

FOUNDATIONS OF COMPUTER PROGRAMMING

Edited By
Dr. Anil Sharma

ISBN: 978-93-87034-75-4

Printed by
EXCEL BOOKS PRIVATE LIMITED
Regd. Office: E-77, South Ext. Part-I, Delhi-110049
Corporate Office: 1E/14, Jhandewalan Extension, New Delhi-110055


+91-8800697053, +91-011-47520129


info@excelbooks.com/ projects@excelbooks.com
internationalalliance@excelbooks.com


www.excelbooks.com



for
Lovely Professional University
Phagwara

CONTENTS

Unit 1:	Foundation of Programming Languages *Manmohan Sharma, Lovely Professional University*	1
Unit 2:	Introduction to C Language *Kumar Vishal, Lovely Professional University*	19
Unit 3:	Basics - The C Declaration *Anil Sharma, Lovely Professional University*	36
Unit 4:	Operators *Yadwinder Singh, Lovely Professional University*	48
Unit 5:	Managing Input and Output in C *Anuj Sharma, Lovely Professional University*	61
Unit 6:	Decision-making and Branching *Balraj Kumar, Lovely Professional University*	91
Unit 7:	Decision-making and Looping *Mandeep Kaur, Lovely Professional University*	126
Unit 8:	Arrays *Kanika Sharma, Lovely Professional University*	155
Unit 9:	Strings *Sarabjit Kumar, Lovely Professional University*	168
Unit 10:	Pointers *Anil Sharma, Lovely Professional University*	187
Unit 11:	Functions *Anil Sharma, Lovely Professional University*	209
Unit 12:	Union and Structure *Sarabjit Kumar, Lovely Professional University*	237
Unit 13:	File Handling in C *Anil Sharma, Lovely Professional University*	266
Unit 14:	Additional in C *Avinash Bhagat, Lovely Professional University*	282

SYLLABUS

Foundations of Computer Programming

Objectives: It imparts programming skills to students. Students will be able to:

- Understand the structure of a C/C++ language program including the use of variable definitions, data types, functions, scope and operators.
- Be able to develop, code, and test well structured programs using: if-then logic, while, do, and for loops, functions, arrays, strings and string functions

Sr. No.	Description
1.	***Introduction:*** ANSI C standard, Overview of Compiler and Interpreters, Structure of C Program, Programming rules, Execution
2.	***Basics – The C Declarations:*** C Character Set, keywords, **:** Identifiers, data types, operators, constants and variables **Operators & Expressions**
3.	***Input/ Output in C:*** Formatting input & output functions.
4.	***Decision-making Statements:*** if, else if **Control Statements:** For, do while, while. Control transfer statements - break, continue.
5.	***Arrays and Strings:*** Defining arrays; I/O of arrays, I/O of string data; built-in library functions to manipulate strings, array of strings
6.	***Pointer:*** Introductions, Features, Declaration, Pointers and Arrays, pointers to pointers ,Pointers and strings, Void Pointers
7.	***Functions:*** Defining and accessing a functions, passing arguments – call by value, function prototypes, recursive functions **Storage Classes:** Storage classes and their usage
8.	***Structures & Unions:*** Defining and processing structures, array of structures, nested structures, Unions & difference from Structures
9.	***Files:*** Opening, reading, writing & Closing file
10.	***Additional in C:*** Dynamic memory allocation, Memory models, Linked List

Unit 1: Foundation of Programming Languages

Notes

CONTENTS

Objectives

Introduction

1.1 Programming Language

1.2 Assembly Language

1.3 Assembly Program Execution

1.4 Assembler

1.5 Assembly Program and its Components

1.6 Machine Level Language

1.7 Higher Level Languages

1.8 Compiling High Level Language

1.9 Some High Level Languages

1.10 Summary

1.11 Keywords

1.12 Self Assessment

1.13 Review Questions

1.14 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain programming language
- Describe assembly language
- Describe how to execute assemble program
- Explain higher level language
- Describe some high level language

Introduction

Computer is an electronic device which works on the instructions provided by the user. As the computer does not understand natural language, it is required to provide the instructions in some computer understandable language. Such a computer understandable language is known as Programming language.

A computer programming language consists of a set of symbols and characters, words, and grammar rules that permit people to construct instructions in the format that can be interpreted by the computer system.

Notes

Computer Programming is the art of making a computer do what you want it to do.

Computer programming is a field that has to do with the analytical creation of source code that can be used to configure computer systems. Computer programmers may choose to function in a broad range of programming functions, or specialize in some aspect of development, support, or maintenance of computers for the home or workplace. Programmers provide the basis for the creation and ongoing function of the systems that many people rely upon for all sorts of information exchange, both business related and for entertainment purposes.

1.1 Programming Language

Different programming languages support different styles of programming. The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute.

The basic instructions of programming language are:

1. ***Input:*** Get data from the keyboard, a file, or some other device.
2. ***Output:*** Display data on the screen or send data to a file or other device.
3. ***Math:*** Perform basic mathematical operations like addition and multiplication.
4. ***Conditional execution:*** Check for certain conditions and execute the appropriate sequence of statements.
5. ***Repetition:*** Perform some action repeatedly, usually with some variation.

1.2 Assembly Language

Assembly languages are also known as second generation languages. These languages substitute alphabetic symbols for the binary codes of machine language.

In assembly language, symbols are used in place of absolute addresses to represent memory locations.

Mnemonics are used for operation code, i.e., single letters or short abbreviations that help the programmers to understand what the code represents.

e.g.: MOV AX, DX.

Here mnemonic MOV represents 'transfer' operation and AX, DX are used to represent the registers.

One of the first steps in improving the program preparation process was to substitute letter symbols mnemonics for the numeric operation codes of machine language. A mnemonic is any kind of mental trick we use to help us remember. Mnemonics come in various shapes and sizes, all of them useful in their own way.



Example: A computer may be designed to interpret the machine code of 1111 (binary) or 15 (decimal) as the operation 'subtract', but it is easier for human being to remember is as SUB.

Notes

Use of Symbols Instead of Numeric of OpCodes

All computers have the power of handling letters as well as numbers. Hence, a computer can be taught to recognize certain combination of letter or numbers. It can be taught to substitute the number 14 every time it sees the symbol ADD, substitute the number 15 every time it sees the symbol SUB, and so forth. In this way, the computer can be trained to translate a program written with symbols instead of numbers into the computer's own machine language. Then we can write program for the computer using symbols instead of numbers, and have the computer do its own translating. This makes it easier for the programmer, because he can use letters, symbols, and mnemonics instead of numbers for writing his programs.



Example: The preceding program that was written in machine language for adding two numbers and printing out the result could be written in the following way:

```
CLA   A
ADD   B
STA   C
TYP   C
HLT
```

Which would mean "take A, add B, store the result in C, type C, and halt." The computer by means of a translating program, would translate each line of this program into the corresponding machine language program.

Advantages of Assembly Language

The main advantages of assembly language are:

1. Assembly language is easier to use than machine language.
2. An assembler is useful for detecting programming errors.
3. Programmers do not have to know the absolute addresses of data items.
4. Assembly languages encourage modular programming.

Disadvantages of Assembly Language

The main disadvantages of assembly language are:

1. Assembly language programs are not directly executable.
2. Assembly languages are machine dependent and, therefore, not portable from one machine to another.
3. Programming in assembly language requires a higher level of programming skill.

1.3 Assembly Program Execution

An assembly program is written according to a strict set of rules. An editor or word processor is used for keying an assembly program into the computer as a file, and then the assembler is used to translate the program into machine code.

Notes

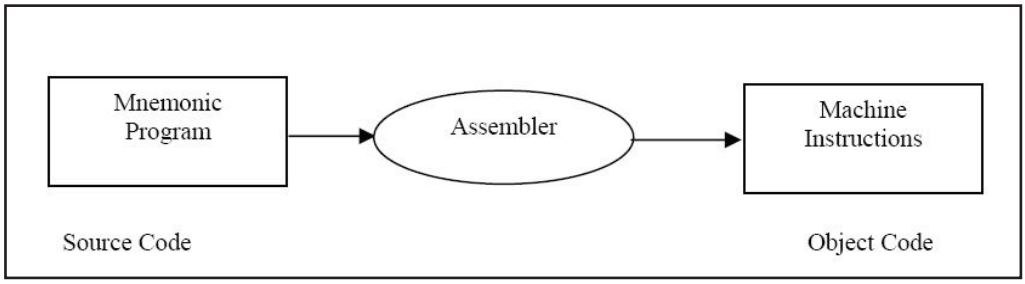
There are two ways of converting an assembly language program into machine language:

1. Manual assembly
2. By using an assembler.

Manual Assembly: It was an old method that required the programmer to translate each opcode into its numerical machine language representation by looking up a table of the microprocessor instructions set, which contains both assembly and machine language instructions. Manual assembly is acceptable for short programs but becomes very inconvenient for large programs. The Intel SDK-85 and most of the earlier university kits were programmed using manual assembly.

Using an Assembler: The symbolic instructions that you code in assembly language is known as - Source program.

An assembler program translates the source program into machine code, which is known as object program.



The steps required to assemble, link and execute a program are:

1. The assembly step involves translating the source code into object code and generating an intermediate .OBJ (object file) or module.

   The assembler also creates a header immediately in front of the generated .OBJ module; part of the header contains information about incomplete addresses. The .OBJ module is not quite in executable form.

2. The link step involves converting the .OBJ module to an .EXE machine code module. The linker's tasks include completing any address left open by the assembler and combining separately assembled programs into one executable module.

   The linker:

   (a) combines assembled module into one executable program

   (b) generates an .EXE module and initializes with special instructions to facilitate its subsequent loading for execution.

3. The last step is to load the program for execution. Because the loader knows where the program is going to load in memory, it is now able to resolve any remaining address still left incomplete in the header. The loader drops the header and creates a program segment prefix (PSP) immediately before the program is loaded in memory.

Tools Required for Assembly Language Programming

The tools of the assembly process described may vary in details.

Notes

Editor

The editor is a program that allows the user to enter, modify, and store a group of instructions or text under a file name. The editor programs can be classified in two groups.

1. Line editors
2. Full screen editors

Line editors, such as EDIT in MS DOS, work with the manage one line at a time. Full screen editors, such as Notepad, Wordpad etc. manage the full screen or a paragraph at a time. To write text, the user must call the editor under the control of the operating system. As soon as the editor program is transferred from the disk to the system memory, the program control is transferred from the operating system to the editor program. The editor has its own command and the user can enter and modify text by using those commands. Some editor programs such as WordPerfect are very easy to use. At the completion of writing a program, the exit command of the editor program will save the program on the disk under the file name and will transfer the control to the operating system. If the source file is intended to be a program in the 8086 assembly language the user should follow the syntax of the assembly language and the rules of the assembler.

Linker

For modularity of your programs, it is better to break your program into several sub routines. It is even better to put the common routine, like reading a hexadecimal number, writing hexadecimal number, etc., which could be used by a lot of your other programs into a separate file. These files are assembled separately. After each file has been successfully assembled, they can be linked together to form a large file, which constitutes your complete program. The file containing the common routines can be linked to your other program also. The program that links your program is called the linker.

The linker produces a link file, which contains the binary code for all compound modules. The linker also produces link maps, which contains the address information about the linked files. The linker however does not assign absolute addresses to your program. It only assigns continuous relative addresses to all the modules linked starting from the zero. This form a program is said to be relocate-able because it can be put anywhere in memory to be run.

Loader

Loader is a program which assigns absolute addresses to the program. These addresses are generated by adding the address from where the program is loaded into the memory to all the offsets. Loader comes into action when you want to execute your program. This program is brought from the secondary memory like disk. The file name extension for loading is .exe or .com, which after loading can be executed by the CPU.

Debugger

The debugger is a program that allows the user to test and debug the object file. The user can employ this program to perform the following functions.

1. Make changes in the object code.
2. Examine and modify the contents of memory.
3. Set breakpoints, execute a segment of the program and display register contents after the execution.

Notes

4. Trace the execution of the specified segment of the program and display the register and memory contents after the execution of each instruction.
5. Disassemble a section of the program, i.e., convert the object code into the source code or mnemonics.

In summary, to run an assembly program you may require your computer:

1. A word processor like notepad
2. MASM, TASM or Emulator
3. LINK.EXE, it may be included in the assembler
4. DEBUG.COM for debugging if the need so be.

Errors

Two possible kinds of errors can occur in assembly programs:

1. ***Programming errors:*** They are the familiar errors you can encounter in the course of executing a program written in any language.
2. ***System errors:*** These are unique to assembly language that permit low-level operations. A system error is one that corrupts or destroys the system under which the program is running - In assembly language there is no supervising interpreter or compiler to prevent a program from erasing itself or even from erasing the computer operating system.


Task Discuss the roles of assembler in computer programming.

1.4 Assembler

An assembly program is used to transfer assembly language mnemonics to the binary code for each instruction, after the complete program has been written, with the help of an editor it is then assembled with the help of an assembler.

An assembler works in two phases, i.e., it reads your source code two times. In the first pass the assembler collects all the symbols defined in the program, along with their offsets in symbol table. On the second pass through the source program, it produces binary code for each instruction of the program, and give all the symbols an offset with respect to the segment from the symbol table.

The assembler generates three files. The object file, the list file and cross reference file. The object file contains the binary code for each instruction in the program. It is created only when your program has been successfully assembled with no errors. The errors that are detected by the assembler are called the symbol errors.


Example: MOVE AX1, ZX1 ;

In the statement, it reads the word MOVE, it tries to match with the mnemonic sets, as there is no mnemonic with this spelling, it assumes it to be an identifier and looks for its entry in the symbol table. It does not even find it there therefore gives an error as undeclared identifier.

List file is optional and contains the source code, the binary equivalent of each instruction, and the offsets of the symbols in the program. This file is for purely documentation purposes. Some of the assemblers available on PC are MASM, TURBO etc.

1.5 Assembly Program and its Components

Notes

Sample Program:

In this program we just display:

Line Numbers	Offset	Source Code
0001		DATA SEGMENT
0002	0000	MESSAGE DB "HAVE A NICE DAY!$"
0003		DATA ENDS
0004		STACK SEGMENT
0005		STACK 0400H
0006		STACK ENDS
0007		CODE SEGMENT
0008		ASSUME CS: CODE, DS: DATA SS: STACK

0009	Offset	Machine	Code
0010	0000	B8XXXX	MOV AX, DATA
0011	0003	8ED8	MOV DS, AX
0012	0005	BAXXXX	MOV DX, OFFSET MESSAGE
0013	0008	B409	MOV AH, 09H
0014	000A	CD21	INT 21H
0015	000C	B8004C	MOV AX, 4C00H
0016	000F	CD21	INT 21H
0017			CODE ENDS
0018			END

The details of this program are:

The Program Annotation

The program annotation consists of three columns of data: line numbers, offset and machine code.

1. The assembler assigns line numbers to the statements in the source file sequentially. If the assembler issues an error message; the message will contain a reference to one of these line numbers.
2. The second column from the left contains offsets. Each offset indicates the address of an instruction or a datum as an offset from the base of its logical segment, e.g., the statement at line 0010 produces machine language at offset 0000H of the CODE SEGMENT and the statement at line number 0002 produces machine language at offset 0000H of the DATA SEGMENT.
3. The third column in the annotation displays the machine language produce by code instruction in the program.

Segment numbers: There is a good reason for not leaving the determination of segment numbers up to the assembler. It allows programs written in 8086 assembly language to be almost entirely

Notes

relocatable. They can be loaded practically anywhere in memory and run just as well. Program1 has to store the message "Have a nice day$" somewhere in memory. It is located in the DATA SEGMENT. Since the characters are stored in ASCII, therefore it will occupy 15 bytes (please note each blank is also a character) in the DATA SEGMENT.

Missing offset: The xxxx in the machine language for the instruction at line 0010 is there because the assembler does not know the DATA segment location that will be determined at loading time. The loader must supply that value.

Program Source Code: Each assembly language statement appears as:

{identifier} Keyword {{parameter},} {;comment}.

The element of a statement must appear in the appropriate order, but significance is attached to the column in which an element begins. Each statement must end with a carriage return, a line feed.

Keyword: A keyword is a statement that defines the nature of that statement. If the statement is a directive then the keyword will be the title of that directive; if the statement is a data-allocation statement the keyword will be a data definition type. Some examples of the keywords are: SEGMENT (directive), MOV (statement) etc.

Identif ers: An identifier is a name that you apply to an item in your program that you expect to reference. The two types of identifiers are name and label.

1. Name refers to the address of a data item such as counter, arr etc.
2. Label refers to the address of our instruction, process or segment.



Example: MAIN is the label for a process as:

```
MAIN PROC FAR

A20: BL,45 ; defines a label A20.
```

Identifier can use alphabet, digit or special character but it always starts with an alphabet.

Parameters: A parameter extends and refines the meaning that the assembler attributes to the keyword in a statement. The number of parameters is dependent on the Statement.

Comments: A comment is a string of a text that serves only as internal document action for a program. A semicolon identifies all subsequent text in a statement as a comment.

Directives

Assembly languages support a number of statements. This enables you to control the way in which a source program assembles and list. These statements, called directives, act only when the assembly is in progress and generate no machine-executable code. Let us discuss some common directives.

1. ***List:*** A list directive causes the assembler to produce an annotated listing on the printer, the video screen, a disk drive or some combination of the three. An annotated listing shows the text of the assembly language programs, numbers of each statement in the program and the offset associated with each instruction and each datum. The advantage of list directive is that it produces much more informative output.
2. ***HEX:*** The HEX directive facilitates the coding of hexadecimal values in the body of the program. That statement directs the assembler to treat tokens in the source file that begins with a dollar sign as numeric constants in hexadecimal notation.

Notes

3. ***PROC Directive:*** The code segment contains the executable code for a program, which consists of one or more procedures defined initially with the PROC directive and ended with the ENDP directive.

   Procedure-name PROC FAR ; Beginning of Procedure

   Procedure-name ENDP FAR ; End Procedure

4. ***END DIRECTIVE:*** ENDS directive ends a segment, ENDP directive ends a procedure and END directive ends the entire program that appears as the last statement.

5. ***ASSUME Directive:*** An .EXE program uses the SS register to address the base of stack, DS to address the base of data segment, CS to address base of the code segment and ES register to address the base of Extra segment. This directive tells the assembler to correlate segment register with a segment name.

   Example:

   ASSUME SS: stack_seg_name, DS: data_seg_name, CS: code_seg_name.

6. ***SEGMENT Directive:*** The segment directive defines the logical segment to which subsequent instructions or data allocations statement belong. It also gives a segment name to the base of that segment.

The address of every element in a 8086 assembly program must be represented in segment - relative format. That means that every address must be expressed in terms of a segment register and an offset from the base of the segmented addressed by that register. By defining the base of a logical segment, a segment directive makes it possible to set a segment register to address that base and also makes it possible to calculate the offset of each element in that segment from a common base.

An 8086 assembly language program consists of logical segments that can be a code segment, a stack segment, a data segment, and an extra segment.

A segment directive indicates to assemble all statements following it in a single source file until an ENDS directive.

Code Segment

The logical program segment is named code segment. When the linker links a program it makes a note in the header section of the program's executable file describing the location of the code segment when the DOS invokes the loader to load an executable file into memory, the loader reads that note. As it loads the program into memory, the loader also makes notes to itself of exactly where in memory it actually places each of the program's other logical segments. As the loader hands execution over to the program it has just loaded, it sets the CS register to address the base of the segment identified by the linker as the code segment. This renders every instruction in the code segment addressable in segment relative terms in the form CS: xxxx.

The linker also assumes by default that the first instruction in the code segment is intended to be the first instruction to be executed. That instruction will appear in memory at an offset of 0000H from the base of the code segment, so the linker passes that value on to the loader by leaving an another note in the header of the program's executable file.

The loader sets the IP (Instruction Pointer) register to that value. This sets CS:IP to the segment relative address of the first instruction in the program.

Notes

Stack Segment

8086 Microprocessor supports the Word stack. The stack segment parameters tell the assembler to alert the linker that this segment statement defines the program stack area.

A program must have a stack area in that the computer is continuously carrying on several background operations that are completely transparent, even to an assembly language programmer, for example, a real time clock. Every 55 milliseconds the real time clock interrupts. Every 55 ms the CPU is interrupted. The CPU records the state of its registers and then goes about updating the system clock. When it finishes servicing the system clock, it has to restore the registers and go back to doing whatever it was doing when the interruption occurred. All such information gets recorded in the stack. If your program has no stack and if the real time clock were to pulse while the CPU is running your program, there would be no way for the CPU to find the way back to your program when it was through updating the clock. 0400H byte is the default size of allocation of stack. Please note if you have not specified the stack segment it is automatically created.

Data Segment

It contains the data allocation statements for a program. This segment is very useful as it shows the data organization.

Def ning Types of Data

The following format is used for defining data definition:

Format for data definition:

{Name} <Directive> <expression>

Name: A program references the data item through the name although it is optional.

Directive: Specifying the data type of assembly.

Expression: Represent a value or evaluated to value.

The list of directives are given below:

Directive	Description	Number of Bytes
DB	Define byte	1
DW	Define word	2
DD	Define double word	4
DQ	Define Quad word	8
DT	Define 10 bytes	10

DUP Directive is used to duplicate the basic data definition to 'n' number of times

ARRAY DB 10 DUP (0)

In the above statement ARRAY is the name of the data item, which is of byte type (DB). This array contains 10 duplicate zero values; that is 10 zero values.

EQU directive is used to define a name to a constant

CONST EQU 20

Type of number used in data statements can be octal, binary, haxadecimal, decimal and ASCII. The above statement defines a name CONST to a value 20.

Some other examples of using these directives are:

Notes

```
TEMP      DB    0111001B      ; Binary value in byte operand
                              ; named temp
VALI      DW    7341Q         ; Octal value assigned to word
                              ; variable
Decimal   DB    49            ; Decimal value 49 contained in
                              ; byte variable
HEX       DW    03B2AH        ; Hex decimal value in word
                              ; operand
ASCII     DB    'EXAMPLE'     ; ASCII array of values.
```

1.6 Machine Level Language

Although computers can be programmed to understand many different computer languages, there is only one language understood by the computer without using a translation program, this language is called the machine language or the machine code of the computer. Machine code is the fundamental language of a computer and is normally written as strings of binary 1s and 0s. the circuitry of a computer is wired in such a way that it immediately recognizes the machine language and converts it into the electrical signals needed to run the computer.

An instruction prepared in any language has a two part format, as shown in Figure 1.1. The first part is command or operation, and it tells the computer what function to perform. Every computer has an operation code or opcode for each of its functions. The second part of the instruction is the operand, and it tells the computer where to find or store the data or other instructions that are to be maintained. Thus, each instruction tells the control unit of the CPU what to do and the length and location of the data field are involved in the operation. Typical operations involve reading, adding, subtracting, writing and so on.

Figure 1.1: Instruction Format



We already know that all commuters use binary digits (0s and 1s) for performing operations. Hence, most computers machine language consists of strings of binary numbers and is the only one the CPU directly understands. When stored inside the computer, the symbols which make up the machine language program are made up of 1s and 0s.

Example: A typical program instruction to print out a number on the printer might be.

10110011111101001110110011000011100 1

The program to add two numbers in memory and print the result look something like the following:

00100000000000011001110 01

00111100000011111100011 1

Notes

100111100011101100110101

101100010101010101110000

000000000000000000000000

This is obviously not a very easy language to learn, partly because it is difficult to read and understand and partly because it is written in a number system with which we are not familiar. But it will be surprising to note that some of the first programmers, who worked with the first few computers, actually wrote their programs in binary form as above.

Since human programmers are more familiar with the decimal number system, most of them preferred to write the computer instructions in decimal, and leave the input device to convert these to binary. In fact, without too much effort, a computer can be wired so that instead o fusing long numbers. With this change, the preceding program appears as follows:

10001471

14002041

30003456

50773456

00000000

The set of instruction codes, whether in binary or decimal, which can be directly understood by the CPU of a computer without the help of a translating program, is called a machine code or machine language. Thus, a machine language program need not necessarily be coded as strings of binary digits (1s and 0s). it can also be written using decimal digits if the circuitry of the computer being used permits this.

Advantages and Limitations of Machine Language

Programs written in machine language can be executed very fast by the computer. This is mainly because machine instructions are directly understood by the CPU writing a program in machine language has several disadvantages which are discussed below.

1. ***Machine dependent:*** Because the internal design of every type of commuter is different from every other type of computer and needs different electrical signals to operate, the machine language also is different from computer to computer. It is determined by the actual design or construction of the LU, the control unit, and the size as well as the word length of the memory unit. Hence, suppose after becoming proficient in the machine code of a particular computer, a company decides to change to another computer, the programmer may be required to learn a new machine language and would have to rewrite all the existing programs.
2. ***Diff cult to program:*** Although easily used by the computer, machine language is difficult to program, it is necessary for the programmer either to memorize the dozens of code numbers for the commands in the machine's instruction set or to constantly refer to keep track of the storage location of data and instructions. Moreover, a machine language programmer must be an expert who knows about the hardware structure of the computer.
3. ***Error code:*** For writing programs in machine language, since a programmer has to remember the opcodes and he must also keep track of the storage location of data and instructions, it becomes very difficult fro him to concentrate fully on the logic of the problem. This frequently results in program errors. Hence, it is easy to make errors while using machine code.

Notes

4. ***Diff cult to modify:*** It is difficult to correct or modify machine language programs. Checking machine instructions to locate errors is about as tedious as writing them initially. Similarly, modifying a machine language program at a later date is so difficult that many programmers would prefer to code the new logic afresh instead of incorporating the necessary modifications in the old program.

In short, writing a program in machine language is so difficult and time consuming that it is rarely used today.



Task Computer only understand 0 and 1 but we enter all values in the form of alphabets how computer accept of understand these values.

1.7 Higher Level Languages

We have talked about programming languages as COBOL, FORTRAN and BASIC. They are called high level programming languages. The program shown below is written in BASIC to obtain the sum of two numbers.

```
LET     X       =       7
LET     Y       =       10
LET     sum     =       X+Y
PRINT   SUM
END
```

The time and cost of creating machine and assembly languages was quite high. And this was the prime motivation for the development of high level languages.

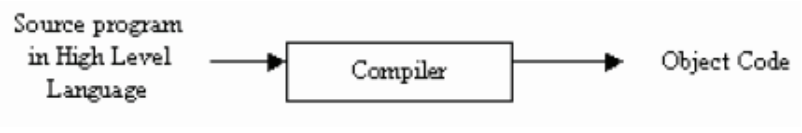
Because of the difficulty of working with low-level languages, high-level languages were developed to make it easier to write computer programs. High level programming languages create computer programs using instructions that are much easier to understand than machine or assembly language code because you can use words that more clearly describe the task being performed.



Example: High-level languages include FORTRAN, COBOL, BASIC, PASCAL, C, C++ and JAVA.

1.8 Compiling High Level Language

Since a high level source program must be translated first into the form the machine can understand, this is done by a software called Compiler which takes the source code as input and produces as output the machine language code of the machine on which it is to be executed. This is illustrated in figure.



Notes

During the process of translation, the Compiler reads the source programs statement-wise and checks the syntax (grammatical) errors. If there is any error, the computer generates a printout of the errors it has detected. This action is known as diagnostics.

A compiler is a computer program (or set of programs) that transforms source code written in a computer language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). A program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a language translator, source to source translator, or language converter. A language rewriter is usually a program that translates the form of expressions without a change of language.

There is another type of software, which also does the translation. This is called an Interpreter. The Compiler and Interpreter have different approaches to translation. The following table lists the differences between a Compiler and an Interpreter.

Compiler	Interpreter
Scans the entire program first and then translates it into machine code	Translates the program line by line.
Converts the entire program to machine code; when all the syntax errors are removed execution takes place.	Each time the program is executed, every line is checked for syntax error and then converted to equivalent machine code.
Slow for debugging (removal of mistakes from a program)	Good for fast debugging
Execution time, is less	Execution time is more.

Advantages of High-level Programming Language

The various advantages of high-level programming language are:

1. ***Readability:*** Programs written in these languages are more readable than assembly and machine language.
2. ***Portability:*** Programs could be run on different machines with little or no change. We can, therefore, exchange software leading to creation of program libraries.
3. ***Easy debugging:*** Errors could easily be removed (debugged).
4. ***Easy Software development:*** Software could easily be developed. Commands of programming language are similar to natural languages (ENGLISH).

1.9 Some High Level Languages

Some popular high-level languages are:

1. ADA
2. APL
3. BASIC
4. Pascal
5. FORTRAN
6. COBOL

Notes

7. C
8. LISP
9. RPG

ADA

ADA was named after lady Augusta Ada Byron (the first computer programmer). It was designed by the US Defence Department for its real time applications. It is suitable for parallel processing.

APL

Developed by Dr Kenneth Aversion at IBM, APL is a convenient, interactive programming language suitable for expressing complex mathematical expressions in compact formats. It requires special terminals for use.

BASIC (Beginners All-purpose Symbolic Instruction Code)

BASIC was developed by John Kemeny & Thomas Karthy at Dartmouth College. It is a widely known and accepted programming language. It is easy to use and is almost coded in real-time conversational mode. This language provides good error diagnostics but has no self-structuring or self-documentation.

Pascal

Developed in 1968, Pascal was named after a French inventor Blaise Pascal and was developed by a Swiss programmer NiKolus Wirth. Pascal was the first structured programming language and it is used for both scientific and business data processing applications.

FORTRAN (FORmula TRANslation)

Developed by IBM in 1957, it is one of the oldest and most widely used high level languages. It is widely used by scientists and engineers as this language has huge libraries of engineering and scientific programs. This language is suitable for expressing formulae, writing equations, and performing iterative calculations.

Various versions of FORTRAN are:

1. FORTRAN I (1957)
2. FORTRAN II (1958)
3. FORTRAN IV (1962)
4. FORTRAN 77 (1978)

COBOL (COmmon Business Oriented Language)

Cobol is a structured and self documented language. It was developed by a committee of business, industry, government, and academic representatives called codasyl (conference on data SYstem Languages) commissioned by US government in 1959. Statements of Cobol language resemble English language expressions and it makes them easy to understand and use.

Notes

C

Developed by Denis Ricthie at the Bell Laboratories in 1970, it is a general purpose programming language, which features economy of expression, modern control flow and data structures, and a rich set of operators. Its programs are highly portable (machine independent). C language supports modular programming through the use of functions, subroutines, and macros.

LISP (List Processor)

Developed in 1960 by Prof. John McGrthy, Lisp and Prolog (programming logic) are the primary languages used in artificial intelligence research and applications.

RPG (Report Program Generator)

RPG is an important business oriented programming language developed by IBM in late 1960s. It is primarily used for preparing written reports.

Advantages of RPG

The various advantages of RPG are:

1. RPG is problem oriented.
2. It is easy to learn and use.
3. Limited programming skills are required.



Task Specify the statement "A compiler is a computer program that transforms source code written in a computer language into another computer language."

1.10 Summary

- In this unit, we discussed about various categories of programming languages starting from machine language to fourth generation language.
- The concepts of compilers and interpreters have also been introduced along with this discussion. Finally we have discussed about basic elements of any programming language.

1.11 Keywords

Compiler: A compiler is a computer program that transforms source code written in a computer language into another computer language.

Computer Programming: Computer programming is a field that has to do with the analytical creation of source code that can be used to configure computer systems.

Debugger: The debugger is a program that allows the user to test and debug the object file.

Loader: Loader is a program which assigns absolute addresses to the program.

1.12 Self Assessment

Notes

Choose the appropriate answers:

1. Which one is not the basic instruction of a programming language?
   (a) Input
   (b) Math
   (c) Match
   (d) Repetition
2. For stop the program what used in machine language
   (a) STA
   (b) TYP
   (c) CLA
   (d) HLT
3. Translates the source program into machine code known as
   (a) Translator
   (b) Assembler
   (c) Coder
   (d) Transfer
4. PSP stands for
   (a) Popular Stack Point
   (b) Program Segment Prefix
   (c) Program Stack Prefix
   (d) None of the above
5. An assembler works in
   (a) 4 phases
   (b) 1 phase
   (c) 2 phases
   (d) 3 phases

Fill in the blanks:

6. A ..... is a statement that defines the nature of that statement.
7. A ..... is a computer program that transforms source code written in a computer language into another computer language.
8. ..... is a structured and self documented language.
9. ..... is a program which assigns absolute addresses to the program.
10. Programs could be run on different machines with little or no change known as .....

State whether the following statements are true or false:

11. Assembly languages are also known as second generation languages.
12. Assembly program is not written according to a strict set of rules.

Notes

13. Linker produces a link file, which contains the binary code for all compound modules.
14. 8086 Microprocessor supports the Word stack.

1.13 Review Questions

1. Describe assembly language in detail. Also explain the advantages of the same.
2. Describe various tools required for assembly language programming.
3. What do you mean by stack segment?
4. Explain various advantages and limitations of machine language.
5. Write short notes on the following:
   (a) PASCAL
   (b) COBOL
   (c) BASIC
   (d) C
6. Distinguish between OPCODE and OPERAND.
7. List various version of FORTRAN language.

Answers: Self Assessment

1.	(c)	2.	(d)	3.	(b)	4.	(b)
5.	(c)	6.	keyword	7.	compiler	8.	COBOL
9.	Loader	10.	portability	11.	True	12.	False
13.	False	14.	True				

1.14 Further Readings



Books

Brian Kerrighen and Dennis Ritchie, *The C Programming language*

D. Bharioke, *Fundamentals of IT,* Excel Books

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer,* Prentice-Hall International, 1982.

R.G.Dromey, *How to solve it by Computer,* 2007, Pearson Education, India

Seymour Lipschutz, Essentials *Computer Mathematics,* Schaums' Outlines Series, 2004, Tata McGraw Hill.

Turban Rainer Cotter, *Introduction to IT,* John Wiley & Sons.

V. Rajaraman, *Fundamentals of Computer,* Prentice Hall of India.

Yashvant Kanetkar, Let us C

Unit 2: Introduction to C Language

Notes

CONTENTS

Objectives

Introduction

2.1 Origin and Development of C Language

2.2 About C

2.3 Evolution of C

2.4 Compilers and Interpreters

2.5 Structure of a C Program

2.6 Functions

2.7 Compiling a C Program

2.8 Programming Rules and Execution

2.9 Summary

2.10 Keywords

2.11 Self Assessment

2.12 Review Questions

2.13 Further Readings

Objectives

After studying this unit, you will be able to:

- Know how C language evolved
- State the concepts of compilers and interpreters
- Identify structure of a C program
- Discuss how C program is compiled

Introduction

The programming language C was originally developed by Dennis Ritchie of Bell Laboratories and was designed to run on a PDP-11 with a UNIX operating system. Although it was originally intended to run under UNIX, there has been a great interest in running it under the MS-DOS operating system on the IBM PC and compatibles. It is an excellent language for this environment because of the simplicity of expression, the compactness of the code, and the wide range of applicability. Also, due to the simplicity and ease of writing a C compiler, it is usually the first high level language available on any new computer, including microcomputers, minicomputers,

Notes

and mainframes. It allows the programmer a wide range of operations from high level down to a very low level, approaching the level of assembly language. There seems to be no limit to the flexibility available.

2.1 Origin and Development of C Language

C is a general-purpose, structured programming language. Structured Languages have a characteristic programme structure and associated set of static scope rules.

C was originated in Bell Telephone Laboratories presently known as AT & T Bell Laboratories by Dennis Ritchie in 1970. The Kernighan and Ritchie description is commonly referred to as "K&R C". Following the publication of the K & R description, computer professionals, impressed with C's many desirable features, began to promote the use of the language. Since 1980's, the popularity of C has become widespread. The American National Standards Institute (ANSI) proposed a standardised definition of the C language (ANSI committee X3J11). Most commercial C compilers and interpreters are expected to adopt the ANSI standard.

C has the feature of high level programming language as well as the low-level programming. It works as a bridging gap between machine language and the more conventional high-level languages. This feature of C Language made it most popular for system programming as well as application programming.

2.2 About C

C is often termed as a middle level programming language because it combines the power of a high level language with the flexibility of a low level language. High-level languages have lot of built-in features and facilities, which result in high programming efficiency and productivity. Low-level languages, on the other hand, are designed to give more efficient programs and better machine efficiency.

C is designed to have a good balance between both extremes. Programs written in C give relatively high machine efficiency as compared to the high level languages (though not as good as low level languages). Similarly, C language programs provide relatively high programming efficiency as compared to the low level languages (though not as high as those provided by high level languages). Thus, C can be used for a whole range of applications with equal ease and efficiency.

There are several features which make C a very suitable language for writing system programs. These are as follows:

1. C is a machine independent and highly portable language.
2. It is easy to learn as it has only as few as 32 keywords.
3. It has a comprehensive set of operators to tackle business as well as scientific applications with ease.
4. Users can create their own functions and add them to the C library to perform a variety of tasks.
5. C language allows manipulation of BITS, BYTES, and ADDRESSES at hardware level.
6. It has a large library of functions.
7. C operates on the same data types as the computer, so the codes need very little data conversion, if at all. Therefore, codes generated are fast and efficient.

Notes



Did u know? C is general programming language C is a link between low level to high level language.

2.3 Evolution of C

By the late fifties, there were many computer languages into existence. However, none of them were general purpose. They served better in a particular type of programming application more than others. Thus, while FORTRAN was more suited for engineering programming, COBOL was better for business programming. At this stage people started thinking that instead of learning so many languages for different programming purposes, why not have a single computer language that can be used for programming any type of application.

In 1960, to this end, an international committee was constituted which came out with a language named ALGOL-60. This language could not become popular because it was too general and highly abstract.

In 1963, a modified ALGOL-60 by reducing its generality and abstractness, a new language, CPL(Combined Programming Language) was developed at Cambridge University. CPL, too turned out to be very big and difficult to learn.

In 1967, Martin Richards, at Cambridge University, stripped down some of the complexities from CPL retaining useful features and created BCPL (Basic CPL). Very soon it was realized that BCPL was too specific and much too less powerful.

In 1970, Ken Thompson, at AT&T labs., developed a language known by the name B as another simplification to CPL. B, too, like its predecessors, turned out to be very specific and limited in application.

In 1972, Ritchie, at AT&T, took the best of the two BCPL and B, and developed the language C. C was truly a general purpose language, easy to learn and very powerful.

In 1980, Bjarne Stroustrup, at Bell labs., took C to its next phase of evolution, by incorporating features of Object Oriented programming, reincarnating C into its new avatar C++. By and large, C remains the mother language for programming even today.



Task Give two examples of low level language.

2.4 Compilers and Interpreters

Note that the only language a digital computer understands is binary coded instructions. Even the above implementation will not execute on a computer without further translation into binary (machine) code. This translation is not done manually, however. There are programs available to do this job. These translation programs are called compilers and interpreters.

Compilers and interpreters are programs that take a program written in a language as input and translate it into machine language. Thus a program that translates a C program into machine code is called C compiler; BASIC program into machine code is called a BASIC compiler and so on.

Therefore, to implement an algorithm on a computer you need to have a compiler for the language you have chosen for writing the program for the algorithm.

Notes

A number of different compilers are available these days for C language. ANSI, Borland C, Turbo C, etc. are only few of the popular C compilers. As a matter of fact, these software tools are little more than just compiler. They provide a complete environment for C program development. They include, among others, an editor to allow Program writing, a Compiler for compilation of the same, a debugger for debugging/testing the program , and so forth. Such tools are referred to as IDE (Integrated Development Environment) or SDK (Software Development Kit).



Did u know? You know the full form of COBOL

COBOL: Common Business Oriented Language

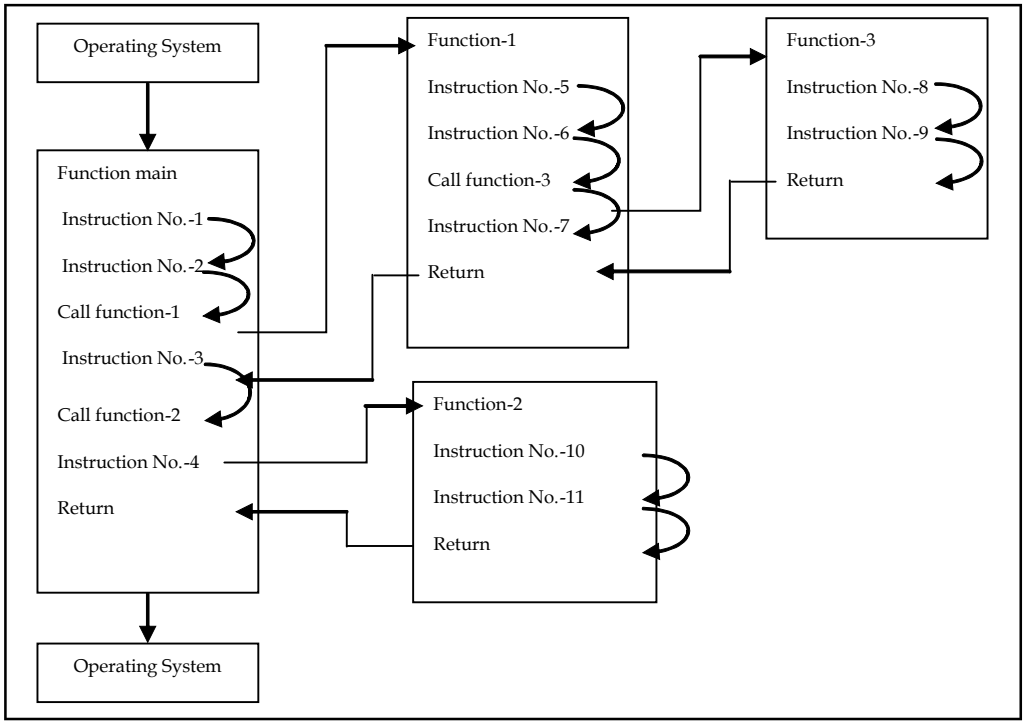
2.5 Structure of a C Program

Every C program consists of one or more distinct units called functions. Each function has unique name. One and only one of the constituent functions of a C-program must be named main(). It is the main function, which is executed when the program is run. A function may call another function, which executes and return computed value to the calling function as depicted in the figure.

Each function has a name, an optional list of input parameters (also called arguments) with their individual data-type, and a return data-type T.

A compound statement is a group of zero or more statements enclosed within curly braces {}. Compound statements may be nested i.e., one compound statement may exist inside another one.

A C-statement is a valid C-expression delimited by semi-colon.



Notes

The following rules are applicable to all C-statements:

1. Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank space is allowed within a word.
2. Most of the C-compilers are case-sensitive, and hence statements are entered in small case letters.
3. C has no specific rules about the position at which different parts of a statements be written. Not only can a C statement be written anywhere in a line, it can also be split over multiple lines. That is why it is called free-format language.
4. A C-statement ends with a semi-colon(;).

Task What will be the value of d if d is a float after the operation d = 2 / 7.0?

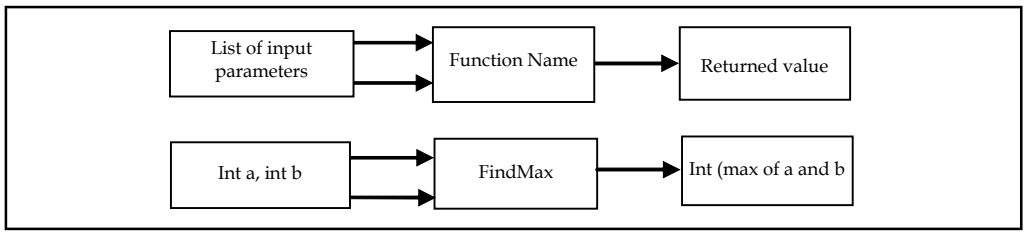
2.6 Functions

Every C program is structured as an assembly of one or more distinct units called functions. Each function comprises of a set of valid C statements and is designed to perform a specific task. Each function is given a unique name for reference purposes and is treated as a single unit by the C-compiler. A function name is always followed by a pair of parenthesis, i.e., ( ).

The statements within a function are always enclosed within a pair of braces { }. Every C program must necessarily contain a special function named main( ). The program execution always starts with this function. The main function is normally, but not necessarily, located at the beginning of the program. The group of statements within main( ) are executed sequentially. When the closing brace of the main function is reached, program execution stops, and the control is handed back to the operating system.

Whenever a function is called, it returns to the caller the value it is supposed to return.

Schematically, a C-function may be depicted as:



The C-program code for this function looks like:

```
Int  FindMax(int  a,  int  b)
          {
          statement1;
          statement2;
          return  (p);  //p  is  the  integer  value  returned  by
          //this  function
}
```

Notes

In plain English it reads: function, whose name if FindMax, takes two integer type arguments, process it some way (i.e. executes the statements contained within) and returns an integer value to the caller, which can be used by other statements of the program.

Statements

Single C language instruction delimited by a semi-colon is called a statement. Statements are written in accordance with the grammar of C language. Blank space is inserted between words to improve the readability. However, no blank spaces are allowed within a word. Usually all C statements are entered in lower case letters. C is a free form language. There are no restrictions regarding the format of writing C language statements. The statements can start and terminate anywhere on the line. They can also be split over multiple lines. Every C language statement always ends with a semicolon.

2.7 Compiling a C Program

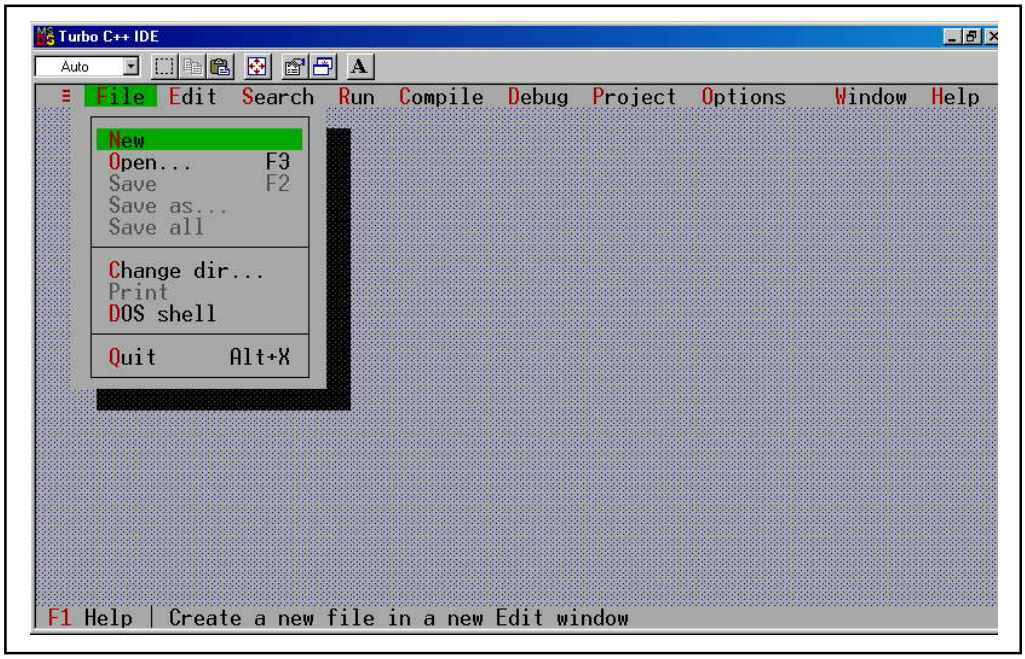
Compiling a C program depends on the compiler as well as the operating system under use. Supposing that you are using Turbo C compiler on DOS platform, the compilation of a C program stored in a file called TEST.C, would look like as shown below:

```
C:\>tc  test.c
```

Here, TC is the name of Turbo C compiler. The compilation would produce an object file called TEST.OBJ. Object files, thus produced, contain the machine language translation of the C program. However, this file is yet not executable. To make it an executable program you need to LINK the object files with library files, when TEST.EXE is produced.

