250

SOLVED PROGRAM

Write a 'C' program to convert a decimal number in its binary format using bitwise operators.

```
#include<stdio.h>
main()
{ int dec;
  unsigned int mask=32768;
  printf("\nEnter the decimal number :");
  scanf("%d",&dec);
  while(mask>0)
  {  if((dec & mask) == 0)
       printf("0");
     else
       printf("1");
       mask=mask>>1;
  }
 getch();
```

EXERCISES

A. Predict the output

1.
```
main()
{  union bbb }
       { struct
          int a:1;
          int b:1;
          int c : 1;
          int d : 1;
          int e : 1;
          int f : 1;
          int g : 1;
          int h : 1;
       } aaa;
   char x;
union bbb pqr;
pqr.aaa.a = pqr.aaa.b = pqr.aaa.c = pqr.aaa.d = 1;
pqr.aaa.e = pqr.aaa.f = pqr.aaa.g = pqr.aaa.h = 1;
printf("%d",pqr.x);   }
```

B. Programming exercises

1. Write a function rightrot (x,m) which rotates integer x right by m position and returns the result.
2. Write a function leftrot (x,m) which rotates bits of integer x to the left by m position.
3. Write a program to accept a hexadecimal number and reverse the nibbles of the number, i.e., 3A25 should be reversed to 52A3.
4. Accept a hexadecimal number and a nibble number (1,2,3 or 4). Convert the number such that only the specified nibble is inverted and all others remain the same.
5. Write a function setbits (x,m,y) that returns x with its leftmost m bits set to the right most m bits of y.
6. Write a program to find the number of bits required for storing a character, integer and long integer without using the sizeof( ) operator.
7. Write a program to convert an integer into its equivalent binary number using bitwise operators.

   Hint: use masking.

C. Review questions

1. What is masking? Explain giving an example.
2. What are bit fields?
3. How is date internally stored? Show the format.
4. Explain the bitwise operators giving examples.

Collection of Questions asked in Previous Exams PU

1. Using the bitwise operator add two positive integer numbers (without using + operator). **[Apr. 2009 – *10 M*]**
2. Write a short note on Applications of bitwise operators. **[Apr. 2010 – *5 M*]**
3. Write a note on Masking using bitwise operator. **[Oct. 2010 – *5 M*]**

VISION

13 Graphics In C

13.1 INTRODUCTION

So far we have limited ourselves to text input and output. However in order to make the output 'look' better or to design an application which uses images, shapes, etc. we have to use graphics. All computer games, animation, multimedia applications extensively use graphics.

In this chapter, we shall be studying some basic graphics concepts and see the use of simple graphics functions in the C languages.

13.2 BASIC CONCEPTS

i. **Graphics.h, Graphics.lib:** Graphics.h is a header file that contains the definitions of constants and prototypes of graphics functions.

   Graphics.lib is a library file, which contains function definitions.

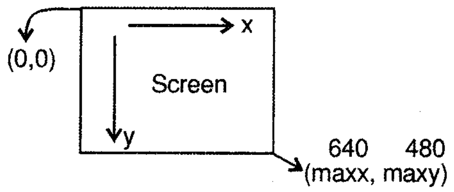
ii. **Graphics mode:** In order to use graphics in the program, a user has to switch from the default 'text' mode to the graphics mode. There are various modes depending upon the monitor used and the display adapter.

   Automatic selection of a mode can be done by the standard function initgraph( ).

iii. **Resolution:** The display screen (for graphics) is divided into number 'dots' called pixels. The more the pixels the clearer is the image.

   The total numbers of pixels on the screen in the graphics is called the resolution.

Example: The Video Graphics Array (VGA) adapter provides a maximum resolution 640×480 pixels in 16 colours.



iv. **initgraph( ):** This standard library function selects the graphics mode offering the best resolution and stores the corresponding mode number in the variable **gm.**

v. **Graphics drivers:** Device drivers are programs, which communicate with specific devices. A program communicates with these drivers, which in turn communicate with various devices.