Most often you shall be using an IDE for program development and therefore, you do not need to compile and link the programs explicitly as shown above. Every IDE provides a user friendly set of commands to carry out the compilation and linking automatically.

We will consider how Turbo C IDE may be used to develop and compile a C program. When you start the Turbo C IDE, the IDE provides all the necessary commands that let you write, compile and link your programs. A typical Turbo C IDE is displayed below.



Notes

Clearly, the IDE is menu driven. All the commands are grouped into menus such as; File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. You may take your time to navigate through the menus and see what command they offer to you.

To just give you a feel, we will interact with the IDE in form of a session. Let us create a program file called TEST.C containing the following program:

```
#include <stdio.h>

main ()

{

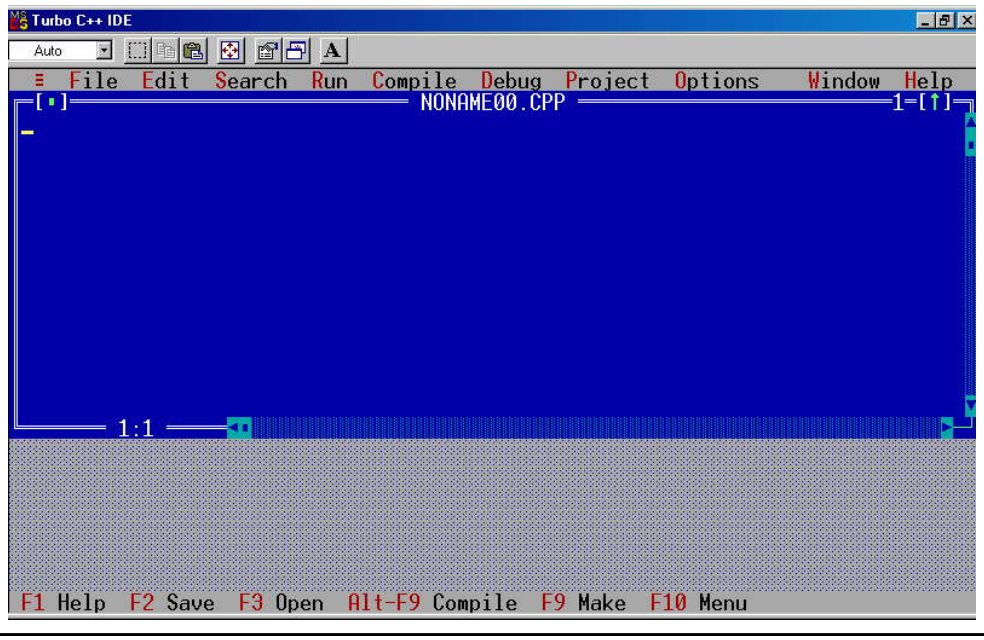
printf("this is the first C program");

}
```

Never mind if you do not understand what this program means. It is just to demonstrate how you would write your programs and compile them.

To write the program into Test.C file:

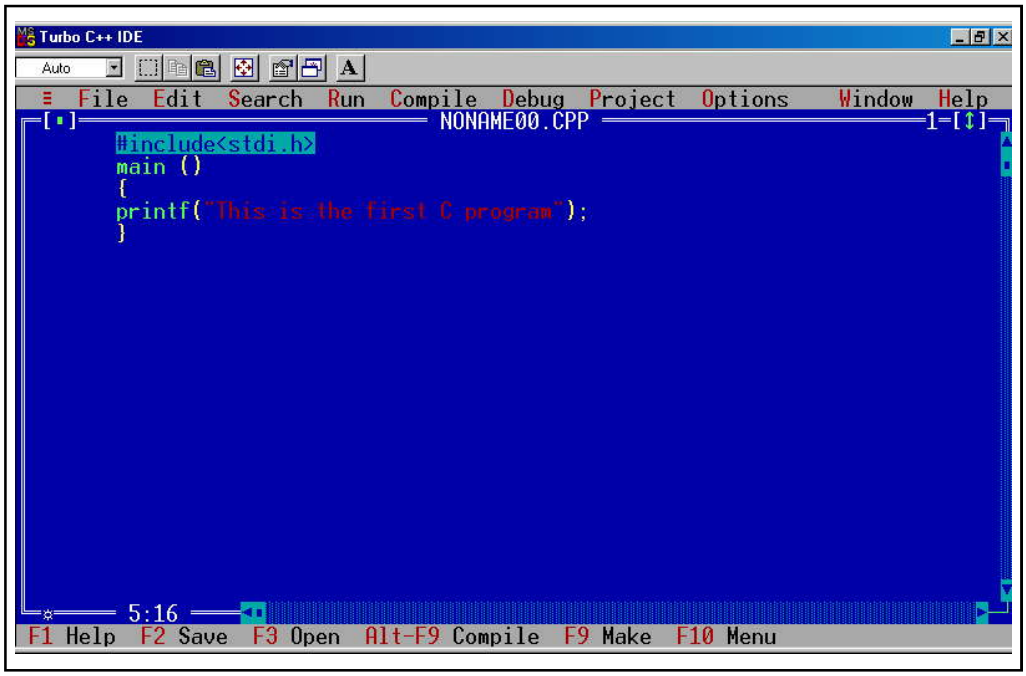
1. Go to file menu.
2. Click at New command. Turbo C IDE opens a blank file for you, as shown below:



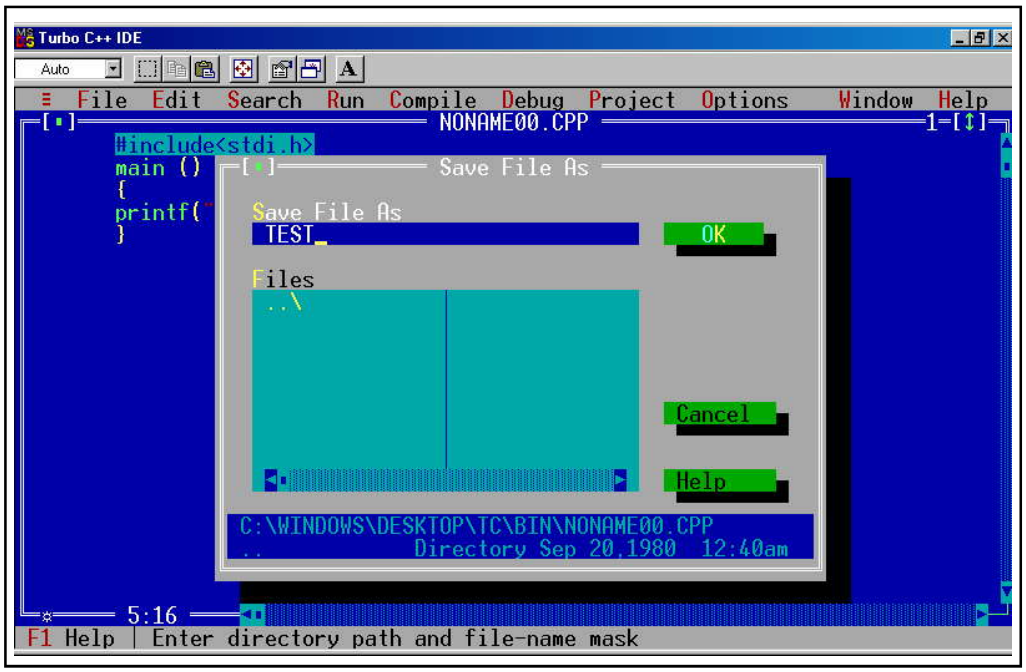
Notes Turbo C names the file as NONAME00.C. You can now enter your program in this file and save it with whatever name you wish to assign to it.

Notes

3. Type the program as shown below:



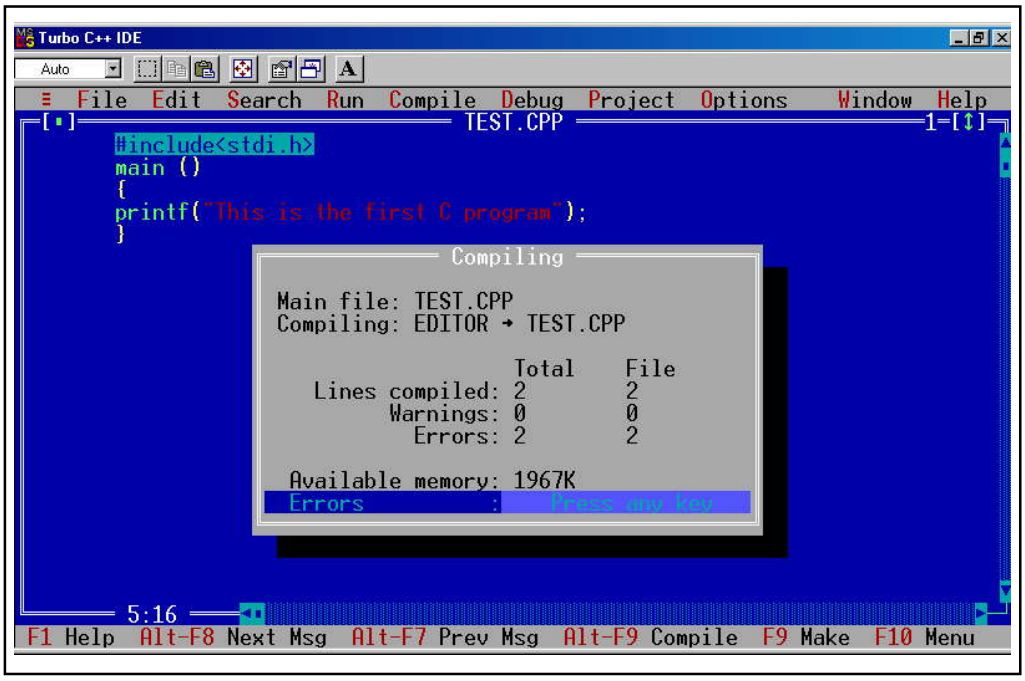
Now that you have entered the program into the opened file, save it as TEST.C or whatever name that suits you. Remember, the IDE automatically saves the file with .C extension. All the C programs must be saved with .C extension. To save this file, go to File menu once again. Click at Save command. A dialog window appears as shown below:



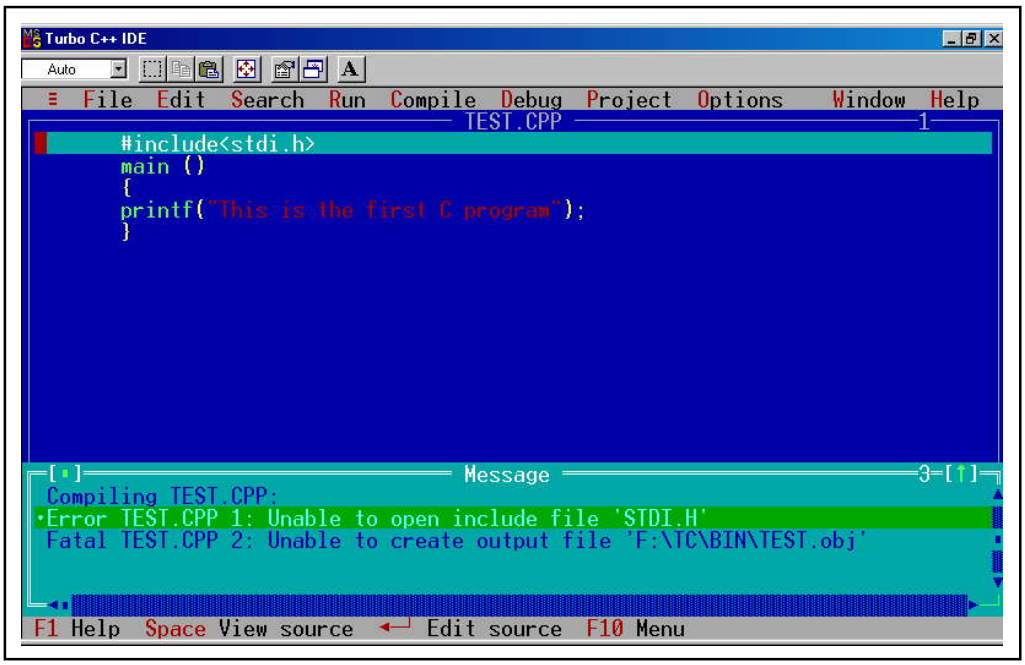
Type in the name – TEST and press OK. The file will be saved on the disk.

Notes

You can now compile the program by clicking at compile command available in Compile menu. Alternatively we can press Alt+F9. The compilation result is displayed in compile dialogue window as shown below.



In this case, the compiler reports that there is an error in the program and that the compilation could not proceed. In order to see the errors just press any key when the following window appears.



Notes


Note

There is an error at line no.1 - "Unable to open include file 'STDI.H'". Obviously the include file name should have been STDIO.H instead of STDI.H. Correct this and recompile. This time the program would compile successfully.

1. Now that you have object program ready, you can now link it by clicking at link command in compile menu. This produces TEST.EXE.
2. In order to run the program, go to Run menu and click at Run command. You will see the output of the program as shown below.

Once you are through with the programming session, you can quit from the IDE by clicking at Exit command available in File menu.

2.8 Programming Rules and Execution

To begin with, let us write a simple C language program. Enter the following program in a Source file, Compile it, link it and run it to see the output.



Lab Exercise

Program:

```
/* Program to print Hello C on the screen */
#include <stdio.h>
main ( )
{
printf ("Hello C");
}
```

Although, it is extremely primitive program that simply prints - Hello C - on the screen, nonetheless it includes a number of useful features of a C program. A little explanation of the components of this program is presented below.

The first line is a comment that explains the purpose of the program.

#include or any statement beginning with # character, is known as compiler directive. A compiler directive is a command to compiler to translate the program in a certain way. These statements are not converted into machine language but only perform some other task.

A function (as also a variable) must be defined or declared before it is used. This program uses two functions – main() and printf(). printf() function prints whatever is handed over to it as input argument. More on printf() later. While main() is a being defined here, printf() is not. Actually, printf() function is defined in a library file - stdio.h.

#include compiler directive commands the compiler to copy the contents of the specified file at this line. Thus, #include<stdio.h> will copy the contents of the file stdio.h in its place. As said earlier this file contains definition of printf() function, its definition (as also any other definition that file may have) is copied here. Now on, you can use printf() function in the rest of the program.

Notes

The third line is a heading for the function main( ). It takes no arguments.

```
{begins the body of the function main().
Printf("Hello C"); statement prints "Hello C' on the screen.
} terminates the function main().
```



Lab Exercise

Program:

```
# include <stdio.h>
   main( )
   {
        char c1, c2;
        int i1, i2;
        c1 = 'a';
        c2 = 'b';
        i1 = 65;
        i2 = 66;
        printf ("c1 and c2 as character values are: %c,
%c \n", c1, c2);
printf ("c1 and c2 as integer values are: %d, %d
\n", c1, c2);
        printf ("i1 and i2 as character values are: %c,
%c \n", i1, i2);
printf ("i1 and i2 as character values are: %d,
%d \n", i1, i2);
        }
output: c1 and c2 as character values are: a, b
        c1 and c2 as integer values are: 97, 98
        i1 and i2 as character values are: A, B
        i1 and i2 as integer values are: 65, 66.
```

Explanation

Numeric data are stored in the memory in their binary form while the character data has to be codified as a unique integer and that code number is stored in the internal storage. The integer equivalents of alphabets are:

Lower case : a – z ≡ 97 – 122

Upper case : A – Z ≡ 65 – 90

Notes

In the above program, when characters are displayed in the integer format, the corresponding ASCII codes are displayed. Similarly, when integers are displayed in the character format, their equivalent character is displayed.



Task

Which of the following is allowed in a C Arithmetic instruction?

1. [ ]
2. { }
3. ( )
4. None of the above



Case Study

Write down our first C program. It would simply calculate simple interest for a set of values representing principle, number of years and rate of interest.

```
/* Calculation of simple interest */
/* Author LPU Date: 12/02/2011 */
main( )
{
int p, n;
float r, si;
p = 1000;
n = 3;
r = 8.5;
/* formula for simple interest */
si = p * n * r / 100;
printf ( "%f" , si );
}
```

Questions

1. Write a program in C addition of two numbers.
2. Write a simple program in C to show "I love my India" on screen.

Notes

2.9 Summary

- C is a programming language developed at AT & T's Bell Laboratories of USA in 1972.
- It was designed and written by a man named Dennis Ritchie.
- In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc. No one pushed C.
- It wasn't made the 'official' Bell Labs language. Thus, without any advertisement C's reputation spread and its pool of users grew.
- Ritchie seems to have been rather surprised that so many programmers preferred C to older languages like FORTRAN or PL/I, or the newer ones like Pascal and APL. But, that's what happened.

2.10 Keywords

Circular Linked List: A linear linked list in which the last element points to the first element, thus, forming a circle.

Doubly Linked List: A linear linked list in which each element is connected to the two nearest.

Linear List: A one-dimensional list of items.

Linked List: A dynamic list in which the elements are connected by a pointer to another element.

NULL: A constant value that indicates the end of a list.

2.11 Self Assessment

Choose the appropriate answers:

1. C language has been developed by
   (a) Ken Thompson
   (b) Dennis Ritchie
   (c) Peter Norton
   (d) Martin Richards
2. C programs are converted into machine language with the help of
   (a) An Editor
   (b) A compiler
   (c) An operating system
   (d) None of the above
3. From the given program

```
void main()
{
    int a=10,b=20;
    char x=1,y=0;
    if(a,b,x,y)
   {
```

Notes

```
            printf("EXAM");
        }
    }
```

What is the output?

(a) XAM is printed

(b) exam is printed

(c) Compiler Error

(d) Nothing is printed

Fill in the blanks:

4. ..... may be inserted between two words to improve the readability of the statement.

5. A ..... is an entity whose value can change during program execution.

6. C program consists of one or more distinct units called .....

7. Constants are the ..... that remain unchanged during the execution of a program and are used in assignment statements.

2.12 Review Questions

1. What will be the output of:

(a)
```
main( )
{
  int i = - 3;
  i = - i - i + i;
  printf ("%d", i);
}
```

(b)
```
main( )
{
  int i, j, k;
  i = 10;
  j = 5;
  k = i > J;
  printf ("%d", k);
}
```

(c)
```
main( )
{
  int p, q, r;
  p = 7, q = 3;
  r = (p > q) && (q < p);
```

Notes

```
    printf ("%d", r);
}
(d) main( )
{
    unsigned char a;
    a = OxFF + 1;
    printf ("%d", a);
}
(e) main( )
{
    int a, b, c, d, e;
    a = 5;
    b = 6;
    c = 12;
    d = 11;
    e = (a ! = b) ? (e < = (! d) ? a: b): c;
    printf ("e = % d", e);
}
(f) main( )
{
    float a = 1; b;
    int m = 3, n = 5;
    b = (a ? m: n) / 2.0;
    printf ("%f", b );
}
(g) main( )
{
    int a, b, c, d;
    a = 4;
    b = + + a;
    c = A - -;
    d = - - A;
    a + = 1 - 2 * 2;
    printf ("a = %d, b = %d, c = %d, d = %d", a, b, c, d);
}
```

Notes

(h)
```
main( )
{
  int x, y, z;
  x = (y = 5, z = y+2, y + z);
  x = y = 5; z = x = 9;
  printf ("x = %d, y = %d, z = %d", x, y, z);
}
```

(i)
```
main( )
{
  int i = -3, j = 2, k = 0, m;
  m = ++ i && ++k && ++k;
  printf ("%d, %d, %d", i, j, k);
```

(j)
```
main( )
{
  int i = -3, j = 2, k = 0 m;
  m = ++j && ++i | | ++ k;
  printf ("\n %d, %d, %d", i, j, k, m);
}
```

2. When was 'C' developed?
3. Who is the author of 'C' language?
4. What is the role of compiler in C programming?
5. "A function name is always followed by a pair of parenthesis, i.e., ( )." Explain
6. Write a sample program "My Best Friend" and explain step by step its execution process.
7. Describe C program structure in detail.
8. Write a program in C "I am a student of /!!!!!!!!!!!!!!!/ BCA".
9. Distinguish between low level language and high level language.
10. Write a program to display any matter in two lines.

Answers: Self Assessment

1.	(b)	2.	(b)	3.	(d)	4.	Blank spaces
5.	Variable	6.	Functions	7.	Fixed values		

2.13 Further Readings

Notes



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication, 2008.

B.W. Kernighan and D.M. Ritchie, "T*he Programming Language*", Prentice Hall of India, New Delhi.

Byron Gottfried, "*Programming with C*", Tata McGraw Hill Publishing Company Limited, New Delhi.

Greg W Scragg, Genesco Suny, *Problem Solving with Computers*, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer*, Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 3: Basics – The C Declaration

CONTENTS

Objectives

Introduction

3.1 C Character Set

3.2 Keywords or Reserved Words

3.3 Identifiers

3.4 Constants in C

3.5 Data Types

3.6 Additional Data Types

3.7 Variables

3.8 Declaration of Variables

3.9 Summary

3.10 Keywords

3.11 Self Assessment

3.12 Review Questions

3.13 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe C program structure
- Explain identifiers and constants in C
- Explain data types in C
- Describe how to declare variables in C

Introduction

In a C program, data items (variables and constants) can be arithmetically manipulated using operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation on data stored in variables. The variables that are operated are termed as operands.

3.1 C Character Set

Character set of a language is set of all the symbols used to write a program in that language. They have been taken from English language. The characters in C are grouped into four categories:

1. Letters : A - Z or a – z
2. Digits : 0 - 9

Notes

3. Special Symbols : ~ . ‘ . ! @ # % ^ & * ( ) _ - + = | \ { } [ ]: ; “ ‘ < >, . ? /

4. White spaces : blank space, horizontal tab, carriage return, new-line, form-feed

The entire C program should be written using these characters alone. Inclusion of any other character would produce error in the program.

3.2 Keywords or Reserved Words

Every language contains certain words that have specific predefined meaning associated with them. Such words are known as Keywords. For instance, eat, sleep, book, have specific meaning in English language. Other words, which are not keywords, represent an object such as India, Rajan, etc., are referred to as literals.

C language also has keywords and literals (also called identifiers). In order to avoid problems, keywords should not be used as variable names (or identifiers). A list of keywords employed in C language is presented below.

auto	double	int	strcut
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

It is interesting and as a matter of fact quite relieving that C language (the language you are about to learn) has very limited vocabulary as compared to the English language!



Did u know? C is case sensitive language.

3.3 Identifiers

From characters (letters) of a language one can form words. Words are just a collection of characters of that language.



Example: Mango, dog, cat, master, man etc. are valid word in English language. Some words may have a pre-defined meaning, as the word – cat – in English; while some may not have such a fixed meaning and may be taken as a word representing a noun or an object.

Similarly, words can be formed from the characters in C language. Words formed in language C are termed as identifiers. C identifiers can be of two types:

1. Keywords or reserved words, and

2. User-defined identifiers



Task Long is reserved keyword in C. What about Long (Small).

Notes

3.4 Constants in C

A constant is a token with fixed value that does not change. It can be stored at a location in the memory of the computer and can be referenced through that memory address. There are four basic types of constants in C, viz. integer constants, floating-point constants, character constants and string constants. Composite types may also have constants.

Integer and floating-point constants represent numbers. They are often referred to collectively as numeric-type constants.

C imposes the following rules while creating a numeric constant type data item:

1. Commas and blank spaces cannot be included within the constants.
2. The constant can be preceded by a minus (-) sign if desired.
3. Value of a constant cannot exceed specified maximum and minimum bounds. For each type of constant, these bounds will vary from one C-compiler to another.

Constants are the fixed values that remain unchanged during the execution of a program and are used in assignment statements. Constants can also be stored in variables.

The declaration for a constant in C language takes the following form:

```
const <datatype> <var_name> = <value>;

Ex.: const float pi = 22/7;
```

This declaration defines a constant named pi whose value remains 22/7 throughout the program in which it is defined.

C language facilitates five different types of constants.

1. Character
2. Integer
3. Real
4. String
5. Logical

Character Constants

A character constant consists of a single character, single digit, or a single special symbol enclosed within a pair of single inverted commas. The maximum length of a character constant is one character.

Ex. : 'a' is a character constant

Ex. : 'd' is a character constant

Ex. : 'P' is a character constant

Ex. : '7' is a character constant

Ex. : '*' is a character constant

Integer Constants

An integer constant refers to a sequence of digits and has a numeric value. There are three types of integers in C language: decimal, octal and hexadecimal.

Notes

Decimal integers	1, 56, 7657, -34 etc.
Octal integers	076, -076, 05 etc. (preceded by zero, 0)
Hexadecimal integers	0x56, -0x5D etc. (preceded by zero, 0x)

No commas or blanks are allowed in integer constants.

Real or Floating Point Constants

A number with a decimal point and an optional preceding sign represents a real constant.



Example: 34.8, –655.33, .2, –.56, 7.



Notes Note that 7 is an integer while 7. or 7.0 is real.

Another notation (called scientific notation) for real constants consists of three parts:

1. A sign (+ or 0) preceding the number portion (optional).
2. A number portion.
3. An exponent portion following the number portion (optional). It starts with E or e followed by an integer. This may or may not be preceded by a sign.



Example:

Valid representations	Invalid Representations
+ . 72	12 (no decimal)
+ 72.	7.6 E + 2.2 (non integer exponent)
+ 7.6 E + 2	1.2 E9229892 (very large exponent)
24.4 e - 5	

String Constants

A string constant is a sequence of one or more characters enclosed within a pair of double quotes (" "). If a single character is enclosed within a pair of double quotes, it will also be interpreted as a string constant and not a character constant.



Example:

1. `"Welcome To C Programming \ n"`
2. `"a"`

Actually, a string is an array of characters terminated by a NULL character. Thus, "a" is a string consisting of two characters, viz. 'a' and NULL('\0').

Logical Constants

A logical constant can have either true value or false value. In C, a non-zero value is treated as true while 0 is treated as false.

Notes

Task Point out the error, if any, in this C statement:

int = 314.562 * 150 ;

3.5 Data Types

The range of different types of data that a programming language can handle is one of the factors that determine the power of the programming language. C language is very powerful in this sense. Almost all types of data can be represented and manipulated in C program.

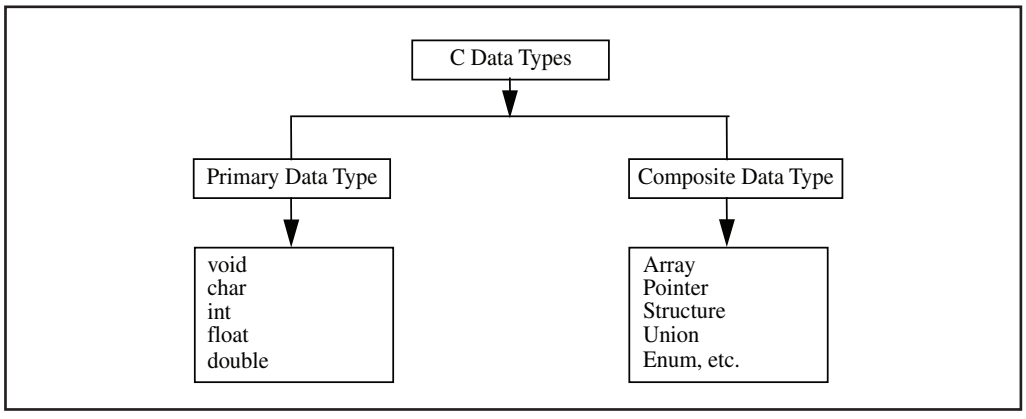
Inside a digital computer, at the lowest level, all data and instructions are stored using only binary digits (0 and 1). Thus, decimal number 65 is stored as its binary equivalent: 0100 0001. Also the character "A" is stored, as binary equivalent of 65(A's ASCII): 0100 0001. Both the stored values are same but represent different type of values. How's that?

Actually, the interpretation of a stored value depends on the type of the variable in which the value is stored even if it is just 0100 0001 as long as it is stored on the secondary storage device. Thus, if 0100 0001 is stored in an integer type variable, it will be interpreted to have integer value 65, whereas, if it is stored in character type of variable, it will represent "A".

Therefore, the way a value stored in a variable is interpreted is known as its data type. In other words, data type of a variable is the type of data it can store.

Every computer language has its own set of data types it supports. Also, the size of the data types (number of bytes necessary to store the value) varies from language to language. Besides, it is also hardware platform dependent.

C has a rich set of data types that is capable of catering to all the programming requirements of an application. The C-data types may be classified into two categories: Primary and Composite data types as shown in figure.



C has two distinct categories of data types – primary, and composite. Primary data types are the ones that are in-built with C language while composite data types allow the programmers to create their own data types.

There are five primary data types in C language.

1. ***char:*** stores a single character belonging to the defined character set of C language.
2. ***int:*** stores signed integers. e.g., positive or negative integers.
3. ***f oat:*** stores real numbers with single precision (precision of six digits after decimal points).

Notes

4. ***double:*** stores real numbers with double precision, i.e., twice the storage space required by float.

5. ***void:*** specify no values.

The following table shows the meaning and storage spaces required by various primary data types.

Data Type	Meaning	Storage Space	Format	Range of Values
char	A character	1 byte	%c	ASCII character set
int	An integer	2 bytes	%d	– 32768 to +32767
float	A single precision floating point number	4 bytes	%f	$-3.4*10^{38}$ to $+3.4*10^{38}$
double	A double precision floating point number	8 bytes	%lf	-1.7×10^{308} to $+1.7*10^{308}$
void	valueless or empty	0 byte	–	–

3.6 Additional Data Types

Primary C data types may have different sizes that enable them to store large range of values. This is indicated in a program by appending a keyword before the data type – called data type qualifier.

For instance, short, long, signed and unsigned are data type qualifiers for int basic type. Thus an integer type data may be defined in C as short int, int, unsigned int, long int. The range of values and size of these qualified data-types is implementation dependent. However, short is smaller than or equal int, which in turn, is smaller than long. Unsigned int contains larger range since it does not store negative integers.

Also known as derived data types, composite data types are derived from the basic data types. They are five in number.

1. ***Array:*** Sequence of objects, all of which are of same types and have same name.

   Example: int num [5];

   Reserves a sequence of five locations of 2 bytes, each, for storing integers num[0], num[1], num[2], num[3] and num[4].

2. ***Pointer:*** Used to store the address of any memory location.

3. ***Structure:*** Collection of variables of different types.

   Example: A structure of employee's data, i.e., name, age and salary.

4. ***Union:*** Collection of variables of different types sharing common memory space.

5. ***Enumerated:*** Its members are the constants that are written as identifiers though data type they have signed integer values. These constants represent values that can be assigned to corresponding enumeration variables.

Enumeration may be defined as:

```
enum tag { member1, member2 …. member n};

E.g.: enum colors {red, green, blue, cyan};

colors foreground, background;
```

Notes

In the first line, an enumeration named "colors" which may have any one of the four colors defined in the curly braces. In the second line, variables of the enumerated data type "colors" are declared.



Task Data types always used on the basis of their length value suggest the range of float data type.

3.7 Variables

A variable is an entity whose value can change during program execution. A variable can be thought of as a symbolic representation of address of the memory space where values can be stored, accessed and changed. A specific location or address in the memory is allocated for each variable and the value of that variable is stored in that location.

Each variable has a name, data-type, size, and the value it stores. All the variables must have their type indicated so that the compiler can record all the necessary information about them; generate the appropriate code during translation and allocating required space in memory.

Every programming language has its own set of rules that must be observed while writing the names of variables. If the rules are not followed, the compiler reports compilation error. Rules for Constructing Variable Name in C language are listed below:

1. Variable name may be a combination of alphabets, digits or underscores. Sometimes, an additional constraint on the number of characters in the name is imposed by compilers in which case its length should not exceed 8 characters.
2. First character must be an alphabet or an underscore (_).
3. No commas or blank spaces are allowed in a variable name.
4. Among the special symbols, only underscore can be used in a variable name.

   E.g.: emp_age, item_4, etc.
5. No word, having a reserved meaning in C can be used for variable name.



Note C language is a case-sensitive language which differentiates between lower case and upper case. Thus, CAT is different from Cat, cAT, CaT. Although any word complying with the rules cited above can be used as the variable name, it is advised that you create variable names that have some meaning. Thus, you may chose sum as the variable name for storing sum of numbers rather than choosing X.

3.8 Declaration of Variables

C language is strongly typed language, meaning that all the variables must be declared before their use. Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

Notes

In C language, a variable declaration has the form:

```
<Type-specifier>    <comma-separated-list-of-variables>;
```

Here <type-specifier> is one of the valid data types (e.g. int, float, char, etc.). List-of-variables is a comma-separated list of identifiers representing the program variables.



Example:

```
a.    int i, j, k; //creates integer variables i,j and K
b.    char ch; //creates a character type variable ch
```

Once variable has been declared in the above manner, the compiler creates a space in the memory and attaches the given name to this space. This variable can now be used in the program.

A value is stored in a variable using assignment operation. Assignment is of the form:

```
<Variable-name> = <value>;
```

Obviously, before assignment, the variable must be declared.



Example:

```
int i, j;
j = 5;
i = 0;
```

C also allows assignment of a value to a variable at the time of declaration. It takes the following form:

```
<Type-specifier>        <variable_name> = <value>;
e.g.    :               int I = 5;
```

This declaration not only creates an interior type variable I, but also stores a value 5 at the same time.

The process of assigning initial values to the variable is known as initialization. More than one variable can be initialized in one statement using multiple assignment operators.



Example:

```
i.      int i = 5, j=3;
ii.     int j, m;
        j = m = 2;
```

However, there is an exception worth noting. Consider the following example:

```
int i, j = 2, k;
```

The assignments will be made as follows:

```
i = 0
j = 2
k = 1063 (a garbage value, unintialized)
```

Let us consider some of programming examples to illustrate the matter further.



Example:

```
/* Example of assignments */
        /* declaration */
                int a1, b1;
```

Notes

```
/* declaration & assignment */
                int var = 5;
                int a, b = 6, c;
/* declaration & multiple assignment */
                int p, q, r, s;
                p = q = r = s = 5;
}
values stored in various variables are:
                var = 5
                a = 0, b = 6
c = garbage value
                p = 5, q = 5, r = 5, s = 5
```



Task Write a program in C to show the variable declaration.



Case Study

Data types are provided to store various types of data that is processed in real life. A student's record might contain the following data types: name, roll number, and grade percentage. For example, a student named Anil might be assigned roll number 5 and have a grade percentage of 78.67. The roll number is an integer without a decimal point, the name consists of all alpha characters, and the grade percentage is numerical with a decimal point. C supports representation of this data and gives instructions or statements for processing such data. In general, data is stored in the program in variables, and the kind of data the variable can have is specified by the data type. Using this example, grade percentage has a float data type, and roll number has an integer data type. The data type is attached to the variable at the time of declaration, and it remains attached to the variable for the lifetime of the program. Data type indicates what information is stored in the variable, the amount of memory that can be allocated for storing the data in the variable, and the available operations that can be performed on the variable. For example, the operation S1 * S2, where S1 and S2 are character strings, is not valid for character strings because character strings cannot be multiplied.

Program

```
// the program gives maximum and minimum values of data type
#include <stdio.h>
main()
{
int i,j ;// A
```

Contd...

Notes

```
i = 1;
while (i > 0)
{
j = i;
i++;
}
printf ("the maximum value of integer is %d\n",j);
printf ("the value of integer after overflow is %d\n",i);
}
```

Explanation

1. In this program there are two variables, i and j, of the type integer, which is declared in statement A.
2. The variables should be declared in the declaration section at the beginning of the block.
3. If you use variables without declaring them, the compiler returns an error.

3.9 Summary

- C is a programming language. It has letters, words, sentences and a well-defined grammar. Character set of a language is set of all the symbols used to write a program in that language.
- Words formed in language C are termed as identifiers. C identifiers can be of two types – keywords or reserved words, and user-defined identifiers. Every language contains certain words that have specific predefined meaning associated with them. Such words are known as Keywords.
- A token is a group of characters separated from other group of characters by one or more white space. A constant is a token with fixed value that does not change.
- A variable is an entity whose value can change during program execution. A variable can be thought of as a symbolic representation of address of the memory space where values can be stored, accessed and changed.
- C language is a case-sensitive language which differentiates between lower case and upper case. C language is strongly typed language, meaning that all the variables must be declared before their use. C has two distinct categories of data types – primary, and composite. Primary data types are the ones that are in-built with C language while composite data types allow the programmers to create their own data types.

3.10 Keywords

Character set: Character set is a set of characters which are submitted to the compiler and interpreted in various contents as characters, names/identifiers, constants and statements.

Constant: A named data item whose value does not change throughout the execution of the program

Notes

Identif er: A string of characters representing the name to identify a variable, function etc.

Keywords: Sentence can contain certain words which are used for specific purposes. These words are called keywords. Example to show a message the keyword is 'Display'.

Variable: A named location in the memory that can store a value of specified type.

White space: The characters that separate one identifier from the other like space, tab, carriage return, new line etc.

3.11 Self Assessment

Choose the appropriate answers:

1. 'd' is a
   (a) Character constant
   (b) Integer constant
   (c) Logical constant
   (d) None
2. A character variable can at a time store
   (a) 1 character
   (b) 8 characters
   (c) 254 characters
   (d) None of the above
3. The maximum value that an integer constant can have is
   (a) -32767
   (b) 32767
   (c) 1.7014e+38
   (d) –1.7014e+38

Fill in the blanks:

4. Words formed in language C are termed as .....
5. ..... and floating-point constants represent numbers.
6. The maximum length of a character constant is ..... character.
7. A string constant is a sequence of one or more characters enclosed within a pair of .....
8. The C-data types may be classified into two categories: Primary and ..... data types
9. ..... are the fixed values that remain unchanged during the execution of a program and are used in assignment statements.
10. An integer constant refers to a sequence of digits and has a ..... value.

Notes

3.12 Review Questions

1. What is the process of assigning values to the variable?
2. Write in detail about the various data type available in C.
3. What is a variable? How to declare and initialize variables in C?
4. Give a brief description of the various derived data types available in C.
5. What is a string constant? List down some rules for writing strings in C.
6. What will be the value of the variable t at the end of execution of each of the following set of codes?

   (a) int t = -3;

   t = t – t + t;

   (b) int a = 8, b = 3; float t;

   t = a/b;

   (c) int a = 8, b = 3; float t;

   t = (float) a / b;

   (d) int a = 8, b = 3; float t;

   t = (float) a / (float) b;

Answers: Self Assessment

1.	(a)	2.	(a)	3.	(b)	4.	identifiers
5.	Integer	6.	one	7.	double quotes (" ")	8.	Composite
9.	Constants	10.	numeric				

3.13 Further Readings



Books

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, *Problem Solving with Computers*, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer*, Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 4: Operators

CONTENTS

Objectives

Introduction

4.1 Operators

4.1.1 Arithmetic Operators

4.1.2 Relational Operators

4.1.3 Logical Operators

4.1.4 Assignment Operators

4.1.5 Increment and Decrement Operators

4.1.6 Conditional Operators

4.1.7 Bitwise Operators

4.1.8 Special Operators

4.2 Arithmetic Expression

4.3 Valuation of Expression

4.4 Precedence of Arithmetic Operator

4.5 Type Conversions in Expression

4.5.1 Automatic Type Conversion

4.5.2 Casting a Value

4.6 Operator Precedence and Associativity

4.7 Summary

4.8 Keywords

4.9 Self Assessment

4.10 Review Questions

4.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain arithmetic operators
- Describe conditional operators
- Describe arithmetic expression
- Explain type conversion in expression

Notes

Introduction

A combination of constants, variables and operators that confirm to the grammatical rules of the language C and evaluate to some valid value is called an expression. The effect that operators bring about on their operands is called operation.

An expression that confirms to the rules of grammar of C language is referred to as valid or well-formed expression. A valid or well expression always evaluates to a single value of a valid C data type. Accordingly, C expression can be of the following types:

1. ***Numerical expressions*** always evaluates to a numeric value on which arithmetic operations can be performed. They can be further divided into the following two categories:

   (a) *Integer expression:* those evaluating to integer value

   (b) *Real expression:* those evaluating to a real (floating point) value

   Thus, 3 + 5 is an integral expression and 3.8 – 6.97 is a real expression.

2. ***Logical or conditional expressions*** always result into either of the two values – true or false.

   Thus 3 > 5 and x <= 7 are conditional expressions.

4.1 Operators

C operators can be classified into a number of categories. They include:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

4.1.1 Arithmetic Operators

Arithmetic operators work on numeric type of operands. C provides all the basic arithmetic operators. There are five arithmetic operators in C.

Operator	Purpose
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

The division operator (/) requires the second operand to be non-zero, though the operands need not be integers. When an integer is divided by another integer, the result is also an integer. In such instances the division is termed as integer division. Consider the following:

```
int x;

x = 10;
```

Notes

What do you expect the value of x/4 to be? If you guessed 2.5, you are wrong. The result is of course 2.5 however, since it is integer division (division operation in which both the operands are integers), the result 2.5 will be truncated to 2 to make the result an integer. In case you wish to get the correct value you make this division a float type as x/4.0.

The % operator is known as modulus operator. It produces the remainder after the division of two operands. The second operand must be non-zero.

Rest all the other operators work in their normal arithmetic way. Normal BODMAS rules are also applicable to these arithmetic operators.

4.1.2 Relational Operators

Relational operator is used to compare two operands to see whether they are equal to each other, unequal, or one is greater or lesser than the other.

While the operands can be variables, constants or expressions, the result is always a numeric value equivalent to either true or false. As mentioned earlier a non-zero result indicates true and zero indicates false. C language provides six relational operators.

= =	equal to
! =	not equal to
<	less than
< =	less than or equal to
>	greater than
> =	greater than or equal to

A simple relation contains only one relational expression and takes the following form:

<ae-1> <relational operator> <ae-2>

<ae-1> and <ae-2> are arithmetic expressions, which may be constants, variables or combination of these. The value of the relational operator is either 1 or 0. If the relation is true, result is 1 otherwise it is 0.



Example:

Expressions	Result
6.3 < = 15	True
2.5 < -2	False
-10 > = 0	False
10 < 8+3	True

4.1.3 Logical Operators

More than one relational expression can be combined to form a single compound relational expression using logical operators. Logical operators are used to combine two or more relational expressions. C provides three logical operators. A compound relation behaves identically producing either true or false value.

Operator	Meaning	Result
&&	Logical AND	True if both the operands are true
¦¦	Logical or	True if both the operands are true
!	Logical not	True if the operand is false and vice versa

Notes

Example:

1. (age > 50 && weight < 80) will evaluate to true if age is more than 50 and also weight is less than 80. Otherwise it will evaluate to false.
2. (a < 0 ¦ ch = = 'a') will evaluate to true if a is less than 0 while ch is equal to 'a', false otherwise.
3. (!(a < 0)) will evaluate to true if a is greater than or equal to 0, false otherwise.

4.1.4 Assignment Operators

Assignment operators are used to store the result of an expression to a variable. The most commonly used assignment operator is (=). Be careful not to mistake assignment operator (=) for mathematical equality operator which is indicated by the same symbol.

An expression with assignment operator is of the following form:

```
<identifier> = <expression>;
```

When this statement is executed, <expression> is evaluated and the value is stored in the <identifier>.

Note Data type of both the sides should be either the same or compatible to each other.

Let us consider the following usage of assignment operator in C language.

```
int i;

i = 5;

i = i + 10;
```

The value now stored in the variable "i" will be 15. In this program, the current value stored in variable i is 5. Thus, while executing i = i+10, the right hand side will be evaluated to give a value 15. This value will then be assigned to the left hand side. As a result, the current value of I after execution of this statement will become 15.

C language provides a short cut to write arithmetic assignment expressions which takes the following form:

```
<Variable> op = <expression>;
```

This statement is identical to:

```
<Variable> = <Variable> op <expression>;
```

Thus,	i=i+3	can be written as	i+=3
	Sum=sum-2	can be written as	sum-=2
	i=i*5	can be written as	i*=5

The advantages of using this form of assignment operators are:

1. The statement is more efficient and easier to read.
2. What appears on the L.H.S need not to be repeated and therefore it becomes easier to write for long variable names. Consider the following C code that illustrates this point.

Notes

```
int averylongvariablename;
averylongvariablename = 2;
while (averylongvariablename < 20)
{
averylongvariablename*= averylongvariablename;
}
```

4.1.5 Increment and Decrement Operators

C has two very useful operators ++ and -- called increment and decrement operators respectively. These are generally not found in other languages. These operators are unary operators as they require only one operand. The operands must be a variable name and not a constant.

The increment operator (++) adds one to the current value of the operand and stores the result back into the operand, while the decrement operator (--) subtracts one from the operand and stores the decremented value back into the operand.

There are two different forms of increment and decrement operators. When they are used before the operand, it is termed as prefix, while when used after the operand, they are termed as postfix operators.



Example:

```
int i = 5;
i++;
++i;
--i;
i--;
```

When used in an isolated C statement, both prefix and postfix operators have the same effect, but when they are used in expressions, each of them has a different effect.

In expressions, postfix operator uses the current value and then increments/decrements while in the prefix form the value is incremented/decremented first and then used in the expression. Consider the following examples:

```
E.g.:  b = a ++;
       this is postfix increment expression. This statement is equivalent to:
{b = a;
a = a+1;}
E.g.   b = - - a;
       this is prefix decrement expression. This statement is equivalent to:
       { a= a-1;
       b = a; }
```

Consider the following C code that illustrates the usage of postfix and prefix increment operators.

```
int a = 10; b = 0;          //a = 10 and b = 0
a++;                        //a = 11 and b = 0
b = ++a;                    //a = 12 and b = 12
b = a++;                    //a = 13 and b = 12
```

Notes

4.1.6 Conditional Operators

C provides a ternary operator called the conditional operator which is represented by :?. The syntax of this operator is given below.

```
A?B:C
```

Where "A" is a conditional expression resulting in either of the two values – true or false. The value generated by this operator in the expression depends on the value of the conditional expression "A". If the value of "A" is true then the expression evaluates to "B" otherwise it results in "C".