Graphics drivers are device drivers applicable only in the graphics mode. Turbo C offers many graphics derivers (with extension BGI). One of them has to be selected depending upon the adapter used.

vi. **DETECT macro:** DETECT is a predefined macro which does the task of selecting the appropriate graphics driver. This value has to be stored into a variable for further use (gd).

vii. **Exiting the graphics mode:** The graphics mode can be exited by using the closegraph( ) function. This function deallocates the memory allocated to various graphics objects and exits the graphics mode.

viii. **Restoring the text mode:** The function restorecrtmode( ) can be used to restore the screen mode to the settings prior to the graphics settings.

Simple Graphics Program

To enter the graphics mode, display the text "Welcome to Graphics" in the center of the screen and then exit the graphics mode.

```
#include<graphics.h>
main()
{ int gm, x,y, gd = DETECT;
  initgraph(&gd, &gm, "");
  x = getmaxx();   /* Get maximum screen coordinates */
  y = getmaxy();
  outtextxy(x/3, y/2, "Welcome to Graphics");
  getch();
  closegraph();
  restorecrtmode();
}
```

i. getmaxx( ) and getmaxy( ) are functions which return the maximum x and y coordinates respectively in the current screen mode.

ii. outtextxy is a function which displays text at the specified x and y coordinates on screen.

13.3 DRAWING SIMPLE GRAPHICS OBJECTS

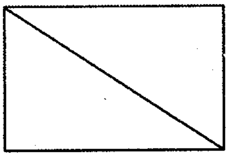
13.3.1 Drawing a Line

Method 1: Using the line() function

Syntax: `void line(int x1, int y1, int x2, int y2)`

It draws a line in the current colour using the current line style and thickness, between the two points (x1, y1) and (x2, y2) without updating the **current position (CP).**

Example:
```
x = getmaxx();
y = getmaxy();
  line(0,0,x,y)
```

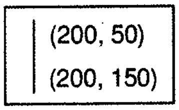


Method 2: Using moveto() and lineto()

The cp can be changed to a specific position using moveto( ) and the lineto( ) function can be used to draw a line from the cp to a specified position.

After lineto( ) the cp changes to the end point of the line.

Example
```
moveto(200, 50);    /* change C.P. to (200, 50) */
lineto(200, 150);   /* draws a line from (200, 50) to (200, 150) */
```



Method 3: Using linerel( )

This function draws a line relative to the c.p. The cp can be changed using either moveto( ) or moverel( ) functions. The c.p. is advanced by the specified offsets

Syntax: `void linerel(int dx, int dy);`

Example:
```
moveto(200, 50);
linerel(0,100);
```

13.3.2 Setting Line Style

Lines of different styles can be drawn by using the **setlinestyle( )** function followed by any line drawing function.

Syntax: `setlinestyle(int linestyle, unsigned upattern, int thickness)`

The linestyles enumerated in graphics.h are:

0	Solid Line
1	Dotted Line
2	Center Line (Alternate dots and dashes)
3	Dashed Line
4	User-defined line

The second parameter is applicable only if the first is user-defined.

The thickness parameter can take values from 1 to 3.

Example:

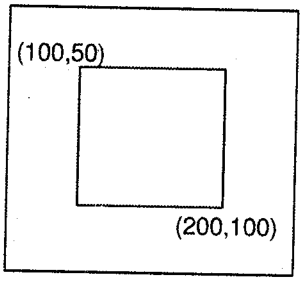
```
setlinestyle(3,15,1);
line(0,0,200,150);
```

13.3.3 Drawing a Rectangle

The rectangle( ) function draws a rectangle in the current line style, thickness and colour. It does not fill the rectangle.

Syntax: `void rectangle(int left, int top, int right, int bottom)`

Example: `rectangle(100,50,200,100);`



13.3.4 Drawing a Circle

The circle( ) function is used to draw a circle with specified center and radius.

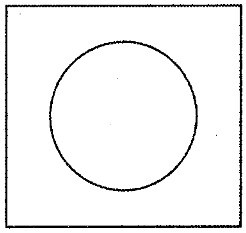
Syntax: `void circle(int x, int y, int radius)`

Example:

```
x = getmaxx();
y = getmaxy();
circle(x/2, y/2, 75);
```

Program: **Write a C program to display the following output:**

PU
Apr. 2010 – *5 M*



```
#include<stdio.h>
#include<graphics.h>
main()
{
   int gm,x,y,gd=DETECT;//macro
   initgraph(&gd,&gm,"");// graphics mode is initialized
   circle(125,75,20);
   rectangle(100,50,150,100);
   restorecrtmode();
   getch();
}
```

13.3.5 Drawing an Ellipse

The ellipse( ) function can be used to draw an elliptical arc in the current thickness and colour. The linestyle does not affect the ellipse.

Syntax: `void ellipse(int x, int y, int startangle, int endangle, int xradius, int yradius)`

- centre of ellipse – (x,y)
- x radius and y radius are the horizontal and vertical axes respectively.

Example: `ellipse(x/2, y/2, 0, 360, 100, 50);`

13.3.6 Drawing an Arc

The arc function draws an arc from a specific start angle to end angle with respect to a specified centre and with a given radius.

Syntax: `void arc(int xc, int yc, int startangle, int endangle, int rad);`

Example:
```
x = getmaxx();
y = getmaxy();
arc(x/2, y/2, 0, 90, 100);
```

13.3.7 Drawing Polygons

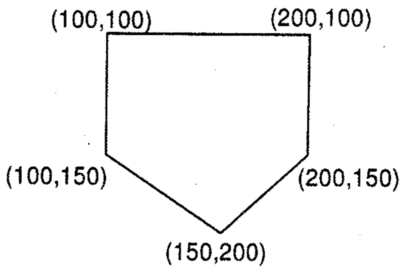
A polygon with n vertices can be drawn using the **drawpoly** function.

Syntax: `void drawpoly(int num, int * polypoints);`

This function draws a polygon with num-1 vertices using the current linestyle, thickness and colour. Polypoints points to an array of num *2 integers. Each pair of integers gives the x and y coordinates of a vertex. Coordinates of (num) vertices have to be given where the coordinates of first vertex matches co-ordinates of num th vertex.

Example

```
int coords[] = {100,100,200,100,200,150,150,200,100,150,100,100};
drawpoly(6, coords);
```



13.3.8 Filling Images

The setfillstyle function is used to set a fill pattern and colour. There are many predefined fill styles as shown in the table.

Name	Value	Description
EMPTY_FILL	0	Fill with background colour
SOLID_FILL	1	Solid fill
LINE_FILL	2	Fill with -----
LTSLASH_FILL	3	Fill with ////
SLASH_FILL	4	Fill with ////,thick lines
BKSLASH_FILL	5	Fill with \\\\, thick lines
LTBKSLASH_FILL	6	Fill with \\\\
HATCH_FILL	7	Light hatch fill
XHATCH_FILL	8	Heavy cross-hatch fill
INTERLEAVE_FILL	9	Interleaving line fill
WIDE_DOT_FILL	10	Widely spaced dot fill
CLOSE_DOT_FILL	11	Closely spaced dot fill
USER_FILL	12	User-defined fill pattern

Syntax: `void setfillstyle(int pattern, int colour);`

The colour parameters can take a value from 0 to 15. A user can also specify the colour name instead of its integer value.

Example:
```
setfillstyle(4,MAGENTA);
bar(100, 100, 200, 200);
rectangle(100, 100, 200, 200);
```

Note: The bar function is similar to rectangle but it does not draw the boundary but fills the inside whereas the rectangle function does not fill the inside.

13.3.9 Pattern with a Difference

The setfillpattern( ) is like setfillstyle( ), except that you use it to set a user_defined 8×8 pattern rather than a predefined pattern.

The setfill pattern function is used to select a user defined fill pattern.

Syntax: `setfillpattern(char * pattern, int colour)`

The first parameter is a pointer to a sequence of 8 bytes, with each byte corresponding to 8 pixels in the pattern. Whenever a bit in a pattern byte is set to 1, the corresponding pixel will be plotted. The second parameter specifies the colour in which the pattern would be drawn.

13.3.10 Setting Colours

The setcolour function is used to set a specific drawing colour.

Syntax: `void setcolor(int color);`

The color parameter can take a value from 0 to 15 (as shown in table below). The current color is used to draw graphics and output text.