4.1.7 Bitwise Operators

You know that a numeric value is stored in a variable in binary form. Bitwise operators are used for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators are applicable to integer data types only. A list of different bit wise operators available in C language and their corresponding meaning is presented below:

Table 4.1: Some Bitwise Operators

Operator	Meaning
&	Bitwise Logical AND
¦	Bitwise Logical OR
^	Bitwise Logical XOR
<<	Left shift
>>	Right shift
~	One's complement

| ***(Bit-wise OR):*** Binary operator takes two operands of int type and performs bit-wise OR operation. With assumption that int size is 8-bits:

```
int a = 5;                    [binary : 0000 0101]

int b = 9;                    [binary : 0000 1001]

a | b yields                  [binary : 0000 1101]
```

& (Bit-wise AND): Binary operator takes two operands of int type and performs bit-wise AND operation. With same assumption on int size as above:

```
int a = 5;                    [binary : 0000 0101]

int b = 9;                    [binary : 0000 1001]

a & b yields                  [binary : 0000 0001]
```

^ ***(Bit-wise Logical XOR):*** XOR gives 1 if only one of the operand is 1 else 0. With same assumption on int size as above:

```
int a = 5;                    [binary : 0000 0101]

int b = 9;                    [binary : 0000 1001]

a ^ b yields                  [binary : 0000 1100]
```

<< ***(Shift left):*** This operator shifts the bits towards left padding the space with 0 by given integer times.

```
int a = 5;                    [binary : 0000 0101]

a << 3 yields                 [binary : 0010 1000]
```

Notes

>> ***(Shift right):*** This operator shifts the bits towards right padding the space with 0.

```
int a = 5;                    [binary : 0000 0101]

a >> 3 yields                 [binary : 0000 0000]
```

~ ***(one's complement operator):*** It is a unary operator that causes the bits of its operand to be inverted so that 1 becomes 0 and vice-versa. The operator must always precede the operand and must be integer type of all sizes. Assuming that int type is of 1 byte size:

```
inr a = 5;                    [binary: 0000 0101]

~a;                           [binary: 1111 1010]
```

4.1.8 Special Operators

C language also provides number of special operators which have no counter parts in other languages. These operators include comma operator, sizeof operator, pointer Operators (& and *), and member selection operators (. and ->). Pointer operators will be discussed while introducing pointers and member selection operators will be discussed with structures and union.

We will discuss comma operator and sizeof operator in this section.

Comma Operator

This operator is used to link the related expressions together.



Example:

```
int x, y, z;

z = (x = 10, y=20, x+y);
```

Here, the first statement will create three integer type variables – x, y and z. In the second statement, right-hand side will be evaluated first. Consequently, 10 will be stored in variable x, then 20 will be stored in variable y, and then values in x and y will be multiplied result of which will be stored in variable z. Thus, the value stored in the variable z will be 200 at the end of execution.

Sizeof Operator

The sizeof operator works on variables, constants and even on data types. It returns the number of bytes the operand occupies in the memory.

Consider the following C code for illustration.

```
sizeof(int);          //Gives number of bytes occupied by an

                      //integer type variable

sizeof(float);        //Gives number of bytes occupied by a

                      //float type variable
```

The output of this code will be 2, 4. Don't get disheartened if you get different result. This is only because the machine on which this program was run allotted 2 bytes for int type and 4 bytes for float type. The result that you get depends on the number of bytes allocated to these types on your machine. Nevertheless in all cases sizeof operator returns the number of bytes occupied by its operand on that particular machine.



Task If a four-digit number is input through the keyboard, write a program to obtain the sum of the first and last digit of this number.

Notes

4.2 Arithmetic Expression

An expression is a combination of variables, constants and operators arranged according to syntax of the language. Some examples of expressions are:

```
c = (m + n) * (a - b);
```

```
temp = (a + b + c) / (d - c);
```

Expression is evaluated by using assignment statement.

Such a statement is of the form

```
variable = Expression;
```

The expression on the L.H.S is evaluated first, then the value is assigned to the variable. But all the relevant variables must be assigned the values before the evaluation of the expression.



Did u know? Avoid your variable name same as the keyword reserved in C language.

4.3 Valuation of Expression

By using assignment statement we can evaluate an expression.

```
variable=expression;
```

The expression is evaluated first and then a value is assigned to the variable on the left hand side.



Example: `temp = ((f*cos(x)/sin(y))+(g*sin(x)/cos(y)))`

All relevant variables must be assigned values before the evaluation of the expression.

4.4 Precedence of Arithmetic Operator

The precedence of arithmetic operators shown in the following table

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
* / %	Multiplication Division Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

4.5 Type Conversions in Expression

4.5.1 Automatic 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 the higher type. Given below is the sequence of rules that are applied while evaluating expressions.

Notes

Op-1	Op-2	Result
long double	any	long double
double	any	double
float	any	float
unsigned long int	any	unsigned long int
long int	any	long int
unsigned int	any	unsigned int

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it.

However, the following changes are introduced during the final assignment.

1. float to int causes truncation of the fractional part.
2. double to float causes rounding of digits.
3. long int to int causes dropping of the excess higher order bits.

4.5.2 Casting a Value

Casting a value is forcing a type conversion in a way that is different from the automatic conversion. The process is called type cast. The general form of casting is

```
(type_desired) expression;
where   type_desired: standard C data types and
                expression: constant, variable or an expression.
```



Example:

```
1.   #include<stdio.h>
                main( )
                {
                        int production, sale;
                        float ratio;
                        ratio = (float) production / sale;
                        printf ("%f\n",ratio);
                }
```

in expression ratio = (float) production / sale; production is converted to float, otherwise decimal part of the result of division would be lost and ratio would represent a wrong figure.

2. Notice how a cast affects the value of the following example:

```
#include<stdio.h>
main( )
{
        int a, b;
        float c, d;
        a = 1;
        c = 3.1415;
        b = (int) c;/*b will receive the value 3 */
        d = (float) a / (float) b;      /* d = 0.333 */
}
```

4.6 Operator Precedence and Associativity

Notes

An expression may contain more than one operator. Which operator will execute first depends on its precedence. Precedence defines the sequence in which operators are to be applied on the operands, while evaluating the expressions involving more than one operators.

Operators of same precedence are evaluated from left to right or right to left, depending upon the level. This is known as associativity property of an operator. A complete list of operator precedence as applicable in C language is presented below. Some of the operators you are already familiar with, others will be covered elsewhere in the book.

Table 4.2: Summary of Precedence and Associativity

Description	Operators	Associativity
Function expression	( )	Left→Right
Array expression	[ ]	Left→Right
Structure operator	.	Left→Right
Unary Minus	-	Right→Left
Increment/Decrement	++ --	Right→Left
One's complement	~	Right→Left
Negation	!	Right→Left
Address of	&	Right→Left
Value at address	*	Right→Left
Type cast	(type)	Right→Left
Size in bytes	sizeof	Right→Left
Multiplication	*	Left→Right
Division	/	Left→Right
Modulus	%	Left→Right
Addition	+	Left→Right
Subtraction	-	Left→Right
Left shift	<<	Left→Right
Right shift	>>	Left→Right
Less than	<	Left→Right
Less than or equal to	< =	Left→Right
Greater than	>	Left→Right
Greater than or equal to	> =	Left→Right
Equal to	= =	Left→Right
Not equal to	! =	Left→Right
Bitwise AND	&	Left→Right
Bitwise XOR	^	Left→Right
Bitwise OR	|	Left→Right
Logical AND	&&	Left→Right
Logical OR	||	Left→Right
Conditional	?:	Right→Left
Assignment	=	Right→Left
	* = / = % =	Right→Left
	+ = - = & =	Right→Left
	^ = | =	Right→Left
	<< = >> =	Right→Left
Comma	,	Right→Left

Notes



Task Two numbers are input through the keyboard into two locations C and D. Write a program to interchange the contents of C and D.

4.7 Summary

Precedence defines the sequence in which operators are to be applied on the operands, while evaluating the expressions involving more than one operator.

4.8 Keywords

Expression: A combination of identifiers and operators according to some rule that yields a value

Operator Proceeding: The precedents of an operator determine the order in which expression will be evaluated

Operator: A symbol that works on one or more values to yield another value

The Size of Operator: The size of operator which is used to measure the date sizes. It a unary compile type operator that is to return the length of the variable or parenthesized type specifiers.

Token: A token is a group of characters separated from other group of characters by one or more white space

Type Casting: Specifying the data type before a value in order to convert one data type to another compatible data type

4.9 Self Assessment

Choose the appropriate answers:

1. Which one is not a operator?
   - (a) Bitwise operator
   - (b) Comma operator
   - (c) Local operator
   - (d) Assignment operators
2. Relational operator generally used for
   - (a) Compare two operands
   - (b) Addition two operands
   - (c) Multiplication two operands
   - (d) None of the above
3. Symbol used to represent increment operator is
   - (a) +-
   - (b) ++
   - (c) - -
   - (d) **

4. Conditional operator represented by

   (a) :?

   (b) :_

   (c) :*

   (d) :%

5. Array expression represented by

   (a) []

   (b) ()

   (c) ++ - -

   (d) //++

6. Which one is not a relational operator?

   (a) = =

   (b) ! +

   (c) >

   (d) > =

Fill in the blanks:

7. ..... are used to store the result of an expression to a variable.

8. Bitwise operators are used ..... of data at bit level.

9. An expression may contain more than ..... operator.

10. ..... operators work on numeric type of operands.

4.10 Review Questions

1. What are the different classes of operators available in C language?

2. Define the term "Expression". Explain the various types of expression in C.

3. What are the various logical and relational operators supported by C. Explain them with proper examples.

4. Draw a table that will provide a complete list of operators, their precedence level and their rules of association.

5. List down the advantages and limitations of using conditional operator in a C program.

6. Write short notes on:

   (a) Shorthand assignment operators

   (b) Bitwise operators

7. What will be the output of the following program:

```
main( )
{
int i = 32, j = 65, k, l, m, n, o, p ;
k = i | 35 ; l = ~k ; m = i & j ;
```

Notes

```
n = j ^ 32 ; o = j << 2 ; p = i >> 5 ;
printf ( "\nk = %d l = %d m = %d", k, l, m ) ;
printf ( "\nn = %d o = %d p = %d", n, o, p ) ;
}
```

8. Write a program addition of 1-10 number with the help of arithmetic operator.
9. Ramesh's basic salary is input through the keyboard. His dearness allowance is 40% of basic salary, and house rent allowance is 20% of basic salary. Write a program to calculate his gross salary.
10. Temperature of a city in Fahrenheit degrees is input through the keyboard. Write a program to convert this temperature into Centigrade degrees.

Answers: Self Assessment

1.	(c)	2.	(a)	3.	(b)	4.	(a)
5.	(a)	6.	(b)	7.	Assignment operators		
8.	for manipulation	9.	one	10.	Arithmetic		

4.11 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication: 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, *Problem Solving with Computers*, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer*, Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 5: Managing Input and Output in C

Notes

CONTENTS

Objectives

Introduction

5.1 Input/Output Function

5.2 Reading and Writing a Character

5.2.1 getchar() and putchar()

5.2.2 getch() and putch()

5.2.3 getche()

5.2.4 gets() and puts() Functions

5.2.5 clrscr() Function

5.3 Formatted Input/Formatted Output

5.3.1 printf()

5.3.2 Escape Sequences

5.3.3 scanf()

5.3.4 sign (*)

5.3.5 The Asterisk Sign (*)

5.4 Summary

5.5 Keywords

5.6 Self Assessment

5.7 Review Questions

5.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain the input/output function
- Know how to read and write a character
- Describe formatted input/output

Introduction

A computer program probably would serve no useful purpose if a user cannot interact with the program. In most programming assignments it is necessary that the program reads input values from the user console and produces the intended useful output to the user.

The peculiarity about C is that unlike other high level languages, C does not have any built-in I/O statements as part of its syntax. In this unit, we will see how C manages various I/O

Notes

operations. We shall also learn about various C's standard library functions along with their respective header files. This unit is not intended to be a complete treatment of these topics, but it provides enough information so that you can start writing real programs.

5.1 Input/Output Function

I/O operations deal with the transfer of data to peripheral devices such as monitor, key board, printer or secondary storage etc. As C has no provision for receiving data from input devices (such as keyboard) nor for sending data to the output devices (such as monitor), all I/O operations are carried out through library functions such as printf(). A library is nothing more than one or more files that contain a group of predefined functions. The developers of C compiler have written several standard I/O functions and put them in the library called on "C standard library".

These library functions can be accessed from the standard library by different methods, depending upon the compiler and required functions. Some compliers automatically search libraries for called functions. In most, the programmer must explicitly state the library file name during the linking process, thus only the required functions will be included in the executable program.

Functions of this type are the resident of a special file with the extension.h, (such as stdio.h) generally called as header files. A header file can be inserted into a C program file using #include compiler directive as shown below.

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

The #include directive instruct the compiler to read the particular file i.e., stdio.h (standard input output header file) and replace this line with the contents of stdio.h file. Similarly, to make use of other predefined functions, their respective header files must be included in a program so that the declaration of the function becomes available to the program.

C has a rich set of standard I/O library functions. However, these I/O functions are not the part of C's formal definition. C's standard library function for I/O can be broadly divided in to the following categories:

1. Port I/O functions
2. Disk I/O function
3. Console I/O function

Port I/O function deals with the different I/O operators on various ports such as mouse port, printer port etc. The detailed study of the port I/O functions is beyond the scope of this text.

Disk I/O functions are used for manipulating files as the secondary storage devices like floppy disk or hard disk. Disk I/O functions are nothing but the file handling functions as files are located in secondary storage generally on disk.

In its most general form the word 'console' refers to the standard input and output devices. Console I/O functions deal with these standard input or output devices, which are often defined as the keyboard and monitor by default. These functions accept input from the keyboard and produce output on the screen.

C takes all input and output as stream of characters. A stream is nothing, but a series of bytes. C language treats all streams equally i.e., whether a program gets input from the keyboard, a disk file or a modem, it consider it as only a stream of characters.

Different steams are used to represent different kinds of data flow. In C, there are three streams associated with console I/O operations.

1. ***stdin:*** A stream that supplies data to the program i.e., standard input, usually from the keyboard.

Notes

2. ***stdout:*** A stream that receives data from the program i.e., standard output; usually to the monitor.

3. ***stderr:*** A stream used to keep error messages separate from program's output i.e., standard error; usually points to your terminal screen.

C provides many functions for performing console I/O operations. These function permits the transfer of information between the computer's standard input and output devices (i.e., keyboard and monitor). Few of them give formatting control over input and output operations. Where as some of them doesn't allow to control the format of I/O operations.

From this aspect, console I/O operations can be further categories as:

1. Unformatted console I/O functions

2. Formatted console I/O functions

To access these functions, it is necessary to include the standard I/O library header file. The header file stdio.h contains the declaration for these functions. Therefore, always include the header file stdio.h in your C program before using these console I/O functions.



Task Point out the errors, if any, in this program:

```
main( )
{
        int ival ;
        scanf ( "%d\n", &n ) ;
                printf ( "\nInteger Value = %d", ival ) ;
}
```

5.2 Reading and Writing a Character

Unformatted console I/O functions don't allow input and output to be formatted as per the user requirements. In this category, we have:

1. Character I/O functions

2. String I/O functions

The functions that program input/output of one character at a time are known as character I/O functions. These are the most fundamental I/O functions as they deal with the individual character value. Following functions can be used for inputting a character from the keyboard:

1. getchar()

2. getch()

3. getche()

Where as the output of a character as the monitor, the following functions can be used:

1. putchar()

2. putch()

Beside these, getc() and putc() can also be used for one same purpose.

Let us see the working of above mentioned console I/O functions with the help of programs.

Notes

5.2.1 getchar() and putchar()

getchar() function is used for reading a character from the keyboard. The syntax for the getchar() function is:

```
Character_variable = getchar(void);
```

Where character_variable is any valid C variable name. The word void indicates that no argument is needed for calling the function. Following statement reads a character and stores it in variable ch, that is of type char obviously.

```
ch = getchar();
```

The getchar() waits for the character input until a character in typed at the keyboard. The typed character is echoed to the monitor and before assigning this value to the character variable appeared on the left side, it requires a carriage return (enter key) to be type by the user. getchar() function returns a value called EOF (End of File) if an error occurs.

Typically, the value of EOF is 1, though this may vary from compiler to compiler.

The putchar() is complementary function of getchar(). It is used to display a character on the monitor. The syntax for the putchar() function is:

```
putchar(character_variable);
```

Where character_variable refers to some previously declared character variable. The following statement displays a character value on the monitor whatever is stored inside ch at the current cursor position.

```
putchar(ch);
```

We can also use putchar() with character value directly, as shown below.

```
putchar('V');
```

This statement will display the character V as the monitor, whereas the statement

```
putchar('\n');
```

would cause the cursor on the screen to move to the beginning of the next line.

This function also returns EOF if there occurs an error.

The following program illustrates the working of these functions. This program will accept a character from the keyboard and will print it on the monitor screen.



Lab Exercise

```
#include<stdio.h>
void main()
     {
     char ch;              /*declare a variable ch of char type*/
     printf("\n Enter the character:");
     ch = getchar(); /*will read a character from the    keyboard*/
     printf("\n Typed character is:");
     putchar(ch); /*will print the value of ch on to the  monitor*/
     }
```

Notes

The output of the program:

```
Enter any character: R↵
Typed character is: R
(Where the sign (↵) represents pressed enter key)
```

Technically, both the function i.e., getchar() and putchar() uses integer values to perform their respective operations, as character values are internally represented by their associated ASCII codes. ASCII codes are nothing but the integer numbers represented in decimal format. For instance, the character value 'a' is equivalent to numeric value 97 as this number is the ASCII code for letter 'a'. Thus, functions can also be applied directly with the ASCII codes as used in the following program.



Lab Exercise

```
#include<stdio.h>
void main()
{
char ch1 = 97; ch2 = '\n';  /* ch1 = 97, as ASCII code directly */
putchar(ch1);
putchar(ch2);
putchar('b');
putchar('\n');
putchar (99);       /* ASCII code of letter 'c' */
putchar (10);       /* ASCII code of new line character i.e., '\n' */
}
printf("thank you");
}
```

The output of the program:

a

b

c

Thank you

As the difference of 'a' and 'A' in 32 (i.e., 97 - 65 = 32), program can also be written to convert a lowercase character in to upper case and vice-versa, as listed below:



Lab Exercise

```
#include<stdio.h>
void main()
{
     char ch;
```

Notes

```
        printf("\n Enter any character in lowercase:");
        ch = getch();
        printf("\n Typed character in uppercase is:");
        purchar(ch-32);
}
```

The output of the program:

Enter any character in lowercase: r ↵

Types character in uppercase is: R



Note This program will run successfully if it gets proper input; otherwise output may be unexpected.

5.2.2 getch() and putch()

getch() and putch() also serve the same purpose as their preceding character I/O functions i.e., getch() is used to accept a character from the keyboard and putch() is used to print a character on the monitor.

Although getch() and putch() has the same syntax format as of getchar() and putchar() i.e.,

```
Character_variable = getch(void);
```

And

```
putch(Character_variable);
```

respectively, but still these are slightly different from them.

Unlike getch(), neither the user is required to press enter key after typing a character nor the typed character is echoed to the monitor while using getch(). This value could only be visualized on the monitor by using character output function.

This added advantage of getch() could be proved beneficial in the application where user want to hide the input, for instance, password security.

The working of putch() is exactly the same as of putchar(). The character output function putch() is only the mirror image if the character input function is getch().

Consider the following Program, which demonstrates the use of getch() and putch() function:



Lab Exercise

```
#include<stdio.h>
void main()
{
char ch;
printf("\n Type any character and press enter key to see  the output");
ch = getchar();
printf("\n Typed character is:");
```

Notes

```
putch(ch);
ch = getch();
printf("\n Type another character and see the output without pressing enter
key:");
ch = getch();
printf("\n Typed character is:");
putch(ch);
}
```

The output of the program:

1. Type any character and press enter key to see the output: R ↵
2. Typed character is: R
3. Type another character and see the output without pressing enter key: Typed character is : V



Note Second time input character didn't echoed on the monitor and even it didn't require enter key to be typed to proceed further; obviously due to getch().

Another possible use of getch() is to temporarily halt the execution of a program intentionally. Ironically, this is the most general use of getch(), found in almost all the C programs implementing practically.

As you know that the execution of a C program is very fast. While working with editor such as turbo editor, you can not observe the output properly because after successful execution, control quickly moves back to editor environment.

In order to cop up with this situation, getch() can be used to halt the execution temporarily as it waits for a character to be input from the keyboard. The execution remains in the same state (i.e., waiting state), until and unless, there is an input of any key.

To see this particular effect of getch(), try to execute the following program with and without using getch() at the end of the program.



Lab Exercise

```
#include<stdio.h>
void main()
{
printf("\n Press any key to continue....");
getch(); /* to halt the execution temporarily*/
}
```

The output(without using getch()) couldn't be observed. The output by using getch():

Press any key to continue.....

You can continue hereafter only by press a key from the keyboard.

Notes

5.2.3 getche()

You can also use getch() for receiving a character from the keyboard. The getche() is basically:

```
getch() + e
```

Where the letter 'e' stands for echoes. Accordingly, it doesn't wait for the enter key to be typed and also echo the typed character to the monitor. It bears the same syntax as by other family member (like getchar() and getch())i.e.,

```
Character_variable = getche (void);
```

The advantage of using getche() over getchar() in that as soon as it accepts the character from the keyboard, the character is immediately assign to the variable of left hand side, without pressing the enter key.

The advantage of using getche() with the getch() is that it echoes the character to the monitor before assigning it to the variable of left hand side. That is, character appears instantly on the monitor as soon as the user types it.

The following program illustrates the working of all character I/O functions described in this section.



Lab Exercise

```
#include<stdio.h>
void main()
{
     char ch1, ch2, ch3;
     printf("\n enter the first character:");
     ch1 = getch();
     printf("\n don't worry, enter the second character:");
     ch2 = getch();
     printf("\n OK, enter the last character:");
     ch3 = getch();
     printf("\n all the character you'd typed, are: \n");
     putchar(ch1);
     putch('\n');
     putch(ch2);
     putchar('\n');
     putchar(ch3);
}
```

The output of the program:

Enter the first character:

Don't worry, enter the second character: b

OK, enter the last character: c ↵

All the characters you'd typed, are:

Notes

```
a       (for instance)
b
c
```

Execute the program carefully as the effects of character input functions could also be observed during the execution of the program.



Task What would be the output of this program?

```
main( )
{
        printf ( "More often than \b\b not \rthe person who \
        wins is the one who thinks he can!" ) ;
}
```

5.2.4 gets() and puts() Functions

The limitation of character I/O functions is that they cannot handle more than one character at a time. Where as strings are used frequently in real life program. A string is nothing, but a sequence of characters. The functions which facilitates the transfer of string between the computer and the standard I/O devices are known as string I/O functions. Following function can be used for handling strings I/O:

1. gets()
2. puts()

gets() function is used to accept a string from the keyboard whereas puts() function is used to print a string on the monitor. Besides these I/O functions, C's standard library also provides several functions for various string handing operations. Let's first discuss gets() and puts() in this section.

The gets() function receives a sequence of characters i.e., a string entered at the keyboard and store them in a variable (essentially as Array of type char) mentioned with it. The general syntax of gets() is:

```
gets(character Array);
```

Where character Array is a valid C variable declare as an array of character type. The concept of arrays has been discussed in chap(9). The gets() function reads character from the standard input stream until a new line character ('\n') is read. The newline character is suppressed and a null ('\0') character is, then, appended in the end. This string is, then, stored in the memory address provided to by argument. That is why a string in C is also called as a character array terminated by a null character ('\0').



Example:

```
char str[11];
gets(str);
```

In the above code, str is a character array and can store a string of 10 valid characters, as the 11th space is reserved for null character ('\0') with which a string is always terminated with.

Notes

The function gets() will accept a string of maximum 10 characters and will store it in a memory address pointed to by str.

The puts() function, in contrast with gets(), writes a sequence of characters i.e., a string on the monitor and moves the cursor to the next line i.e., it ends with a newline character ('\n') automatically. The general syntax of puts() is:

```
puts(character array);
```

Where character array refers to some previously declared array of type char.

The following statement will display a string on the monitor whatever is stored inside str and will also causes the cursor to be moved in the next line.

```
puts(str);
```

We can also print string literals on the screen by using puts(), as shown below:

```
puts("Rudraksh");
```

White spaces are allowed in the string as a string ends with a null character.

The following statement is a valid use of puts() function.

```
puts(" Hello! Rudraksh");
```

Both the function i.e., gets() and puts() return an EOF if there occurs an error or no characters are read (i.e., a null string).

The following program demonstrate the use of gets() and puts(). This program will accept a string from the keyboard and will print it on the monitor screen.



Task Write down two functions xgets( ) and xputs( ) which work similar to the standard library functions gets( ) and puts( ).



Lab Exercise

Program

```
#include<stdio.h>
void main()
{
char str [11]; /* declar a character array str of size 11 */
printf("\n enter a string (Maximum 10 characters):");
gets(str); /* will read a string from the keyboard*/
printf("\n the entered string \n:");
puts(str); /* will print the under of str on the monitor and */
print(" /* and advances the cursor to the .......*/ thank you");
}
```

Output:

Enter a string (maximum 10 characters): Rudraksh ↵

Notes

The entered string is: Rudraksh

Thank you



Note puts() ends with a new line characters that is why "thank you" appeared in the next line without giving any newline character with printf().

As mentioned earlier, white spaces and tabs are allowed as a part of input string i.e., a string may consist of multiple words. The length of the string is limited by the declaration of the string variable. As soon as the enter key is pressed, a null character ('\0') is automatically added in the end of the string and is the indication that the input of the string is completed.

Here in another program that reads a line of text into the computer and then writes it back on the monitor.



Lab Exercise

Program

```
#include<stdio.h>
void main()
{
     char sentence[80]; /* Maximum number of character    that a line */
     /* can have */
     printf("\n enter any sentence of single line: \n");
     gets(sentence);
     printf("\n The entered sentence is: \n");
     puts(sentence);
}
```

Output:

Enter any sentence of single line:

The quick brown fox jumps over the little lazy dog. ↵

The entered sentence is:

The quick brown fox jumps over the little lazy dog.

5.2.5 clrscr() Function

As input and output progresses interactively, the screen of the user console gets cluttered with I/O text. At times the programmer needs to clear the screen. C provides a library function for this purpose – clrscr() (acronym for clear the screen). The prototype of this function is defined in the standard library file – conio.h (acronym for console I/O). Therefore, this file must be included if the function is called in the program.

The syntax of the function is very simple. The function requires no argument. When called it simply clears the screen of the text and pushes the cursor to the first character position, i.e., on the left top corner of the screen.

Notes

Writing a user-friendly program is more an art than a technique. I/O functions are profusely used to achieve this user-friendliness as is elucidated in the following example.

The following code snippet expects the user to enter two integer values through the keyboard. However, when you run the program the intention is not explicitly expressed.



Lab Exercise

```
#include<stdio.h>
void main()
{
     int a, b, c;
     scanf(" %d %d", a, b);
     c= a + b;
     printf(" \n the sum of %d and %d is %d, a, b, c);
}
```

When the program is run the cursor waits for the user to interact and enter two integer values but the user may not know what to enter. A better way is to prompt to the user with a user-friendly message as is in the following version of the same program.

```
#include<stdio.h>
void main()
{
        int a, b, c;
        printf("\n Enter any two numbers:");
        scanf(" %d %d", a, b);
        c= a + b;
        printf(" \n the sum of %d and %d is %d, a, b, c);
}
```

This time the program prints a message asking the user to enter two numeric values. The power of user-friendliness of a program becomes evident in menu driven programs. Herein a menu of different actions is presented to the user when the program runs.

5.3 Formatted Input/Formatted Output

In this category, we have functions that allow input and output operations to be performed in a fixed format. Formatting of I/O operators deals with some of the following issues like:

1. How much field width is required to display the various values on the monitor?
2. How many decimal places are required to display the fractional part of a real number?
3. Should data values be left aligned or right aligned, and how much?
4. How much space between two data values is to be given?
5. How various type of data i.e. integer, character, and string can be used together I/O operators, etc.

Notes

The two most frequently used functions for formatted I/O are printf() and scanf(). The printf() is used to display the formatted data items on the standard output device normally the monitor whereas scanf() is used to read the formatted data input from the standard input device normally the keyboard. However, both the functions are slower than the previous classes of functions because of their greater complexity. These functions are defined in the header file stdio.h and return EOF if there occurs an error or end of file. Let's discuss these functions in detail one by one.

5.3.1 printf()

The printf() in one of the most important and useful functions to display data on monitor. We have seen the use of printf() for printing messages in the various example given previously in this book. For example, the statement

```
printf(" this section will discuss printf() in detail");
```

will simple print this message on the monitor. Beside these text messages, a program frequently required numeric values and the value of other variables to be displayed on the screen.



Example: In order to print the sum of two numbers say a and b, in a new line along with some identifying text, the printf() will take the following form:

```
printf("\n the sum of %d and %d is %d.", a, b, c);
```

If the value of a and b is 5 and 6 respectively, then the output would be as follows:

The sum of 5 and 6 is 11.

Undoubtedly, a little more complicated then printing a simple message. Before getting the detail of the various section of this printf(), Let's discuss the general format of a printf(), shown below:

```
printf("Format string", arg1, arg2,...., argn);
```

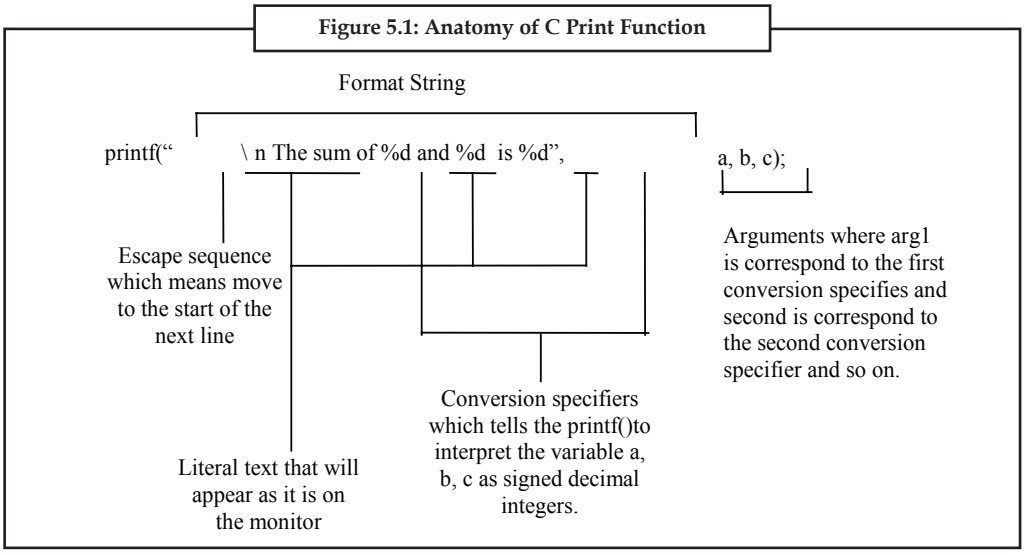
Where format string refers to a string in enclosed in double quotes that contain formatting information and arg1, arg2, ..., argn are arguments (may be constants, variable, or other complex expressions) whose values are formatted and printed according to the specification of the format string.

The format string in a printf() contains the format specifies that defines how the output is formatted. Following are the three possible components of a format string:

1. Literal text that is simply printed as entered in the format string.
2. An escape sequence that begins with a \ (backslash) sign, provides special formatting control.
3. A conversion specifier that begins with a % sign and followed by a single character, that tells printf() how to interpret the arguments being used. To understand the various sections of previously used printf() statement, consider Figure 5.1:

Notes

Figure 5.1: Anatomy of C Print Function



The output of given statement if the value of a and b is 5 and 6 respectively, would be as:

The sum of 5 and 6 is 11

Let's see how this output is evaluated:

The printf() interpret the format string from left to right and start sending the coming characters to the standard output device. As soon as it encounters with the \ (backslash) (that is, the indication of the presence of as escape sequence), it takes action accordingly. Similarly when it encounters with % (conversion specifier) sign, it picks up the corresponding argument and prints its value according to the specified format. This process comes to an end, when it encounters the closing pair of double quotes.

In our case, the first character after the opening pair of double quote is \ , followed by a character n, so the effect of \n will take place i.e., output of the coming characters will start from the starting of the next line. Output up to this stage appears as:

The sum of

Then comes the character % followed by character d (that is, the indication to treat the corresponding variable as assigned decimal integer), so it picks up the variable a and will print its value on the screen. At this stage the output will be looking like as:

The sum of 5

In the same manner, this process will go on until there comes an end point of format string. The final output would be appears as:

The sum of 5 and 6 is 11

The following program will help you to understand the concept more closely as it uses the printf() statement to print the result of the calculations.

Write a program to print the sum of two numbers.



Lab Exercise

Program:

```
#include<stdio.h>
void main()
```

Notes

```
{
    int a, b, c;
    a = 5;
    b = 6;
    c = a + b;
    printf("\n sum = %d", C); /* will print the desired */
                                /* result on the monitor */
}
```

Output:

```
Sum = 11
```



Note Argument list must be corresponding to the conversion specifiers mentioned in the format string i.e., number and the type of arguments must correspond to the conversion specifiers, otherwise unexpected result may come.

Lets discuss the various possible component of a format string in detail.

Literal Text

The format string usually consists of multiple characters. Except for double quotes, escape sequences and conversions specifiers, all characters with in a pair of double quotes will be treated as literal text (string context) and will be display as it is a on the monitor.



Task If a character string is to be received through the keyboard which function would work faster? scanf() or getch()

5.3.2 Escape Sequences

As discussed earlier, escape sequences are used to control the location of output by moving the cursor on the monitor. Any character that is prefix with a backslash is supposed to be treated as an escape sequence. The backslash tells the compiler that this is a special character constant that would otherwise have different meaning to printf(). For example,

Consider the following Table 5.1.

Table 5.1: Character Constant in printf Function

Character Constant	Meaning
"X"	The character X
"\X"	The character that follows this is in hexadecimal
"n"	The character n
"\n"	New line

The following program shows the usage by some of the frequently used escape sequences.

Notes



Lab Exercise

Program

```
#include<stdio.h>

void main()

{

printf("\n 1..\t2..\t3.. here we go..\n");

printf("The question is, \" said Humpty Dumpty,\"which is to be master-
that\'s all.\"");

}
```

Output:

```
1..  2..    3..    here we go..
```

"The questions is, "said Humpty Dumpty," which is to be master-that's all".



Note By default a tab stop is equivalent to 8 columns and the delimiters character i.e., single quotes, double quotes and the back slash can be printed by preceding them with the backslash.

5.3.3 scanf()

As mentioned earlier, in order to write interactive program we must include some statements in a program that could be able to receive the data from user. In this context, we presented a couple of functions like getchar(), gets(), and getch() etc. But a programmer needs more flexibility in terms of:

1. Read the data from keyboard according to a specified format
2. Instruct the compiler to receive the particular type of value from the keyboard. For instance, integer value or floating point value.

Instruct the compiler to read the specified number of digits of a given number.

Reading mixed data types from the keyboard using single function.

But the use of above mentioned functions is restrict with the character values only. There is a need of more flexible and general function that could address the problems mentioned above.

scanf(), the complement of the printf(), can actually be used to read the different type of data from the keyboard in a specified format. Due to what, it is referred to as formatted input functions. Like printf(), scanf() also uses a format string to describe the format of the input, but with some little variations as given below:

1. It doesn't allow escape sequences in the format string.
2. It requires a special operator & called as "address of" to be prefix with the variable identifiers.

So, a scanf() takes the following from:

```
scanf(" format string", arg1, arg2,.......argn);
```

Notes

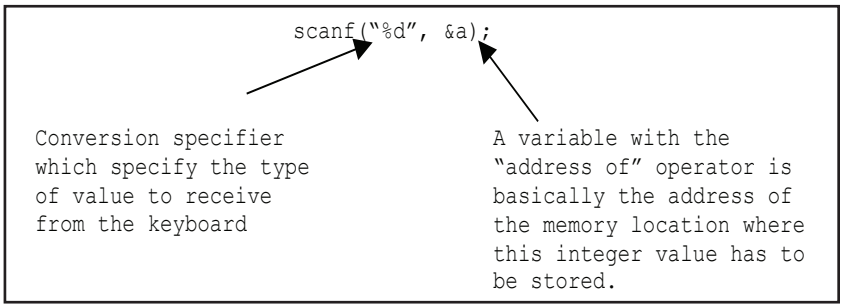
Where format string contains the formatting information by using which the data is to be entered and arg1, arg2,...., argn are the arguments (normally variables preceded by an ampersand &) specify the address of location where the data is stored. Both the section i.e., format string and arguments (within itself also) must be separated by commas.

The format string in a scanf() describe the format of the input and it may contain:

1. Conversion specifiers as in the printf() functions.
2. White space characters i.e., tabs, blanks, and newlines.
3. Other characters than white spaces, that are matching characters and asteric

5.3.4 sign (*)

To have a better understanding of the concept, consider the following statement:



In a very layman language, the above statement can be interpreted as an instruction to the complier to receive an integer value from the keyboard and store it in a variable named a. Where a must be an integer variable declared earlier. Once the value has been stored in the variable, it can be used anywhere for any purpose after this statement. Let's discuss the various component of this statement in detail.

Conversion Specifier in scanf()

As mentioned earlier, a conversion specifier instruct the scanf() to convert the input stream of binary data coming from the keyboard in to the data type specified by the conversion character. For instance, integer in case of %d as it utilizes the same conversion specifiers as the printf() except one i.e., [...]. More than one conversion specifier can be used in a single scanf() to read more than one value.

In such a case, corresponding variables each preceded by & must include in the same statement.



Example: The statement.

```
scanf(" %d %d", &a, &b);
```

Will read two integer values from the keyboard, first value will be assign to a and second to b.

White Spaces in scanf()

When multiple variables are entered in a single scanf(), they can be separated using white space character (i.e., blank space, tabs or new line character). White spaces in the format specifier itself are ignored. So in the input data they will be read but ignored.

Notes

Example: Consider the following statement once again:

Single space



scanf("%d %d", &a, &b);

During runtime, in response to the above statement, you could enter

10 20



single space

As white space can be a tab, so you can also enter as

10 20

tab

A white space can be in the form of newline character, so you could also enter

10 ↵

20

As white spaces are required is input stream, they just can be used to identify the end of each input value.

For real time experience, consider the following program which demonstrate the usage of scanf() to read integer values from the keyboard. This program will accept two numbers from the user and will print their sum on the monitor.



Lab Exercise

Program:

```
#include<stdio.h>
void main()
{
     int a, b, c;
     printf("\n Enter any two numbers:");
     scanf(" %d %d", a, b);
     /* two conversion characters to read two integer     values */
     c= a + b;
     printf(" \n the sum of %d and %d is %d, a, b, c);
     }
```

Notes

Output:

```
First run:
     Enter any two numbers: 5                    6 ↵
                                                 . ↵
                                                 (space)
                    The sum of 5 and 6 is 11
     Second run:
            Enter any two number : 5 _________ 6 ↵
                                              ↵
                                              tab)
            The sum of 5 and 6 is 11.
     Third run:
            Enter any two number : 5 ↵
                                   6 ↵
            The sum of 5 and 6 is 11.
```



Note During input, white spaces are simply ignored so it may be a space, or a tab, or even a newline character.

Let's write another program to receive floating point values from the user. This program will accept the radius of a circle and will print the area and circumference of a circle according to the formula given below:

area = pr2 , circumference = 2pr

where p = 3.14



Lab Exercise

Program

```
#include<stdio.h>
void main()
{
        float r, a, c;
        const float pi = 3.14;
        printf("\n Enter the radius of a circle:");
        scanf(" %f", & r);
        a = pi * r *r ; /* as r power 2 = r * r  */
        c = 2 * pi * r;
        printf("\n Area = % f", a);
```

Notes

```
        Printf("\n circumference = %f", c);
    }
```

Output:

```
Enter the radius of a circle: (user input)
Area =
Circumference =
```



Note For more accuracy of the result, variables may be defined of the type double.

The next segment of the programs will demonstrate the use of scanf() with character values. The first program will accept a character from the user in lower case and will display it in upper case, using printf() we have already written this program using getch() and putchar(). And the second program will receive a character string from the user and will print it on the monitor using printf().



Lab Exercise

Program

```
#include<stdio.h>
void main()
{
char ch;
printf("\n Enter any character in lower case:");
scanf("%c', & ch);
printf(" \n The typed character in upper case is %c", ch);
}
```

Output:

```
Enter any character in lower case: a ↵
The typed character in upper case is: A
```



Note If execution is not provided with the proper input, result may be unexpected.

As receiving string using scanf() is quite different, observe the following program:



Lab Exercise

Program

```
#include<stdio.h>
void main()
```

Notes

```
{
        char name[10];
        printf("\n enter you name:");
        scanf("%s", name);  /* ampersand sign (&) is not required */
        printf("\n God bless you %s", name);
        }
```

Output:

```
Enter your name: Rohan ↵
God Bless you Rohan
```



Note The variable name didn't precede with & operator as it is not required here because character strings are read is to character arrays, and the array same without any index is the address of the first element of the array. The format specifier %s cause the scanf() to read character until a white space is encountered. Then a null character is automatically appended to the array, is the indication of the end of a string.

As mentioned earlier, a single scanf can be used for reading more than and value or the mixed data from the keyboard. Care must be taken while inputting mixed data values from the keyboard as even a little tempering with the input order and type may causes the result to be appear very surprising.



Example: Consider the following segments of code (It is assumed that the variables used in the scanf() has been declared already of the concerning type):

```
scanf("%d%f%c" &a, &b, &c);
```

The given statement is expected to read these different values from the keyboard i.e., first integer, second float and third is of character type. If we input the value as:

```
123         1.23 ↵
```

and typed enter key without giving value, the result may be unexpected as:

```
scanf(" %d %f", &a, &b);
```

If we would provide the input as

9.2 9

instead of

9 9.2

as it is suppose to be, it will store 9 in a and the value of that will be assign to b is totally unpredictable.

```
scanf(" %d %s %f", &rn, name, &marks);
```

If we provide the input other than the required sequence i.e., first roll number (integer value), then name (string value), and the marks (float value) as given below:

```
Aaryan    17      63.4 ↵
```

Notes

then the result may not be the desired one as scanf() will terminate the reading process after encountering with the first value of type string instead of type int.

Here's a complete program which is used different multiple conversion characters in scanf(). This program will accept the rn, name and marks of a student in three subjects and will print his/her percentage and average marks. (Assuming three hundred is the maximum marks)



Lab Exercise

Program

```
#include<stdio.h>
void main()
{
int rn,
char name[10];
        float mks1, mks2, mks3, avg, per;
        printf("\n Enter you roll number, name, and marks in three
        subjects");
        scanf("%d %s %f %f %f", &rn, name, &mks1, &mks2, &mks3);
        avg = (mks1 + mks2 + mks3)/3;
        per = (mks1 + mks2 + mks3) * 100/300;
        printf(" \n %s, roll no = %d  has secured :", name, rn);
        printf("\n Average marks =  % .2f", avg);
        /* (.2) precision specifier is */
        /* used to restrict the output of Average & percentage */
        printf("\n percentage marks = % .2f", per);
        }
```

Output:

Enter your roll number, name, and marks in three subjects: 2 Rishivansh

```
96.0 98.5 100.0 ↵
```

Rishivansh, roll no. = 2 has secured:

Average marks =

Percentage marks =



Note Providing input for marks other then floating point value as 96,98.5,100 may not be acceptable by some complier, thus results may be unexpected.

The conversion specifier %0 and %x can be used with scanf() to receive the integer values from the keyboard in octal and hexadecimal format respectively.

Notes

The format string in scanf(), like in printf() can also be provided with others informations like maximum field width, matching character, asteric sign(*) etc. To perform the additional formatting as input data. Let's discuss these features one by one to explain the other capabilities of scanf().

Maximum Field Width

You must have observed in the programs given before that the continuous non white space characters in the input stream considered to be as a single data item or value. Whereas, whitespace character like space, tabs, or newlines are used to separate the input stream into the various data values of fields.



Example: If the statement

```
scanf("%d %d:", &a, &b);
```

Is provided with this input as

123 4567 ↵

124

then the value 123 will be assign to a and 4567 will be assigned to b.

Where as if the input is

12345 67 ↵

the value 12345 will be assign to a and 67 will be assigned to b.

So it is clear from the above examples that all continuous nonwhite space characters collectively defines a single value. However, it is possible to assign a limited number of character to a particular variable from the set of continuous non white space characters. That is, if the input, for the above example is as:

1234567

then it is feasible to assign 123 to a and 4567 to b, by specify the maximum field width for a variable as in our case 3 for the variable a.

We can do the same by placing an unsigned decimal integer between the percentage sign (%) and conversion character. It may take the focus as given below:

```
%w conversion character
```

where w is an integer number and specify the number of characters to be read and conversion character may be any character from the given list.

For instance, consider the following expression.

```
%3d
```

will be interpreted as read three characters of the coming input value in integer mode and store it in the concern variable.

If the input value is provided according to the specified field width, it will assign as is it to the concerned variables.



Example: The statement its:

```
scanf(" %3d %4d", &a, &b);
```

and input values are 123 4567 then, 123 will be assigned to a and 4567 will be assigned to b.

Notes

If the input value contains lesser character than the specified field width then also they will be assigned as it is as whitespace will work as a separator.



Example: The statement is:

```
scanf("%3d %4d", &a, &b);
```

and input values are 12 45 then 12 will be assigned to a and 45 will be to b. And if the data contains more character then the specified field width then only the specified number of character will be assigned to concerned variable and the characters in the input value beyond the specified width will be assigned to the next variable existing other wise to the first variable in the next scanf() call.



Example: Consider the statement as:

```
scanf(" %3d %4d", &a, &b);
```

and the input values are 12345 6789 then

the value 123 will be assigned to a and 45 to b. The value 6789 is increased and will be assigned to the first variable of the consecutive scanf().

The following segments of code will give you more clear vision of the concept, where input sections in consists of scanf() statement and the input values and the output section will show you the resulting assignments. All the variables used in these statement are supposed to be declared earlier of the concerning type.

Input : scanf("% 3d % 3d % 3d", &a, &b, &c);
Input values = 123456789
Output : a = 123 b = 456 c = 789

Input : scanf("% 3d %3d %3d", &a, &b, &c);
Input values = 1234 56789
Output : a = 123 b = 4 c = 567

Input : scanf("%2d %3f %c", &a, &b, &c);
Input values = 12 1.23 x

Output : a = 12 b = 1.2 c = 3

Input : scanf("% 2d % 5s", &a, b,);
Input value = 1234 Hello

Output : a = 2 s = 34

Input : scanf("% 4s % c", a, & b);
Input value = Hello X

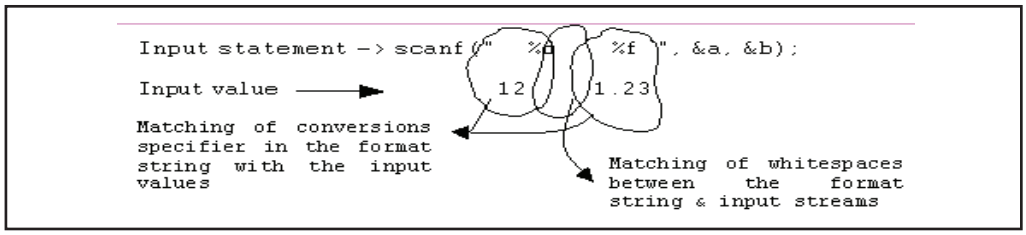
Output : a = Hell b = o

Matching Characters

Matching characters in scanf() is the concept of matching the contents of format string to the input stream. In the program of scanf() given in the previous section, there was a matching between the format specifiers in the format string with the input values in the input stream and between the white space character in format string with the white space character in the input stream.

Consider the figure

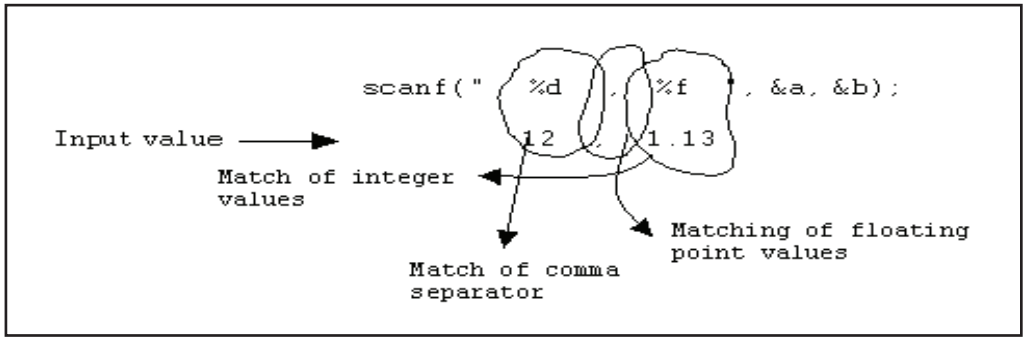
Notes



Sometimes, input for the above statement may be provided as:

```
12,1.23
```

where a comma is used to separate the both data values in input stream, surely won't be accepted by the scanf(), as punctuations do not count as separator. scanf() accept these type of characters() in the input stream at the particular place only if it is included in the format string at the same place. So the above statement could be rewrite as:



Then the character/(s) other than the specifiers or white spaces included in the format string, called as matching characters.

5.3.5 The Asterisk Sign (*)

Sometimes, there are situations when you don't want to assign the particular data value (given in the input stream) to the concerning variable mentioned in the scanf().



Example: The statement

```
scanf("%c %d %f", &a, &b, &c);

and the Input  x 12 1.23
```

Will be assign the values x, 12 and 1.23 to a, b, and c respectively. Now in order to suppress the particular assignment for instance, say character assignment (i.e. a = x), an asterisk sign (*) called as suppression character can be prefix to the particular conversion character (i.e., %*c). When included with in a conversion specification, the input defined by this conversion specifier will be skipped over that is particular assign won't take place. For more clarity consider the following program.

Notes



Lab Exercise

Program

```
#include<stdio.h>
void main()
{
        char a;
        int b;
        float c;
        printf("\n Enter a character, an integer, and a float value:");
        scanf(" %*c,%d,%f", &a, &b, &c);
        printf("\n Assignment that took place\n");
        printf("\na = %c", a);
        prinff("\nb = %d", b);
        printf("\nc = %f", c);
}
```

Output:

```
Enter a character, an integer, and a float value: x,
12, 1.23 ↵
a = x
b = 12
c = 1.23
```



Case Study

Program

```
/* Count chars, spaces, tabs and newlines in a file */
# include "stdio.h"
main( )
{
FILE *fp ;
char ch ;
int nol = 0, not = 0, nob = 0, noc = 0 ;
fp = fopen ( "PR1.C", "r" ) ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
```

Contd...

Notes

```
break ;
noc++ ;
if ( ch == ' ' )
nob++ ;
if ( ch == '\n' )
nol++ ;
if ( ch == '\t' )
not++ ;
}
fclose ( fp ) ;
printf ( "\nNumber of characters = %d", noc ) ;
printf ( "\nNumber of blanks = %d", nob ) ;
printf ( "\nNumber of tabs = %d", not ) ;
printf ( "\nNumber of lines = %d", nol ) ;
}
```

Here is a sample run...

Number of characters = 125

Number of blanks = 25

Number of tabs = 13

Number of lines = 22

Questions

1. Write a program to find the size of a text file without traversing it character by character.
2. Write a program to copy one file to another. While doing so replace all lowercase characters to their equivalent uppercase characters.
3. Write a program that merges lines alternately from two files and writes the results to new file. If one file has less number of lines than the other, the remaining lines from the larger file should be simply copied into the target file.

5.4 Summary

- I/O operations deal with the transfer of data to peripheral devices such as monitor, key board, printer or secondary storage etc.
- A library is nothing more than one or more files that contain a group of predefined functions. In its most general form the word 'console' refers to the standard input and output devices.
- Unformatted console I/O functions doesn't allow input and output to be formatted as per the user requirements. getchar() function is used for reading a character from the keyboard.
- The putchar() is complementary function of getchar(). It is used to display a character on the monitor.

Notes

- The another possible use of getch() is to temporarily halt the execution of a program intentionally. gets() function is used to accept a string from the keyboard whereas puts() function is used to print a string on the monitor.
- Formatted I/O Functions allows input and output operations to be performed in a fixed format.
- The printf() in one of the most important and useful functions to display data on monitor.
- Except for double quotes, escape sequences and conversions specifiers, all characters with in a pair of double quotes will be treated as literal text (string context) and will be display as it is a on the monitor.
- Any character that is prefix with a backslash is suppose to be treated as an escape sequence.
- A conversions specifiers always begin with the percent sign (%) and immediately followed by one or more conversions characters.

5.5 Keywords

ASCII (American Standard Code for Information Interchange): A coding scheme that assigns an integer between 0 and 255 to every character on the keyboard.

Console: The keyboard and monitor interface through which an operator usually interacts with the rest of the computer resources.

EOF: A constant defined in the language C to indicate the end of the file or end of the input.

Header f les: A text file that contains prototype of functions, definitions of constants etc. and which can be included in a C program file to access those functions and constants.

#include compiler directive: This compiler directive instructs the compiler to insert the contents of the specified file in place of this line.

Standard library: A group of in-built functions stored in a single file as a unit.

5.6 Self Assessment

Choose the appropriate answers:

1. Out of the following which one is not the standard I/O library functions.
   (a) Memory I/O functions
   (b) Disk I/O functions
   (c) Port I/O functions
   (d) Console I/O functions
2. Which function used for handling strings I/O?
   (a) lets()
   (b) rets()
   (c) puts()
   (d) xats()

Notes

3. One of the most important and useful functions to display data on monitor is.
   (a) scanf()
   (b) clrscr()
   (c) for()
   (d) printf()
4. Which function used to keep error messages separate from program's output?
   (a) stdout.
   (b) stderr
   (c) stdin
   (d) None of the above

Fill in the blanks:

5. ..... are used to control the location of output by moving the cursor on the monitor.
6. Matching characters ..... is the concept of matching the contents of format string to the input stream.
7. ..... function deals with the different I/O operators on various ports such as mouse port, printer port etc.
8. getchar() function is used for reading a character from the .....
9. An ..... called as suppression character can be prefix to the particular conversion character (i.e., %*c).

State whether the following statements are true or false:

10. getchar() function is used for reading a character from the keyboard.
11. The working of putch() is exactly the same as of putchar().
12. Writing a user-friendly program is not an art than a technique.
13. The printf() is used to display the formatted data items on the standard output device normally the monitor.
14. The scanf() in one of the most important and useful functions to display data on monitor.

5.7 Review Questions

1. Define stdin, stdout, and stderr.
2. Differentiate the followings:
   (a) printf() and puts()
   (b) getche() and getch()
   (c) scanf() and gets()
3. How format string is associated with printf()? Discuss the various possible components of a format string in detail.
4. What happens if one uses variables in scanf() without using the address of operator (&)? Discuss.
5. An amount of rupees, say R, is deposited in a bank for Y years, which pays simple interest at the rate of 'rt' annually. Write a C program that prints the amount after Y years.

Notes

6. Write down two functions xgets( ) and xputs( ) which work similar to the standard library functions gets( ) and puts( ).
7. What is the differences between getchar( ), fgetchar( ), getch( ) and getche( )? With the help of suitable example.
8. Write down two functions xgets( ) and xputs( ) which work similar to the standard library functions gets( ) and puts( ).
9. Write down a function getint( ), which would receive a numeric string from the keyboard, convert it to an integer number and return the integer to the calling function. A sample usage of getint( ) is shown below:

```
main( )
{
int a ;
a = getint( ) ;
printf ( "you entered %d", a )
}
```

10. What is the differences between getchar( ), fgetchar( ), getch( ) and getche( )?

Answers: Self Assessment

1.	(a)	2.	(c)	3.	(d)	4.	(b)
5.	escape sequences	6.	in scanf()	7.	Port I/O	8.	Keyboard
9.	asterisk sign (*)	10.	True	11.	True	12.	False
13.	True	14.	False				

5.8 Further Readings



Books

Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication: 2008

B.W. Kernighan and D.M. Ritchie, "The Programming Language", Prentice Hall of India, New Delhi

Byron Gottfried, "Programming With C", Tata McGraw Hill Publishing Company Limited, New Delhi

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 6: Decision-making and Branching

Notes

CONTENTS

Objectives

Introduction

6.1 Decision-making Control Statement

6.1.1 Sequence

6.1.2 Selection

6.1.3 Iteration

6.2 Branching

6.3 Simple if Statement

6.4 The if-else Statement

6.5 The Nested if Statements

6.6 Nested else if Statement

6.7 else if Ladder

6.8 Switch Statement

6.9 Summary

6.10 Keywords

6.11 Self Assessment

6.12 Review Questions

6.13 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain decision making in C
- Explain branching
- Describe if, if-else statement in C
- Explain switch statement

Introduction

To write a realistic program doesn't mean that collection of the statements arranged in a particular sequence. It requires more than that. Just take an analogy of our real life. Life doesn't go straight all the way as:

1. There are some situations when you have to take decisions like whether to purchase this book or not.
2. There are also some situations where you have to perform the particular action again and again like for better understanding read this unit 5 times continuously.

Notes

In the some manner, hardly we write a computer program that may not encounter these situations. Most of the programs require a statement or set of statements to be executed multiple times or not to execute at all, depending on the circumstances.

6.1 Decision-making Control Statement

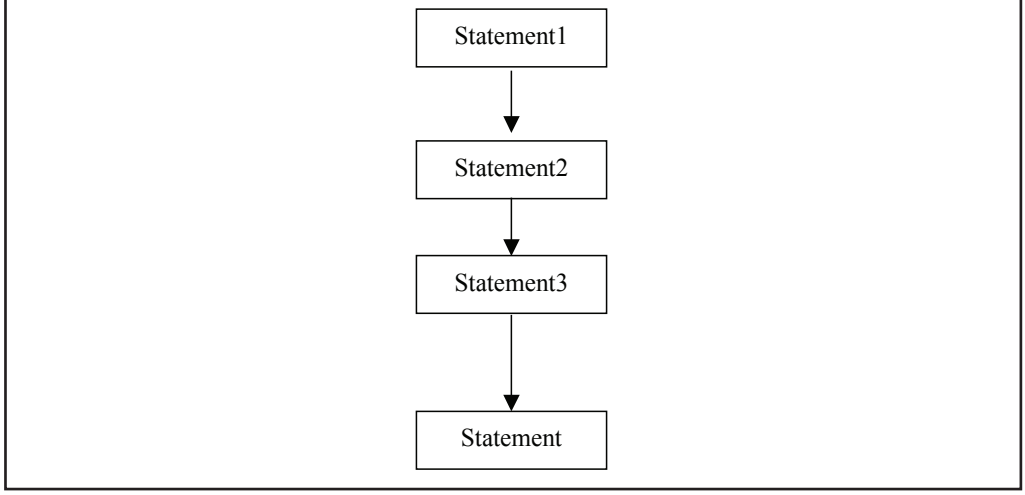
The statement by which we can control the flow of the program execution is called as control flow statement or program control statement. Program statements may be executed sequentially, selectively or iteratively. The C language provides constructs to support sequence, selection and iteration. The combination of one or more of following constructs explain the flow of the program.

1. Sequence
2. Selection
3. Iteration

6.1.1 Sequence

In the sequence construct, as the name implies, statements are executed sequentially i.e. one after the other. In this, neither the statement are repeated nor in the order of execution changed as shown in Figure 6.1.

Figure 6.1: The Sequence Construct



You must have observed in the last unit that the execution of a C program is top down i.e., execution starts with the beginning of the main() function and proceeds, statement by statement, the end of the main () is reached. The following program shows the sequential execution of statement in a C program.



Lab Exercise

Program:

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

Notes

```
printf("\n  First statement");
printf("\n  Second statement");
printf("\n  Third statement");
printf("\n  Second last statement");
printf("\n  Last statement");
}
```

Output:

First Statement

Second Statement

Third Statement

Second Last Statement

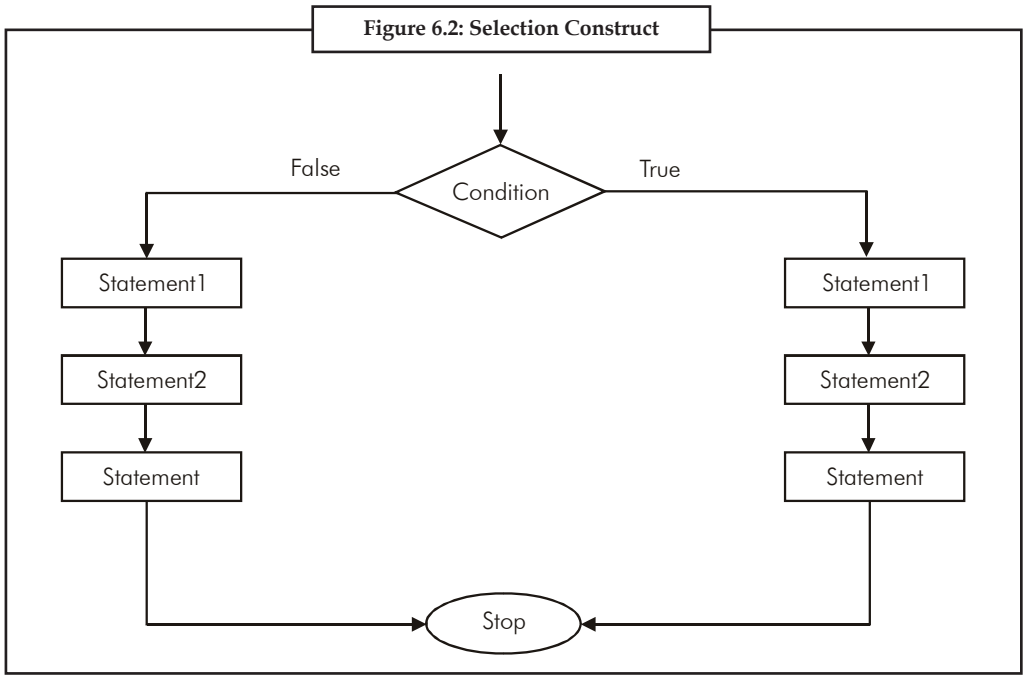
Last Statement

Note Other complex statement may be used in a sequence to demonstrate the concept.

6.1.2 Selection

Sometimes, instead of executing all the statements, only suitable statements are executed depending on the input and the condition involved. In selection construct, the execution of statements depends upon a condition test. If the test evaluates to true, you direct the program to take one course of action, otherwise, you direct the program to do something else. In selection construct, two or more sets of statements are written but only 'one' of these sets is executed as show in the Figure 6.2.

Figure 6.2: Selection Construct



Notes

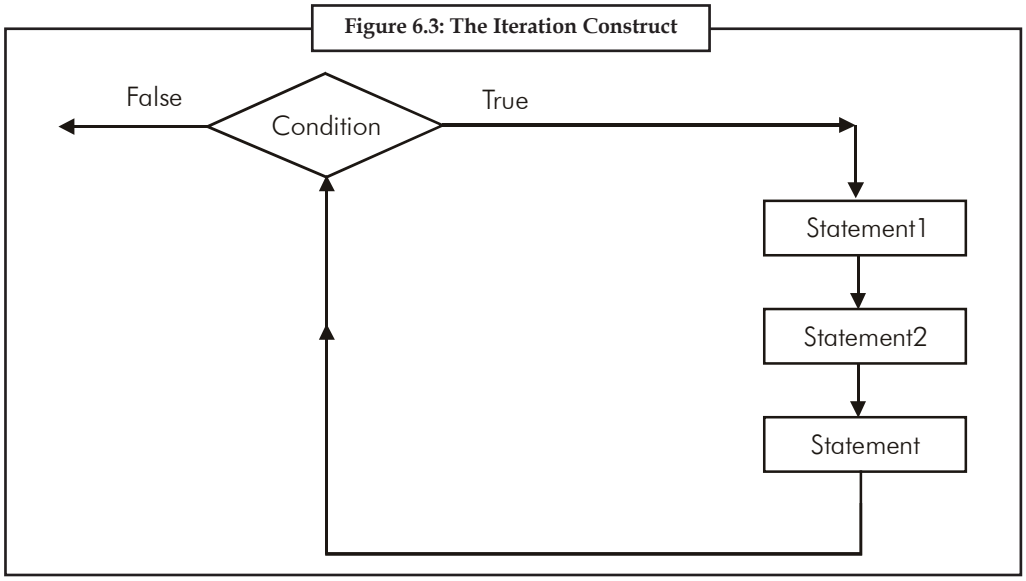
The selection construct can be implemented by means of the if construct. The if construct makes use of relational and logical operators for decision making, as shown in the pseudo code given below:

```
:
if (marks >40) then
        printf "Pass"
else
        printf "Fail"
:
:
```

6.1.3 Iteration

The iteration constructs are an efficient method of handling a series of statements that must be repeated a variable number of times. Sometimes the required number of repetitions is known in advance and sometimes the statements repeats over and over until certain specified conditions on met, as shown in the Figure 6.3.

Figure 6.3: The Iteration Construct



The iteration construct is also called as loop. The statements that are to be executed is called as body of the loop and the condition on which a loop terminates is called as exit condition as demonstrated by the psuedocode given below:

```
:
while (condition is true)
{
statement 1 ;
statement2 ;
:
statement;
}
```

Notes

As mentioned earlier, purely sequenced statements are rarely included in a real program. Every programming language must provide statement to support other constructs. C also provides statements to supports these constructs. Let's discuss these constructs in detail.



Task What would be the output of this program?

```
main( )
{
        int x = 10, y = 20 ;
        if ( x == y ) ;
            printf ( "\n%d %d", x, y ) ;
}
```

6.2 Branching

The branching or selection statement enables you to execute either one section of code or another. The execution of the particular section of code is, actually determined by the evaluation of a test condition. If the condition evaluates to true then the particular section of code will be executed, otherwise, if the condition evaluates to false, another set of code will be executed. In this category C consider the following statements:

1. The simple if statement
2. The if-else statement
3. The nested if statement
4. The else-if statement
5. The switch statement
6. The conditional operator: an alternative statement

6.3 Simple if Statement

In its basic form, the if statement evaluates a test condition (i.e., nothing but an expression) and direct program execution depending on the result of that evaluation. The general form of a simple if statement is as shown below:

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

Where a statement may consist of a single statement, a compound statement or nothing as an empty statement. The expression also referred so as test condition must be enclosed in parentheses, which causes the expression to be evaluated first. If it evaluate to true (i.e., a non-zero value), then the statement associated with it will be executed otherwise ignored and the control will pass to the next statement.



Example: Consider the following statement:

```
:
if (marks > 9)
```

Notes

```
printf ("\n Pass");
:
:
```

The above code fragment will printf "Pass" on the monitor if the value of marks is greater than 9. If the value of marks is not grater than 9, the control simple ignore this statement and will pass to the next statement. The follow program shows the use of simple if as it accepts the marks of a student and printfs his/her result.



Lab Exercise

Program:

```
#include<stdio.h>
void main
{
int marks;
printf( "\n enter your marks");
scanf("%d", marks);
if (marks>9)        /* if construct with test condition */
printf (" \n Pass");        /* statement associated with if */
printf(" Thank you");       //next statement
}
```

output: run 1-> run 2->

```
Enter your marks : 77↵        Enter your marks : 9 ↵
Pass            Thank you
Thank you
```



Note "Pass" has been displayed only if the expression evaluated to true otherwise if it evaluated to false, the control ignores the associated statement and executed the next statement i.e., "Thank you".

As mentioned earlier, an if statement can control the execution of multiple statements, called as compound statement or a block. Where a block is a group of two or more statements enclosed in braces. So if these multiple statements are to be executed than they must be placed with in a pair of braces, as illustrated by the following program.



Lab Exercise

Program:

```
# include<stdio.h>
void main()
```

Notes

```
{
int marks;
printf ("\n  Enter your marks: ");
scanf (" %d ", &marks) ;
if (marks > 39)
{
printf("\n Pass");
printf (" \n Congratulation …");
}
if (marks <40)
{
printf ("\n Fail");
printf ("\n sorry. Good luck next time …");
}
printf ("\n Thank you");
}
```

Output: run 1-> run 2->

Enter your marks: 77↵	Enter your marks : 20↵
Pass	Fail
Congratulations…	Sorry. Good luck next time….
Thank you	Thank You

Note Default an if construct when evaluates to true executes only the first statement associated with it. If multiple statements are not enclosed with in parentheses, results may be unexpected.

Task Write a program if candidate got more than 50% in year exam screen show "Pass" otherwise "Fail".

6.4 The if-else Statement

As you must observed in the previous section that the simple if statement executes a single statement or a group of statement when the given expression evaluates to true (i.e. non-zero value). It does nothing when the expression evaluates to false (i.e. a zero value) and simply moves to next statement of the program.

However, if you want a statement or group of statements to be executed. Only when an expression evaluate to false, you can mention this in else section, as shown below the general format of if else statement.

Notes

```
if (expression)
statement1;
      else
statement 2;
```

If expression evaluates to true, statement 1 is executed. If expression evaluates to false, statement 2 is executed, but never both. Both statement 1 and statement 2, as mentioned earlier, may be single statement, a compound statement, or an empty statement.

Actually, the simple if statement described in previous section is a simplification of its parent statement i.e. if else statement, where the else section in optional. Without it, however, an if-else construct look like a simple if construct.

if-else construct is particularly useful when you have the statements to be executed in both the cases i.e. when the expression evaluates to true or false.



Example: Consider the following statements:

```
:
if (marks > = 40)
printf ("\n Pass");
else
printf ("\n Fail");
:
```

The code segment will display "Pass" on the monitor if the value of marks is greater than or equals to 40. If the marks are less than 40 (obviously the else case), then the statement in the else section will be executed and will printf "Fail" on the monitor. Let's write the Program by using an if-else construct.



Lab Exercise

Program:

```
#include<stdio.h>
void main ()
{
int marks;
printf ("\n Enter your marks:");
scanf ("% d", & marks");
if (marks >=40)
{
     printf ("\n Pass");
     printf ("\n congratulations….");
}
else
{
```

Notes

```
    printf ("\n  Fail");
    printf ("\n Sorry. Good luck next time …");
}
}
```

Output: run1 →

```
Enter your marks: 77→
Pass
Congratulations….
```

run2 →

```
Enter your marks : 30→
Fail
Sorry. Good luck next time…
```



Note else section required their own pair of braces as more than one statement is to be executed when the expression evaluates to false.

Here's another program that demonstrate the use of if else construct. This program will accept a number from the user and will printf whether it is positive or negative.



Lab Exercise

Program:

```
# include<stdio.h>
void main ()
{
int num;
printf ("\n Enter any number:");
scanf ("% d", &num);
if( num < 0)
       printf ("\n Number is a negative number");
else
       printf ("\n Number is a positive number");
}
```

Output: run 1 →

```
Enter any number: 2→
Number in a positive number
Run 2 →
Enter any number: - 2→
Number is a negative number
Run 3->
Enter any number: 0→
Number is a positive number.
```

Notes

Note You can have separate message for zero. But for this the logic of the program requires some changes….

The Figures 6.4 and 6.5, will give you an idea about the working of simple if and if else constructs.

Figure 6.4: Simple if Construct

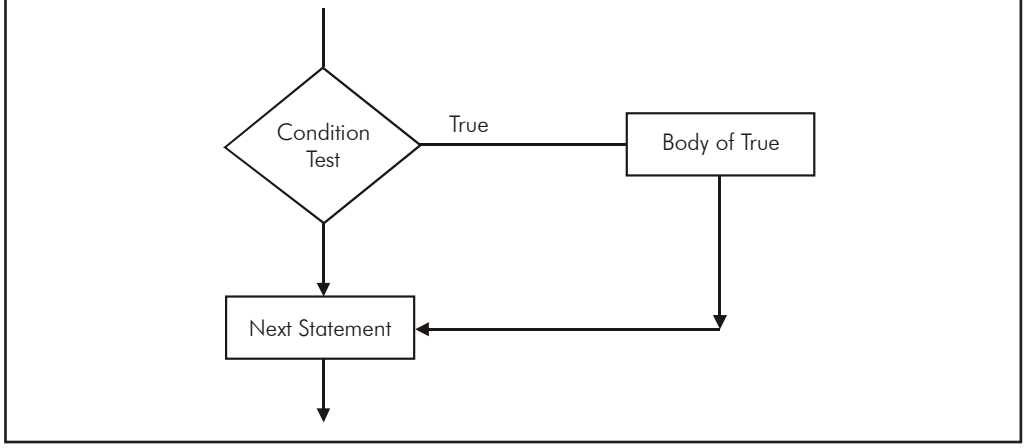
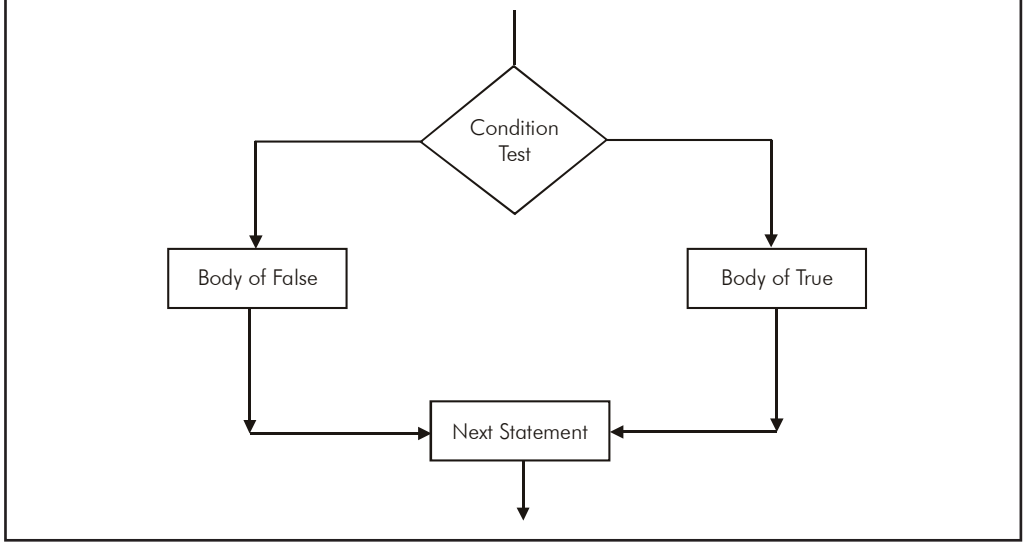


Figure 6.5: if-else Construct



6.5 The Nested if Statements

A simple if or an if-else construct may be placed with in another if or if-else construct. That is simple if or if-else construct may be nested within another if in its if's body or in its else's body.

Notes

The executions of the inner if depends upon its location in the outer construct and upon the value of expression of the outer construct. For instance, consider the following:

```
:
if (expression 1)
{
if  (expression 2)
     statement 1;
else
      statement 2;
}
:
```

In the above segment of code, the inner if executes only if the expression1 evaluates to true. The other possible combination of nested if may take one of the following form:

```
·       if (expression1)
{
if  (expression2)
     statement1;
        }
·       if (expression1)
{
if (expression 2)
  statement1;
else
      statement 2;
}
·       if(expression1)
        {
        statement1;
        }
else
        {
        if(exp2)
              statement 2;
        }
·       if (exp 1)
        {
              statement1;
        }
else
```

Notes

```
        {
        if (exp2)
              statement 2;
        else
              statement3;
        }
        if(exp 1)
        {
        if (exp 2)
              statement1;
        else
              statement 2;
        }
else
        {
        if (exp3)
              statement3;
        else
              statement4;
        }
        if (exp1)
        {
        if (exp2)
              statement1;
        }
else
        {
        if (exp3)
              statement 2;
        }
```

Let's write a couple of programs to explore the various combination of nested if. The following program not only print the request after accepting marks from the students but also print his/her grade.



Lab Exercise

Program

```
# include<stdio.h>
void main ()
```

Notes

```
{
int marks;
printf ("\n Enter your marks:");
scanf ("% d",&marks");
if (marks >=40)
{
            printf ("\n Pass");
            if (marks > = 80)
            printf (" with distinction");
}
else
     printf ("\n Fail");
}
```

Output: run 1 ->

```
Enter your marks : 77↵
Pass
Run  2->
Enter your marks : 88↵
Pass with distinction
Run 3->
Enter your marks : 38↵
Fail
```



Note The execution of the inner if only be there if first expression evaluates to true and the message "with distinction" will printf only if both the expressions evoluates to true.

The next program demonstrates the use of an if else construct in if's body as will as in else's body. We'll write the same programs of marks and result with little variations.



Lab Exercise

Program:

```
#include <stdio.h>
void main ()
{
int marks;
printf ("\n Enter your marks:");
scanf ("% d", &marks");
```

Notes

```
if (marks >=40)
{
            if (marks > = 75)
            printf ("\n very well done");
else
            printf ("\n  well done");
}
else
     {
if (marks > = 20)
            printf ("\n poor");
else
            printf ("\n very poor");
     }
}
```

Output: run 1 - >

```
Enter your marks: 45↵
  Well done
Run  2->
Enter your marks: 85↵
Very well done
Run 3 ->
Enter your marks: 25↵
Poor
Run 4 ->
Enter your marks: 15↵
            Very poor
```



Note Every opening braces must have corresponding closing braces. A mismatch of braces may give unexpected results.



Task Write a program if a manager done business over 10 lakh then display very well done if manager done business more than 5 lakh but less than 10 lakh then display well done, if manager done business more than 3 lakh but less than 5 lakh then display good otherwise display poor performance.

As mentioned earlier, the if constructs also makes use of logical operators for decision making. The following program demonstrate the use of logical operators in if construct. We will write the same program of marks and result for the same purpose.

Notes



Lab Exercise

Program:

```
# include<stdio.h>
void main ()
{
int marks;
printf ("\n Enter your marks:");
scanf ("% d", &marks");
     if (marks > = 75)
             printf ("\n very well done");
if (marks > = 40 && marks < 75)
             printf ("\n well done");
     if (marks > = 20 & & marks < 40)
             printf ("\n poor");
     if (marks < 20)
             printf ("\n very poor");
}
```

Output:

Run 1 →	Run 2 →
Enter your marks : 78↵	Enter your marks : 38↵
Vary well done	Poor



Note The statements that are associated with the expressions consist of logical operator will executes only when both the sub expressions evaluates to true , as in AND relation only true and true holds the value true. (i.e. 1 && 1 = 1).

Sometimes, the nestedness of if constructs increases the complexity and produces an ambiguous situation referred to as dangling else problem. This problem may arise in a nested if statement in the following circumstances:

1. Improper use of braces in nested if statements.
2. Number of if's are more than the number of else clauses.

Consider the following code fragment:

```
:
if (marks > = 40)
     {
             printf ("\n Pass");
             if (marks > = 80)
             printf (" with distinction");
```

Notes

```
else
            printf("\n Fail");
}
```

The indentation of the code shows the intention of the programmer to use else with the outer if. Although the code in syntactically correct. But the improper use of braces will give an unexpected results.

Consider another code fragment given below:

```
if(marks > = 40)
        if(marks > = 80)
        printf(" Pass with distinction");
else
        printf(" Fail");
```

As only the single statement is attracted with if and else, there is no need to put braces. Just reverse of the programmer intention, C will match this else with the preceding if end code will be evaluated as shown below:

```
if(marks > = 40)
        if (marks > = 80)
        printf (" Pass with distinction");
else
        printf (" Fail");
```

To overcome the above mentioned problem of dangling else, pairs of braces are supposed to use at appropriate places. The above-mentioned code fragments could be modified as given below to serve the intended purpose.

```
if (marks > = 40)
{
        printf ("\n Pass");
        if (marks > = 80)
        printf (" Pass with distinction");
}
else
{
        printf ("\n Fail");
}
and
if (marks > = 40)
{
        if (marks > = 80)
        printf (" Pass with distinction");
}
```

Notes

```
else
{
        printf ("\n Fail");
}
```

Respectively.

6.6 Nested else if Statement

Imagine a situation where you have to test number of conditions to get the desired results. These types of particular situations requires nestedness of if-else statements up to a deeper level and it may looks like as:

```
if (expression 1)
statement 1;
            else
        if  (expression 2)
     statement 2;
        else
                  if(expression 3)
                        statement 3;
                else
                     :
                        statement n;
```

The following program demonstrates the use of nested if-else statement up to a deeper level. This program will accept the marks of a student and will display the grade accordingly.



Lab Exercise

```
Program:
# include<stdio.h>
void main ()
{
int marks;
char grade;
printf ("\n Enter your marks:");
scanf ("% d", & marks");
if (marks > = 90)
grade = '0';
else
if (marks > = 80)
grade = 'D';
else
```

Notes

```
if( marks > =75)
grade = 'M';
else
if (marks > = 60)
grade = 'I';
else
if (marks > =50)
grade = 'II';
else
if (marks > 40)
grade = 'III';
else
grade = 'F';
printf( "\n Your grade is : % c", grade),
}
Output: run 1 →
Enter your marks: 77 ↵
Your grade is: M
Run 2 →
Enter your marks: 39 ↵
Your grade is: F
```



Note This whole section of code is actually one statement that is comprised of six hierarchically nested is else constructs, so there is no need to put them in the braces. At any time during the general top to bottom execution of these expressions, if an expression evaluates to true, then the associated statement will be executed and control flow will pass to the statement immediately following the entire nested chain.

Although, the indentation scheme presented in this program is technically correct and there is nothing wrong with the program. However, this style of programming is not recommended as nestedness up to a deepper level is difficult to read. An alternate way to represent these nested if-else constructs is by using else-if construct, whose syntax may looks like as:

```
if(exp 1)
     statement1;
else if (exp2)
     statement 2;
else if (exp3)
     statement 3;
:
```

Notes

```
else if (expn)
     statementn ;
else
     statementx;
statementnext;
```

As mentioned earlier, the expressions are evaluated in top-down approach.

If any expression evaluates to true, then the statement associated with it is executed and control passes to the statementnext. If none of the expression evaluates to true then the statement in the last else (i.e. statementx) will be executed and then control will pass to statementnext. Though this construct involves nothing new, it only utilizes the free-form nature of C to represent constructs. An else-if construct is nothing, but a well indented nested if-else constructs only.

To come up with the better understanding of the concept, let's write the same program once again by using else-if statements.



Lab Exercise

Program:

```
#include<stdio.h>
void main ()
{
int marks;
char grade;
printf("\n Enter your marks.");
scanf ("% d", &marks);
if (marks > = 90)
grade = 'O';
else if (marks > = 80)
grade = 'O';
else if (marks > = 75)
grade = 'M';
else if (marks > = 60)
grade = 'I';
else if (marks > = 50)
grade = 'II';
else if (marks > = 40)
grade = 'III';
else
grade = 'F';
```

Notes

```
printf("\n Your grade is  % c", grade );
}
```

Output:

```
run 1  →                    run 2 →
Enter your marks: 77↵       Enter you marks: 39↵
Your grade is: M            Your grade is: F
```



Note There is no difference in coding of this program and the previous program. That's the output is also same if provided with same input values. The only difference is of indentation.

The above situation could also be handled using simple if with logical operator && (AND). The following program shows the same.



Lab Exercise

Program:

```
#include<stdio.h>
void main ()
{
int marks;
char grade;
printf ("\n enter your marks.");
scanf ("% d", &marks);
if (marks > = 90)   grade = 'O';
if (marks > = 80 & & marks < 90)   grade = 'D';
if (marks > = 75 & & marks < 80)   grade = 'M';
if (marks > = 60 & & marks < 75)   grade = 'I';
if (marks > = 50 & & marks < 60)   grade = 'II';
if (marks > = 40 & & marks < 50)   grade = 'III';
if (marks <  40) grade 'F';
printf("\n your grade is : % c" , grade );
}
```

Output:

```
run 1 →                     run 2 →
Enter your marks: 77↵       Enter you marks: 39↵
Your grade is: M            Your grade is: F
```



Note All ifs are individual statement and there is no involvement of else. So these ifs may appear in any order without affecting the result or the algorithm in any way.

6.7 else if Ladder

Notes

There is another way of putting *ifs* together when multipath decisions are involved. Multipath decision is a chain of *ifs* in which statement associated with each *else* is an *if*.

It takes the following general form:

```
if (condition 1)
        statement1;
else if (condition 2)
        statement 2;
else if (condition 3)
        statement 3;
statement x;
```

The conditions in elseif ladder are evaluated from the top (of the ladder) downwards. As soon as the true condition is found, associated statement is executed and control is transferred to statement x.

```
#include <stdio.h>
main( )
{
     int unit, cust;
     float charges;
     printf ("Enter Customer No. and Units Consumed: \n");
     scanf ("% d % d", &cust, &unit);
     if (unit < = 200)
        charges = 0.5 *unit;
     else if (unit < = 400)
        charges = 100 + 0.65* (unit - 200);
     else if (unit < = 600)
        charges = 230 + 0.8 * (unit - 600);
        printf ("\n \n Customer No: % charges: %0.2f \n" Cust,  Charges);
}
```

Remember that each *else* is associated with the nearest preceding *if* as is illustrated below:

```
if (condition-1)
        if (condition-2)
                statement-1;
else
                statement-2;
```

Here if condition-1 is true then condition-2 is evaluated. If condition-2 is also true then statement-1 is executed. If condition-2 is false then statement-2 is executed.

Notes



Note If condition-1 is false nothing is executed because there is no *else* part associated with condition-1 even though the indentation of the program suggests that.

6.8 Switch Statement

The switch statement is another convenient tool provided by C to handle the situations in which multiple decisions to be made based on an expression that can have multiple values.

The switch is a multiple branch statement that successively tests the value of an expression against a list of case values and when a match is found, the statement associated with the particular case is executed. The general form of a switch-case statement may books like as:

```
switch (expression)
{
        case value1:   statement1;
        case value2:   statement2;
        case value3:   statement3;
        :
        case valuen:   statementn;
        [default:      statement x ;]
}
statement;
```

Where switch is a keyword and the expression is any expression that evaluates to an integer value, may be of type int, or char, or long. The case is a keyword followed by value 1, value 2, value n. where value 1, value 2, .. value n may be an integer or character constant, normally referred to as case labels. And the statement1, statement2, .. statementn may be single statement or set of statements, or may be an empty statement.

The switch statement evaluates the expression first and then compare the return value against the values value1, value2,.. valuen, and then one of the following happens:

1. If a case is found whose value matches with the value of the expression then the statement associated with that case is executed.
2. If no match is found then the statement followed by the keyword default is executed.
3. If no match is found and there is no default label as it is an optional case, then no action takes place and control passes to the statement next which is a statement immediately followed the switch statements closing braces.

Consider the following program, which gives you an example of using the switch statement. This program will receive a number between 1 to 5 and will display it's English counterpart.



Lab Exercise

Program:

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

Notes

```
void main ()
{
int num;
printf  ("\n Enter any number between 1 to 5 : ");
scanf (" % d", &num ) ;
switch (num)
{
case 1       :       printf("\n One");
case 2       :       printf ("\n Two");
case 3       :       printf ("\n Three");
case 4       :       printf ("\n Four");
case 5       :       printf ("\n Five");
default :    printf ("\n Wrong input");
}
printf ("\n Thank You");
}
```

Output: run 1 → run 2 →

Enter any number between 1 to 5: 2↵ Enter any number between 1 to 5: 4↵

Two Four

Three Five

Four Wrong input

Five Thank you

Wrong input

Thank you

Run 3->

Enter any number between 1 to 5 = 9↵

Wrong input

Thank you

Note There is no need to put braces with the individual case labels as they each contains single statement, although a pair is required to group the entire case section.

You must have observed during the execution of the previous program that the control continues to execute all the statements once a case has been matched, irrespective of the fact whether those statements belong to the case that has been matched or not. This flow procedure is known as "Fall Through" execution. Generally the "Fall through" execution approach is not derivable at

Notes

all because at a particular instance one or only a few blocks of statement one required. In order to overcome the problem of "Fall Through", the following C statements can be used:

1. goto
2. if-else
3. break

Though the break statement is discussed in detail in the coming section, but it is worth mentioning over here that the use of break statement causes an exit from the switch statement and the control passes to the statementnext without executing the statements of the other case labels.

In general it is advisable to use the break statement whenever exclusion of case statement is required. However, a break statement does not require to be put in the default case as the control moves to the statementnext automatically after executing the last statement of switch construct.

The following program shows the use of break statement in the switch case construct. This program does a similar job as previous program, but this time output would not be same as break statement in used as and when required.



Lab Exercise

Program:

```
#include<stdio.h>
void main ()
{
int num;
printf("\n enter any number between 1 to 5 : ");
scanf(" % d", &num ) ;
switch (num)
{
case 1       :       printf("\n one");
             break;
case 2       :       printf("\n Two");
             break;
case3        :       printf("\n Three");
             break;
case 4       :       printf("\n Four");
             break;
case 5       :       printf("\n Five");
             break;
default      :       printf("\n wrong input");
}
printf("\n Thank You");
}
```

Notes

Output: run 1 →

```
Enter any number between 1 to 5 : 2↵
Two
Thank you
run 2 →
Enter any number between 1 to 5 : 4↵
Four
Thank you
Enter any number between 1 to 5 : 9↵
Wrong input
Thank you
```



Note In a switch statement, braces are not needed to group the statements with in an individual case as control continuously executes the statements following the selected case until the break statement or the end of the switch statement is reached.

Although "Fall Through" execution is a problem, it can be proved useful at times. For instance, when you want the same statement is to be executed for more than one value of the expression. Just simply omit the break statements and put all possible case one after the other and then specify the appropriate statement. In this way, the general syntax of switch-case may look like as:

```
switch (Exp)
{
:
case value3:
case value4 :
case value5:
:
}
```

The following program not only demonstrate the use of character value in a switch case but also shows how the problem of "Fall through" execution could be proved beneficial.



Lab Exercise

Program:

```
#include<stdio.h>
void main ()
{
char vowel ;
printf("\n Enter any vowel :");
```

Notes

```
scanf(" % c", &vowel ) ;
switch (vowel)
{
case 'A':
case 'a':   printf("\n The A"); break;
case 'E :
case 'e':   printf("\n The E"); break;
case 'I':
case 'i':   printf("\n The I"); break;
case 'O':
case 'o':   printf("\n The O"); break;
case 'U':
case 'u':   printf("\n The U"); break;
dafault :   print("\n Not a vowel");
}
}
```

Output:

```
run 1 →
Enter any vowel  : A↵
The A
Run 2 →
Enter any vowel  : i↵
The I
Run 3 →
Enter any vowel  : v↵
Not a vowel
```



Note As soon as the expression matches any of the case labels, execution will "Fall Though" the following statement until it reaches to the break statement.

Besides the above-mentioned usages, various other possibilities also exist. The following segment of codes not only provide you a few useful tips about the usage of switch but also some important points that should be taken care while using it.

1. The case labels must be on int constant or a char constant as the switch statement can only marks for equality compressions. One cannot have a case label consist of relational or logical expression, the following case label

   ```
   case a < = 2:
   ```

   is not allowed in a switch statement.

2. A switch statement can also be put within another switch statement, called as nested switch statement.

Notes



Example: The following snippet of code is absolutely right in C.

```
switch (outer)
{
case 1 : switch (inner)
{
case 'a'       : -
        -
case 'b'       : -
        :
}
case 2: break ;
:
}
```

3. The case labels must not be a floating point value as given below:

```
switch (value)
{
case 1.1:  _
  _
case 1.2:  _
  _
:
}
```

4. The case labels must not be a string as given below:

```
switch (value)
{
case "string1":  _
                 _
case "string2":  _
                 _
:
}
```

5. The case labels can not be an expression as given below :

```
switch (value)
{
case 1 + 2:  _
             _
case 1 * 3:  _
             _
case x + y:  _
```

Notes

```
                    _
        :
    }
```

6. The default case may appear anywhere in the switch case construct. But may required a break statement if used somewhere else than the at end point of a switch case construct.



Example: The following code fragment is allowed in C.

```
        switch (exp)
        {
        default  : _
                   break;
        case 1 :   _
                   _
        case 2   :  _
                   _
    :
    }
```

7. No two case labels in the same switch can have similar values. However, this is allowed in case of nested switch statement i.e. outer case label and the inner case label may have same values.



Example: The following snippet of code is allowed in C.

```
switch (outer)
{
case 1 : _
         _
case 2  :  switch (inner)
{
        case 1 :       _
                       _
        case 2 :       _
                       _
        }
case 3 :       _
:
}
```

8. Though it is not necessary to put the case labels in a particular order, they may appear according to the user specifications as shown below:

```
switch (value)
{
```

Notes

```
case 4 :  _
          _
case 1 :  _
         _
          default :      _
                    _
case 2 :  _
                    _
case 10 :  _
           _
:
}
```

9. A mixture of character constants and integer constants are allowed as different case labels in a switch statement as shown below :

```
switch (value)
{
case 'A' :      _
                _
case 66 :       _
                _
case 'C' :      _
                _
case 68 :       _
                _
:
}
```

10. A switch expression may be a literal value, a variable, a complex expression, or may be a function that returns an integer value. So the coding switch statements are allowed in C :

```
switch ('a')    /*      character literal */
          {
          }
switch (5)      /*      integer literal */
          {
          }
switch (x)      /*      A variable */
          {
          }
switch (x + y)          /*      A complex expression */
          {
```

Notes

```
        }
switch( f1() ) /* A function that returns an integer value*/
        {
        }
```

If default case doesn't exist in a switch statement, the control simply passes to the statement next, if in case it doesn't find any match between the expression value and the case labels, as show below :

```
        switch (value)
        {
        case 1 :  _
                  _
        case 2 :  _
                  _
        case 3 :  _
                  _
}
statement ;
```

The main usage of using switch-case construct is write menu driven programs as shown in Figure 6.6.

Figure 6.6: A Sample Menu

```
Main Menu
Enter '+' for addition
Enter '_' for subtracts
Enter '*'for multiplication
Enter your choice :
```



Task If the ages of Ram, Shyam and Ajay are input through the keyboard, write a program to determine the youngest of the three.

As mentioned earlier, the switch statement may be though of as an alternative convenient tool to implement the concept of nested if-else statements.

However, there are some situations when the use of switch statement in much more convenient and in others, we are left with no choice but to use if. The table 6.1 summarizes the difference between switch statement and nested if-else statement on the basis of certain points.

Notes

Table 6.1: Distinction between Switch and Nested if-else Statements

Basis	Switch Statement	Nested if-else
Pair of braces	Not required with in single case level	Required in case of compound statement
Expression	More than two values	Only two values evaluates to true or False doesn't exit.
Problem of "Fall Through" execution	May exists	Doesn't exit
Flexibility handle ranges	Less flexible	More flexible as it can
Floating point	Cannot handle	Can handle
Type of comparison	Equality comparison	Cannot evaluate a logical or relational expression
Break statement	Allowed	Now allowed
Level of indentation	Manageable	Bit complex
Express evaluation	Once	Repeatedly unitil it a match finds
ASCII specification	Can have upto 257case statement	A minimum of 15 levels of nastiness, however most compiler allows.