Colour	Value
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHT GRAY	7
DARK GRAY	8
LIGHT BLUE	9
LIGHT GREEN	10
LIGHT CYAN	11
LIGHT RED	12
LIGHT MAGENTA	13
YELLOW	14
WHITE	15

Example:
```
setcolour(WHITE)
circle(x/2, y/2, 100);
```

13.3.11 Setting Background Colours

The setbkcolour function is used to set the current background colour using the palette.

Syntax: `void setbkcolor(int color);`

The colour parameter can take a value from 0 to 15. If you use EGA (Enhanced Graphics Adapter) or a VGA (Video Graphics Array) and you change the palette colours using setpalette( ) or setallpalette( ). The color value you use might not give you the correct color. This is because the parameter to setbkcolor( ) indicates the entry number in the current palette rather than a specific color.

13.3.12 Setting Palette Colours

The setpalette function is used to change one palette colour.

Syntax: `void setpalette(int colornum., int color);`

Setpalette( ) changes the colournum entry in the palette to colour. The valid colours depend on the current graphics driver and current graphics mode. The setallpalette function is used to change all palette colours.

Syntax: `void setallpalette(struct palettetype * palatte);`

The setallpalette( ) function sets the current palette to the values gives in the palettetype structure pointed to by palette.

You can partially or completely change the colours in the EGA/VGA palette with setallpalette( ).

The MAXCOLORS constant and the prototype structure used by setallpalette( ) are

```
#define MAXCOLORS 15
struct palettetype
{  unsigned char size;
   signed char colors[MAXCOLORS + 1];  };
```

Size gives the number of colours in the palette for the current graphics driver in the current mode.

Colours is an array of size bytes containing the actual row colour numbers for each entry in the palette. If an element of colours is –1, the palette colour for that entry is not changed.

13.3.13 Filling regular and non regular shapes

To fill regular shapes like polygons and ellipses there exist standard library functions like fillpoly( ) and fillellipse( ). These functions fill the polygon or ellipse with the current fill style and current fill colour that may have been set up by calling setfillstyle( ) or setfillpattern( ).

i. **fillpoly( ):** The fillpoly( ) function draws and fills a polygon.

   Syntax: `void fillpoly(int num, int *polypoints);`

   fillpoly( ) draws the outline of a polygon with num points in the current line style and colour and then fills the polygon using the current fill pattern and fill colour, polypoints points to a sequence of (num * 2) integers. Each pair of integers gives the x and y coordinates of a point on the polygon.

ii. **fillellipse( ):** The fillellipse( ) function draws and fills an ellipse.

Syntax: `void fillellipse(int x, int y, int xrad, int yrad);`

fillellipse( ) draws an ellipse using (x, y) as a center point and xrad and yrad as the horizontal and vertical axes and fills it with the current fill colour and fill pattern.

To fill non-regular shapes like the interesting area between an overlapping traingle and circle the floodfill( ) function is used repeatedly.

floodfill( ): The floodfill( ) function flood-fills the bounded region.

Syntax: `void floodfill(int x, int y, int border);`

(x, y) is a 'seed' point within the enclosed area to be filled. The area bounded by the colour border is flooded with the current fill pattern and fill colour.

13.4 OUTPUTTING TEXT

There are various functions in graphics.h which are used to display text and change text settings.

13.4.1 outtextxy() and outtext()

The outtextxy( ) function displays a string at the specified location using the current settings(justification, colour direction, font and size)

Syntax: `void outtextxy(int x, int y, char *s);`

Example:

```
x = getmaxx();
y = getmaxy();
outtextxy(x/2, y/2, "Hello");
```

The outtext ( ) function is similar to the above but it displays the string at the current position (c.p.)

Syntax: `void outtext(char *s);`

Example: `outtext("Hello");`

13.4.2 Changing Text Setting

The settextstyle function allows setting characteristics for text output. It sets

i. font
ii. text direction
iii. character size

Syntax: `void settextstyle(int font, int direction, int charsize);`

iv. font can take a value from 0 to 4
v. Direction can take a value from 0 to 1.
vi. Charsize can take a value from 0 to 7.

Fonts	
Name	Value
DEFAULT_FONT	0
TRIPLEX_FONT	1
SMALL_FONT	2
SANS_SERIF_FONT	3
GOTHIC_FONT	4

Direction	
Symbolic Name	Value
HORIZ_DIR	0
VERT_DIR	1

Example:
```
for(i=0;i<=10;i++)
{  settextstyle(i,HORIZ_DIR, 2);
   outtextxy(x,y, "DEMO");
   x += 20;
   y += 20;  }
```

13.4.3 Text Justification

We can also control the justification (positioning) the text with respect to the C.P. The settextjustify( ) function can be used for this purpose.

Syntax: `void settextjustify(int horiz, int vert);`

The standard values are:

Name	Value	Description
LEFT_TEXT	0	Horizontal
CENTRE_TEXT	1	Horizontal and vertical
RIGHT_TEXT	2	Horizontal
BOTTOM_TEXT	0	vertical
TOP_TEXT	2	vertical

The default values are LEFT_TEXT (for horiz) and TOP_LEFT (for vertical)

Example:
```
settextjustify(1,1);
outtext("Text Demo");
```

13.4.4 Finding text height and width

The textheight and textwidth functions returns the height and width of a string in pixels.

Syntax:
```
int textheight(char *s);
int textwidth(char *s);
```

Example:
```
char str[] = "Demo";
for(j = 1; j< = 4; j++)
{  settextstyle(2,HORIZ_DIR, j);
   outtextxy(x,y, str);
   y + = (textheight(str) + 10);
   x + = (textwidth(str) + 10); }
```

SOLVED PROGRAMS

1. **Write a function to display a bar graph for the runs per over.**

PU
Apr. 2009 – *5 M*

```
#include<graphics.h>
#include<stdlib.h>
#include<stdio.h>
#include<conio.h>
int main(void)
{ int gdriver = DETECT, gmode, errorcode;
  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
   if(errorcode != grOk)
   {  printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1);
   }
   line(10,20,10,250);
   line(10,250,250,250);
   setfillstyle(USER_FILL, getmaxcolor());
   bar(50,250,30,100);
   bar(80,250,100,80);
   bar(140,250,120,100);
   getch();
   closegraph();
   return 0;
}
```

2. **Write a program to demonstrate fill color in circle.**

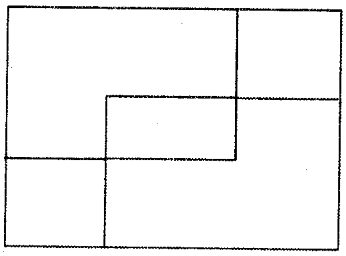
PU
Oct. 2010 – *10 M*

```
#include<stdio.h>
#include<graphics.h>
#include<math.h>
#include<conio.h>
void main(void)
{ int driver,mode,n;
  float angle;
  char string[80];
  driver=DETECT;/*autodetect driver for graphics*/
  clrscr();
  initgraph(&driver,&mode,"");//initializes graphics mode
  setcolor(BLUE);/*sets the color of the circle*/
  circle(300,250,200);
  /*thus,there are now two closed areas bounded by blue*/
  setfillstyle(SOLID_FILL,YELLOW);//fills solid yellow color
  floodfill(300,75,BLUE);
  getch();
  restorecrtmode();/*restores the screen to the original screen */
}
```

EXERCISES

A. Programming exercises

1. Write a program to draw a rectangle in the centre of the screen and fill it with all the fillstyles and all colours.
2. Write a program to display 3 rectangles such that they occupy the screen diagonal as shown. Calculate the height and width of the rectangles.

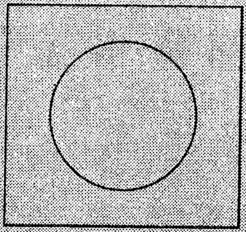


B. Review questions

1. Define the following terms: cp, Resolution, pixel.
2. What do you mean by graph driver and graph mode?
3. Explain the purpose of initgraph( ) and the DETECT macro.
4. Explain the following functions giving their syntax, usage and example.
   i. rectangle ii. circle iii. arc iv. ellipse
5. Which are the various functions that can be used to draw a line.
6. How can line settings be changed?
7. How can polygons be drawn and filled?
8. Explain different functions for outputting text.