Case Study

While purchasing certain items, a discount of 10% is offered if the quantity purchased is more than 1000. If quantity and price per item are input through the keyboard, write a program to calculate the total expenses.

```
/* Calculation of total expenses */
main( )
{
int qty, dis = 0;
float rate, tot;
printf ( "Enter quantity and rate " );
scanf ( "%d %f", &qty, &rate);
if ( qty > 1000 )
dis = 10;
tot = ( qty * rate ) - ( qty * rate * dis / 100 );
printf ( "Total expenses = ₹ %f", tot );
}
```

Here is some sample interaction with the program.

```
Enter quantity and rate 1200 15.50
Total expenses = ₹ 16740.000000
Enter quantity and rate 200 15.50
```

Contd...

Notes

Total expenses = ₹ 3100.000000

In the first run of the program, the condition evaluates to true, as 1200 (value of qty) is greater than 1000. Therefore, the variable dis, which was earlier set to 0, now gets a new value 10. Using this new value total expenses are calculated and printed.

In the second run the condition evaluates to false, as 200 (the value of qty) isn't greater than 1000. Thus, dis, which is earlier set to 0, remains 0, and hence the expression after the minus sign evaluates to zero, thereby offering no discount.

Is the statement dis = 0 necessary? The answer is yes, since in C, a variable if not specifically initialized contains some unpredictable value (garbage value).

Questions

1. ***Write a program:*** The current year and the year in which the employee joined the organization are entered through the keyboard. If the number of years for which the employee has served the organization is greater than 3 then a bonus of ₹ 2500/- is given to the employee. If the years of service are not greater than 3, then the program should do nothing.
2. Write a program to determine whether a number is prime or not. A prime number is one, which is divisible only by 1 or itself.

6.9 Summary

- Most of the programs require a statement or set of statements to be executed multiple times or not to execute at all, depending on the circumstances.
- The statement by which we can control the flow of the program execution is called as control flow statement or program control statement.
- In the sequence construct, as the name implies, statements are executed sequentially i.e. one after the other.
- In selection construct, the execution of statements depends upon a condition test.
- The iteration constructs are an efficient method of handling a series of statements that must be repeated a variable number of times.
- If multiple statements are to be executed than they must be placed with in a pair of braces.
- A simple if or if else construct may be placed with in another if or if-else construct.
- The switch statement is another convenient tool provided by C to handle the situations in which multiple decisions to be made based on an expression that can have multiple values.

6.10 Keywords

Break statement: A statement that terminates the block of statements currently under execution.

Conditional operator: An operator that takes three arguments in which the first one is conditional statement. One of the two next statements is executed depending on the truth-value of the conditional statement just like if-else statement.

Conditional statement: A statement that evaluates to either true or false.

Notes

Continue statement: The statement that ignores execution of further statements and forces the loop to evaluate the loop condition once again.

Default statement: An optional statement in a switch that is executed if none of the conditions evaluates to true.

Switch statement: A multi-selection statement that branches to that statement whose specified condition evaluates to true.

6.11 Self Assessment

Choose the appropriate answers:

1. The C language provides constructs to support
   (a) Sequence
   (b) Selection
   (c) Iteration
   (d) All of the above
2. The branching or selection statement enables you to execute either
   (a) One section of code or another.
   (b) Many section of code or another.
   (c) Only two section of code or another.
   (d) None
3. This simple statement is an example of

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

   (a) The nested if statement
   (b) The else-if statement
   (c) The switch statement
   (d) Simple If statement
4. If there are more than one expression then
   (a) Use only simple if statement
   (b) Use else-if statement
   (c) Both
   (d) Only (b)

Fill in the blanks:

5. The ..... is used to handle the situations in which multiple decisions to be made based on an expression that can have multiple values.
6. A switch expression may be a literal value, a variable, a complex expression, or may be a function that returns an ..... value.
7. Floating point cannot handle by .....
8. The ..... enables you to execute either one section of code or another.

Notes

State whether the following statements are true or false:

9. The iteration construct is also called as loop.
10. A switch statement can also be put within another switch statement.
11. If-else construct is particularly useful when you have only one choice.

6.12 Review Questions

1. Write a program using if-else statement.
2. Explain nested-if statement with example.
3. What do you mean by switch statement? How it used
4. A five-digit number is entered through the keyboard. Write a program to obtain the reversed number and to determine whether the original and reversed numbers are equal or not.
5. Write a program to check whether a triangle is valid or not, when the three angles of the triangle are entered through the keyboard. A triangle is valid if the sum of all the three angles is equal to 180 degrees.
6. Given the length and breadth of a rectangle, write a program to find whether the area of the rectangle is greater than its perimeter. For example, the area of the rectangle with length = 5 and breadth = 4 is greater than its perimeter.
7. What is the use of if-else statement?
8. Define selection in c programming.
9. Write a program in C to enter five integer values as age of five boys and calculate the average age of all the boys.
10. Write a program to calculate the area of a square. All values enter with the help of keyboard.

Answers: Self Assessment

1.	(d)	2.	(a)	3.	(d)	4.	(d)
5.	Switch statement	6.	integer	7.	switch statement	8.	branching
9.	True	10.	True	11.	False		

6.13 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication: 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Notes

Byron Gottfried, "*Programming With C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, "*Problem Solving with Computers*", Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., "*How to Solve it by Computer*", Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 7: Decision-making and Looping

CONTENTS

Objectives

Introduction

7.1 Looping

7.2 for Loop

7.3 While Loop

7.4 do-while Loop

7.5 Jump and Break Statement

7.6 goto Statement

7.7 Summary

7.8 Keywords

7.9 Self Assessment

7.10 Review Questions

7.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain looping concept in C
- Describe do-while loop
- Describe goto statement

Introduction

The C language includes a variety of program control statements that let you control the order of program execution. This unit discusses various iteration based program control statements and how these can be implemented in a program. This unit also discusses some jump statements of C which are break, and continue.

7.1 Looping

Iteration statements are also know as loops or looping statements because the program execution typically loops through the statement more than once. In this category, C provide the following statement or you call loops.

1. for loop
2. while loop
3. do-while loop

Notes

Looping must not continue indefinitely as an analogy to real life you would not like to crack the same joke again and again, so a mechanism is required to break out the loop and to allow the executives of the next set of statements.

Therefore, a general structure has been devised for the implementation of a loop statement. Which can be more understood by understanding the various elements/ parts/components of a loop that controls the number of repetitions as given below:

1. ***Initial Expression(s):*** Initial expression(s) is usually an assignment expression(s) which initializes the control variable(s) of a loop, as they must be initialized before entering in a loop. The initial expression(s) is executed only once, in the beginning of the loop. But if this expression(s) occurs in the loop body, control variable(s) would be reassigned to initial values with every loop pass, and the condition expression would never fail.
2. ***Condition Expression:*** Conditional expression is typically a relational expression that is set up to terminate the execution of a loop. If the condition expression evaluates, to true i.e. 1, the loop body gets executed, otherwise the loop is terminated.

   A condition expression may be evaluated before entering in to a loop or before exiting from the loop called as entry-controlled loop and exit controlled loop respectively. In C, the for loop and while loop are entry-controlled loops where as do while loop is exit-controlled loop.
3. ***Update Expression(s):*** The update expression(s) is essentially an increment expression or decrement expression that changes the value(s) of loop variable(s), so that they could come to the boundary values.

   The update expression(s) normally execute at the end of the loop body. It may appear in the body of loop as it is updating expressions that assign the variable a new updated value every time the loop passes.
4. ***The Loop Body:*** The loop body consists of statement(s) that is supposed to be executed again and again as long as the condition expression evaluator to true i.e. 1. In an entry-controlled loop, the condition expression evaluated first and if it evaluates to true, the loop-body is executed and if it evaluate to false, the loop-body is terminated. Whereas, in exit controlled loop, the loop body executed first and then the condition expression are evaluated. It is evaluate to false i.e. o, the loop is terminated, otherwise repeated.

The above mentioned components are the essential component of a statement to be called as a loop statement. Messing any of them may change the basic meaning of a perfect loop. The for, while and do-while statements of C, comprises of all these essential components, hence referred to as loop statements.

7.2 for Loop

The for loop in C is the simplest, fixed and entry controlled loop. It is simplest as the structure of for loop is divided into two segments i.e. control statement and the body of the loop. All its loop control elements are placed together in the control statement where as body of the loop consists of statements to be executed repeatedly.

It is fixed as number of repetitions is known is advance and can be useful in a situation when you want to do something a fixed number of times.

It is an entry controlled loop as the control statement placed before the loop body i.e. condition expression will be evaluated first. The general form of the far loop is:

```
for(initial expression(s) ; condition expression ; update expression(s))

loop-body;
```

Notes

Example: Consider the following statement:

```
for ( i =1 ; i<= 10;  ++i)

printf(" \n Hello  World!");
```

where i is an integer variable declared already.

```
i=1;  is an initial expression.

i < = 10;  is a conditional expression.

++i;  is an update expression.
```

And the statement

```
printf("\n Hello World!") ;
```

is the body of the loop.

When the above statement is encountered during program execution, the following events occur:

1. Initial expression is evaluated first and i will be assigned an initial value 1 i.e. i =1.
2. Then the condition expression is evaluated i.e. i < =10 and the result will be true as 1 < = 10 is true.
3. Since the condition expression is true, the statement in the loop body is executed i.e. printf("\n Hello World!"); which prints the message Hello World! on the screen.
4. After the execution of the loop body, the update expression i.e. ++i is executed which increment the value of i by 1. In this way after the first execution of the loop the value of i becomes 2 as initially it was 1.
5. After the execution of the update expression the condition expression is again evaluated. If it evaluates to true the sequence is repeated from step no. 3, otherwise the loop terminates.

Note After a certain repetition the condition expression evaluates to false, as the value of I will be greater than 10, then loop will be terminated and the output of the code may appear as:

Hello World!

Hello World!

:

:

Hello World! (10 Times)

Also note that the loop body never executes if condition expression is evaluated to false in its first execution. Figure 7.1 shows the operation of a for loop.

Notes

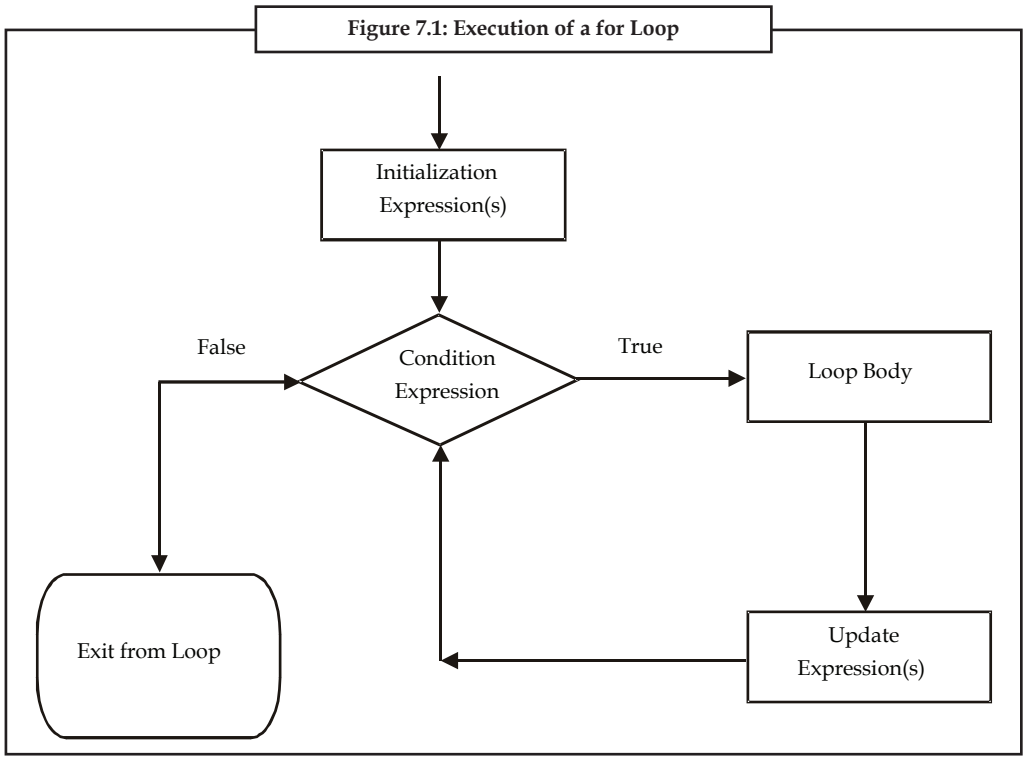


Figure 7.1: Execution of a for Loop

Let's write a complete program that demonstrates the use of for loop. This program will printf find to natural number run bus int for loop.



Lab Exercise

Program:

```
#include<stdio.h>
void main()
{
int  i ;            /* declaring variable */
for ( i=-1; i<=10; ++i)
printf("\n % d", i);
}
```

Output:

```
1
2
3
4
5
6
```

Notes

```
7
8
9
10
```



Note The control statement (for (i = 1, i<=10; ++i)) should not end with the semicolon, otherwise is will be treated as an empty loop discussed later in this unit.

Here's another program another which gives you an idea about the for loop capabilities.

This program will print the sum of first 10 natural numbers.



Lab Exercise

Program:

```
#include<stdio.h>
void main()
{
int i,sum= 0; // the variable sum and I is declared of type int
// sum is initialized to 0
for ( i = 1 ; i<=10 ;++ i )
{
sum = sum + i ;
}
printf("\n The sum of first 10 natural numbers is %d.",sum);
}
```

Output:

The sum of first 10 natural numbers is 55.

Note that printf() statement is out of the for loop body. If it was in loop body the output would have been like this;

The sum of first 10 natural numbers is 1.

The sum of first 10 natural numbers is 2.

The sum of first 10 natural numbers is 6.

The sum of first 10 natural numbers is 55.

What should be the contents of a loop body, is totally depends upon the logic.

Let's write the previous program of printing the largest value among four numbers once again by using simple if with for loop. This approach is more desirable as the algorithm is quite simple, resultant a less complex code.

Notes



Lab Exercise

```
Program:
#include<stdio.h>
void main()
{
int i, num, max = 0 ;
printf("\n Enter any four numbers \n");
for ( i = 0 ; i < =3 ; i ++ )
{
scanf("% d", &num);
if (num > max)
max = num ;
}
printf("\n highest = %d ", max);
}
Output:
Enter any four numbers
5
4
3
8
highest = 8
```



Note The variable i is used as a counter variable which counts the repetition of the loop 4 times as initially i = 0 to i < =3. In order to receives four number we have to repeat the loop 4 times and the counter variable may have initially assigned with any value. And so the condition expression can also be set. For instance, the control statement may be written as:

```
for ( i =10; i < = 13 ; i++)
```

Let's explore the different possibilities of using for loop as C offer several variations that increase the flexibility and applicability of for loop. The following segment of code not only provides you a better understanding of the concept but also give you some guidelines for using for loop.

1. In the previous examples, the for loop is used to count up i.e. incrementing a counter from one value to another. You also can use it to countdown i.e. decrementing the counter variable. For example:

```
for ( i = 10 ; i> = 1; - - i)
```

Notes

2. You can also update the counter by a value other than 1, say by 5 as given below:

```
for ( i = 5; i< = 50; i = i + 5)
```

3. The for loop is quite flexible you can skip any of these (initialization expression, conditional expression or update expression) or all of these from the control statement. For example, you can skip the initialization expression if the particular variable has been initialized previously in the program, But you must use the semi colon separator as shown below:

```
int i = 1;
for ( ; i < = 10 ; ++i)
```

4. Similarly you can also omit the updation expression as shown below:

```
int i = 1;
for ( ; i < = 10; )
{
++i;
}
consider another example as
for ( i =1, i ! = 99;  )
scanf(" % d", & num);
```

5. According to the above statement, the loop will execute until the user enter 99. This approach of coding can be apply to user the for loop as a variable loop rather than fixed.

   As you know, the initialization expression executed once when the for statement in first reached. After that it doesn't required for the rest of process. If the particular variable has been declared and initialized previously in the program, then this place can be used for any valid C expression. For example, the code segment

```
i=1;
for (printf( "\n Output ); i < = 10; ++ i)
printf("\n % d", i);
Will produce the same output i.e. printing of first 10 natural numbers as:
Output
  1
  2
  :
  10
```

6. Another example of missing updation expression could be as given below. These statements will also printf first ten natural number on the monitor.

```
int     i = 1 ;
for (i = 1; i < = 10;  )
printf("\n % d", i + +);
```

7. An infinite for loop can be created by skipping the conditional expression as show below:

```
for (i = 100;       ; - - i)
printf(" \n  Infinite loop");
```

Notes

8. An infinite for loop can also be written as given below:

```
for (   ;    ;  )
puts (" Infinite loop");
```

9. Following is also the example of infinite loop as it accepts the numbers continuously from the user.

```
for ( scanf( "%d", & i) ;    ; i ++)
```

10. First ten natural numbers could also be print as the monitor by using the following code:

```
for ( i = 1; i++ < 10;   )
printf("\n %d", i);
```

11. If a for loop doesn't contain even a single statement in the loop body is called an empty loop. For example, the following loop

```
for (i = 0; i < 10000 ; i ++);
```

is an empty loop and can be used as time delay loop, which are often used in programs.

12. By even writing an empty loop still you can printf first ten natural number on the monitor by mentioning all the work to be done in the for statement itself, as given below:

```
for (i = 1; i < = 10; printf("\n % d", i + + ));
```

13. A for loop may contain multiple expressions in initialization section and/or updation section, must be separated by commas. The following code demonstrate the use of multiple expression in initialization sections.

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

The output of the above code is to similar as printf first 10 natural numbers.

14. The following for statement illustrates the use of multiple expressions in updation section:

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

The output of the above code is as follow:

```
i = 1   j = 10
i = 2   j = 9
i = 3   j = 8
i = 4   j = 7
i = 5   j = 6
```

15. A conditional expression can not have multiple expression like initialization and updation expression, but it may contain several conditions linked together using logical operators. For example, consider the following for loop:

```
int i = 1;
int j = 10;
for (  ; (i < = j) & & (j > = i) ;  )
printf("\n i = % d     j = %d,   i++, j - -);
```

Notes

The execution of the loop body in depends upon the individual values of both the sub expressions as they both are true then only printf() statement will be executed. In case any sub expression evaluates to false the loop will be terminated as both the sub expressions are link together using && (AND) relationship.

16. An infinite loop can also be configured by missing the updation expression as shown below:

```
for (i = 1 ; i < = 10;   )
printf("\n % d",i),
```

17. In all the examples given above, the control variables of the loop has been assigned with integer values. However, it is not necessary as control variable can even be a float. Following is the example of incrementing a counter using floating point value:

```
for (i = 0.0; i < 0.9; i = i + 0.1)
printf(" % .2 f" , i);
```

Before going for the next topic, let's write one more complete program using for loop this program will accept 10 numbers from the user and will print the total number of positive, negative and zeros input by the user.



Lab Exercise

Program:

```
#include<stdio.h>
void main ()
{
     int num, p = n = z = 0, i ;
     printf("\n Enter any ten numbers: \n") ;
     for (i = 1 ; i <= 10 ; ++ i)
     {
     scanf(" % d", &num)
     if (num > 0 )
            + + p ;
     else if (num < 0)
            + + n;
     else
            + + z ;
}
printf(" \n Total no. of positives = % d", p);
printf(" \n Total no. of negatives = % d", n);
printf("\n Total no of zeros = % d", z);
}
Output:
```

Notes

```
Enter any ten numbers
- 1    -2     1     5     6     0     -4     -1     2     3
```

Total no. of positives = 5

Total no. of negative = 4

Total no. of zero = 1



Note To avoid unexpected results, if a variable is used as a container then it should be initialized properly.



Task Two numbers are entered through the keyboard. Write a program to find the value of one number raised to the power of another.

7.3 While Loop

The second type of loop, the while loop is an entry controlled loop as it tests the conditions first and if the condition is true, then only the control will enter into the loop body.

When each iteration of the loop is finished, the control returns to the while statement which perform the condition test again as so on. But if the condition in false the first time, no iteration of the loop executes and control passes to the statement next to loop statement. In this way, it is a sort of variable loop as we do not know the exact number of iteration. The statements repeats over and over until certain specified conditions are met.

The while loop has the following form:

```
while (condition expression)
loop body;
```

Where the loop-body may contains a single statement, a compound statement or an empty statement. The while loop iterates the loop body as long as the specified condition expression evaluates to true.

The while loop doesn't explicitly contains the initialization expression and update expressions of the loop. These two expressions are normally provided by the programmers as the initialization expression(s) should be placed before the loop begins and updation expression(s) should be inside the loop body. By using all these expressions the general farm of while loop may looks like as:

```
:
initialization expression(s);
while (conditional expression)
{
:
:                                          Loop Body
updation expression;
}
```

Notes

Example: Consider the following segment of code:

```
i = 1 ;
while ( i < = 10)
{
printf("\n Hello  World!");
+ + i;
}
where i is an integer variable declared already
i = 1;              is an initial expression
i < = 10;           is a conditional expression
++i ;       is an update expression.
```

and the statements between the { and } forms the body of the loop. But the braces can be discarded, if there is only one statement in the loop body.

When the program execution readers a while statement, the following events occur:

1. First of all the conditional expression is evaluated i.e. i < 10.
2. The conditional expression is evaluated to true as i was 1 initially and 1 < 10 is true. But if it evaluate to false, the loop will be terminated and the control moves to the first statement following loop body.
3. Since the condition expression is true, the loop body will be executed i.e. the printf statement and the updation expression.
4. With the closing braces (}), it is assumed that the loop is finished and the control moves back to the while statement, which repeats the test again and proceeds accordingly.

Note Next time due to the updation expression the value of i will be 2 as previously it was 1. With the every next execution the value of I will be increased. After a certain repetitions the conditional expression evaluates to false as the value of i will be greater than 10. Then this loop will be terminated. The output as this code may appear as:

Hello World!

Hello World!

.

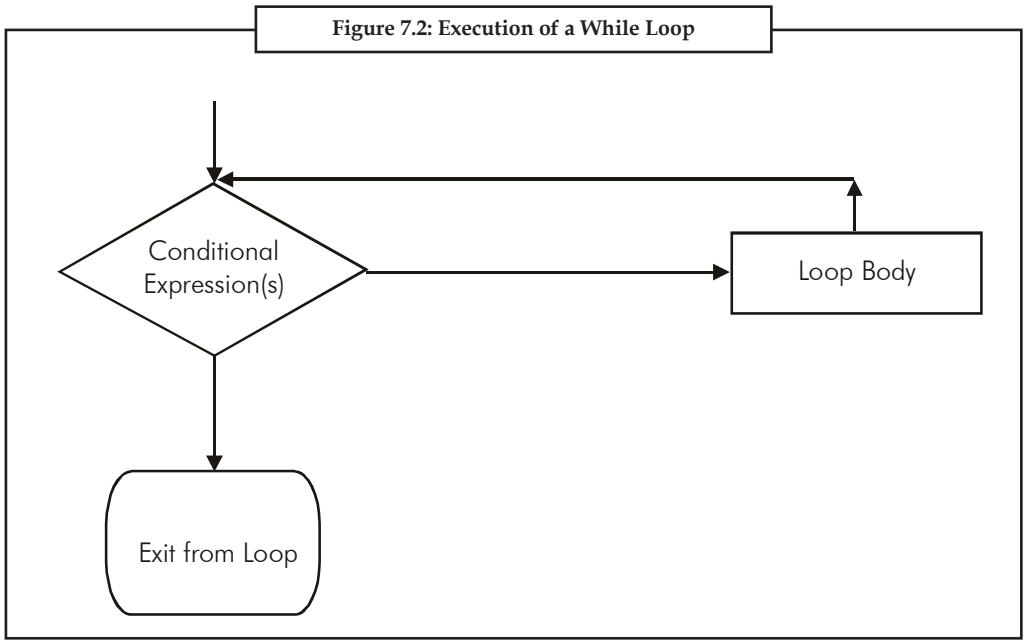
.

Hello World! (10 times)

The Figure 7.2 illustrates the working of a while loop.

Notes

Figure 7.2: Execution of a While Loop



Lets write a complete program that demonstrate the use of a while loop.

This program will print first 10 natural numbers using while loop.

Lab Exercise

```
Program:
#include<stdio.h>
void main()
{
int i = 1;
while ( i < = 10)
{
printf("\n % d" , i);
++i ;
}
}
Output:
1
2
3
.
.
10
```

Notes



Note The while statement (while (i <= 10)) should not end with a semicolon, otherwise it will be treated as an empty loop.

Here's another program which gives you an idea about the while loop capabilities. This program will print the sum of as many numbers as user wants.



Lab Exercise

```
Program:
#include<stdio.h>
void main ()
{
int num, sum = 0 ;
char reply = 'y' ;
while ( reply =  = 'y')
{
printf("\n Enter the number to add:");
scanf( " % d" ,&num) ;
sum = sum + num;
printf("\n Continue (y/n):");
scanf( " % c" ,&reply) ;
}
printf("\n The sum of all the numbers is = %d",sum);
}
Output:
Enter number to add : 4 ↵
Continue (y/n) : y ↵
Enter the number to add : 8 ↵
Continue (y/n): y ↵
Enter the number to add: 8 ↵
Continue (y/n) : n ↵
The sum of all the numbers is = 20
```



Note Reply is initialized with ' y' to get in side the loop body, the very first time. The next iterations of the computer body will depend upon the user's response. Input must be provided in a required way, otherwise unexpected results may appear.

Let's rewrite the same program of printing the largest value, with little variation by using while loop. This program will print the highest number from all the number input by user.

Notes



Lab Exercise

```
Program:
#include<stdio.h>
void main()
{
int number, max = 0;
char reply = 'y',
while (reply = = 'y')
{
printf("\n Enter any positive number :" ) ;
scanf(" % d", &num);
if (number > max)       max=num;
printf("\n Continues (y/n) :" );
scanf ("%c", &reply);
}
printf("\n Highest among all the input numbers is = % d", max);
}
Output:
Run 1:
Enter any positive number : 5 ↵
Continue (y/n) :y ↵
Enter any positive the number : 9 ↵
Continue (y/n):y↵
Enter any positive number : 2 ↵
Continue (y/n) : n ↵
Highest among all the input numbers is = 9
Run 2:
Enter any positive number : 5 ↵
Continue (y/n) : n ↵
Highest among all the input numbers is = ↵
```

C also allowed a while loop to be written in variations. As variations in while loop no only increases the flexibility to use it but also increase your logic sense. The following statement of code explores some of the variations of while loop.

1. The while loop doesn't care about the initialization expression and the updation expression as it only has to deal with conditional expression. So the particular variables may be initialized or updated according to the programmer choice.

Notes



Example: The following statements will print first ten natural numbers in reverse order.

```
i=10;
while ( i>=1)
{
printf ("%d \n", i);
- -i;
}
take another example, the following  statement will print the table of 5.
i = 5;
while ( i <= 5)
{
printf("\n % d", i );
i += 5;
          }
```

2 The while loop can also be written without the initialization expression and the updation expression. However, this is the real use of while loop.



Example:

```
while ( ch = getchar () ! = EOF )
putchar (ch);
```

This loop reads a character from the keyboard and displays it on the monitor, as long as the character in not a EOF (i.e. ^z).

3. Missing an updation expression sometimes may cause while loop to be executed infinitely.



Example:

```
i=1;
while(i<=10)
printf("\n%d",i);
```

The following could also be the example of infinite while loop:

```
i=1;
i=2;
while (i++<=j++)
----
```

4. First ten natural number could also be printf on the monitor by using the following:

```
i=1;
while (i<=10)
printf("\n % d", i++);
```

5. In the same manner, the sum of first 10 natural numbers can also be calculated as follows:

Notes

```
i=1;
sum=0;
while (i<=10)
sum +=  i++;
```

6. An empty loop can also be configured using while statement and could used as a time delay loop for example.

```
i=1;
while (i++< 10000);
```

7. The conditional expression in while loop may contain several conditions linked together using logical operators.



Example: Consider the following while loop:

```
i=1;
j=10;
while (i<=j && j>=i)
printf(" \n i= % d j= % d", i++, j - -);
```

8. First ten natural number could also be print using the following while statement:

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

9. A while loop can also be implemented using floating point values.



Example: Consider the following code:

```
i=0.0 ;
while ( i< =0.9)
{
printf("\n % .2f", i);
i = i + 0.1 ;
}
```

The above code will print the numbers from 0.0 to 0.9 on the monitor.

Before going for the next topic, here's present another useful program using while loop. This program reads the contents of a paragraph input by the user and printfs the total number of characters, words, and words used in the paragraph. Assuming enter key in the end of the paragraph.

Notes



Lab Exercise

Program:

```
#include<stdio.h>
{
char ch;
int TC = TW = TV = 0; /*TC = Total characters, w=words, v=vowels*/ printf("\n
Start typing the paragraph in small case  and terminate by ENTER key \n");
while ((ch=getche() ! = '\n')
{
if ((ch ==  ' ')) || (ch ==  '\t '))
TW=TW+1;
if((ch== 'a')||(ch== 'e')||(ch == 'i')||(ch =='0')|| (ch=='u'))
TV=TV+1;
TC=TC+1;
}
TW=TW+1;   /* For the last word */
printf("\n Total characters = % d", TC);
printf("\n Total words = % d", TW);
printf("\n Total vovels = %d", TV);
}
```

Output:

```
Start typing the paragraph in small case and terminate by ENTER key
I love my India
Total character = 15
Total words = 4
Total vowels = 6
```



Note If statement containing comparisons for vovels may be changed as follows, to enable the program for receiving capital letters:

if((ch == 'a') || (ch == 'A') || (ch == 'e') || …….);)

7.4 do-while Loop

C's third loop statement is the do-while loop, is an exit controlled loop i.e. it tests the conditions after having executed the statement with in loop body. This means unlike the for and while loops, a do while loop always executes at least once. The statement of the do-while loop is as follows:

```
do
{
```

Notes

```
loop-boody ;
}while (conditional expression) ;
```

The braces { } can be discarded when the loop-boody contains a single statement. The do-while loop iterates the loop body as long as the specified condition is true while testing the condition at the end of the loop each time, rather than at the beginning, as is done by the for and the while loop.

Like while loop, do-while loop also doesn't contain the initialization and updation expression as part of loop statement. However, these expressions can be associated with do-while loop by the programmer according to required logic. Then the new form of do-while loop may looks like as:

```
Initialization expression(s);
do
{
Loop body;
Updating expression;
}while ( conditional expression(s) );
```



Example: Consider the following segment of code :

```
i=1;
do
{
 printf("\n Hello World!");
++i;
} while ( i<=10);
```

where i is an integer variable declared already

i=1; is an initial expression.

i < = 10; is a conditional expression.

++i; is an update expression.

When the program control reaches at a 'do while' loop, the following events occur:

1. The loop body will be executed i.e. the print statement and the updation statement.
2. The conditional expression will be evaluated i.e. i <=10.
3. The conditional expression will evaluates to true as the value as i is 2 this time (initial i=1).
4. Since the condition expression is true, the control will move back to execute the loop-body once again.



Note After a certain repetition the condition expression will evaluates to false as the value as I will be greater than 10, the loop will be terminated.

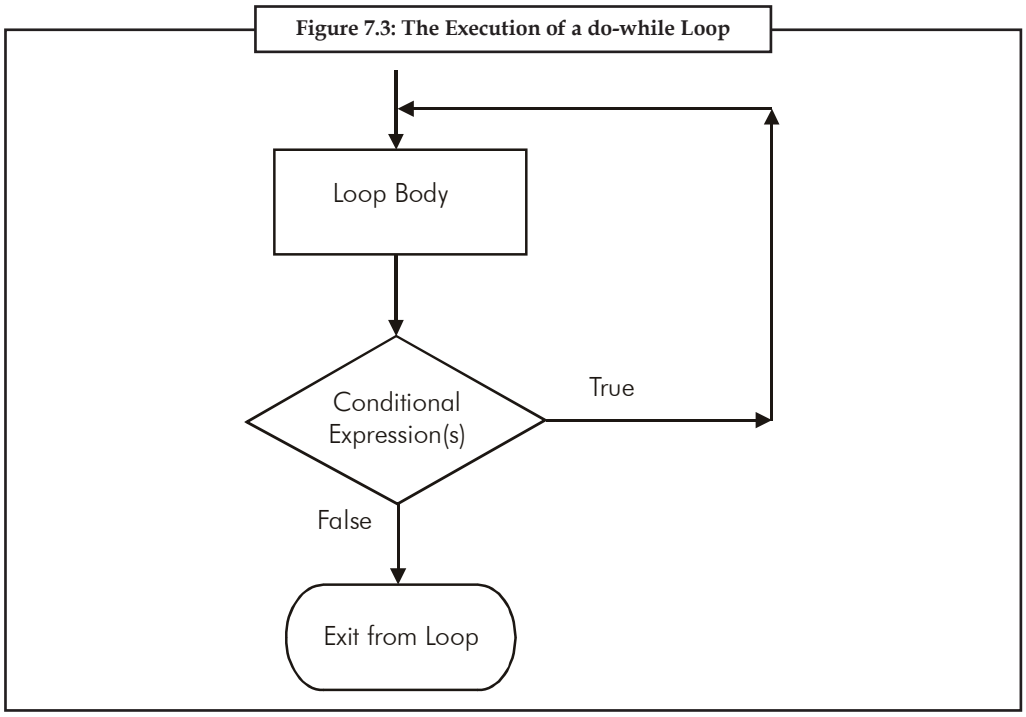
Notes

The output of the above code may looks likes as:

```
Hello World !
Hello World !
:
Hello World !  (10 times)
```

The Figure 7.3 demonstrates the working of a do-while loop.

Figure 7.3: The Execution of a do-while Loop



The do-while loop is well suited for the problems where number of repetitions is not known in advance. But this is sure that the loop will be executed at least once. The following segment of codes will give a clear picture of the concept:

```
for ( i = 11; i <=10; i++)
        printf("\n%d,"i);
        printf ("\n Thank you");
        Output:
        Thank you
i=11;
        while (i<=10)
        {
        printf("\n%d", i);
        ++i;
        }
        printf("\n Thank you");
        Output:
        Thank you
```

Notes

```
i=11;
        do
        {
        printf("\n% d", i);
        ++i;
        } while (i<=10);
        printf("\n Thank you");
        Output:
        11
        Thank you
```

From the above segment of code, this is clear that a do-while loop is always executed at least once, regardless of the outcome of the condition. Because the condition expression in evaluated at the end, instead of the beginning of the loop.

Let's write a complete program that demonstrate the use of a do-while loop. This program will print the first 10 natural number using do while loop.



Lab Exercise

Program:

```
#include<stdio.h>
void main()
{
int i = 1;
do
{
printf ("\n % d", i) ;
++i ;
}while (i<=10);
}
```

Output:

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

Notes



Note The statement while (i<=10) must end with a semicolon. Otherwise system may flag the compile time error.

While writing the program to print the sum of as many numbers as user wants, using while loop, you must have observed that in order to evaluates the condition expression as true the very first time, the control variable has been initializes before the loop accordingly. (i.e. reply = 'y').

Let's rewrite the program using do-while loop. This program won't require the particular variable to be initialize before the loop as do-while loop executes once, surely.



Lab Exercise

```
Program:
#include<stdio.h>
void main()
{
int num, sum = 0;
char reply ;
do
{
printf("\n enter the number to add:");
scanf ("% d", & num );
sum = sum + num ;
printfi("\n  Continue (y/n): " );
scanf("% c", &reply);
} while(reply == 'Y'|| reply == 'y') ;
printf("\n The sum of all the numbers is = % d", sum);
}
Output:
Enter number to add : 1↵
Continue (y/n) : y↵
Enter the number to add :2↵
Continue (y/n) : y↵
Enter the number to add : 3↵
Continue (y/n) : n↵
The sum of all the numbers is = 6
```



Note This program will execute once even if the variable reply initialized with n (i.e. reply = 'n'). And the next iteration of the loop body is totally depends upon the user input.

Let's write another interesting program using do-while loop. This program will printf the alphabet A to Z along with their ASCII codes.

Notes



Lab Exercise

Program:

```
#include<stdio.h>
void main ()
{
char ch = 'A',
do
{
printf("\n The ASCII code of % c is % d. ", ch, ch);
} while ( ++ ch < = 90 ) ;
}
```

Output:

```
The ASCII code of A is 65.
The ASCII code of B is 66.
:
The ASCII code of Z is 90.
```



Note The char variable ch may be initialized as ch = 65 and the conditional expression may be given as while (++ ch < 'Z').

The do-while loop resembles the while loop in both syntax and operation. All the variations that could be implemented with while loop may also be applied on do-while loop. But still do-while loop is used less frequently than while and for loops as it is most appropriate when the statement(s) associated with the loop must be executed at least once. The most common of the do-while loop is to write the menu selection program, where the menu is appeared on the monitor at least once. And then depending upon the user's response it is either repeated or terminated.

The following program is an example of menu selection program using do-while loop. This program provides a menu with three choices. The user has to select one of the three choices, and then the program performs the selected operation. This process will be continuing until user selects the particular choice to terminate the loop.



Lab Exercise

Program:

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

Notes

```
int a,b;
char choice ;
do
{
printf("\n          Main Menu                                   ");
printf("\n ------------------------------------------------");
printf("\n1. ADDITION                                         ");
printf("\n2. SUBTRACTION                                      ");
printf(" \n3. QUIT                                           ");
printf("\n Enter your choice:");
scanf("%c",&choice);
if( choice =='1' || choice == '2')
{
printf("\n Enter the first & second number :");
scanf("%d %d, &a,&b);
}
switch (choice)
{
case '1' :  printf("\n Result = % d", a + b) ;
break;
case '2' :  printf("\n Result = % d", a - b);
break;
default :   printf("\n Wrong input");
}
} while (choice ! = '3' ) ;
printf("\n Thank you") ;
}
```

Output:

```
Run 1:
                              Main Menu
------------------------------------------------------------
                           1.        ADDITION
                           2.        SUBTRACTION
                           3.        QUIT
Enter your choice: 1↵
Enter the first and second number : 5 6↵
Result = 11
                           Main Menu
------------------------------------------------------------
```

Notes

```
                        1.      ADDITION
                        2.      SUBTRACTION
                        3.      QUIT
Enter your choice : 9↵
wrong input
Main Menu
-----------------------------------------------------------
                          1.      ADDITION
                          2.      SUBTRACTION
                          3.      QUIT
enter your choice : 3↵
Thank you
```

Notes To come out from the program execution if the choice entered is other than the given one, the menu is simply redisplayed until user entered the connect one. For the desired operation, program required the input accordingly.

7.5 Jump and Break Statement

What if you need to exit from a loop statement even before the test condition becomes false? You can use the break statement. The break statement is used to terminate loops or to exit from a switch (discussed later). When break is encountered inside any C loop, the loop is immediately exited without testing the loop condition and control automatically passes to the first statement after the loop. It can be used within a while, a do-while, a for or a switch statement. The break statement is written simply as

```
break;
```

The break statement does not have any operand.

Following C code snippets illustrate use of break statement to exit from various C loops. In each situation, the loop will continue to execute as long as the current value for the integer variable x does not exceed 10. However, the computation will break out of the loop if a negative value for x is detected.

While Loop

```
scanf ("%d", &x);
while (x <= 10)
{
        if (x < 0)
        {
                printf ("Negative value entered!!\n");
                break;
        }
```

Notes

```
                scanf ("%d", &x);
}
```

do-while Loop

```
do
{
        scanf ("%d", &x);
        if (x < 0)
        {
                printf ("Negative value entered");
                break;
        }
} while (x < = 10);
```

for Loop

```
for (i = 1; x < = 10; ++i)
{
        scanf ("%f", &x);
        if (x < 0)
        {
                printf ("Negative value entered!!");
                        break;
                }
}
```

When break is used in nested while, do-while, for or switch statements, it will cause a transfer of control out of the immediate enclosing statement, but not out of the outer surrounding statements.

Consider the following code snippet in which a while loop is nested within a for loop.

```
for (i = 0; i < = n; ++i)
{
        while (( c = getchar( ))! = '\n')
        {
                if (c = '*') break;
        }
}
```

The internal while loop terminates if the character variable c is assigned an asterisk (*). However, the for loop will continue to execute. Thus, if the value of i is less than n when the break occurs, the program will increment i and make another pass through the for loop.

Notes



Task Write a program to print all prime numbers from 1 to 300. (Use nested loops, break and continue)

7.6 goto Statement

In earlier programming languages goto was very popular looping construct to branch to one particular statement from another one unconditionally. No condition is checked for looping directly. Due to inherent problems associated with goto branching its use it generally discouraged.

For the reasons of backward compatibility, C supports the goto statement to branch unconditionally from one point to another in the program. A goto statement uses an identifier, called label, which specifies the statement to which branching would start execution after a goto has been encountered. A label is any valid identifier name, and must be followed by a colon. A label is placed immediately before the statement where the control is to be transferred.

The general forms of goto and label statements are shown below:

goto label;

…..

…..

label: statements;

. . . . . .

statement;

The label: can by anywhere in the program either before or after the goto label; statement.



Note There is no built-in mechanism for the flow of execution to come back from where it branched.

The following C program evaluates the cube of numbers read from the terminal. Due to the unconditional goto statement at the end, the control is always transferred back to the input statement running the program infinite loop.

```
#include <stdio.h>
main( )
{
        double x, y;
        read:
                scanf ("%f", &x);
                if (x < 0)
                        goto read;
                y = x * x * x;
                printf ("Cube of %f  is %f \n", x, y);
                goto read;
}
```

Notes

The goto statement can be used to transfer the control out of a loop or nested loops when certain peculiar conditions are encountered as shown in the following code snippet.

```
. . . . . .
while (. . . . . )
{
        for (. . . . . )
        {
        . . . . . .
        if (. . . . . .) goto program_end;       // Jump out of loop
        . . . . . .
        }
        . . . . . .
}
program_end:
```

It is advised to avoid using goto as far as possible. But it is not incorrect to use it to enhance the readability of the program or to improve the execution speed.

7.7 Summary

- The for loop in C is the simplest, fixed and entry controlled loop. An infinite for loop can be created by skipping the conditional expression.
- A conditional expression cannot have multiple expression like initialization and updation expression, but it may contain several conditions linked together using logical operators.
- The second type of loop, the while loop is an entry controlled loop as it tests the conditions first and if the condition is true, then only the control will enter into the loop body. An empty loop can also be configured using while statement and could used as a time delay loop.
- C's third loop statement is the do while loop, is an exit controlled loop i.e. it tests the conditions after having executed the statement with in loop body. Unlike the for and while loops, a do while loop always executes at least once.
- The break statement is used in a program to skip the particular part of program code. The another jump statement continue is the compliment of the break statement.
- Instead of forcing termination, it causes the control to jump to the beginning of the loop. The goto statement is C's another jump statement which causes a program control to jump immediately to an executed statement elsewhere in the function.

7.8 Keywords

Control Statements: The statements that allow programmers to alter the sequential flow of execution of the program and control the flow are called control statements.

For Loop: A for loop allows execution of a statement (or a block of statements) repeatedly a number of times.

While Loop: In case the number of times a statement is to be executed is not known in advance, while loop is used.

7.9 Self Assessment

Notes

Choose the appropriate answers:

1. Which one is not include in loop?
   (a) For loop
   (b) While loop
   (c) If loop
   (d) Do-while loop
2. Which loop is well suited for the problems where number of repetitions is not known in advance?
   (a) For loop
   (b) Do-while loop
   (c) While loop
   (d) None
3. exit controlled loop is the other name of
   (a) for loop
   (b) while loop
   (c) do-while loop
   (d) None of the above

Fill in the blanks:

4. ..... statement can be used to exit from an infinite loop.
5. Do-while is a ..... loop.
6. Conditional expression is typically a ..... that is set up to terminate the execution of a loop.

State whether the following statements are true or false:

7. The braces { } can be discarded when the loop-boody contains a single statement.
8. do-while loop also contain the initialization and updation expression as part of loop statement.
9. A do-while loop will execute at least once irrespective of the value of the conditional expression.
10. Due to inherent problems associated with goto branching its use it generally discouraged.

7.10 Review Questions

1. What do you mean by looping?
2. Describe for loop with the help of suitable example.
3. Differentiate while loop and do-while loop.
4. What is the advantage of break statement in while loop?
5. Write a program to find the factorial value of any number entered through the keyboard.

Notes

6. Write a program to print all the ASCII values and their equivalent characters using a while loop. The ASCII values vary from 0 to 255.
7. Write a program to find the range of a set of numbers. Range is the difference between the smallest and biggest number in the list.
8. Write a program to calculate overtime pay of 10 employees. Overtime is paid at the rate of ₹ 12.00 per hour for every hour worked above 40 hours. Assume that employees do not work for fractional part of an hour.
9. Write a program to print out all Armstrong numbers between 1 and 500. If sum of cubes of each digit of the number is equal to the number itself, then the number is called an Armstrong number. For example, 153 = (1 * 1 * 1) + (5 * 5 * 5) + (3 * 3 * 3)
10. Write a program to enter the numbers till the user wants and at the end it should display the count of positive, negative and zeros entered.
11. Write a program to find the range of a set of numbers. Range is the difference between the smallest and biggest number in the list.

Answers: Self Assessment

1. (c)	2. (b)	3. (c)	4. Exit-controlled
5. Break	6. relational expression	7. True	8. False
9. True	10. True		

7.11 Further Readings


Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication: 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, "*Problem Solving with Computers*", Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., "*How to Solve it by Computer*", Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C


Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 8: Arrays

Notes

CONTENTS

Objectives

Introduction

8.1 Arrays

8.1.1 Advantages of Arrays

8.1.2 Types of Arrays

8.2 One-dimensional Array

8.3 Two-dimensional and Multi-dimensional Array

8.4 Array Declaration

8.5 Array Initialization

8.5.1 One-dimensional Array

8.5.2 Two-dimensional Arrays

8.5.3 Multi-dimensional Array

8.6 Accessing Elements of an Array

8.7 Summary

8.8 Keywords

8.9 Self Assessment

8.10 Review Questions

8.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain arrays
- Describe two dimensional array
- Describe array initialization

Introduction

An array is a group of data items of same data type that share a common name. Ordinary variables are capable of holding only one value at a time. If we want to store more than one value at a time in a single variable, we use arrays.

An array is a collective name given to a group of similar quantities. Each member in the group is referred to by its position in the group.

Arrays are alloted the memory in a strictly contiguous fashion. The simplest array is one dimensional array which is simply a list of variables of same data type. An array of one dimensional arrays is called a two dimension array.

Notes

8.1 Arrays

Arrays are allocated the memory in a strictly contiguous fashion. The simplest array is one dimensional array which is a list of variables of same data type. An array of one dimensional arrays is called a two dimensional array; array of two dimensional arrays is three dimensional array and so on.

The members of the array can be accessed using positive integer values (indicating their order in the array) called subscript or index. Look at an array of integers as shown below:

200	120	-78	100	0
a[0]	a[1]	a[2]	a[3]	a[4]

The description of this array is listed below:

Name of the array	:	a
Data type of the array	:	integer
Number of elements	:	5
Valid index values	:	0, 1, 2, 3, 4
Value stored at the location a[0]	:	200
Value stored at the location a[1]	:	120
Value stored at the location a[2]	:	-78
Value stored at the location a[3]	:	100
Value stored at the location a[4]	:	0

8.1.1 Advantages of Arrays

Arrays offer a number of advantages, some of which are elucidated below:

1. If only a limited number of variables of a particular data type is required ion a program, one can choose the variable names to suite the situation. Let us say we require five integer type variables, we can define them as follows:

```
int v_one, v_two, v_three, v_four, v_five;
```

   Now, consider if we require hundred integer type variables, is the above approach convenient? Obviously not. We can, instead, use an array of integer type having 100 elements as shown below:

```
int num[100];
```

2. Array elements can be accessed using index. Therefore, all the elements can be processed in a desired manner in a single for loop that runs for each element, as shown below:

```
for(i=0; i<100; i++)
num[i]=num[i]+10;
```

   In a single for loop, all the elements have been incremented by 10.

3. Since array elements are physically created contiguously in the memory, they can be accesses using pointers (as you will learn later). Therefore, there are more than one way to reference array elements.

Notes

8.1.2 Types of Arrays

According the number of subscripts required to access an array element, arrays can be of following types:

1. One-dimensional array
2. Multi-dimensional array



Task What would be the output of this program?

```
main( )
{
        int sub[50], i ;
        for ( i = 0 ; i <= 48 ; i++ ) ;
        {
            sub[i] = i ;
            printf ( "\n%d", sub[i] ) ;
        }
}
```

8.2 One-dimensional Array

A list of items can be given one variable name using only one subscript and such a variable is called a one dimensional array.



Example: If we want to store a set of five numbers by an array variable number. Then it will be accomplished in the following way:

```
int number [5];
```

This declaration will reserve five contiguous memory locations capable of storing an integer type value each, as shown below:

number [0]	number [1]	number [2]	number [3]	number [4]

As C performs no bounds checking, care should be taken to ensure that the array indices are within the declared limits. Also, indexing in C begins from 0 and not from 1.

8.3 Two-dimensional and Multi-dimensional Array

It is possible to have an array of more than one dimensions. Two dimensional array (2-D array) is an array of number of 1-dimensional arrays.

A two dimensional array is also called a matrix. Consider the following table:

	Item1	Item2	Item3
Sales 1	300	275	365
Sales 2	210	190	325
Sales 3	405	235	240
Sales 4	260	300	380

Notes

This is a table of four rows and three columns. Such a table of items can be defined using two dimensional arrays.

General form of declaring a 2-D array is

```
data_type array_name [row_size] [colum_size];
```



Example: `int marks [4] [2];`

It will declare an integer array marks of four rows and two columns. An element of this array can be accessed by the manipulation of both the indices. printf ("%d", marks [2] [1]) will print the element present in third row and second column.

C allows arrays of three or more dimensions. Multi-dimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript.

The general form of a multi-dimensional array is

```
data_type      array_name [s1] [s2] [s3] . . . [sm];
E.g.:          int survey [3] [5] [12];
               float table [5] [4] [5] [3];
```

Here, survey is a 3-dimensional array declared to contain 180 integer type elements. Similarly, table is a 4-dimensional array containing 300 elements of floating point type.

Let us consider some applications of multidimensional array programming.

1. Sorting an integer array.

```
# include <stdio.h>
void main( )
{
        int arr [5];
        int i, j; temp;
        printf ("\n Enter the elements of the array:"};
        scanf ("%d", & arr [i]);
        for (i = 0; i < = 4; i ++);
        {
                for (J = 0; J < = 3; J ++)
                if (arr [J] > arr [J+1])
                {
                        temp = arr [J];
                        arg [J] = arr [J+1];
                        arr [J+1] = temp;
                }
        }
        printf ("\ n The Sorted array is:");
        for (i = 0; i < 5; i++)
        printf ("\ t %d", arr [i]);
}
```

2. To insert an element into an existing sorted array (Insertion Sort). **Notes**

```
# include <stdio.h>
main( )
{
int i, k, y, x [20], n;
for (i = 0; i < 20; i++)
x [ i] = 0;
printf ("\ Enter the number of items to be inserted:\n");
        scanf ("%d", &n);
        printf ("\n Input %d values \n", n);
        for (k = 0; k < n; k++)
        {
        scanf ("%d", &x [k]);
        y = x [x]
        for (i = k-1; i > = 0 && y < x [i]; i - -)
                x [i+1] = x[i];
                x [i+1] = y;
        }
        printf ("\n The sorted numbers are:");
        for (i = 0; i < n; i++)
        printf ("\n %d", x [i]);
}
```

3. Accept character string and find its length.

We will solve this question by looping instead of using Library function strlen( ).

```
# include <stdio.h>
void main( )
{
        char name [20];
        int i, len;
        printf ("\n Enter the name:");
        scanf ("%s", name);
        for (i = 0; name [i] ! = '\0'; i++);
        Len = i - 1;
        print f("\n Length of array is % d", len);
}
```

Notes

Character Arrays

Just as a group of integers can be stored in an integer array, group of characters can be stored in a character array or "strings". The string constant is a one dimensional array of characters terminated by null character ('\0'). This null character '\0' (ASCII value0) is different from 'O' (ASCII value 48).

The terminating null character is important because it is the only way the function that works with string can know where the string ends.



Example: `Static char name [ ] = {'K', 'R', 'I', 'S', 'H', '\0'};`

This example shows the declaration and initialization of a character array. The array elements of a character array are stored in contiguous locations with each element occupying one byte of memory.

K	R	I	S	H	N	A	'\0'
4001	4002	4003	4004	4005	4006	4007	4009



Notes

1. Contrary to the numeric array where a 5 digit number can be stored in one array cell, in the character arrays only a single character can be stored in one cell. So in order to store an array of strings, a 2-dimensional array is required.
2. As scanf( ) function is not capable of receiving multi word string, such strings should be entered using gets( ).



Task Point out the errors, if any, in this program:

```
main( )
{
        int i, a = 2, b = 3 ;
        int arr[ 2 + 3 ] ;
        for ( i = 0 ; i < a+b ; i++ )
        {
            scanf ( "%d", &arr[i] ) ;
            printf ( "\n%d", arr[i] ) ;
        }
}
```

8.4 Array Declaration

Arrays are defined in the same manner as ordinary variables, except that each array name must be accompanied by the size specification.

Notes

The general form of array declaration is:

```
data_type array_name [size];
```

data-type specifies the type of array, size is a positive integer number or symbolic constant that indicates the maximum number of elements that can be stored in the array.



Example: `float height [50];`

This declaration declares an array named height containing 50 elements of type float.

The compiler will interpret first element as height [0]. As in C, the array elements are induced for 0 to [size-1].

Two dimensional arrays can be declared similarly, as shown below:

```
data_type array_name[size1][size2];
```

For instance, the following array (named b) is array of 2 arrays of integer type of size 5 elements:

```
int b[2][5];
```

The array b has 10 (2 * 5) elements, each capable of storing an integer type data, referenced as:

```
b[0][0]    b[0][1]    b[0][2]    b[0][3]    b[0][4]
b[1][0]    b[1][1]    b[1][2]    b[1][3]    b[1][4]
```

Multidimensional arrays can be declared on the similar lines. A three dimensional array (named c) of int type has been declared below:

```
Int c[2][2][5];
```

The array c has 20 (2 * 2 * 5) elements, each capable of storing an integer type data, referenced as:

```
c[0][0][0]    c[0][0][1]    c[0][0][2]    c[0][0][3]    c[0][0][4]
c[0][1][0]    c[0][1][1]    c[0][1][2]    c[0][1][3]    c[0][1][4]
c[1][0][0]    c[1][0][1]    c[1][0][2]    c[1][0][3]    c[1][0][4]
c[1][1][0]    c[1][1][1]    c[1][1][2]    c[1][1][3]    c[1][1][4]
```

8.5 Array Initialization

8.5.1 One-dimensional Array

The elements of an array can be initialized in the same way as the ordinary variables, when they are declared. Given below are some examples which show how the arrays are initialized.

```
static int num [6] = {2, 4, 5, 45, 12};
static int n [ ] = {2, 4, 5, 45, 12};
static float press [ ] = {12.5, 32.4, -23.7, -11.3};
```

In these examples note the following points:

1. Till the array elements are not given any specific values, they contain garbage value.
2. If the array is initialized where it is declared, its storage class must be either static or extern. If the storage class is static, all the elements are initialized by 0.

Notes

3. If the array is initialized where it is declared, mentioning the dimension of the array is optional.

8.5.2 Two-dimensional Arrays

Two dimensional arrays may be initialized by a list of initial values enclosed in braces following their declaration.

```
E.g.:  static int table[2][3] = {0, 0, 0, 1, 1, 1};
```

initializes the elements of the first row to 0 and the second row to one. The initialization is done by row.

The aforesaid statement can be equivalently written as

```
static int table[2][3] = {{0, 0, 0}, {1, 1, 1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two dimensional array in the form of a matrix as shown below:

```
static int table[2][3] = {{0, 0, 0},
                          {1, 1, 1}};
```

The syntax of the above statement. Commas are required after each brace that closes off a row, except in the case of the last row.

If the values are missing in an initializer, they are automatically set to 0. For instance, the statement

```
static int table [2] [3] = {{1, 1},
                            {2}};
```

will initialize the first two elements of the first row to one, the first element of the second row to two, and all the other elements to 0.

When all the elements are to be initialized to 0, the following short cut method may be used.

```
static int m [3] [5] = {{0}, {0}, {0}};
```

The first element of each row is explicitly initialized to 0 while other elements are automatically initialized to 0.

While initializing an array, it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional. Thus, the following declarations are acceptable.

```
static int arr [2] [3] = {12, 34, 23, 45, 56, 45};
static int arr [ ] [3] = {12, 34, 23, 45, 56, 45 };
```

8.5.3 Multi-dimensional Array



Example: Example of initializing a 4-dimensional array:

```
static int arr [3] [4] [2] = {{{2, 4}, {7, 8}, {3, 4}, {5, 6},},
                              {{7, 6}, {3, 4}, {5, 3}, {2, 3}, },
                              {{8, 9}, {7, 2}, {3, 4}, {6, 1}, }    };
```

Notes

In this example, the outer array has three elements, each of which is a two dimensional array of four rows, each of which is a one dimensional array of two elements.

8.6 Accessing Elements of an Array

Once an array is declared, individual elements of the array are referred using subscript or index number. This number specifies the element's position in the array. All the elements of the array are numbered starting from 0. Thus number [5] is actually the sixth element of an array.

Consider the program given above. It has entered 6 values in the array num. Now to read values from this array, we will again use for Loop to access each cell. The given program segment explains the retrieval of the values from the array.

```
for (count = 0; count < 6; count ++)
{
        printf ("\n %d value =", num [count]);
}
```

Data can be inserted into array by treating the array elements just like any other variable.

If an integer value is to be read from keyboard into an array element (say c[2][3][0]), the following code snippet would do the job:

```
Scanf("%d", &c[2][3][0]);
```

In order to read values in the entire array for loop may be used as explained by the following examples:

```
main( )
{
        int num [6];
        int count;
        for (count = 0; count < 6; count ++)
        {
                printf ("\n Enter %d element:" count+1);
                scanf ("%d", &num [count]);
        }
}
```

In this example, using the for loop, the process of asking and receiving the marks is accomplished. When count has the value zero, the scanf( ) statement will cause the value to be stored at num [0]. This process continues until count has the value greater than 5.

Notes



Task Twenty-five numbers are entered from the keyboard into an array. The number to be searched is entered through the keyboard by the user. Write a program to find if the number to be searched is present in the array and if it is present, display the number of times it appears in the array.



Case Study

Each element of the array has a memory address. The following program prints an array limit value and an array element address.

```
Program:
#include <stdio.h>
void printarr(int a[]);
main()
{
    int a[5];
    for(int i = 0;i<5;i++)
    {
        a[i]=i;
    }
    printarr(a);
}
void printarr(int a[])
{
    for(int i = 0;i<5;i++)
    {
        printf("value in array %d\n",a[i]);
    }
}
void printdetail(int a[])
{
    for(int i = 0;i<5;i++)
    {
        printf("value in array %d and address is %16lu\n",a[i],&a[i]);
\\ A
    }
}
```

Contd...

Notes

Explanation

1. The function printarr prints the value of each element in arr.
2. The function printdetail prints the value and address of each element as given in statement A. Since each element is of the integer type, the difference between addresses is 2.
3. Each array element occupies consecutive memory locations.
4. You can print addresses using place holders %16lu or %p.

Questions

1. Write a program to add two 6 x 6 matrices.
2. Write a program to multiply any two 3 x 3 matrices.
3. Write a program to sort all the elements of a 4 x 4 matrix.
4. Write a program to obtain the determinant value of a 5 x 5 matrix.

8.7 Summary

- An array is a group of memory locations related by the fact that they all have the same name and same data type.
- An array including more than one dimension is called a multidimensional array.
- The size of an array should be a positive number. If an array in declared without a size and in initialized to a series of values it is implicitly given the size of number of initializers.
- Array subscript always starts with 0. Last element's subscript is always one less than the size of the array e.g., an array with 10 elements contains element 0 to 9. Size of an array must be a constant number.

8.8 Keywords

Array: A user defined simple data structure which represents a group of same type of variables having same name each being referred to by an integral index

Multidimensional array: An array in which elements are accessed using multiple indices

One dimensional array: An array in which elements are accessed using a single index

Subscript/Index: The integral index by which an array element is accessed

Two dimensional array: An array in which elements are accessed using two indices

8.9 Self Assessment

Choose the appropriate answers:

1. Array is a group of data items of
   (a) Same data type that share a common name
   (b) Same data type that share a uncommon name
   (c) Not data type that never common name
   (d) None of the above

Notes

2. The general form of array declaration is
   (a) array_name [size];
   (b) data_type array_name [size];
   (c) data_type [size];
   (d) None
3. What will be the output of the following program if the input is - "tomorrow never comes!".

```
main( )
        {
                char letter [80];
                int count;
                for (count =  0; count < 80;  count++)
                        letter[count] = getchar( );
                for (count =  0; count < 80;  count++)
                putchar (toupper (letter[count]));
                }
```

Fill in the blanks:

4. The members of the array can be accessed using positive integer values called .....
5. While initializing a two dimensional array, it is necessary to mention the ..... dimension, whereas the ..... is optional.
6. The ..... is a one dimensional array of characters terminated by null character ('\0').

State whether the following statements are true or false:

7. All the members of an array share a common name and memory location.
8. Array elements contain garbage values till the time they are initialized.
9. 3-dimensional array declared to contain 180 integer type elements.
10. Array element can be accessed using index.

8.10 Review Questions

1. Explain the usefulness of Arrays in C.
2. What do you mean by 'Array'? How it can be declared & initialized in a C program?
3. Draw a diagram to represent the internal storage of an Array.
4. Describe the different types of Array. Give suitable programs.
5. Find the smallest number in an array using pointers.
6. If an array arr contains n elements, then write a program to check if arr[0] = arr[n-1], arr[1] = arr[n-2] and so on.
7. Write a program to copy the contents of one array into another in the reverse order.

Notes

8. How will you initialize a three-dimensional array threed[3][2][3]? How will you refer the first and last element in this array?
9. Write a program to pick up the largest number from any 5 row by 5 column matrix.
10. Write a program to obtain transpose of a 4 x 4 matrix. The transpose of a matrix is obtained by exchanging the elements of each row with the elements of the corresponding column.
11. Write a program that interchanges the odd and even components of an array.

Answers: Self Assessment

1. (a) 2. (b) 3. tomorrow never comes
4. subscript 5. second (column), first dimension (row)
6. string constant 7. False 8. True 9. True
10. True

8.11 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication, 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer,* Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 9: Strings

CONTENTS

Objectives

Introduction

9.1 Strings

9.2 Sequential Fixed Length Structure

9.3 Declaring and Initializing String

9.4 Build-in-Library Functions to Manipulate Strings

9.4.1 strlen( )

9.4.2 strcpy( )

9.4.3 strcat( )

9.4.4 strcmp( )

9.5 Reading and Writing Strings

9.5.1 Character String Input Function

9.5.2 Character String Output Function

9.6 Putting String Together

9.7 Comparison of two String

9.8 String Handling Functions

9.9 Summary

9.10 Keywords

9.11 Self Assessment

9.12 Review Questions

9.13 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain strings
- Describe reading and writing strings
- Explain string handling functions

Introduction

Numeric data are not the only data types that are processed on a computer. Very often the data to be processed are in textual form such as words, names, addresses etc. This type of data care stored and processed using string type variables. C does not have an explicit string data type.

Notes

However, the same can be simulated using character array. This lesson deals with strings and string manipulation as is done in C.

9.1 Strings

A string is defined in C as an array of characters. Each string is terminated by the NULL character, which indicates end of the string. A string constant is denoted by any set of characters included in double-quote marks.

The NULL character is automatically appended to the end of the characters in a string constant when they are stored. Within a program, the NULL character is denoted by the escape sequence '\ 0'. A string constant represents an array whose lower bound is 0 and whose upper bound is the number of characters in the string.

In deciding on a data representation for a given data object one must take into consideration the cost of performing different operations using that representation. In addition, a hidden cost resulting from the necessary storage management operations must also be taken into account.

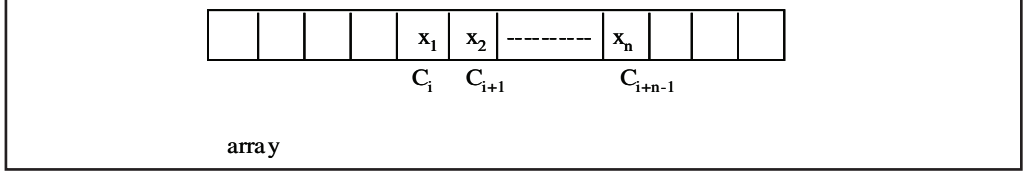
Strings are stored in three types of structures:

1. Fixed length structure
2. Variable length structure
3. Linked structure

9.2 Sequential Fixed Length Structure

In this representation successive characters of a string will be placed in consecutive character positions. The string S = 'x_1---x_n' could then be represented as in Figure 9.1 with s as a pointer to the first character.

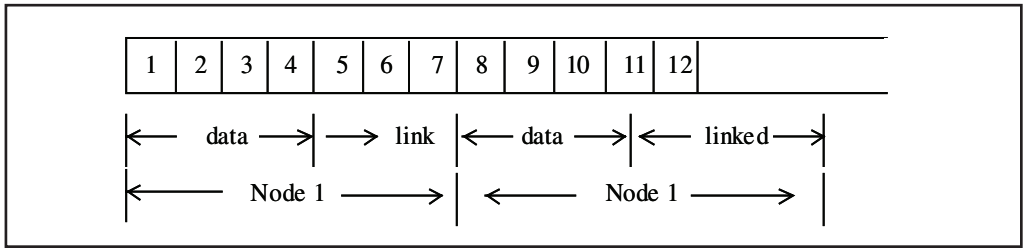
Figure 9.1: Sequential Representation of S = 'x_1----x_n'



Now, if we want to pick a substring of size k from the string of size n, the time required to achieve this would be O(k) plus the time needed to locate a free space big enough to hold the string.

Linked List Fixed Size Nodes

The available memory is divided into nodes of fixed size. Each node has two fields: Data and Link. The size of a node is number of characters that can be stored in the DATA fields.



Notes

In the above figure memory is divided into nodes of size 4 with a link field that is two characters long. Deletion of a substring can be carried out by replacing all characters in this substring by 0 and freeing nodes in which the data fields consist of only 0's.

Storage compaction can be carried out when there are no free nodes. String representation with variable size is similar.

In the purest form of a linked list representation of strings, each node would be of size one. Normally this would represent extreme wastage of space. With a link field of size two characters, this would mean that only 1/3 of available memory would be available to store string information while the remaining 2/3 will be used only for link information.



Task What would happen if you assign a value to an element of an array whose subscript exceeds the size of the array?

9.3 Declaring and Initializing String

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the null character, which is simply the character with the value 0. (The null character has no relation except in name to the null pointer. In the ASCII character set, the null character is named NUL.) The null or string-terminating character is represented by another character escape sequence, \0.

Because C has no built-in facilities for manipulating entire arrays (copying them, comparing them, etc.), it also has very few built-in facilities for manipulating strings.

In fact, C's only truly built-in string-handling is that it allows us to use string constants (also called string literals) in our code. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the \0 character.



Example: We can declare and define an array of characters, and initialize it with a string constant:

```
char string[] = "Hello, world!";
```

In this case, we can leave out the dimension of the array, since the compiler can compute it for us based on the size of the initializer. This is the only case where the compiler sizes a string array for us, however; in other cases, it will be necessary that we decide how big the arrays and other data structures we use to hold strings are.

To do anything else with strings, we must typically call functions. The C library contains a few basic string manipulation functions, and to learn more about strings, we'll be looking at how these functions might be implemented.

Since C never lets us assign entire arrays, we use the `strcpy` function to copy one string to another:

```
#include <string.h>
char string1[] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

The destination string is strcpy's first argument, so that a call to strcpy mimics an assignment expression (with the destination on the left-hand side). Notice that we had to allocate string2

Notes

big enough to hold the string that would be copied to it. Also, at the top of any source file where we're using the standard library's string-handling functions (such as strcpy) we must include the line

```
#include <string.h>
```

which contains external declarations for these functions.

Since C won't let us compare entire arrays, either, we must call a function to do that, too. The standard library's strcmp function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically "less than'' the second string, or a positive number if the first string is "greater." (Roughly speaking, what it means for one string to be "less than'' another is that it would come first in a dictionary or telephone book, although there are a few anomalies.) Here is an example:

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
        printf("strings are equal\n");
else    printf("strings are different\n");
```

This code fragment will print "strings are different''. Notice that strcmp does not return a Boolean, true/false, zero/nonzero answer, so it's not a good idea to write something like

```
if(strcmp(string3, string4))
        ...
```

because it will behave backwards from what you might reasonably expect. (Nevertheless, if you start reading other people's code, you're likely to come across conditionals like if(strcmp(a, b)) or even if(!strcmp(a, b)). The first does something if the strings are unequal; the second does something if they're equal. You can read these more easily if you pretend for a moment that strcmp's name were strdiff, instead.)

Another standard library function is strcat, which concatenates strings. It does not concatenate two strings together and give you a third, new string; what it really does is append one string onto the end of another. (If it gave you a new string, it would have to allocate memory for it somewhere, and the standard library string functions generally never do that for you automatically.) Here's an example:

```
char string5[20] = "Hello, ";
char string6[] = "world!";
printf("%s\n", string5);
strcat(string5, string6);
printf("%s\n", string5);
```

The first call to printf prints ``Hello, '', and the second one prints "Hello, world!", indicating that the contents of string6 have been tacked on to the end of string5. Notice that we declared string5 with extra space, to make room for the appended characters.

If you have a string and you want to know its length (perhaps so that you can check whether it will fit in some other array you've allocated for it), you can call strlen, which returns the length of the string (i.e. the number of characters in it), not including the \0:

```
char string7[] = "abc";
int len = strlen(string7);
printf("%d\n", len);
```

Notes

Finally, you can print strings out with printf using the %s format specifier, as we've been doing in these examples already (e.g. printf("%s\n", string5);).

Since a string is just an array of characters, all of the string-handling functions we've just seen can be written quite simply, using no techniques more complicated than the ones we already know. In fact, it's quite instructive to look at how these functions might be implemented. Here is a version of strcpy:

```
mystrcpy(char dest[], char src[])
{
int i = 0;
while(src[i] != '\0')
        {
        dest[i] = src[i];
        i++;
        }
dest[i] = '\0';
}
```

We've called it mystrcpy instead of strcpy so that it won't clash with the version that's already in the standard library. Its operation is simple: it looks at characters in the src string one at a time, and as long as they're not \0, assigns them, one by one, to the corresponding positions in the dest string. When it's done, it terminates the dest string by appending a \0. (After exiting the while loop, i is guaranteed to have a value one greater than the subscript of the last character in src.) For comparison, here's a way of writing the same code, using a for loop:

```
for(i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];
dest[i] = '\0';
```

Yet a third possibility is to move the test for the terminating \0 character out of the for loop header and into the body of the loop, using an explicit if and break statement, so that we can perform the test after the assignment and therefore use the assignment inside the loop to copy the \0 to dest, too:

```
for(i = 0; ; i++)
        {
        dest[i] = src[i];
        if(src[i] == '\0')
                break;
        }
```

(There are in fact many, many ways to write strcpy. Many programmers like to combine the assignment and test, using an expression like (dest[i] = src[i]) != '\0'. Here is a version of strcmp:

```
mystrcmp(char str1[], char str2[])
{
int i = 0;
while(1)
```

Notes

```
        {
        if(str1[i] != str2[i])
                return str1[i] - str2[i];
        if(str1[i] == '\0' || str2[i] == '\0')
                return 0;
        i++;
        }
}
```

Characters are compared one at a time. If two characters in one position differ, the strings are different, and we are supposed to return a value less than zero if the first string (str1) is alphabetically less than the second string. Since characters in C are represented by their numeric character set values, and since most reasonable character sets assign values to characters in alphabetical order, we can simply subtract the two differing characters from each other: the expression str1[i] - str2[i] will yield a negative result if the i'th character of str1 is less than the corresponding character in str2. (As it turns out, this will behave a bit strangely when comparing upper- and lower-case letters, but it's the traditional approach, which the standard versions of strcmp tend to use.) If the characters are the same, we continue around the loop, unless the characters we just compared were (both) \0, in which case we've reached the end of both strings, and they were both equal. Notice that we used what may at first appear to be an infinite loop--the controlling expression is the constant 1, which is always true. What actually happens is that the loop runs until one of the two return statements breaks out of it (and the entire function).



Note When one string is longer than the other, the first test will notice this (because one string will contain a real character at the [i] location, while the other will contain \0, and these are not equal) and the return value will be computed by subtracting the real character's value from 0, or vice versa. (Thus the shorter string will be treated as ``less than'' the longer.)

Finally, here is a version of strlen:

```
int mystrlen(char str[])
{
int i;
for(i = 0; str[i] != '\0'; i++)
        {}
return i;
}
```

In this case, all we have to do is find the \0 that terminates the string, and it turns out that the three control expressions of the for loop do all the work; there's nothing left to do in the body. Therefore, we use an empty pair of braces {} as the loop body. Equivalently, we could use a null statement, which is simply a semicolon:

```
for(i = 0; str[i] != '\0'; i++)
        ;
```

Notes

Empty loop bodies can be a bit startling at first, but they're not unheard of.

Everything we've looked at so far has come out of C's standard libraries. As one last example, let's write a substr function, for extracting a substring out of a larger string. We might call it like this:

```
char string8[] = "this is a test";

char string9[10];

substr(string9, string8, 5, 4);

printf("%s\n", string9);
```

The idea is that we'll extract a substring of length 4, starting at character 5 (0-based) of string8, and copy the substring to string9. Just as with strcpy, it's our responsibility to declare the destination string (string9) big enough. Here is an implementation of substr. Not surprisingly, it's quite similar to strcpy:

```
substr(char dest[], char src[], int offset, int len)

{

int i;

for(i = 0; i < len && src[offset + i] != '\0'; i++)

        dest[i] = src[i + offset];

dest[i] = '\0';

}
```

If you compare this code to the code for mystrcpy, you'll see that the only differences are that characters are fetched from src[offset + i] instead of src[i], and that the loop stops when len characters have been copied (or when the src string runs out of characters, whichever comes first).

In this unit, we've been careless about declaring the return types of the string functions, and (with the exception of mystrlen) they haven't returned values. The real string functions do return values, but they're of type ``pointer to character,'' which we haven't discussed yet.

When working with strings, it's important to keep firmly in mind the differences between characters and strings. We must also occasionally remember the way characters are represented, and about the relation between character values and integers.

As we have had several occasions to mention, a character is represented internally as a small integer, with a value depending on the character set in use. For example, we might find that 'A' had the value 65, that 'a' had the value 97, and that '+' had the value 43. (These are, in fact, the values in the ASCII character set, which most computers use. However, you don't need to learn these values, because the vast majority of the time, you use character constants to refer to characters, and the compiler worries about the values for you. Using character constants in preference to raw numeric values also makes your programs more portable.)

As we may also have mentioned, there is a big difference between a character and a string, even a string which contains only one character (other than the \0).



Example: 'A' is not the same as "A". To drive home this point, let's illustrate it with a few examples.

If you have a string:

```
char string[] = "hello, world!";
```

Notes

you can modify its first character by saying

```
string[0] = 'H';
```

(Of course, there's nothing magic about the first character; you can modify any character in the string in this way. Be aware, though, that it is not always safe to modify strings in-place like this; we'll say more about the modifiability of strings in a later unit on pointers.) Since you're replacing a character, you want a character constant, 'H'. It would not be right to write

```
string[0] = "H";                /* WRONG */
```

because "H" is a string (an array of characters), not a single character. (The destination of the assignment, string[0], is a char, but the right-hand side is a string; these types don't match.)

On the other hand, when you need a string, you must use a string. To print a single newline, you could call

```
printf("\n");
```

It would not be correct to call

```
printf('\n');                   /* WRONG */
```

printf always wants a string as its first argument. (As one final example, putchar wants a single character, so putchar('\n') would be correct, and putchar("\n") would be incorrect.)

We must also remember the difference between strings and integers. If we treat the character '1' as an integer, perhaps by saying

```
int i = '1';
```

we will probably not get the value 1 in i; we'll get the value of the character '1' in the machine's character set. (In ASCII, it's 49.) When we do need to find the numeric value of a digit character (or to go the other way, to get the digit character with a particular value) we can make use of the fact that, in any character set used by C, the values for the digit characters, whatever they are, are contiguous. In other words, no matter what values '0' and '1' have, '1' - '0' will be 1 (and, obviously, '0' - '0' will be 0). So, for a variable c holding some digit character, the expression

```
c - '0'
```

gives us its value. (Similarly, for an integer value i, i + '0' gives us the corresponding digit character, as long as 0 <= i <= 9.)

Just as the character '1' is not the integer 1, the string "123" is not the integer 123. When we have a string of digits, we can convert it to the corresponding integer by calling the standard function atoi:

```
char string[] = "123";

int i = atoi(string);

int j = atoi("456");
```

Notes



Task What would be the output of this program?

```
main( )
{
        char s[ ] = "Get organised! learn C!!" ;
        printf ( "\n%s", &s[2] ) ;
        printf ( "\n%s", s ) ;
        printf ( "\n%s", &s ) ;
        printf ( "\n%c", s[2] ) ;
}
```

9.4 Build-in-Library Functions to Manipulate Strings

With every C compiler a large set of useful string handling library functions are provided. Table 9.1 lists the more commonly used functions along with their purpose.

Table 9.1

Functions	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one strings to uppercase
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters o one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("I" denotes that this function ignores case)
stricmp	Compares two strings without regards to case (identical to strcmpi)
strnicmp	Compares first n characters if two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strchr	Finds last occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given striung in another string
strset	Sets all characers of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

Out of the above list, We shall discuss the functions strlen( ), strcpy( ), strcat( ) and strcmp( ), since these are the most commonly used functions. This will also illustrate how the library functions in general handle strings. Let us study these functions one by one.

Notes

9.4.1 strlen( )

This function counts the number of characters present in a string. Its usage is illustrated in the following program.



Lab Exercise

```
main( )
{
char arr[ ] = "Bamboozled" ;
int len1, len2 ;
len1 = strlen ( arr ) ;
len2 = strlen ( "Humpty Dumpty" ) ;
printf ( "\nstring = %s length = %d", arr, len1 ) ;
printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

The output would be...

string = Bamboozled length = 10

string = Humpty Dumpty length = 13



Note In the first call to the function strlen( ), we are passing the base address of the string, and the function in turn returns the length of the string. While calculating the length it doesn't count '\0'. Even in the second call,

```
len2 = strlen ( "Humpty Dumpty" ) ;
```

what gets passed to strlen( ) is the address of the string and not the string itself. Can we not write a function xstrlen( ) which imitates the standard library function strlen( )?

9.4.2 strcpy( )

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. Here is an example of strcpy( ) in action...



Lab Exercise

```
main( )
{
char source[ ] = "Sayonara" ;
char target[20] ;
strcpy ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
```

Notes

```
printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

source string = Sayonara

target string = Sayonara

On supplying the base addresses, strcpy( ) goes on copying the characters in source string into the target string till it doesn't encounter the end of source string ('\0'). It is our responsibility to see to it that the target string's dimension is big enough to hold the string being copied into it. Thus, a string gets copied into another, piece-meal, character by character. There is no short cut for this. Let us now attempt to mimic strcpy( ), via our own string copy function, which we will call xstrcpy( ).



Lab Exercise

```
main( )
{
char source[ ] = "Sayonara" ;
char target[20] ;
xstrcpy ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
xstrcpy ( char *t, char *s )
{
while ( *s != '\0' )
{
*t = *s ;
s++ ;
t++ ;
}
*t = '\0' ;
}
```

The output of the program would be...

source string = Sayonara

target string = Sayonara



Note Having copied the entire source string into the target string, it is necessary to place a '\0' into the target string, to mark its end.

Notes

9.4.3 strcat( )

This function concatenates the source string at the end of the target string. For example, "Bombay" and "Nagpur" on concatenation would result into a string "BombayNagpur". Here is an example of strcat( ) at work.



Lab Exercise

```
main( )
{
char source[ ] = "Folks!" ;
char target[30] = "Hello" ;
strcat ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

source string = Folks!

target string = HelloFolks!



Note Note that the target string has been made big enough to hold the final string. I leave it to you to develop your own xstrcat( ) on lines of xstrlen( ) and xstrcpy( ).

9.4.4 strcmp( )

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, strcmp( ) returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is a program which puts strcmp( ) in action.



Lab Exercise

```
main( )
{
char string1[ ] = "Jerry" ;
char string2[ ] = "Ferry" ;
int i, j, k ;
i = strcmp ( string1, "Jerry" ) ;
j = strcmp ( string1, string2 ) ;
k = strcmp ( string1, "Jerry boy" ) ;
```

Notes

```
printf ( "\n%d %d %d", i, j, k ) ;
}
```

And here is the output...

0 4 -32



Task Write a program in C that converts all lowercase characters in a given string to its equivalent uppercase character.

9.5 Reading and Writing Strings

9.5.1 Character String Input Function

% ws or % wc can be used as the specification for reading character strings. The specifier % terminates reading a string at the encounter of blank space. Some versions of scanf( ) support the following conversion specification for strings.

```
% [characters]        and    % [^characters]
```

The specification % [characters] means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character.

The specification % [^character] does exactly the reverse, i.e., characters specified after circumflex (^) are not permitted.

gets( ) function is used to read a character entered at the keyboard and places it at the address pointed to by its character pointer argument.

Characters are entered until the enter key is pressed.

```
syntax: char * gets (char *a);
```

where a is the character array.

9.5.2 Character String Output Function

puts( ) function writes its string argument to the screen followed by the new line.

```
syntax: char * puts (const char * a);
```

puts( ) function takes less space then printf( ). It is faster than printf( ). It does not output numbers or does format conversions as puts( ) outputs character string only.



Example:

```
# include <stdio.h>
# include <conio.h>
main( )
{
        char str [50]
        gets (str);
```

```
        puts (str);
    }
```

Notes

9.6 Putting String Together

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

int main()
{
  char first[100];
  char last[100];
  char full_name[200];

  strcpy(first, "firstName");
  strcpy(last, "secondName");

  strcpy(full_name, first);

  strcat(full_name, " ");
  strcat(full_name, last);
   printf("The full name is %s\n", full_name);
   return (0);
   }
```

9.7 Comparison of two String

```
#include <string.h>
i = strcmp( s1, s2 );
Where:
const char *s1, *s2;
```

are the strings to be compared.

```
int i;
```

gives the results of the comparison. "i" is zero if the strings are identical. "i" is positive if string "s1" is greater than string "s2", and is negative if string "s2" is greater than string "s1". Comparisons of "greater than" and "less than" are made according to the ASCII collating sequence.

9.8 String Handling Functions

A close analysis of the essential string-handling facilities required of any text creation and editing system (formal or otherwise) should lead to the following list of primitive functions:

1. Create a string of test
2. Concatenate two strings to form another string

Notes

3. Search and replace (if desired) a given substring within a string
4. Test for the identity of a string
5. Compute the length of a string

String related functions are grouped into string.h header file. It contains the following functions among others:

1. ***char * strcat(char *dest, const char *src):*** This function appends one string to another returning a pointer to concatenated string. It appends a copy of src to the end of dest. The length of the resulting string is strlen(dest) + strlen(src).

   strings.

2. ***int strcmp(const char *s1, const char *s2):*** This function compares two strings. The string comparison starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The returned values are integers as follows:

   ```
   <  0  if s1 <  s2
   = 0  if s1 = = s2
   > 0  if s1 >  s2
   ```

3. ***char *strcpy(char *dest, const char *src):*** This function copies string src to dest stopping after the terminating null character has been moved. The return value is dest.
4. ***int strlen(const char *s):*** This function returns the length of a string (i.e., the number of characters in s), not counting the terminating null character.
5. ***int strncmp(const char *s1, const char *s2, int maxlen):*** This function compares portions of two strings s1 and s2 looking at no more than maxlen characters. The string comparison starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until maxlen characters have been examined. It returns an int value based on the result of comparing s1 (or part of it) to s2 (or part of it) as given below:

   ```
   <  0  if s1 <  s2
   = 0  if s1 == s2
   >  0  if s1 >  s2
   ```

Some Examples of String

Let us have a look at few of the string operations.

Algorithm to f nd the Length of String

```
# define STRSIZE 80
char string [STRSIZE]
strlen (string)
        char string [ ];
                {
                        int i;
                        for (i = 0; string [i] ! = '\0;, i++);
```

Notes

```
        return (i);
}       /* end strlen * /
```

Algorithm to Concatenate two Strings

```
void strcat (char s1[ ], char s2[ ] )
    {
        int i, j;
        for (i = 0; s1[i] ! = '\0' ; i++);
        for (j = 0; s2[j] ! = '\0'; s1 [i++] = s2 [j++]);
    }
```

Algorithm to f nd a Substring in given String

```
void substr (char s1[ ], int i, int j, char s2[ ])
    {
        int k, m;
        for (k = i, m = 0; m < j; s2[m++] = s1[k++]);
        s2[m] = '\0';
    }       /* end substr * /
```

Task Write a program that takes a set of names of individuals and abbreviates the first, middle and other names except the last name by their first letter.

9.9 Summary

- A string is defined in C as an array of characters. Each string is terminated by the NULL character, which indicates end of the string.
- A string constant is denoted by any set of characters included in double-quote marks.
- The NULL character is automatically appended to the end of the characters in a string constant when they are stored. Within a program, the NULL character is denoted by the escape sequence '\ 0'.
- A string constant represents an array whose lower bound is 0 and whose upper bound is the number of characters in the string.
- Strings are stored in three types of structures - Fixed length structure, Variable length structure, and Linked structure.

9.10 Keywords

gets( ): A C library function used to read a character entered at the keyboard and to place it at the address pointed to by its character pointer argument

puts( ): A C library function that writes its string argument to the screen followed by the new line

Notes

strcat(): The C library function that appends one string to another returning a pointer to concatenated string

strcmp(): The C library function that compares two strings

string.h: A C header file that contains string manipulating library functions

String: An array of characters terminated by the NULL character

9.11 Self Assessment

Choose the appropriate answers:

1. Out of the following which one is not the returned value by using int strcmp(const char *s1, const char *s2)
   (a) < 0 if s1 < s2
   (b) = 0 if s1 = = s2
   (c) 0 if s1 > s2
   (d) 1 if s1 < s2
2. A string is defined in C as an
   (a) array of characters
   (b) array of integer
   (c) character of integer
   (d) None of the above

Fill in the blanks:

3. ..... character is denoted by the escape sequence '\ 0'.
4. The ..... means that only the characters specified within the brackets are permissible in the input string.
5. Each node has two fields: Data and .....

State whether the following statements are true or false:

6. puts( ) function takes less space then printf( ).
7. A string is defined in C as an array of characters.
8. int strcmp not compare two string.
9. gets( ) function is used to read a character entered at the keyboard.
10. Strings in C are represented by arrays of characters.

9.12 Review Questions

1. Write a C program that reads a sentence from the keyboard and prints the frequency of each letter.
2. How can you create a string type C variable? Can they be assigned to each other in the same way as other data types? Explain.
3. Write a program that converts a string like "124" to an integer 124.

Notes

4. Write a program that replaces two or more consecutive blanks in a string by a single blank. For example, if the input is

   Grim return to the planet of apes!!

   the output should be

   Grim return to the planet of apes!!

5. Can an array of pointers to strings be used to collect strings from the keyboard? If not, why not?

6. Write a program to sort a set of names stored in an array in alphabetical order.

7. Write a program to delete all vowels from a sentence. Assume that the sentence is not more than 80 characters long.

8. Write a program that uses an array of pointers to strings str[ ]. Receive two strings str1 and str2 and check if str1 is embedded in any of the strings in str[ ]. If str1 is found, then replace it with str2.

   char *str[ ] = {

   "We will teach you how to...",

   "Move a mountain",

   "Level a building",

   "Erase the past",

   "Make a million",

   "...all through C!"

   } ;

   For example if str1 contains "mountain" and str2 contains "car", then the second string in str should get changed to "Move a car".

9. Write a program that takes a set of names of individuals and abbreviates the first, middle and other names except the last name by their first letter.

10. A factory has 3 division and stocks 4 categories of products. An inventory table is updated for each division and for each product as they are received. There are three independent suppliers of products to the factory:

    (a) Design a data format to represent each transaction.

    (b) Write a program to take a transaction and update the inventory.

    (c) If the cost per item is also given write a program to calculate the total inventory values.

Answers: Self Assessment

1.	(d)	2.	(a)	3.	NULL	4.	specification %
5.	Link	6.	True	7.	True	8.	False
9.	True	10.	True				

Notes

9.13 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication, 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer,* Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 10: Pointers

Notes

CONTENTS

Objectives
Introduction
10.1 Pointers
10.2 Accessing the Address of a Variable
10.3 Pointer Declaration
10.3.1 Address Operator - &
10.3.2 Indirection Operation - *
10.4 Pointer Variables
10.5 Initialization of Pointer Variables
10.6 Accessing a Variable through its Pointer
10.7 Chain of Pointers
10.7.1 Pointer to Pointers
10.7.2 Two-dimensional Arrays and Pointers
10.8 Pointer Expression
10.9 Pointer Increment and Scale Factors
10.10 Pointer and Arrays
10.11 Pointers and Character Strings
10.12 Array of Pointers
10.13 Void Pointers
10.14 Summary
10.15 Keywords
10.16 Self Assessment
10.17 Review Questions
10.18 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss the concepts of pointers
- Identify pointer increment and scale factors
- State the array of pointers
- Know about void pointers

Notes

Introduction

Computers use their memory for storing instructions of the programs as well as the values of the variables. Since memory is a sequential collection of storage cells each cell has an address associated with it. Whenever we declare a variable, the system allocates, somewhere in the memory, a memory location and a unique address is assigned to this location. Whenever a value is assigned to this variable the value gets stored in the location having a unique address in the memory associated with that variable. Therefore, the values stored in memory can be manipulated using their addresses.

Pointer is an extremely powerful mechanism to write efficient programs. Incidentally, this feature makes C stand out as the most powerful programming language. Pointers are the topic of this unit.

10.1 Pointers

A memory variable is merely a symbolic reference given to a memory location. Now let us consider that an expression in a C program is as follows:

```
int a = 10, b = 5, c;

c = a + b;
```

The above expression implies that a, b and c are the variables which can hold the integer data. Now from the above mentioned statement let us assume that the variable 'a' occupies the address 3000 in the memory, 'b' occupies 3020 and the variable 'c' occupies 3040 in the memory. Then the compiler will generate the machine instruction to transfer the data from the location 3000 and 3020 into the CPU, add them and transfer the result to the location 3040 referenced as c. Hence we can conclude that every variable holds two values:

Address of the variable in the memory (l-value)

Value stored at that memory location referenced by the variable. (r-value)

Pointer is nothing but a simple data type in C programming language, which has a special characteristic to hold the address of some other memory location as its r-value. C programming language provides '&' operator to extract the address of any object. These addresses can be stored in the pointer variable and can be manipulated.

The syntax for declaring a pointer variable is,

```
<data type> *<identifier>;
```



Example:

```
int n;

int *ptr; /* pointer to an integer*/
```

The following statement assigns the address location of the variable n to ptr, and ptr is a pointer to n.

```
ptr=&n;
```

Since a pointer variable points to a location, the content of that location is obtained by prefixing the pointer variable by the unary operator * (also called the indirection or dereferencing operator) like, *<pointer_variable>.

Notes

Example:

```
# include<stdio.h>
main()
{
int a=10, *ptr;
ptr=&a;                /* ptr points to the location of a */
printf("The value of a pointed by the pointer ptr is: %d", *ptr);
/* printing the value of a pointed by ptr through the pointer ptr*/
}
```

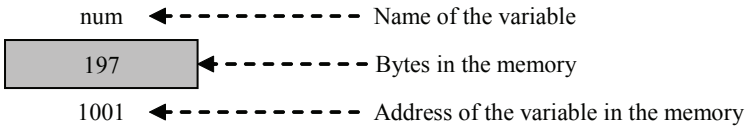
A null value can be assigned to a pointer when it does not point to any data or in the other words, as a good programming habit every pointer should be initialized with the null value. A pointer with a null value assigned to it is nothing but a pointer which contains the address zero.

The precedence of the unary operators '&' and '*' are same in C language. Here as a special case we can mention that '&' operator cannot be used or applied to any arithmetic expression, it can only be used with an operand which has unique address.

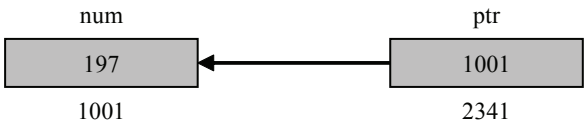
Pointer is a variable which can hold the address of a memory location. The value stored in a pointer type variable is interpreted as an address. Consider the following declarative statement:

```
int num = 197;
```

This statement instructs the compiler to reserve a 2-byte memory location (assuming that the target machine stores an int type in two bytes) and to put the value 84 in that location. Assume that a system allocates memory location 1001 for num. Diagrammatically it can be shown as:



As the memory addresses are numbers, they can be assigned to some other variable. Let ptr be the variable which holds the address of variable num. We can access the value of num by the variable ptr. Thus, we can say "ptr points to num". Diagrammatically, it can be shown as:



Note ptr is itself a variable therefore it will also be stored at a location in the memory having some address (2341 in above case). Here we say that – ptr is a pointer variable which is currently pointing to an integer type variable num which holds the value 197.

Task A string can be declared as a character array or a variable of type char *. Discuss with suitable example (With the help of C program)

Notes

10.2 Accessing the Address of a Variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable? This can be done with the help of the operator & available in C. The operator & immediately preceding a variable return the address of the variable associated with it.



Example: The statement

P = &quantity;

Would assign the address 5000 to the variable p. The & operator can be remembered as 'address of'.

The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

& 125 (pointing at constant).

Int x[10];

&x (pointing at array names).

&(x+y) (pointing at expressions).

If x is an array, then expression such as

&x[0] and &x[i+3]

are valid and represent the addresses of 0th and (i+3)th elements of x

10.3 Pointer Declaration

Since pointer variables contain address that belongs to a separate data type, they must be declared as pointers before we use them. Pointers can be declared just a any other variables. The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

The above statement tells the compiler three things about the variable pt_name.

1. The asterisk (*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type data type.



Example: The statement

```
int *p;
```

declares the variable p as a pointer variable that points to an integer data type (int). The type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.

Given below are some more examples of pointer declaration.

Pointer declaration	Interpretation
Int *rollnumber;	Create a pointer variable rollnumber capable of pointing to an integer type variable or capable of holding the address of an integer type variable
char *name;	Create a pointer variable name capable of pointing to a character type variable or capable of holding the address of a character type variable
float *salary;	Create a pointer variable salary capable of pointing to a float type variable or capable of holding the address of a float type variable

Notes

10.3.1 Address Operator - &

Once a pointer variable has been declared, it can be made to point to a variable by assigning the address of that variable to the pointer variable. The address of a variable can be extracted using address operator - &.

An expression having & operator generates the address of the variable it precedes. Thus, for example,

```
&num
```

produces the address of the variable num in the memory. This address can be assigned to any pointer variable of appropriate type (i.e., the data type of variable num) using an assignment statement such as p = # which causes p to point to num. That is, p now contains the address of num.

The assignment shown above is known as pointer initialization. Before a pointer is initialized, it should not be used. A pointer variable can be initialized in its declaration itself.