Collection of Questions asked in Previous Exams PU

1. Write a function to display a bar graph for the runs per over. [Apr. 2009 – 5 M]
2. Write a C program to display the following output: [Apr. 2010 – 5 M]



3. Write a program to demonstrate fill color in circle. [Oct. 2010 – 10 M]

14 Command Line Arguments

14.1 INTRODUCTION

So far we have been using main with an empty pair of parenthesis. In environments that support C, there is a way to pass arguments or parameters to main when it begins executing, i.e., at runtime.

1
PU
Apr. 2009 – 5 M
★ Write a short note on Command line arguments.

These arguments are called command line arguments because they are passed from the command line during run time.

main is called with two arguments:

i. **int argc:** argument count which is the number of command line arguments the program was called or invoked with.

ii. **char * argv[ ]:** Argument vector. It is an array of pointers each pointing to a command line argument.

Declaration of main

When main has to accept command line arguments, it has to be declared differently. It is declared as

```
main(int argc, char *argv[])
{  ____
   ____
   ____  }
```

i. The subscripts for argv[ ] are 0 to argc-1.

ii. argv[0] is the name of the executable file of the program.

iii. It is not necessary to use the words argc and argv and any others will also do. However, they are used conventionally, so it is better to stick to them.

iv. The arguments have to be separated by white spaces. If a space is to be given as a part of an argument, the argument along with the spaces can be specified in double quotes.

Examples

1. A simple program is the program display which echoes its command line arguments on the screen. If the command is given as

```
Display argument1 10 abcd
```

The output should be

```
argument1 10 abcd
```

For this example argc = 4 and the arguments will be stored as:

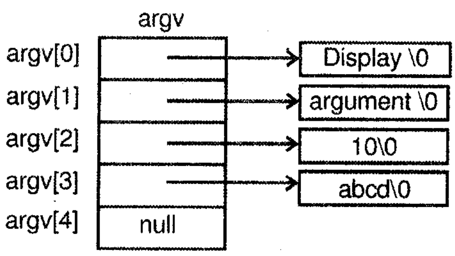


Figure 14. 1

The program will be:

Displays command line arguments.

```
#include<stdio.h>
main(int argc, char *argv[])
{
int i;
for(i = 1; i< argc ; i++)
  printf("%s%s",argv[i]," ");
}
```

2. **Write a program using command line arguments to find reverse of a given three digit number.**

1
PU
Oct. 2009 – *5 M*

```
#include<stdio.h>
#include<conio.h>
void main(int argc, char *argv[])
{
  clrscr();
  while(--argc >= 0)
  {
    printf("%s        %s", argv[argc], "     ");
  }
  getch();
}
```

14.2 ADVANTAGES OF COMMAND LINE ARGUMENTS

i. Arguments can be supplied during runtime. Therefore the program can accept different arguments at different times.

ii. There is no need to change the source code to work with different inputs to the program.

Example: If a program is to be written without using command line arguments for copying the contents of one file to another, both filenames will have to be specified in the program.

By using command line arguments, the program can be run with different file names every time since the code in the program will refer to them using argv[ ].

iii. There's no need to recompile the program since the source code is not changed.

A user can specify file names as command line arguments and the program can perform operations on the specified file.

The same program can be used for different files since the filenames will be supplied at runtime.

Example as follows

A program to copy the contents of one file to another can be run for different source and target files since the program accepts them as command line arguments.

The following program illustrates copying one file to another. The filenames are accepted as command line arguments.

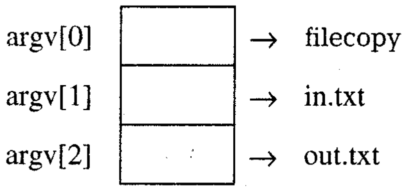
Filecopy using command line arguments.

```
#include<stdio.h>
main(int argc,char *argv[])
{ FILE fp1,fp2;    /*File pointers for the two files  */
  char ch ;
  if(argc!=3) {
    printf("\n Invalid number of arguments");
    printf("\n Usage is <prog name> <source> <target>");
    getch();
    exit();  }
    if((fp1 = fopen(argv[1],"r"))== NULL) {
    printf("\n Error opening source file");
    exit();  }
    if((fp2 =fopen(argv[2],"w"))== NULL)  {
    printf("\n Error opening target file");
    exit();  }
  while (!feof(fp1))       /*copy from source to */
  {                          /*target character by */
    ch = getc(fp1);        /*character*/
    putc(ch,fp2);    }
    fclose(fp1);
    fclose(fp2);
  }
```

To run this program if the following arguments are given at the command line.

filecopy in.txt out.txt

They will be stored as



Note: A program, which uses command line arguments, can be executed as follows:

i. By selecting the "Arguments" option from the menu bar and then specifying the arguments.

ii. From the DOS prompt. An executable program has to be first created and then the file name along with the arguments can be given at the system prompt.