```
int x;
int *p = &x;
```

statement declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. This is an initialization of p, not *p. On the contrary, the statement

```
int *p = &x, x;
```

is invalid because the target variable x is not declared before the pointer.

10.3.2 Indirection Operation - *

Since a pointer type variable contains an assigned address of another variable the value stored in the target variable can be obtained using this address. The value store in a variable can be referred to using a pointer variable pointing to this variable using indirection operator (*).



Example: Consider the following code.

```
int x = 109;
int *p;
p = &x;
```

Then the following expression

```
*p
```

represents the value 109.

10.4 Pointer Variables

The actual address of a variable is not known immediately. We can determine the address of a variable using 'address of' operator (&). We have already seen the use of 'address of' operator in the scanf( ) function.

Another pointer operator available in C is "*" called "value a address" operator. It gives the value stored at a particular address. This operator is also known as 'indirection operator'.



Example:

```
main( )
{
```

Notes

```
int i = 3;
printf ("\n Address of i: = %u", & i); /* returns the address * /
printf ("\t value i = %d", * (&i)); /* returns the value of address of i */
}
```

10.5 Initialization of Pointer Variables

Since pointer variables contain address that belong to a separate data type, they must be declared as pointers before we use them.

The declaration of a pointer variable takes the following form:

```
data_type *pt_name
```

This tells the compiler three things about the variable pt_name.

1. The asterisk (*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type data type.

Example: int *p; declares the variable p as a pointer variable that points to an integer data type. The type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as p = &quantity; which causes p to point to quantity. That is, p now contains the address of quantity. This is known as pointer initialization. Before a pointer is initialized, it should not be used. A pointer variable can be initialized in its declaration itself.

Example: int x, *p=&x; statement declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. This is an initialization of p, not *p. On the contrary, the statement int *p = &x, x; is invalid because the target variable x is declared first.

10.6 Accessing a Variable through its Pointer

Consider the following statements:

```
int q, * i, n;
q = 35;
i = & q;
n = * i;
```

i is a pointer to an integer containing the address of q. In the fourth statement we have assigned the value at address contained in i to another variable n. Thus, indirectly we have accessed the variable q through n. using pointer variable i.

Task Write a program in C to show the initialization of pointers.

10.7 Chain of Pointers

Notes

10.7.1 Pointer to Pointers

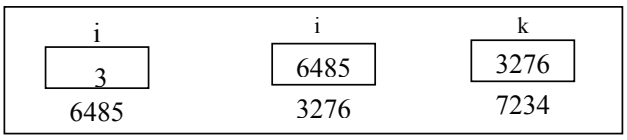
Pointer is a variable, which contains address of a variable. This variable itself could be another pointer. Thus, a pointer contains another pointer's address as shown in the example given below:



Lab Exercise

```
main()
{
     int i = 3; int *j, **k;
     j = &i; k = &j;
     printf ("Address of i = %d\n", &i);
     printf ("Address of i = %d\n", j);
     printf ("Address of i = %d\n", *k);
     printf ("Address of j = %d\n", &j);
     printf ("Address of j = %d\n", *k);
     printf ("Address of k = %d\n\n", &k);
     printf ("Value of j = %d\n", j);
     printf ("Value of k = %d\n", k);
     printf ("Value of i = %d\n", i);
     printf ("Value of i = %d\n", *(&i));
     printf ("Value of i = %d\n", *j);
     printf ("Value of i = %d\n", **k);
}
```

The following figure would help you in tracing out how a program prints the output.



Output: Address of i = 6485

Address of i = 6485

Address of i = 6485

Address of j = 3276

Address of j = 3276

Address of k = 7234

Value of j = 6485

Notes

Value of k = 3276

Value of i = 3

Value of i = 3

Value of i = 3

Value of i = 3

10.7.2 Two-dimensional Arrays and Pointers

A two dimensional array can be defined as a pointer to a group of contiguous one dimensional arrays. A two dimensional array declaration can be written as:

```
data_type (*ptvar)[expression2];
```

```
rather than data_type array [expression1] [expression2];
```

This can be generalized to higher dimensional arrays, that is,

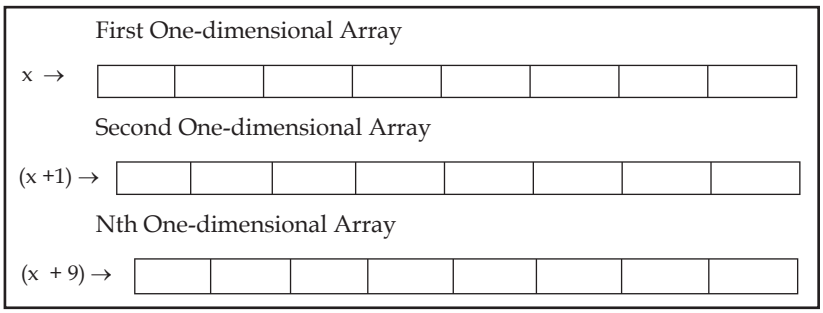
```
data_type (*ptvar) [expression2] [expression3] ... [expression n];
```

```
replaces data_type array [expression1] [expression2]...[expression n];
```

In these declarations data_type refers to the data type of the array, ptvar is the name of the pointer variable, array is the corresponding array name, and expression1, expression 2 ------ expression n are positive valued integer expressions that indicate the maximum number of array elements associated with each subscript.

Example: Suppose that x is a two dimensional integer array having 10 rows and 20 columns. We can declare x as int (*x) [20]; rather than int x[10] [20];

In the first declaration, x is defined to be a pointer to a group of contiguous, one dimensional, 20-element integer arrays. Thus, x points to the first 20-elements array, which is actually the first row (row 0) of the original two dimensional array. Similarly, (x + 1) points to the second 20-elements array, which is the second row (row 1) of the original two dimensional array, and so on, as illustrated below.



Now consider a three dimensional floating-point array t.

This array can be defined as float (*t) [20] [30]; rather than float t [10] [20] [30]

In the first declaration, t is defined as a pointer to a group of contiguous, two-dimensional, 20 × 30 floating-point arrays. Hence, t points to the first 20 × 30 arrays, (t+1) points to the second 20 × 30 array, and so on.

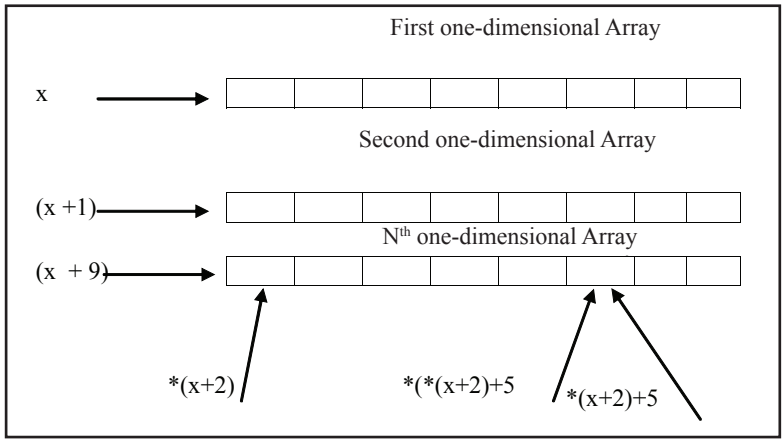
An individual array element within a multi-dimensional array can be accessed by repeatedly using the indirection operator. Usually, however, this procedure is more awkward than the conventional method for accessing an array element. The following example illustrates the use of the indirection operator.

Notes

Suppose that x is a two dimensional integer array having 10 rows and 20 columns, as declared in the previous example.

The item in row 2, column 5 can be accessed by writing either x[2][5] or *(*(x + 2) + 5)

The second form requires some explanation. First, note that x is a pointer to row 0 so (x + 2) is a pointer to row 2. Therefore, the object of this pointer, *(x + 2), refers to the entire row. Since row 2 is a one dimensional array, *(x + 2) is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence, (*(x + 2) + 5) is a pointer to element 5 (the sixth element) in row 2. The object of this pointer, *(*(x + 2) + 5), therefore, refers to the item in column 5 of row 2, which is x[2][5].



10.8 Pointer Expression

Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers. For example, if p1 and p2 are properly declared and initialized pointers, then following statements are valid.

y = *p1 * *p2; /multiply values stored in variables pointed to by *p1/and *p2

sum = sum + *p1; /increment sum by the value stored in the variable/pointed to by p1

The pointer may point to any location in the memory therefore you should be careful while using pointers in your programs.

10.9 Pointer Increment and Scale Factors

We have seen that the pointers can be increment like

```
p1 = p2 +2;
```

```
p1 = p1 +1;
```

and so on. Remember, however, an expression like

```
p1++;
```

will cause the pointer p1 to pint to the next value of its type. For example if p1 is an integer pointer with an initial value, say 2800, then after the operation p1 = p1 +1, the value of p1 will be 2802, and 2801, that is when we increment a pointer, its value is increased by the 'length; of the data type that it point to. This length called the scale factor.

For an IBM PC, the length of various data types are as follows:

```
charater              1 byte
integers              2 bytes
```

Notes

```
floats                  4 bytes
long integers           4 bytes
doubles                 8 bytes
```

The number of bytes used to store various data depends on the system and can be found by making use of the sizeof operator. For example, if x is a variable, then sizeof(x) return the number of bytes needed for the variable.

10.10 Pointer and Arrays

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

The array declared as:

```
static int x[5] = {1, 2, 3, 4, 5};
```

is stored as follows:

Elements	x[0]	x[1]	x[2]	x[3]	x[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[0] is stored. That is,

```
x = &x[0] = 1000
```

If we declare p as an integer pointer, then we can make the pointer p to point to the array x by the

assignment statement

```
p = x ;
```

which is equivalent to

```
p = &x[0];
```

Now we can access every value of x using p++ to move from one element to another. The relationship between p and x is shown below:

p = &x[0] (=1000)

p+1 = &x[1] (=1002)

p+2 = &x[2] (=1004)

p+3 = &x[3] (=1006)

The address of an element is calculated using its index and the scale factor of the data type, i.e.,

Address of x[3] = Base Address + (3 × Scale Factor of int) = 1000 + (3 × 2) = 1006

When handling arrays, instead of using array indexing, we can use pointers to access array elements, as *(p+3) gives the value of x[3]. The pointer accessing method is much faster than array indexing. &x[i] and (x+i) both represent the address of the ith element of x. x[i] and *(x+i) both represent the contents of that address, the value of the ith element of x. The two terms are interchangeable.

Notes

When assigning a value to an array element such as x[i], the left side of the assigned statement may be written as either x[i] or as *(x+i). Thus, a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element. While assigning an address to an identifier, a pointer variable must appear on the left side of the assignment statement. Expressions such as x, (x+1) and &x[i] cannot appear on the left side of an assignment statement because it is not possible to assign an arbitrary address to an array name or an array element.


Task Write a program in C to show the pointer declaration.

10.11 Pointers and Character Strings

As we have seen in strings, a string in C is an array of characters ending in the null character (written as '\0'), which specifies where the string terminates in memory. Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string. The value of a string is the (constant) address of its first character. Thus, it is appropriate to say that a string is a constant pointer.

A string can be declared as a character array or a variable of type char *. The declarations can be done as shown below:

```
char country[ ] = "INDIA";

char *country = "INDIA";
```

Each initialize a variable to the string "INDIA". The second declaration creates a pointer variable country that points to the letter I in the string "INDIA" somewhere in memory.

Once the base address is obtained in the pointer variable country, *country would yield the value at this address, which gets printed through,

```
printf ("%s", *country);
```

Here is a program that dynamically allocates memory to a character pointer using the library function malloc at run-time. An advantage of doing this way is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional character array.


Example: Write a program to test whether the given string is a palindrome or not.

```
/* Program tests a string for a palindrome using pointer notation */
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
main()
{
        char *palin, c;
        int i, count;

        short int palindrome(char,int); /*Function Prototype */
        palin = (char *) malloc (20 * sizeof(char));
        printf("\nEnter a word: ");
```

Notes

```
do
{
        c = getchar( );
        palin[i] = c;
        i++;
}while (c != '\n');

i = i-1;
palin[i] = '\0';
count = i;
if (palindrome(palin,count) == 1)
        printf ("\nEntered word is not a palindrome.");
else
        printf ("\nEntered word is a palindrome");
}
short int palindrome(char *palin, int len)
{
        short int i = 0, j = 0;
        for(i=0 , j=len-1; i < len/2;i++,j--)
        {
                if (palin[i] == palin[j])
                        continue;
                else
                        return(1);
        }
        return(0);
}
```

```
Output:
Enter a word: malayalam
Entered word is a palindrome.
Enter a word: abcdba
Entered word is not a palindrome.
```



Task Identify the errors in the following pointer arithmetic:

1. int *a;
   a = a - a;
2. int *a;
   a = a * 2;
3. int *a;
   a = 5;

Array of Pointers to Strings

Notes

Arrays may contain pointers. We can form an array of strings, referred to as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character, so each entry in an array of strings is actually a pointer to the first character of a string. Consider the following declaration of a string array:

```
char *country[ ] = {
"INDIA", "CHINA", "BANGLADESH", "PAKISTAN", "U.S"
};
```

The *country[ ] of the declaration indicates an array of five elements. The char* of the declaration indicates that each element of array country is of type "pointer to char". Thus, country [0] will point to INDIA, country[1] will point to CHINA, and so on.

Thus, even though the array country is fixed in size, it provides access to character strings of any length. However, a specified amount of memory will have to be allocated for each string later in the program, for example,

```
country[ i ] = (char *) malloc(15 * sizeof (char));
```

The country character strings could have been placed into a two-dimensional array but such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when a large number of strings are stored with most strings shorter than the longest string.

As individual strings can be accessed by referring to the corresponding array element, individual string elements be accessed through the use of the indirection operator.

For example, * (* country + 3) + 2) refers to the third character in the fourth string of the array country. Let us see an example below.



Example: Write a program to enter a list of strings and rearrange them in alphabetical order, using a one-dimensional array of pointers, where each pointer indicates the beginning of a string:



Lab Exercise

```
/* Program to sort a list of strings in alphabetical order using an array
of pointers */
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# include <string.h>
void readinput (char *[ ], int);
void writeoutput (char *[ ], int);
void reorder (char *[ ], int);
main( )
{
        char *country[ 5 ];
        int i;
```

Notes

```
        for (i = 0; i < 5; i++)
        {
        country[ i ] = (char *) malloc (15 * sizeof (char));
        }
        printf ("Enter five countries on a separate line\n");
        readinput (country, 5);
        reorder (country, 5);
        printf ("\nReordered list\n");
        writeoutput (country, 5);
        getch( );
}
void readinput (char *country[ ], int n)
{
        int i;
        for (i = 0; i < n; i++)
        { scanf ("%s", country[ i ]); }
        return;
}
void writeoutput (char *country[ ], int n)
{
        int i;
        for (i = 0; i < n; i++)
        { printf ("%s", country[ i ]);
        printf ("\n"); }
        return;
}
void reorder (char *country[ ], int n)
{
        int i, j;
        char *temp;
        for (i = 0; i < n-1; i++)
        {
        for (j = i+1; j < n; j++)
        {
        if (strcmp (country[ i ], country[ j ]) > 0)
        {
        temp = country[ i ];
        country[ i ] = country[ j ];
        country[ j ] = temp;
        }
```

Notes

```
        }
        }
        return;
}
Output:
Enter five countries on a seperate line
INDIA
BANGLADESH
PAKISTAN
CHINA
SRILANKA
Reordered list
BANGLADESH
CHINA
INDIA
PAKISTAN
SRILANKA
```

The limitation of the string array concept is that when we are using an array of pointers to strings we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using scanf( ).

10.12 Array of Pointers

A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such situations the newly defined array will have one less dimension than the original multi-dimensional array. Each pointer will indicate the beginning of a separate (n - 1) dimensional array.

In general terms, a two dimensional array can be defined as one dimensional array of pointers by writing

```
data_type *array[expression1];
```

rather than the conventional array definition data_type array[expression1] [expression2];

Similarly, a n dimensional array can be defined as a (n-1) dimensional array of pointers by writing

```
data_type  *array[expression1][expression2]...[expressionn-1];
```

rather than the conventional array definition data_type array[expression1] [expression2]... [expressionn];

In these declarations data_type refers to the data type of the original n dimensional array, array is the array name, and expression1, expression2, . . ., expression n are positive-valued integer expressions that indicate the maximum number of elements associated with each subscript.

The array name and its preceding asterisk are not enclosed in parentheses in this type of declaration. Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

Notes

Moreover, note that the last (the rightmost) expression is omitted when defining an array of pointers, whereas the first (the leftmost) expression is omitted when defining a pointer to a group of arrays.

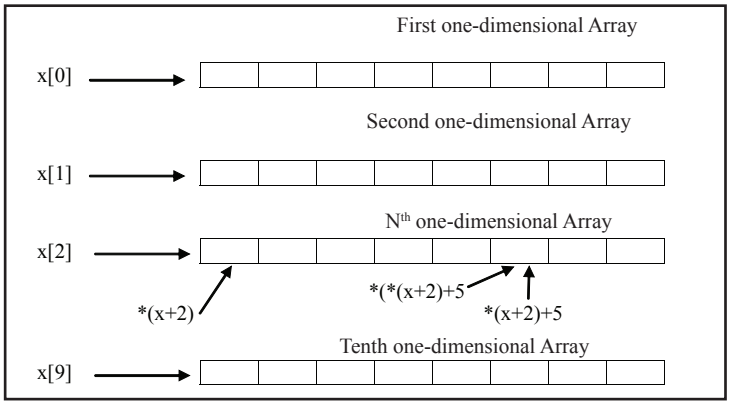
When a n dimensional array is expressed in this manner, an individual array element within the n dimensional array can be accessed by a single use of the indirection operator. The following example illustrates how this is done.

Suppose that x is a two dimensional integer array having 10 rows and 20 columns, we can define x as a one dimensional array of pointers by writing int *x[10];

Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row, and so on. The number of elements within each row is not explicitly specified.

An individual array element, such as x[2][5], can be accessed by writing *(x[2] + 5). In this expression, x[2] is a pointer to the first element in row 2, so that (x[2] + 5) points to element 5 (actually, the sixth element) within row 2. The object of this pointer, *(x[2] + 5), therefore, refers to x[2] [5].

These relationships are illustrated below:



10.13 Void Pointers

A void pointer is a C convention for "a raw address." The compiler has no idea what type of object a void.

Pointer "really points to." If you write

```
int *ip;
```

ip points to an int. If you write

```
void *p;
```

p doesn't point to a void!

In C and C++, any time you need a void pointer, you can use another pointer type. For example, if you have a char*, you can pass it to a function that expects a void*. You don't even need to cast it. In C (but not in C++), you can use a void* any time you need any kind of pointer, without casting. (In C++, you need to cast it).

A void pointer is used for working with raw memory or for passing a pointer to an unspecified type.

Some C code operates on raw memory. When C was first invented, character pointers (char *) were used for that. Then people started getting confused about when a character pointer was a string, when it was a character array, and when it was raw memory.

Notes

At first glance, a void pointer seems to be of limited, if any, use. However, when combined with the ability to cast such a pointer to another type, they turn out to be quite useful and flexible.

Consider the example of the previous section, where we constructed a function pointer to a function of type void and argument int. Such a function pointer in this form could not be used for a void function with a different type of argument (for example, float). This can be done, however, through the use of void pointers, as the following example illustrates.

```
#include <stdio.h>
void use_int(void *);
void use_float(void *);
void greeting(void (*)(void *), void *);
int main(void) {
  char ans;
  int i_age = 22;
  float f_age = 22.0;
  void *p;
  printf("Use int (i) or float (f)? ");
  scanf("%c", &ans);
  if (ans == 'i') {
    p = &i_age;
    greeting(use_int, p);
  }
  else {
    p = &f_age;
    greeting(use_float, p);
  }
  return 0;
}
void greeting(void (*fp)(void *), void *q) {
  fp(q);
}
void use_int(void *r) {
  int a;
  a = * (int *) r;
  printf("As an integer, you are %d years old.\n", a);
}
void use_float(void *s) {
  float *b;
  b = (float *) s;
  printf("As a float, you are %f years old.\n", *b);
}
```

Notes

Although this requires us to cast the void pointer into the appropriate type in the relevant subroutine (use_int or use_float), the flexibility here appears in the greeting routine, which can now handle in principle a function with any type of argument.



Task

What would be the output of this program?

```
main( )
{
        int i = 3 ;
        printf ( "\nAddress of i = %u", &i ) ;
        printf ( "\nValue of i = %d", i ) ;
}
```



Case Study

A pointer is a variable whose value is also an address. As described earlier, each variable has two attributes: address and value. A variable can take any value specified by its data type. For example, if the variable i is of the integer type, it can take any value permitted in the range specified by the integer data type. A pointer to an integer is a variable that can store the address of that integer.

Program

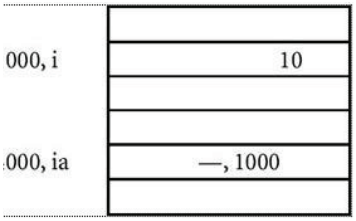
```
#include <stdio.h>
main ()
{
   int i;          //A
   int * ia;       //B
   i = 10;         //C
   ia = &i;   //D
    printf (" The address of i is %8u \n", ia);          //E
    printf (" The value at that location is %d\n", i);   //F
    printf (" The value at that location is %d\n", *ia); //G
  *ia = 50;                                  //H
    printf ("The value of i is %d\n", i);                //I
}
```

Explanation

1. The program declares two variables, so memory is allocated for two variables. i is of the type of int, and ia can store the address of an integer, so it is a pointer to an integer.
2. The memory allocation is as follows:

Contd...

Notes



3. i gets the address 1000, and ia gets address 4000.
4. When you execute i = 10, 10 is written at location 1000.
5. When you execute ia = &i then the address and value are assigned to i, thus i has the address of 4000 and value is 1000.
6. You can print the value of i by using the format %au because addresses are usually in the format unsigned long, as given in statement E.
7. Statement F prints the value of i, (at the location 1000).
8. Alternatively, you can print the value at location 1000 using statement G. *ia means you are printing the value at the location specified by ia. Since i has the value for 1000, it will print the value at location 1000.
9. When you execute *ia = 50, which is specified by statement H, the value 50 is written at the location by ia. Since ia specifies the location 1000, the value at the location 1000 is written as 50.
10. Since i also has the location 1000, the value of i gets changed automatically from 10 to 50, which is confirmed from the printf statement written at position i.

10.14 Summary

- Pointers are often passed to a function as arguments by reference. This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form.
- There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array.
- Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.
- As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as dynamic memory allocation.
- The library routine malloc can be used for this purpose.

10.15 Keywords

Array of Pointer: A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays.

Pointer: It is a variable which can hold the address of a memory location rather than the value at the location.

Pointer Expression: Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers.

Notes

10.16 Self Assessment

Choose the appropriate answers:

1. Pointer is a
   (a) Memory variable
   (b) Simple data type
   (c) Both of the above
   (d) None of the above
2. Pointer is a variable
   (a) Which contains address of a variable
   (b) Not contains the address of a variable
   (c) Pointer contains another pointer's address
   (d) Both (a) and (c)
3. Out of the following which one is not the length of data types for an IBM PC
   (a) Integers 2 bytes
   (b) Floats 2 bytes
   (c) Long integers 4 bytes
   (d) Doubles 8 bytes

Fill in the Blanks:

4. The value store in a variable can be referred to using a pointer variable pointing to this variable using ..... operator.
5. A pointer is a type of ..... which holds the address of another variable.
6. When an array is passed to a function as an argument, only ..... is passed.
7. Runtime memory allocation is known as .....
8. An individual array element within a multi-dimensional array can be accessed by repeatedly using the ..... operator.

State whether the following statements are true or false:

9. It is possible to return multiple values using a return statement.
10. A pointer variable can be initialized in its declaration itself.
11. It is possible to multiply a pointer with a constant.
12. A pointer can contain the address of another pointer variable.

10.17 Review Questions

1. Define 'Pointer'. List down the various advantages of using pointers in a C program.
2. How pointer are initialized and implemented in C? Write a program to explain the concept.
3. Explain with the help of a C program, the concept of Pointer Arithmetic in C.

Notes

4. How printer in C incorporates the concept of Arrays? Write a suitable program to demonstrate the concept.
5. Differentiate the followings:
   (a) Pointer and arrays
   (b) Pointer to a variable and pointer to a pointer
   (c) Pointer and variable
   (d) Value in a function and address in a function
6. Twenty-five numbers are entered from the keyboard into an array. Write a program to find out how many of them are positive, how many are negative, how many are even and how many odd.
7. What would be the output of the following programs:

```
main( )
{
        int b[ ] = { 10, 20, 30, 40, 50 };
        int i;
        for ( i = 0 ; i <= 4 ; i++ )
                printf ( "\n%d" *( b + i ) ) ;
}
```

8. Write a function to calculate the factorial value of any integer entered through the keyboard.
9. Write a function power (a, b), to calculate the value of a raised to b.
10. A positive integer is entered through the keyboard, write a program to obtain the prime factors of the number. Modify the function suitably to obtain the prime factors recursively.

Answers: Self Assessment

1. (c) 2. (d) 3. (b) 4. indirection
5. variable 6. address of the first element of the array
7. dynamic memory allocation 8. indirection
9. True 10. True 11. False 12. True

10.18 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C",* Pearson Education, Year of Publication, 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language",* Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C",* Tata McGraw Hill Publishing Company Limited, New Delhi

Notes

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer,* Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 11: Functions

Notes

CONTENTS

Objectives
Introduction
11.1 Need for User-defined Function
11.2 A Multifunction Program
11.3 Elements of User-defined Functions
11.4 Definition of Functions
11.5 Return Value and their Types
11.6 Function Calls
11.7 Function Declaration
11.8 Category of Functions
11.9 No Argument and no Return Values
11.10 Argument but no Return Values
11.11 Arguments with Return Values
11.12 No Argument but Returns a Value
11.13 Functions that Return Multiple Values
11.13.1 Call by Value
11.13.2 Call by Reference
11.14 Function Prototype
11.15 Recursive Functions
11.16 Storage Classes and their Usage
11.16.1 Automatic Variable
11.16.2 External Variable
11.16.3 External Declaration
11.16.4 Static Variable
11.16.5 Register Variable
11.17 Summary
11.18 Keywords
11.19 Self Assessment
11.20 Review Questions
11.21 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- State the need for user defined functions
- Identify category of functions
- Describe functions that return multiple values
- Discuss recursive functions
- Explain the storage classes and their usage

Introduction

A function is a programming unit which can be identified by a unique name. Once defined, it can be called by a program. It may take zero or more inputs when called. What is to done with the received input(s) is determined by the code written inside the definition of the function. After carrying out the specified manipulation the function generates a single output. This output is passed on to the caller of the function.



Note A C program is nothing but a group of related functions.

11.1 Need for User-defined Function

Why write separate functions at all? Why not squeeze the entire logic into one function, main( )? Two reasons:

1. Writing functions avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of a triangle. If later in the program you want to calculate the area of a different triangle, you won't like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a 'section of code' that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.
2. Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

11.2 A Multifunction Program

One of the merits in the C language is the usage of a function because the function. In C always behave like a traditional function or a procedure. A function may call one more function and so on. There no restriction in C for calling the number of functions in a program. It is advisable that the complex problem may be decomposed into a small easily manageable part and define a function. The control will be transferred from calling the portion of a program to the called function block. If the called function is executed successfully, then control will be transferred back to the calling the portion of a program. There is always overhead of a transfer of the control between calling portion and a called function block.

Notes

Example: The multifunction program segment is shown below

```
function1 ()
{
            --------------
            --------------
            function2 ();
            --------------
            --------------
            function4 ();
}

function2 ()
{

            --------------
            --------------
            function3 ();
            --------------
            --------------
}

function3 ();
{
            --------------
            --------------
}

function4 ()
{
        --------------
        --------------
}
```

Lab Exercise

Program:

```
A program to demonstrate the transfer of control between the multifunction
program.
Main()
{
```

Notes

```
        int j = 10;
        printf ("Inside the main() function\n");
        function1 ();
        printf ("after the function 1\n");
        printf ("main function () \n");
        printf ("j = %d\n", j);
}
function1 ()
{
        int i,n;
        n = 3;
        for(i = 0; i<=n-1; ++){
        printf("inside a function 1\n");
        printf("i = %d\n", i);
        function2 ();
        }
}
function2 ()
{
        printf ("transfer of control\n);
        printf ("inside a function 2\n");
}
```

11.3 Elements of User-defined Functions

Functions are classified as one of the derived data types in C. We can therefore define functions and use them like any other variables in C programs. It is therefore not a surprise to note that there exist some similarities between functions and variables in C.

1. Both function names and variable names are considered identifiers and therefore they must adhere to the rules for identifiers.
2. Like variables, functions have types associated with them.
3. Like variables, function names and their must be declared and defined before they are used in a program.

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition.
2. Function call
3. Function declaration.

The function definition is an independent program, module that is specially written to implement the requirements if the function. In order to use this function we need to invoke it is a required place in the program. This is known as the function call. The program that calls the function is referred to as calling program or calling function.

11.4 Definition of Functions

Notes

A function is a self-contained block of executable code that can be called from any other function. In many programs, a set of statements are to be executed repeatedly at various places in the program and may with different sets of data, the idea of functions comes in mind. You keep those repeating statements in a function and call them as and when required. When a function is called, the control transfers to the called function, which will be executed, and then transfers the control back to the calling function (to the statement following the function call). Let us see an example as shown below:



Example:

```
/* Program to illustrate a function*/
#include <stdio.h>
main ()
{
void sample( );
printf("\n You are in main");
}
void sample( )
{
printf("\n You are in sample");
}
Output:
You are in sample
You are in main
```

Here we are calling a function sample ( ) through main( ) i.e. control of execution transfers from main( ) to sample( ) , which means main( ) is suspended for some time and sample( ) is executed. After its execution the control returns back to main( ), at the statement following function call and the execution of main( ) is resumed.

The syntax of a function is:

```
return data type function_name (list of arguments)
{
datatype declaration of the arguments;
executable statements;
return (expression);
}
```

where,

1. Return data type is the same as the data type of the variable that is returned by the function using return statement.
2. A function_name is formed in the same way as variable names/identifiers are formed.
3. The list of arguments or parameters are valid variable names as shown below, separated by commas: (data type1 var1,data type2 var2,…….. data type n var n) for example (int x, float y, char z).

Notes

4. Arguments give the values which are passed from the calling function.
5. The body of function contains executable statements.
6. The return statement returns a single value to the calling function.



Task Define the functions factorial( ), prime( ) and fibonacci( ) in a file, say 'myfuncs.c'. Do not define main( ) in this file.



Example: Let us write a simple function that calculates the square of an integer.

```
/*Program to calculate the square of a given integer*/
/* square( ) function */
{
int square (int no)           /*passing of argument */
int result ;                  /* local variable to function square */
result = no*no;
return (result);              /* returns an integer value */
}
/*It will be called from main()as follows */
main( )
{
int n ,sq;                    /* local variable to function main */
printf ("Enter a number to calculate square value");
scanf("%d",&n);
sq=square(n);                 /* function call with parameter passing */
printf ("\nSquare of the number is : %d", sq);
}                             /* program ends */
Output:
Enter a number to calculate square value: 5
Square of the number is: 25
```

11.5 Return Value and their Types

If a function has to return a value to the calling function, it is done through the return statement. It may be possible that a function does not return any value; only the control is transferred to the calling function. The syntax for the return statement is:

```
return (expression);
```

We have seen in the square( ) function, the return statement, which returns an integer value.

Important Points

Notes

1. You can pass any number of arguments to a function but can return only one value at a time.



Example: The following are the valid return statements

(a) return (5);

(b) return (x*y);



Example: The following are the invalid return statements

(a) return (2, 3);

(b) return (x, y);

2. If a function does not return anything, void specifier is used in the function declaration.



Example:

```
void square (int no)
{
int sq;
sq = no*no;
printf ("square is %d", sq);
}
```

3. All the function's return type is by default is "int", i.e. a function returns an integer value, if no type specifier is used in the function declaration.



Examples:

(a) square (int no); /* will return an integer value */

(b) int square (int no); /* will return an integer value */

(c) void square (int no); /* will not return anything */

4. What happens if a function has to return some value other than integer? The answer is very simple: use the particular type specifier in the function declaration.



Example: Consider the code fragments of function definitions below:

(a) *Code Fragment – 1:*

```
char func_char( …….. )
{
char c;
……
……
……
}
```

Notes

(b) *Code Fragment – 1:*

```
float func_float (……..)
{
float f;
…….
…….
…….
return(f);
}
```

Thus from the above examples, we see that you can return all the data types from a function, the only condition being that the value returned using return statement and the type specifier used in function declaration should match.

5. A function can have many return statements. This thing happens when some condition based returns are required.



Example:

```
/*Function to find greater of two numbers*/


int greater (int x, int y)
{
if (x>y)
return (x);
else
return (y);
}
```

6. And finally, with the execution of return statement, the control is transferred to the calling function with the value associated with it.

   In the above example, if we take x = 5 and y = 3, then the control will be transferred to the calling function when the first return statement will be encountered, as the condition (x > y) will be satisfied. All the remaining executable statements in the function will not be executed after this returning.

11.6 Function Calls

A function can be called by specifying its name followed by a list of arguments enclosed in parentheses and separated by commas. If a function call does not require any arguments, an empty pair of parenthesis must follow the function name.

The arguments appearing in the function call are referred to as actual arguments, in contrast to the formal arguments that appear in the first line of function definition.



Lab Exercise

```
e.g.:       /* Program to find square of given number */
```

Notes

```
        main( )
        {
                float square (float);           /* function prototype dec/n*/
                float a, b;
                printf ("\n Enter the number:");
                scanf ("%f", &a);
                b = square (a);                 /* calling of function with */
                                                / * actual arguments */
                printf ("Square of entered no. is = %f" , b);
        }
   float square (x)     / * function definition with format l argument * /
   float x;                             /* format l  argument declaration * /
   {
        float y;                /* Local variable declaration
        y = x * x;
        return (y);
   }
Output:
Enter the number: 2
Square of the entered number is = 4
```

11.7 Function Declaration

A function is declared in the following manner:

```
<return_data_type>   <function_name>(arg1, arg2, arg3)
<data_type_1> arg1; <data_type_2> arg2; <data_type_3> arg3;
{
        statement-1;
        statement-2
                :
                :
        statement-n;
        return(<expression of return_data_type>);
}
```

Example: The following function (name being getsq) returns the square of the input number of float type. Clearly the <return_data_type> will also be float type.

```
        float getsq(x)
        float x;
        {
```

Notes

```
                return(x*x);
        }
```

Another form of a function definition is:

```
<return_data_type><function_name>(formal argument list)
{
        statement-1;
        statement-2;
        ----------
        statement-n;
        return (<expression of return_data_type>);
}
```

Where formal argument list is a comma separated list of variables and their corresponding data types.

The following function, addthem, takes two int type arguments and returns the sum of the two.

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

The <return_data_type> always represents the data type of the value which is returned. The type specification can be omitted if the function returns an integer or a character.

An empty pair of parenthesis must follow the function name if the function definition does not include any arguments.

The argument declarations follow the first line. Each formal argument must have the same data type as its corresponding actual argument.

The remainder of the function definition is a compound statement that defines the action to be taken by the function. It is referred to as the body of the function.

The last statement in the body of function is return (expression). It is used to return the computed result, if any, to the calling program.



Task What would be the output of this program?

```
main( )
{
        printf ( "\nOnly stupids use C?" ) ;
        display( ) ;
}
display( )
{
        printf ( "\nFools too use C!" ) ;
        main( ) ;
}
```

Notes

11.8 Category of Functions

We categorize a function's invoking (calling) depending on arguments or parameters and their returning a value. In simple words, we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.

The various types of calling functions are:

1. With no arguments and with no return value.
2. With no arguments and with return value.
3. With arguments and with no return value.
4. With arguments and with return value.

11.9 No Argument and no Return Values

Any function which has no arguments and does not return any values to the calling function, falls in this category. These type of functions are confined to themselves i.e. neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function are only program control will be transferred.



Example:

```
/* Program for illustration of the function with no arguments and no return
value*/

/* Function with no arguments and no return value*/

#include <stdio.h>

main()

{

void message();

printf("Control is in main\n");

message();                              /* Type 1 Function */

printf("Control is again in main\n");

}

void message()

{

printf("Control is in message function\n");

}                                       /* does not return anything */
```

Output:

```
Control is in main

Control is in message function

Control is again in main
```

Notes

11.10 Argument but no Return Values

If a function includes arguments but does not return anything, it falls in this category. One way communication takes place between the calling and the called function.

Before proceeding further, first we discuss the type of arguments or parameters here. There are two types of arguments:

1. Actual arguments
2. Formal arguments

Let us take an example to make this concept clear:



Example: Write a program to calculate sum of any three given numbers.

```
#include <stdio.h>
main()
{
int a1, a2, a3;
void sum(int, int, int);
printf("Enter three numbers: ");
scanf ("%d%d%d",&a1,&a2,&a3);
sum (a1,a2,a3);                        /* Type 3 function */
}
/* function to calculate sum of three numbers */
void sum (int f1, int f2, int f3)
{
int s;
s = f1+ f2+ f3;
printf ("\nThe sum of the three numbers is %d\n",s);
}
```

Output

Enter three numbers: 23 34 45

The sum of the three numbers is 102

Here f1, f2, f3 are formal arguments and a1, a2, a3 are actual arguments.

Thus we see in the function declaration, the arguments are formal arguments, but when values are passed to the function during function call, they are actual arguments.



Note The actual and formal arguments should match in type, order and number.

11.11 Arguments with Return Values

Notes

In this category, two-way communication takes place between the calling and called function i.e. a function returns a value and also arguments are passed to it. We modify above example according to this category.



Example: Write a program to calculate sum of three numbers.

```
/*Program to calculate the sum of three numbers*/
#include <stdio.h>
main ( )
{
int a1, a2, a3, result;
int sum(int, int, int);
printf("Please enter any 3 numbers:\n");
scanf ("%d %d %d", & a1, &a2, &a3);
result = sum (a1,a2,a3);      /* function call */
printf ("Sum of the given numbers is : %d\n", result);
}
/* Function to calculate the sum of three numbers */
int sum (int f1, int f2, int f3)
{
return(f1+ f2 + f3);          /* function returns a value */
}
```

Output

Please enter any 3 numbers:

3 4 5

Sum of the given numbers is: 12



Task Point out the errors, if any, in this program:

```
main( )
{
        int a ;
        a = message( ) ;
}
message( )
{
        printf ( "\nViruses are written in C" ) ;
        return ;
}
```

Notes

11.12 No Argument but Returns a Value

Suppose, if a function does not receive any data from calling function but does send some value to the calling function, then it falls in this category.



Example: Write a program to find the sum of the first ten natural numbers.

```
/* Program to find sum of first ten natural numbers */
#include <stdio.h>
int cal_sum()
{
int i, s=0;
for (i=0; i<=10; i++)
s=s + i;
return(s);     /* function returning sum of first ten natural numbers */
}
main()
{
int sum;
sum = cal_sum();
printf("Sum of first ten natural numbers is % d\n", sum);
}
```

Output

Sum of first ten natural numbers is 55

11.13 Functions that Return Multiple Values

A function must be called with proper number of properly types arguments failing which the compiler will report compilation error. There are two different ways in which arguments may be passed to function when calling it – call by value and call by reference.

11.13.1 Call by Value

Call by value means sending the values of the arguments to functions. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value or call by value.



Lab Exercise

```
e.g.:       /*  A simple C program containing a function that alters the
value of its argument. */
```

Notes

```
#include <stdio.h>
main()
{
     int a = 2;
     printf("\na = %d (from main, before calling the function)",a);
     modify(a);
     printf("\na = %d (from main, after calling the function)",a);
}
modify (int a)
{
a * = 3;
printf("\na = %d (from the function, after being modified)",a);
return;
}
```

output: a = 2 (from main, before calling the function)

a = 6 (from the function, after being modified)

a = 2 (from main, after calling the function)

The original value of a (i.e.=2) is displayed when main is executed. This value is then passed to the function modify, where it is multiplied by three and the new value of the formal argument that is displayed within the function. Finally, the value of a within main (i.e., the actual argument) is again displayed, after control is transferred back to function main from function modify.

These results show that a is not altered within main, even though the corresponding value of a is changed within modify.

Passing an argument by value has certain advantages and disadvantages.

On the positive side, it allows a single valued actual argument to be written as an expression rather than being restricted to a single variable. Moreover, if the actual argument is expressed as a single variable, it protects the value of this variable from alterations within the function.

On the negative side, it prevents information from being transferred back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information.

11.13.2 Call by Reference

Call by reference means sending the addresses of the arguments to the called function. In this method the addresses of actual arguments in the calling function are copied into formal arguments of the called functions. Thus using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. Using a call by reference intelligently, it is possible to make a function return more than one value at a time, which involves the study of pointer.

Notes

11.14 Function Prototype

Before defining the function, it is desired to declare the function along with its prototype. In function prototype, the return value of function, type, and number of arguments are specified. The declaration of all functions statement should be first statement in

```
main( ).
```

The general form of function declaration using ANSI Prototype is

```
data_type function_name (type1 arg1, type2 arg2 - - - - );

where arg1, arg2. . . are the list of arguments.
```

Function prototypes are desirable because they facilitate error checking between calls to a function and corresponding function definition. They also help the compiler to perform automatic type conversions on function parameters. When a function is called, actual arguments are automatically converted to the types in function definition using normal rules of assignment.

11.15 Recursive Functions

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computation in which each action is stated in terms of previous result.

In order to solve a problem recursively, two conditions must be satisfied:

1. The problem must be written in recursive form.
2. The problem statement must include a stopping condition.



Example:

```
/*To calculate the factorial of an integer recursively * /
# include <stdio.h>
main( )
{
        int n;
        long int fact (int);
        printf ("\n n = ");
        scanf ("%d", &n);
        printf ("\n n! = % ld" fact (n));
}
long int fact (int n)
{
        if (n < = 1)
        return 1;
        else
        return (n * factorial (n-1));
}
```

The Tower of Hanoi

Notes

The Towers of Hanoi is a well-known children's game, played with three poles and a number of different sized discs. Each disc has a hole in the centre, allowing it to be stacked around any of the poles. Initially, the discs are stacked on the leftmost pole in the order of decreasing size, i.e., the largest at the bottom and the smallest at the top.

The object of the game is to transfer the discs from the leftmost pole to the rightmost pole, without ever placing a larger disc on top of a smaller disc. Only one disc may be moved at a time, and each disc must always be placed around one of the poles.

The general strategy is to consider one of the poles to be the origin, and another to be the destination. The third pole will be used for intermediate storage, thus allowing the discs to be moved without placing a larger disc over a smaller one. Assume there are n discs, numbered from smallest to largest.

If the discs are initially stacked on the left pole, the problem of moving all n discs to the right pole can be stated in the following recursive manner:

1. Move the top n-1 discs from the left pole to the centre pole.
2. Move the nth disc (the largest disc) to the right pole.
3. Move the n-1 discs from the centre pole to the right pole.

The problem can be solved in this manner for any value of n greater than 0 (n=0 represents a stopping condition).

The program consists of main ( ), which merely reads in a value for n and then initiates the computation by calling a function transfer. In this first function call, the actual parameters will be specified as character constants, that is,

```
transfer (n, 'L', 'R', 'C' );
```

where,

n	number of discs
L	represents the left pole
R	represents the right pole
C	represents the centre pole

This function call specifies the transfer of all n discs from the leftmost pole (the origin) to the rightmost pole (the destination), using the centre pole for intermediate storage.



Lab Exercise

```
e.g.:    /* Program to solve the TOWERS OF HANOI problem using recursion. */
     #include<stdio.h>
     main()
     {
     void transfer(int, char, char, char);
     int n;
```

Notes

```
        printf("\nWelcome to the TOWERS OF HANOI.");
        printf("\nHow many discs?");
        scanf("%d", &n);
        printf("\n");
        transfer(n, 'L', 'R', 'C');
        }
void transfer(int n, char from, char to, char temp)
/* transfer n discs from one pole to another number of discs; from = origin;
to = destination; temp = temporary storage   */
{
        if (n > 0)
        {
                /* move n-1 discs from origin to temporary */
                transfer(n-1, from, temp, to);
                /* move nth disc from origin to destination */
                printf("\nMove disc %d from %c to %c\n", n, from, to);
                /* move n-1 discs from temporary to destination */
                transfer(n-1,temp,to,from);
        }
        return;
}
```


Task

Write a function power (a, b), to calculate the value of a raised to b.

The function transfer receives a different set of values for its arguments each time the function is called. These sets of values will be pushed onto the stack so that each set is independent of the others. They are then popped from the stack at the proper time during the execution of the program.

It is this ability to store and retrieve these independent sets of values that allows the recursion to work.

Sample Programs

To generate positive prime numbers:

```
#include <stdio.h>
main( )
{
        int no, x min, x max, J;
        printf ("kEnter the lower and upper limit of the numbers:");
        scanf ("%d %d", &xmin, &xmax);
```

Notes

```
        printf ("\n prime numbers are:");
        for (J = x min; J < = x max; J++)
        prime (J);
}
prime (int no)
{
int i; flag = 0
        if (no < = 3) printf ("\n % d", no);
        else
{
        for (i = 2; i < = (no/2); i++)
        {
        if (no % i = = 0)
        {
        flag = 0; break;
        }
        else flag = 1;
        }
        if (flag = = 1) printf ("\n % d", no);
        }
}
```

11.16 Storage Classes and their Usage

There are two different ways to characterize variables:

1. by data types
2. by storage class

Data types refers to the type of information while storage class refers to the life-time of a variable and its scope within the program.

A variable in C can have any one of the four storage classes.

1. Automatic variable
2. External variable
3. Static variable
4. Register variable

11.16.1 Automatic Variable

The scope of an automatic variable is confined to that function in which it is declared. It is created when the function is called and destroyed automatically when the function is exited. Hence the name is Automatic.

Notes

By default, a variable declared inside a function with storage class specification is an automatic variable. Automatic variable values cannot be changed accidently by what happens in some other functions in the program.



Example:

```
main( )
{
        int m = 1000;
        function 2( );
        printf ("%d \n", m);
}
function 1( )
{
        int m = 10;
        printf ("%d \n", m);
}
function 2( )
{
        int m = 100;
        function 1( );
        printf ("%d \n", m);
        }
```

output: 10

100

1000

11.16.2 External Variable

An external variable is also known as a global variable. It is not confined to a single function. Its scope extends from the point of definition through the remainder of the program.

External variables can be accessed from any function that falls within their scope. They are declared outside a function. If a local variable and a global variable have the same name, local variable will have precedence over global in the function where it is declared.



Example:

```
int count;
main
{
        count = 10;
        - - - - - -
```

Notes

```
        - - - - -
        - - - - -
}
function ( )
{
        int count = 0;
        - - - - - - -
        - - - - - - -
        count ++;
}
```

When the function references the variable count, it will be referencing only its local variable, not the global one. The value of count in main( ) will not be affected.



Example:

```
/* illustration of working of global variable
        int x;
        main( )
        {
                x = 10;
                printf ("x = %d \n", x);
                printf ("x = %d \n", fun1( ));
                printf ("x = %d \n", fun2( ));
                printf ("x = % d \n", func3( ));
        }
        fun1( )
        {
                x = x + 10;
                return x;
        }
        fun2( )
        {
                int x = 1;
                return x;
        }
                fun3( )
        {
                x = x+10;
                return (x);
        }
```

Output: x = 10

Notes

x = 20

x = 1

x = 30

11.16.3 External Declaration

In the program segment discussed just previously, the main cannot access the variable y as it has been declared after the main function. This problem can be solved by declaring the variable with the storage class extern.



Example:

```
main()
{
        extern int y;  /* external declaration */
        --------
}
fun1()
{
        extern int y;  /* external declaration */
        --------
}
int y;                          /*definition */
```

The external declaration of y inside the functions informs the compiler that y is an integer type defined somewhere else in the program.



Note The extern declaration does not allocate storage space for variable.

11.16.4 Static Variable

Static variables are defined within a function in the same manner as automatic variables, except that the variable declaration must begin with the static storage class designation.



Example: `static int x;` or `static float y;`

A static variable is initialized only once, when the program is compiled. It is never initialized again.

A static variable may be either an internal type or an external type, depending on the place of declaration. Internal static variables are those which are declared inside a function. The scope of internal static variables extends upto the end of the function in which they are defined. Therefore, internal static variables are similar to auto variables, except that they remain in existence (alive) throughout the remaining program. Therefore, internal static variables can be used to retain values between function calls.

Notes



Example:

```
/* Illustration of static variable */
main()
{
        int i;
        for(i=1; i<=3; i++)    stat();
}
stat()
{
        static int x = 0;
        x = x+1;
        printf("x = %d;\t", x);
}
```

Output: x = 1; x = 2; x = 3;

An external static variable is declared outside of all functions and is available to all the functions in that program. The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files also.



Task Write a function that receives 5 integers and returns the sum, average and standard deviation of these numbers. Call this function from main( ) and print the results in main( ).

11.16.5 Register Variable

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored) since a register access is much faster than a memory access and keeping the frequently accessed variables in the register will lead to faster execution of programs.

For example, Loop control variables.

This is done as given below:

```
register int count;
```

Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, C will automatically convert register variables into non-register variables once the limit is reached.

Notes



Case Study

When a function is written before main it can be called in the body of main. If it is written after main then in the declaration of main you have to write the prototype of the function. The prototype can also be written as a global declaration.