(For the above examples, filecopy.exe has to be created first)

SOLVED PROGRAMS

1. **Encrypt a.txt 2**

Program to accept a filename and a key as command line arguments and encrypt the file using the key.

```
#include<stdio.h>
main(int argc, char * argv[])
{ FILE *fp1, *fp2;
  int key;
  char ch;
  fp1 = fopen(argv[1], "r");
  fp2 = fopen("out.txt", "w");
  if((fp1 == NULL) || (fp2 == NULL))
  { printf("Error");
    exit();   }
  key = atoi(argv[2]);     /*Convert the string to integer */
  while(!feof(fp1))
  { ch = fgetc(fp1);
    fputc(ch+key, fp2);     }
  fcloseall();
}
```

2. Using command line accepts source file name and target file name. Copies the all contents of source file into target file and save the target file. Your program should handle the errors.

PU
Oct. 2008 – *10 M*

Ans

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main(int argc, char *argv[])
{
FILE *sf, *tf;
char ch;
if(argc!= 3)
{
    puts("Improper Number of Arguments");
    exit();
}
sf = fopen(argv[1],"r");
if(sf == NULL)
{
      puts("Cannot open the source file");
      exit();
}
      tf = fopen(argv[2], "w");
      if(tf == NULL)
      {
        puts("cannot open target file");
        fclose(tf);
        exit();
      }
      while(1)
      {
        ch = fgetc(sf);
        if(ch == EOF)
        break;
        else
        fputc(ch, tf);
      }
        fclose(sf);
        fclose(tf);
}
```

EXERCISES

A. Programming exercises

1. Accept three file names as command line arguments.

   First two contain roll numbers and names. The roll numbers are in ascending orders. Merge these two into the third file such that the third file still remains sorted on roll number.
2. WAP to accept the filenames as command line arguments, concatenate contents of second file to first and store in the third.
3. WAP to display the command line arguments with the maximum number of characters.

B. Review questions

1. Is it necessary to use the words argc and argv to store Command Line Arguments?
2. What are the Command Line Arguments?
3. What are the advantages of Command Line Arguments?
4. What is the significance of argv[0]?

Collection of Questions asked in Previous Exams PU

1. Using command line accepts source file name and target file name. Copies the all containt of source file into target file and save the target file. Your program should handle the errors. **[Oct. 2008 – *10 M*]**
2. Write short note on Command line argument. **[Apr. 2009 – *5 M*]**
3. Write a program using command line argument to find reverse of a given three digit number. **[Oct. 2009 – *5 M*]**
4. Write a program, which will accept file name and string from command prompt and append the string in given file. **[Oct. 2010 – *10 M*]**

Suggestive Readings:

1. [Ritchie,1993] Ritchie, D. The development of the C language. ACM Sigplan Notices. 28, 201-208 (1993)
2. [Kernighan Ritchie,1978] Kernighan, B. & Ritchie, D. The C Programming Language Prentice Hall. Englewood Cliffs, New Jersey. (1978)
3. [Balasubramanian,2016] Balasubramanian, S. A brief history of the C programming language. International Journal of Computer Applications. (2016)
4. [Alexander,2019] Alexander, M. The advantages of learning C programming. (educba,2019), https://www.educba.com/advantages-of-learning-C-programming/
5. [Doyle,2013] Doyle, B. C Programming: From Problem Analysis to Program Design. (Cengage Learning,2013)
6. [Donovan,2019] Donovan, B. A brief history of C. (IEEE Computer Society,2019)
7. [Ghezzi, et. al.,2018] Ghezzi, C., Jazayeri, M. & Mandrioli, D. Fundamentals of software engi-neering. (Prentice-Hall, Inc.,1991)