```
Program:
Case 1:
#include <stdio.h>
main ( )
{
    int i;
    void (int *k)          // D
    i = 0;
    printf (" The value of i before call %d \n", i);
    f1 (&i);          // A
    printf (" The value of i after call %d \n", i);
}
void (int *k)             // B
{
    *k = *k + 10;         // C
}
Case 2:
#include <stdio.h>
void (int *k)             // B
{
    *k = *k + 10;         // C
}
main ( )
{
    int i;
    i = 0;
    printf (" The value of i before call %d \n", i);
    f1 (&i);          // A
    printf (" The value of i after call %d \n", i);
}
Case 3:
#include <stdio.h>
```

Contd...

Notes

```
void f1(int *k)                    // B
{
    *k = *k + 10;          // C
}
.
main ( )
{
    int i;
    i = 0;
printf ("The value of i before call %d \n", i);
    f1 (&i);           // A
printf ("The value of i after call %d \n", i);
}
```

Explanation

In Case 1, the function is written after main, so you have to write the prototype definition in main as given in statement D.

In Case 2, the function is written above the function main, so during the compilation of main the reference of function f1 is resolved. So it is not necessary to write the prototype definition in main.

In Case 3, the prototype is written as a global declaration. So, during the compilation of main, all the function information is known.

Questions

1. Write a function which receives a float and an int from main( ), finds the product of these two and returns the product which is printed through main( ).
2. Write a function that receives 5 integers and returns the sum, average and standard deviation of these numbers. Call this function from main( ) and print the results in main( ).
3. Write a function that receives marks received by a student in 3 subjects and returns the average and percentage of these marks. Call this function from main( ) and print the results in main( ).

11.17 Summary

- In this unit, we learnt about "Functions": definition, declaration, prototypes, types, function calls datatypes and storage classes, types function invoking and lastly Recursion.
- All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function.
- We have seen that the functions, which do not return any value, must be declared as "void", return type.
- A function can return only one value at a time, although it can have many return statements.
- A function can return any of the data type specified in 'C'.

Notes

11.18 Keywords

Call by Reference: It means sending the addresses of the arguments to the called function.

Data types: It refers to the type of information while storage class refers to the life-time of a variable and its scope within the program.

Function Call: A function can be called by specifying its name followed by a list of arguments enclosed in parentheses and separated by commas.

Return Statement: Information is returned from the function to the calling portion of the program via return statement.

11.19 Self Assessment

Choose the appropriate answers:

1. Out of the following which one is not the element of user defined function.
   (a) Function initialization
   (b) Function call
   (c) Function definition
   (d) Function declaration
2. Which one is a types of calling functions?
   (a) With no arguments and with no return value.
   (b) With no arguments and with return value
   (c) With arguments and with no return value
   (d) All of the above

Fill in the blanks:

3. The scope of a ..... variable is not confined to a single function.
4. By default, a variable declared inside a function with storage class specification is .....
5. A function must be called with proper ..... and ..... of arguments failing which the compiler will report compilation error.
6. Passing by value is restricted to a ..... transfer of information.
7. Two arguments are passed to the main( ) function are ..... and .....
8. We can pass array elements to function by calling them either by ..... or .....
9. The problem statement of a recursive function must include a ..... condition.

State whether the following statements are true or false:

10. The function being called may/may not be declared before it is called either in full or in prototype.
11. In the call by value method, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change.
12. C allows function nesting.

Notes

11.20 Review Questions

1. Takes two integer inputs and produces the remainder when the larger is divided by the smaller.
2. Swaps the two given integers.
3. What do you mean by function call.
4. Describe return value and their types.
5. Evaluates the following series for a specified n:

   $1^2 + 2^2 + 3^2 + 4^2 + ……..n^2$
6. A positive integer is entered through the keyboard. Write a function to obtain the prime factors of this number.
7. Write a function which receives a float and an int from main( ), finds the product of these two and returns the product which is printed through main( ).
8. Write a function that receives marks received by a student in 3 subjects and returns the average and percentage of these marks. Call this function from main( ) and print the results in main( ).
9. Given three variables x, y, z write a function to circularly shift their values to right. In other words if x = 5, y = 8, z = 10 after circular shift y = 5, z = 8, x =10 after circular shift y = 5, z = 8 and x = 10. Call the function with variables a, b, c to circularly shift values.
10. Write a function to compute the distance between two points and use it to develop another function that will compute the area of the triangle whose vertices are A(x_1, y_1), B(x_2, y_2), and C(x_3, y_3). Use these functions to develop a function which returns a value 1 if the point (x, y) lines inside the triangle ABC, otherwise a value 0.
11. Write a function to find the binary equivalent of a given decimal integer and display it.

Answers: Self Assessment

1.	(a)	2.	(d)	3.	External or gobal		
4.	an automatic variable			5.	number, type		
6.	one-way	7.	argc, argv	8.	value, reference		
9.	stop	10.	False	11.	True	12.	False

11.21 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication, 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

Notes

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer,* Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 12: Union and Structure

Notes

CONTENTS

Objectives
Introduction
12.1 Structure Definition
12.2 Giving Values to Members
12.3 Structure Initialization
12.4 Comparison of Structure Variables
12.5 Array within Structures
12.6 Structures within Structures
12.7 Passing Structures to Functions
12.8 Structure Pointers
12.9 Union – Definition and Declaration
12.10 Accessing a Union Member
12.11 Union of Structures
12.12 Initialization of a Union Variable
12.13 Uses of Union
12.14 Use of User-defined Type Declarations
12.14.1 typedef
12.14.2 enum (Enumerated Data Type)
12.15 Differences between Union and Structure
12.16 Summary
12.17 Keywords
12.18 Self Assessment
12.19 Review Questions
12.20 Further Readings

Objectives

After studying this unit, you will be able to:

- Define structure
- Identify array within structure
- Define union and deceleration
- Explain union of structures
- Differentiate between union and structure

Notes

Introduction

As you've learned, arrays can be used to collect groups of variables of the same type. The question now is how to aggregate pieces of data that are not identically typed. The answer is that C is an easily extensible language. It can be extended by defining data types that are constructed from the fundamental types. That is you can group variables of different types with a data type called a structure.

In this unit you'll learn to use structures to collect data items that have different data types. The following topics are concerned in this unit:

1. Declaring and defining structures
2. Assigning values to structure members
3. Array within structures
4. Structure within structures
5. Passing structures to functions

12.1 Structure Definition

A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together. The structure definition creates a format that may be used to declare structure variables in a program later on.

The general format of structure definition is as follows:

```
struct tag_name
{
        data_type member1;
        data_type member2;
        - - - - - - -
        - - - - - - -
};
```

A keyword struct declares a structure to hold the details of fields of different datatypes.

At this time, no variable has actually been created. Only a format of a new data type has been defined.

Consider the following example:

```
struct addr
{
        char name [30];
        char street [20];
        char city [15];
        char state [15];
        int pincode;
};
```

Notes

The keyword struct declares a structure to hold the details of fine fields of address, namely, #name, street, city, state, pin code. The first four members are character array and fifth one is an integer.

Creating Structure Variables

The structure declaration does not actually create variables. Instead, it defines data type only. For actual use a structure variable needs to be created. This can be done in two ways:

1. Declaration using tagname anywhere in the program.



Example:
```
struct book
{
        char name [30];
        char author [25];
        float price;
}
struct book book1, book2;
```

2. It is also allowed to combine structure declaration and variable declaration in one statement.

   This declaration is given below:

```
struct person
{
char * name;
int age;
char *address;
}
p1, p2, p3;
```

   While declaring structure variables along with their definition, the use of tag_name is optional.

```
struct
{
        char *name;
        int age;
        char *address;
}
p1, p2, p3;
```

12.2 Giving Values to Members

As the members are not themselves variables they should be linked to the structure variables. The Link between a member and a variable is established using member operator '.' which is also known as dot operator.

Notes This can be explained using following example:



Example:

```
/ * Program to define a structure and assign value to members*/
   struct book
{
        char * name;
        int pages;
        char *author;
};
main( )
{
        struct book b1;
        printf ("\n Enter Values:");
        scanf ("%s %d %s", b1.name, &b1.page, b1.author);
        printf ("%s, %d, %s, b1.name, b1.page, b1.author);
}
```

12.3 Structure Initialization

A structure variable can be initialized as any other data type.

```
        main( )
{
        static struct
{
        int weight;
        float height;
}
        student = {60, 180.75};
```

This assigns the value 60 to student.weight and 180.75 student.height. There is a one-to-one correspondence between the members and their initializing values.

A structure must be declared as static if it is to be initialized inside a function (similar to arrays). The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main( )
{
        struct st_record
        {
                int weight;
                float height;
        }
```

Notes

```
        static struct st_record student1 = {60, 180.75};
        static struct st_record student2 = {53, 170.60};
        - - - - - -
        - - - - - - -
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
        struct st_record / * No static word */
        {
                int weight;
                int height;
}
        student1 = {60, 180.75};
}
main( )
{
        static struct st_record student 2 = {53, 170.60}
        - - - - - -
        - - - - - -
}
```

The initialization of individual structure members within the template is permitted. The initialization must be done only in the declaration of the actual variables.

Task What would be the output of this program?

```
main( )
{
        struct gospel
        {
            int num ;
            char mess1[50] ;
            char mess2[50] ;
        } m ;
        m.num = 1 ;
        strcpy ( m.mess1, "If all that you have is hammer" ) ;
        strcpy ( m.mess2, "Everything looks like a nail" ) ;
        /* assume that the strucure is located at address 1004 */
        printf ( "\n%u %u %u", &m.num, m.mess1, m.mess2 ) ;
}
```

Notes

12.4 Comparison of Structure Variables

Two variables of the same structure type can be compared the same way as ordinary variables.

If person1 and person2 belong to the same structure, then the following operations are valid.

Operation	Meaning
person1 = person2	Assign person2 to person1.
person1 = = person2	Compare all members of person1 and person2 and return 1 if they are equal, 0 otherwise.



Example:

```
struct class
{
        int number;
        char name [20];
        float marks;
}
main( )
{
        int x;
        static struct class student1 = {111, "Rao", 72.50};
        static struct class student2 = {222, "Reddy", 67.00};
        static class student 3;
        student 3 = student 2;
        x = ((student3.number = = student.number) && (student3.marks = =
        student2.marks)) ? 1:0;
if (x = = 1)
{
        printf ("\nStudent2 and Student2 are same \n");
        printf ("%d %s %f\n", student3.number, student3.name,
        student3.marks);
        }
        else
        printf ("\nStudent2 and Student3 are different\n");
        }
```

Output: Student2 and Student3 are same.

222 Reddy 67.000000

Arrays of Structures

Notes

The most common use of structures is in arrays of structures. To declare an array of structures, first the structure is defined then an array variable of that structure is declared. In such a declaration, each element of the array represents a structure variable.



Example: struct class student [100];

It defines an array called student which consists of 100 elements of structure named class.

An array of structures is stored inside the memory in the same way as a multi-dimensional array.



Example:

```
/ * Program to implement an array of structure * /
        struct marks
        {
                int sub1;
                int sub2;
                int sub3;
                int total;
        };
        main( )
{
        int i;
                static struct marks student [3] = {{45, 67, 81, 0}, {75,
53, 69, 0},
                {75, 53, 69, 0}, {57, 36, 71, 0}};
                static struct marks total;
                for (i = 0; i < = 2; i++)
                {
student [i].total = student [i].sub1 + student [i].sub2+student[i] sub3;
total.sub1 = total.sub1 + student [i].sub1;
total.sub2 + = student [i].sub2;
total.sub3 + = student [i].sub3;
total.total = total.total + student [i].total;
                }
printf ("STUDENT \t\t TOTAL \n");
for (i = 0; i < = 2; i++)
printf ("Student ]%d] \t \t %d \n", i+1, student [i].total);
printf ("\n SUBJECT \t\t %d \n %s \t\t %d \n %s\t\t %d", "Subject1",
total.sub1,
```

Notes

```
"Subject2", total.sub2. "Subject3", total.sub3);
printf ("\n GRAND TOTAL = \t\t %d \n", total.total.);
}
```

12.5 Array within Structures

Single or multi-dimensional arrays of type int or float can be defined as structure members.



Example:

```
struct marks
{
        int number;
        float subject [3];
}
student [2];
```

Here the member subject contains three elements, subject [0], subject [1], and subject [2]. These elements can be accessed using appropriate subscripts. For instance, the name student [1].subject [2];would refer to the marks obtained in the third subject by the second student.



Example:

```
        /* Program to implement arrays within a structure. * /
        main( )
        {
struct marks
{
        int sub [3];
        int total;
};
static struct marks student [3] = {45, 76, 81, 0, 75, 53, 69, 0, 57, 36,
71,
0};
static struct marks total;
int i, j;
for (i = 0; i < = 2; i++)
{
        student [i].total + = student [i].sub[j]
        total.sub[j] + = student [i].sub [j];
}
total.total + = student [i].sub[j];
}
total.total + = student [i].total;
```

Notes

```
}
printf ("STUDENT\t\tTOTAL \n");
}
        for (i = 0; i < = 2; i++)
        printf ("Student [%d]\t\t%d\n", i + 1, student [i].total);
        printf("SUBJECT\t\td\n")
        for (j = 0; j < = 2; j++)
        printf ("Subject %d\t\t       %d\n",  j+1, total.sub[j]);
        printf("GRAND TOTAL = \t\t%d\n", total.total);
}
```

Output:

Student	Total
Student[1]	193
Student[2]	197
Student[3]	164
Subject	Total
Subject1	177
Subject2	156
Subject3	221
GRAND TOTAL	554

Task

Point out the errors, if any, in this program:

```
main( )
{
        struct employee
        {
            char name[25] ;
            int age ;
            float bs ;
        } ;
        struct employee e ;
        strcpy ( e.name, "Hacker" ) ;
        age = 25 ;
        printf ( "\n%s %d", e.name, age ) ;
}
```

Notes

12.6 Structures within Structures

Structures within a structure means nesting of structures. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
        char name [20];
        char department [10];
        int basic_pay;
        int dearness_allowance;
        int house_rent_allowance;
        int city_allowance;
}
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. All the items related to allowance can be grouped together and declared under a sub-structure as shown below:

```
struct salary
{
        char name [2];
        char department [10];
        struct
        {
                int dearness;
                int house_rent;
                int city;
        }
        allowance;
}
employee;
```

The salary structure contains a member named allowance which itself is a structure with three members. The members contained in the inner structure, namely, dearness, house_rent and city can be referred to as:

```
        employee.allowance.dearness
        employee.allowance.house_rent
        employee.allowance.city
```

Then inner most member in a nested structure can be accessed by chaining all the concerned structure variables (from outermost to inner most) with the member using dot operator.

Notes

The following statements are invalid:

```
employee.allowance            (actual member is missing)
employee.house_rent           (inner structure variable is missing)
```

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
        struct
        {
        int dearness;
        - - - - -
        }
        allowance, arrears;
}
employee [100];
```

The inner structure has two variables, allowance and arrears. This implies that both of them have the same structure template.

A base member can be accessed as follows:

```
employee[1].allowance.dearness
employee[1].arrears.dearness
```

Tag names can also be used to define inner structures.



Example:

```
struct pay
        {
                int dearness;
                int house_rent;
                int city;
        };
        struct salary
        {
                char name [20];
                char department [10];
                struct pay allowance;
                struct pay arrears;
        }
        struct salary employee [100];
```

The pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

It is also permissible to nest more than one type of structures:

```
struct personal_record
```

Notes

```
        {
        struct name_part name;
        struct date date_of_birth;
        - - - - - -
        - - - - - -
        };
struct personal_record person1;
```

The first member of the structure is name which is of the type struct name_part. Similarly, other members have their structure types.



Task Ten floats are to be stored in memory. What would you prefer, an array or a structure?

12.7 Passing Structures to Functions

There are three methods by which the values of a structure can be transferred from one function to another.

The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables.

The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the entire structure to the called function, changes are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function.

The third approach employs a concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it.

The general format of sending a copy of a structure to the called function is:

```
function_name (structure_variable_name)
```

The called function takes the following form:

```
data_type function_name (st_name)
struct_type st_name;
{
- - - - - -
- - - - - -
return (expression);
}
```

Notes



Notes

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as struct with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type.
3. The return statement is necessary only when the function is returning some data. The expression may be any simple variable or structure or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in calling function for its type, if it is placed after the calling function.

e.g.:

```
/* Program showing passing of structure member as function
        parameters * /
struct stores
{
        char name [20];
        float price;
        int quantity;
};
main( )
{
        struct stores update( );
        float mul( ), p_increment, value;
        int q_increment;
        static struct stores item = {"XYZ", 25.75, 12};
        printf ("\nInput Increment Values:");
        printf ("\nPrice Increment and Quantity Increment\n");
        scanf ("%f %d", &p_increment, &q_increment);
        item = update item, pincrement, qincrement);
        printf ("\nUpdated values of item");
        printf ("\nName              : %s\n", item.name);
        printf ("\nPrice             : %f\n", item.price);
        printf ("\nQuantity   : %d\n", item.quantity);
        value = mul (item);
        printf ("\nValue of the item: %d\n", value);
```

Notes

```
        }
struct stores update (struct stores product, float p, int q)
{
                product.price += p;
                product.quantity += q;
                return (product);
}
float mul (struct stores stock)
{
                return (stock.price *stock.quantity);
}
```

Output:

```
Input Increment Values: Price Increment and Quantity Increment
10 12
Updated values of item
Name                    : XYZ
Price                   : 35.750000
Quantity                : 24
Value of the item       : 858.000000
```

In case of structures having numerous structure elements passing these individual elements would be tedious. In such cases an entire structure can be passed to a function.



Lab Exercise

e.g.:

```
/* Program passing entire structure as function parameter. * /
struct emp
{
        char empname [25];
        char company [25];
        int empno;
}
main( )
{
        static struct emp emp1 = {“Prashant”, SOCEM”, 101};
        display (emp1);
}
display (e)
struct emp e;
```

Notes

```
{
        printf ("%s\n%s\n%d", emp.empname, emp.company,
        emp.empno);
}
```

Output: Prashant
SOCEM
101

12.8 Structure Pointers

A complete structure can be transferred to a function by passing a structure-type pointer as an argument. In principle, this is similar to the procedure used to transfer an array to a function. However, we must use explicit pointer notation to represent a structure that is passed as an argument. A structure passed in this manner will be passed by reference rather than by value. Hence, if any of the structure members are altered within the function, the alterations will be recognized outside the function.



Example:

```
#include <stdio.h>
typedef struct
{
        char *name;
        int acct_no;
        char accttype;
        float balance;
}
record;
/* transfer a structure-type pointer to a function */
main( )
{
        void adjust (record *pt);     /* function declaration * /
        static record customer = {"Smith", 3333, 'C', 33.33};
        printf ("%s %d %c %.2f\n", customer.name, customer.acct_no,
        customer.acct_type, customer.balance);
        adjust (&customer);
        printf ("%s %d %c %.2f\n", customer.name, customer.acct_no,
        customer.acct_type, customer.balance);
}
void adjust (record *pt)
{
```

Notes

```
        pt->name = "Jones";
        pt->acct_not = 9999;
        pt->acct_type = 'R';
        pt->balance = 99.99;
        return;
}
```

This program illustrates the transfer of a structure to a function by passing the structure's address (a pointer) to the function. In particular, customer is a static structure of type record, whose members are assigned an initial set of values. These initial values are displayed when the program begins to execute. The structure's address is then passed to the function adjust where different values are assigned to the member of the structure.

Within adjust, the formal argument declaration defines pt as a pointer to a structure of type record. Also, nothing is explicitly returned from adjust to main.

Within main, the current values assigned to the members of customer are again displayed after adjust has been accessed. Thus, the program illustrates whether or not the changes made in adjust carry over to the calling portion of the program.

Executing the program results in the following output:

Smith 3333 C 33.33

Jones 9999 r 99.99

The value assigned to the members of customer within adjust are recognized within main.

A pointer to a structure can be returned from a function to the calling portion of the program. This feature may be useful when several structures are passed to a function, but only one structure is returned.

As we define a pointer pointing to int, or a pointer pointing to a char, similarly, we can have a pointer pointing to a struct. Such pointers are known as 'structure pointers'. The program given below demonstrates the usage of structure pointer.

```
main( )
{
struct emp
        {
                char empname [25];
                char company [25];
                int empno;
        };
        static struct emp emp1 = {"Prashant", "SOCEM', 101};
        struct emp *ptr;
        ptr = &emp1;
        printf ("%s %s %d\n", emp1.empname,emp1.company,emp1.empno);
        printf ("%s %s %d\n", ptr->company, ptr->empno);
}
```

Notes

In the above program, two types of operators are used to refer to structure elements:

1. Dot Operator
2. Arrow Operator

When the structure is referred to by its name, the structure elements are addressed using dot operators.



Example: `b1.name`

When the structure is referred to by the pointer to structure, the structure elements are addressed using arrow operators.



Example: `ptr->name`

On the left hand side of '.' structure operator, there must always be a structure variable, whereas on the right hand side of the '->' operator there must always be a pointer to a structure.

The following program demonstrates the passing of address of a structure variable to a function.

```
struct emp
{
        char empname [25];
        int empno;
}
main( )
{
        static struct emp emp1 = {Prashant",”socem", 101};
        display (&emp1);
}
display (e)
struct emp *e; /*pointer to a structure */
{
        printf ("%s \n%s\n%d", e->empname, e->empno);
}
```

Output: Prashant
SOCEM
101

In the above example, -> operator is used to access the structure elements using pointer to structure.

12.9 Union – Definition and Declaration

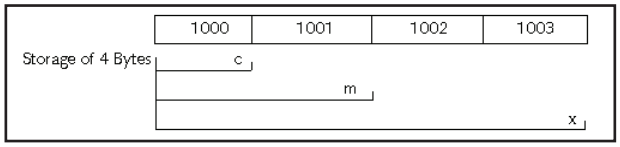
Unions follow the same syntax as structures but differ in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time.

Notes

Like structures, a union can be declared using the keyword union as follows:

```
union item
{
        int m;
        float x;
        char c;
} code;
```

This declaration declares a variable code of type union item. The union contains three members, each with a different data type. However, only one can be used at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.



The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. As shown in the example declaration, the member x requires 4 bytes which is the largest among the members. It is assumed that a float variable requires 4 bytes of storage and the figure above shows how all the three variables share the same address.

Task In an array of structures, not only are all structures stored in contiguous memory locations, but the elements of individual structures are also stored in contiguous locations. Discuss

12.10 Accessing a Union Member

To access a union member, you can use the same syntax that you use for structure members.

Example: code.m, code.x, code.c are all valid member variables.

During accessing, you should make sure that you are accessing the member whose value is currently stored.

Example: The statements such as

```
code.m  = 150;
code.x  = 785;
printf ("%d", code.m);
```

would produce erroneous output (which is machine dependent). The user must keep track of what type of information is stored at any given time.

Thus, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

12.11 Union of Structures

Notes

Just as one structure can be nested within another, a union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union. Here is an example of structures nested in a union.



Lab Exercise

```
main( )
{
      struct a
      {
              int i;
              char c[2];
      };
      struct b
      {
              int j;
              char d[2];
      };
      union z
      {
              struct a key;
              struct b data;
      }strange;
      strange.key.i = 512;
      strange.data.d[0] = 0;
      strange.data.d[1] = 32;
      printf("%d\n", strange.key.i);
      printf("%d\n", strange.data.j);
      printf("%d\n", strange.key.c[0];
      printf("%d\n", strange.data.d[0];
      printf("%d\n", strange.key.c[1]);
      printf("%d\n", strange.data.d[1];
}
```

Output: 512

512

0

0

Notes

```
32
32
```

Structures and unions may be freely mixed with arrays.



Example:

```
union id
{
        char color[12];
        int size;
};
struct clothes
{
        char manufacturer[20];
        float cost;
        union id description;
} shirt, trouser;
```

Now shirt and trouser are structure variable of type clothes. Each variable will contain the following members: a string (manufacturer), a floating-point quantity (cost), and a union (description). The union may represent either a string (color), or an integer quantity (size). Another way to declare the structure variable shirt and trouser is to combine the above two declarations. This is shown as follows:

```
struct clothes
{
        char manufacturer[20];
        float cost;
        union   {
                char color[12];
                int size;
                } description;
} shirt, trouser;
```

This declaration is more concise, though perhaps less straightforward than the original declarations.

An individual union member can be accessed in the same manner as an individual structure member, using the operators " . " and " -> ". Thus, if variable is a union variable, then variable. member refers to a member of the union. Similarly, if ptvar is a pointer variable that points to a union, then ptvar->member refers to a member of that union.



Example:

```
#include <stdio.h>
main()
{
```

Notes

```
        union id
        {
                char color;
                int size;
        };
        struct
        {
                char manufacturer[20];
                        float cost;
                union id description;
        } shirt, trouser;
        printf("%d\n", sizeof(union id));
shirt.description.color = ' w ' ; /* assigns a value to color */
printf("%c %d\n", shirt.description.color, shirt. description. size);
shirt.description.size = 12; /* assigns a value to size */
printf("%c %d\n", shirt.description.color, shirt.description. size);
}
```

12.12 Initialization of a Union Variable

A union variable can be initialized, provided its storage class is either external or static. Only one member of a union can be assigned a value at any one time. The initialization value is assigned to the first member within the union.



Example:

```
        /* Program to demonstrate initialization of union variables. */
        #include <stdio.h>
        main( )
        {
                union id
                {
                        char color[12];
                        int size;
                };
                struct clothes
                {
                        char manufacturer[20];
                        float cost;
                        union id description;
                };
```

Notes

```
static struct clothes shirt = {"American", "25.00", "White"};
printf("%d\n", sizeof(union id));
printf("%s %5.2f", shirt.manufacturer, shirt.cost);
printf("%s %d\n", shirt.description.color, shirt.description.size);
shirt.description.size = 12;
printf("%s %5.2f", shirt.manufacturer, shirt.cost);
printf("%s %d\n", shirt.description.color, shirt.description.size);
}
```

Output:

```
12
American    25.00    White    26743
American    25.00    ~        12
```



Task Write a program that compares two given dates. To store date use structure say date that contains three members namely date, month and year. If the dates are equal then display message as "Equal" otherwise "Unequal".

12.13 Uses of Union

Unions, like structures, contain members whose individual data types may differ from one another. But the members that compose a union share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory.

Unions are useful for applications involving multiple members, where values need not be assigned to all of the members at any one time. Unions are also used wherever the requirement is to access the same memory locations in more than one way. This is often required while calling Basic Input/Output System functions (often simply called BIOS routines) present in the read only memory (ROM) of the computer.

Many DOS based application software's need to access DOS internal data structures. The breakup of these internal data structures however, is not consistent and often changes from one version of DOS to another. Therefore, to make the application programs compatible with different versions of DOS, these programs create unions which take into account the variations in the breakup of these DOS data structures. These programs when executed first test the version member of DOS being used on the machine and then access the appropriate part of the union.

12.14 Use of User-defined Type Declarations

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user defined data type identifier can later be used to declare variables.

12.14.1 typedef

It takes the general form:

```
typedef type indentifier;
```

Notes

where type refers to an existing data type and "identifier" refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user defined ones. The new type is 'new' only in name, but not the data type. typedef cannot create a new type.

Some examples of the type definitions are:

```
typedef int units;                    /* units symbolizes int */

typedef float marks;                  /* marks symbolizes float */
```

Units and marks can be later used to declare variables as follows:

```
units batch1, batch2;  /* batch1 and batch2 are declared as int variable .*/

marks name1 [50], name2 [50]; /* name1 [50] and name2 [50] are declared as 50
element floating point array variables. */
```

The main advantage of typedef is that we can create meaningful data type names for increasing the readability of the program.



Example:

```
struct employee

{

        char name[30];

        int age;

        float bs;

};

     struct employee e;
```

This structure declaration can be made more handy to use when renamed using typedef as shown below:

```
struct employee

{

        char name[30];

        int age;

        float bs;

};

typedef struct employee EMP;

EMP e1, e2;

e1.age = 40;

printf ("%d", e1.age);
```

In this example, by using typedef, a long data type name is replaced by a short and suggestive data type name. Thus, by reducing the length and apparent complexity of data types, typedef can help to clarify source listing and save time and energy spent in understanding a program.

12.14.2 enum (Enumerated Data Type)

It is defined as

```
enum identifier {value 1, value 2, ... value n};
```

Notes

The identifier is a user defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants).

After this definition, we can declare variable to be of this 'new' type as below:

```
enum identifier v1, v2,...vn;
```

The enumerated variables v1, v2, --- vn can only have one of the values value1, value2, ----valuen.

The assignments v1 = value 3; v5 = value 1; are valid.



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;
```



Note The values that are in original declaration, can only be used.

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value1, is assigned 0, value2 is assigned 1, and so on.

The automatic assignments can be overridden by assigning values explicitly to the enumeration constants.



Example: `enum day {Monday =1, Tuesday, ---Sunday};`

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement.



Example: `enum day{Monday, --- Sunday}week_st, week_end;`

Like structures this declaration has two parts:

1. The first part declares the data type and specifies its possible values. These values are called 'enumerators'.
2. The second part declares variables of this data type.

12.15 Differences between Union and Structure

The differences between structure and union are:

1. Union allocates the memory equal to the maximum memory required by the member of the union but structure allocates the memory equal to the total memory required by the members.
2. In union, one block is used by all the member of the union but in case of structure, each member have their own memory space.

Notes

3. Union is best in the environment where memory is less as it shares the memory allocated. But structure can not implemented in shared memory.
4. As memory is shared,ambiguity are more in union,but less in structure.
5. Self referencial union can not be implemented in any datastructure ,but self referencial structure can be implemented.

A structure is a union in which each defined data type will have its own memory.



Example:

```
Struct{
int a;
float b;
}
```

In the above structure b has highest memory but bith a b are having personalised memory. Where as in union the highest memory data type will be memory for entire union. Suppose if a float element is initialized and an int element the int will be first located on 2 bytes and then the next 2 bytes will be the float.



Case Study

Structures are used when you want to process data of multiple data types but you still want to refer to the data as a single entity. Structures are similar to records in COBAL or Pascal.

For example, you might want to process information on students in the categories of name and marks (grade percentages). Here you can declare the structure 'student' with the fields 'name' and 'marks', and you can assign them appropriate data types. These fields are called members of the structure. A member of the structure is referred to in the form of structurename.membername.

```
Program:
struct student      \\ A
{
    char name[30];   \\ B
    float marks;     \\ C
}   student1, student2;      \\ D
main ( )
{
    struct student student3; \\ E
    char s1[30];             \\ F
    float f;                 \\ G
    scanf ("%s", name);      \\ H
    scanf (" %f", & f);  \\ I
    student1.name = s1;      \\ J
```

Contd...

Notes

```
    student2.marks = f;       \\ K
    printf (" Name is %s \n", student1.name);       \\ L
    printf (" Marks are %f \n", student2.marks);    \\ M
}
```

Explanation

1. Statement A defines the structure type student. It has two members: name and marks.
2. Statement B defines the structure member name of the type character 30.
3. Statement C defines the structure member marks of the type float.
4. Statement D defines two structure variables: structure1 and structure2. In the program you have to use variables only. Thus struct student is the data type, just as int and student1 is the variable.
5. You can define another variable, student3, by using the notations as specified in statement E.
6. You can define two local variables by using statements F and G.
7. Statement J assigns s1 to a member of the structure. The structure member is referred to as structure variablename.membername. The member student1.name is just like an ordinary string, so all the operations on the string are allowed. Similarly, statement J assigns a value to student1.marks
8. Statement L prints the marks of student1 just as an ordinary string.

Questions

1. Write a program that compares two given dates. To store date use structure say date that contains three members namely date, month and year. If the dates are equal then display message as "Equal" otherwise "Unequal".
2. There is a structure called employee that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.

12.16 Summary

- Structure is a derived data type used to store the instances of variables of different data types.
- Structure definition creates a format that may be used to declare structure variables in the program later on.
- The structure operators like dot operator "." are used to assign values to structure members.
- Structure variable can be initialized as any other data type. An array of structure can be declared as any other array. In such an array, each element is a structure. Structures may contain arrays as well as structures.
- Union is a memory location that is shared by two or more variables.

Notes

- When union variable is declared, compiler automatically allocates enough storage to hold to largest member of union. Only the unions with storage class external or static can be initialized.
- Unions are useful for applications involving multiple members. They are also used in many DOS based application softwares. typedef and enum are two user defined data types.

12.17 Keywords

Random access f le: A file, which allows accessing its records without restriction on the order of access.

Sequential f le: A file, which allows accessing its records only from the first record onwards.

Structure: A grouped data type created by user.

Structure: A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together.

Structures within structure: It means nesting of structures.

Union: A data type that allows more than one variable to share the same memory area.

User def ned data types: The way you are not creating any new data type but are referring to an existing data type by a different name. Such data types are known as user defined data types.

12.18 Self Assessment

Choose the appropriate answers:

1. A structure is a collection of variables referenced under ..... providing a convenient means of keeping related information together.
   (a) Different name
   (b) Same name
   (c) One name
   (d) Two name
2. The Link between a member and a variable is established using member operator '.' which is also known as .....
   (a) sigma operator
   (b) dot operator
   (c) increment operator
   (d) decrement operator
3. Structures within a structure means
   (a) Nesting of structures
   (b) Difference of structures
   (c) Same as structure
   (d) Array of structure

Notes

Fill in the blanks:

4. The inner structure has two variables, allowance and .....
5. A ..... to a structure can be returned from a function to the calling portion of the program.
6. Unions follow the same syntax as structures but differ in terms of .....
7. A union variable can be initialized, provided its storage class is either external or .....
8. C supports a feature known as ..... that allows users to define an identifier that would represent an existing data type.
9. ..... is a derived data type used to store the instances of variables of different data types.
10. A structure must be declared as static if it is to be initialized ..... a function.

12.19 Review Questions

1. What do you mean by 'Structure'? How it can be declared and initialized in a C program?
2. Draw a diagram to represent the internal storage of a structure.
3. What do you mean by 'Union'? How it can be declared and initialized in a C program?
4. Differentiate the followings:
   (a) Arrays and structures
   (b) Local and global structure
   (c) Array of structure and structure within array
   (d) Structure and union
5. Write short note on:
   (a) Internal storage of union
   (b) Function returning structures
   (c) Structure within a structure
6. What will be the output of the following program? Explain the same.

```
main()
{
        union a
        {
                char Achar;
                int Aint;
        } one;
        one.Achar = 'A';
        printf("Value in Achar = %c\n", one.Achar);
        printf("Value in Aint = %d\n", one.Aint);
        one.Aint += 7;
```

Notes

```
        printf("Value in Achar after modification = %c\n",    one.Achar);
        printf("Value in Aint after modification = %d\n",     one.Aint);
}
```

7. Write a program that compares two given dates. To store date use structure say date that contains three members namely date, month and year. If the dates are equal then display message as "Equal" otherwise "Unequal".
8. Explain the usefulness of structures and unions in C.
9. A record contains name of cricketer, his age, number of test matches that he has played and the average runs that he has scored in each test match. Create an array of structure to hold records of 20 such cricketer and then write a program to read these records and arrange them in ascending order by average runs. Use the qusort( ) standard library function.
10. There is a structure called employee that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.

Answers: Self Assessment

1.	(c)	2.	(b)	3.	(a)		
4.	Arrears	5.	Pointer	6.	Storage	7.	Static
8.	type definition	9.	Structure	10.	inside		

12.20 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C",* Pearson Education, Year of Publication, 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language",* Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming With C",* Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer,* Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 13: File Handling in C

CONTENTS

Objectives

Introduction

13.1 What is a File?

13.2 Defining and Opening a File

13.2.1 Filename

13.2.2 Data Structure

13.3 Closing a File

13.4 Input/Output Operations on Files

13.4.1 getc & putc Functions

13.4.2 getw & putw Functions

13.4.3 fprintf & fscanf Functions

13.4.4 feof( ) Function

13.5 Errors during Input/Output

13.6 Functions for Random Access to Files

13.6.1 ftell( ) Function

13.6.2 rewind( ) Function

13.6.3 fseek( ) Function

13.7 Summary

13.8 Keywords

13.9 Self Assessment

13.10 Review Questions

13.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Know how to define and opening a file
- Perform input/output operation on files
- Identify errors during input/output
- State the functions for random access to files

Notes

Introduction

Storage of data in variables and arrays is temporary. All such data is lost when a program terminates. Files are used for permanent retention of large amounts of data. Computer stores files on secondary storage devices, especially disk storage devices. In this unit, we explain how data files are edited, updated and processed by C programs.

File is a collection of data or set of characters may be a text or program. Basically there are two types of files used in the C language: sequential file and random access file. The sequential files are very easy to create than random access files. The data or text will be stored or read back sequentially. In random access file, the data can be accessed and processed randomly.

13.1 What is a File?

Wherever there is a need to handle large volumes of data, it is advantageous to store data on the disks and read whenever necessary. This method employs the concept of files to store data. A file is a place on disk where a group of related data is stored. C supports a number of functions that have the ability to perform basic file operations, which include:

1. Naming a file
2. Opening a file
3. Reading data from a file
4. Writing data to a file
5. Closing a file

There are two distinct ways to perform file operations in C.

1. Low level I/O Operation (It uses operating system calls)
2. High level I/O Operation (It uses functions in C's Standard I/O library)

Table 13.1: List of High Level I/O Functions

Function Name	Operation
fopen()	Creates a new file for use or opens an existing file for use.
fclose()	Closes a file which has been opened for use.
getc()	Reads a character from a file.
putc()	Writes a character to a file.
fprintf()	Writes a set of data values to a file.
fscanf()	Reads a set of data values from a file.
getw()	Reads an integer from a file.
putw()	Writes an integer to a file.

13.2 Defining and Opening a File

Before storing data in a file in the secondary memory, certain things about the file must be specified to the operating system. These include:

1. Filename
2. Data Structure
3. Purpose

Notes

13.2.1 Filename

It is a string of characters that makes up a valid filename for an operating system. It may contain two parts, a primary name and an optional period with an extension.



Example:

Input.dat

Store

PROG.C

Student.C

Text.out



Task If a file contains the line "I am a boy\r\n" then on reading this line into the array str[ ] using fgets( ) what would str[ ] contain?

13.2.2 Data Structure

Data structure of a file is defined as file in the library of standard I/O function definitions. All files should be declared as of type file before they are used.

Purpose: When we open a file, we must specify what we want to do with the file.

Following is the general format for declaring and opening a file:

```
File    *fp;

fp = fopen ( "filename", "mode");
```

The first statement declares the variable fp as a "pointer to the data type file".

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 a 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:

r Opens the file for reading only.

w Opens the file for writing only.

a Opens the file for appending (or adding) data to it.

Both the filename and mode are specified as string. They should be enclosed in double quotation marks.

Depending on the mode specified, one of the following actions may be performed:

1. When the mode is 'writing', a file with the specified name is created, if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.

Notes

3. If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe, otherwise an error occurs.

Other additional modes of operation are:

r+ The existing file is opened from the beginning for both reading and writing.

w+ Same as w except both for reading and writing.

a+ Same as a except both for reading and writing.

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.



Example:

```
if (fp == NULL)
printf ("File could not be opened.\n");
```

13.3 Closing a File

Once all the operations on a file have been completed, the file is closed. This is done to clear the buffers and flush all the information associated with the file. It also prevents any accidental misuse of the file. In case there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. When there is a need to use a file in a different mode, the file has to be first closed and then reopened in a different mode.

The I/O library supports a function for this of the following form:

```
fclose (file_pointer);
```

This would close the file associated with the file pointer file_pointer.



Example:

.....

```
FILE *p1, *p2;
p1 = fopen ("INPUT", "w");
p2 = fopen ("OUTPUT", "r");
.........
fclose(p1);
fclose(p2);
```

This program opens two files and closes them after all operations on them are completed.

Once a file is closed, its file pointer can be reused for another file. All files are closed automatically whenever a program terminates. However, closing a file as soon as all operations related to it have been completed is a good programming habit.

Notes

13.4 Input/Output Operations on Files

13.4.1 getc & putc Functions

These are analogous to getchar and putchar functions and can handle only one character at a time.

putc can be used to write a character in a file opened in write mode.

A statement like putc (ch, fp1); writes the character contained in the character variable ch to the file associated with file pointer fp1.

Similarly, getc is used to read a character from a file that has been opened in read mode.

The statement c = getc(fp2); would read a character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of getc or putc. The getc will return an end-of-file marker EOF, when end of the file has been reached. The reading should be terminated when EOF is encountered. Testing for the end-of-file condition is important. Any attempt to read past the end of file might either cause the program to terminate with an error or result in an infinite loop situation.

13.4.2 getw & putw Functions

The getw and putw are integer-oriented functions. They are similar to the getc and putc functions and are used to read and write integer values on unix systems.

The general forms of getw and putw are:

```
putw (integer, fp); & getw (fp);
```

13.4.3 fprintf & fscanf Functions

The functions fprintf and fscanf perform I/O operations that are identical to the familiar printf and scanf functions.

The general syntax of fprintf is

```
fprintf (fp, "control string", list);
```

where fp is a file pointer associated with a file that has been opened for writing. The control string contains output specifications for items in the list. The list may include variables, constants and strings.



Example: fprintf (f1,"%s %d %f", name, age, 7.5); here name is an array variable of type char and age in an int variable.

The general syntax of fscanf is

```
fscanf (fp, "control string", list);
```

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the control string.



Example: `fscanf (f2, "%s %d", item, &quantity);`

fscanf also returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF.

Notes

13.4.4 feof( ) Function

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 of the data from the specified file has been read, and returns zero otherwise. If fp is a pointer to the 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.



Task Point out the errors, if any, in the following programs:

```
#include "stdio.h"
main( )
{
        FILE *fp ;
        openfile ( "Myfile.txt", fp ) ;
        if ( fp == NULL )
            printf ( "Unable to open file…" ) ;
}
openfile ( char *fn, FILE **f )
{
        *f = fopen ( fn, "r" ) ;
}
```

13.5 Errors during Input/Output

It is possible that an error may occur during Input/Output operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

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 upto that point, during processing. It returns zero otherwise.

Notes

The statement

```
if (ferror (fp)! = 0)

printf ("An error has occurred. \n");
```

would print the error message, if the reading is not successful.

13.6 Functions for Random Access to Files

To randomly access only a particular part of a file, the following functions are provided in C.

1. ftell
2. rewind
3. fseek

13.6.1 ftell( ) Function

ftell takes a file pointer and returns a number of type long that corresponds to the current position.

This function is useful in saving the current position of a file, which can be used later in the program.

It is used as follows:

```
n = ftell (fp);
```

n would give the relative offset (in bytes) of the current position. This means that n bytes have already been read (or written).

13.6.2 rewind( ) Function

rewind takes a file pointer and resets the position to the start of the file.

For example, the statements

```
rewind (fp);

n = ftell (fp);
```

would assign 0 to n because the file position has been set to the start of the file by rewind.

This function helps us in reading a file more than once, without having to close and open the file.



Note The first byte in the file is numbered as 0, second as 1, and so on. Whenever a file is opened for reading or writing, a rewind is done implicitly.

13.6.3 fseek( ) Function

fseek function is used to move the file position to a desired location within the file. Its syntax is fseek (fileptr, offset, position).

Notes

1. fileptr is a pointer to the file concerned.
2. offset is a number or variable of type long. It specifies the number of positions (bytes) to be moved from the location specified by position.
3. position is an integer number. It can take one of the following three values:

Value	Meaning
0	Beginning of file
1	Current Position
2	End of file

The offset may be positive to move forwards, or negative to move backwards.

The following examples illustrate the operation of the fseek function:

Statement	Meaning
fseek (fp, OL, 0);	Go to the beginning (Similar to rewind)
fseek (fp, OL, 1);	Stay at the current position (Rarely used)
fseek (fp, OL, 2);	Go to the end of the file, past the last character of the file
fseek (fp, x, 0);	Move to (x+1) th byte in the file
fseek (fp, x, 1);	Go forward by x bytes
fseek (fp, -x, 1);	Go backwards by x bytes from the current position



Task Write a program to makes use of the library functions getc and putchar to read and display the data..



Examples:

1. In this example, the text is read into the computer character-by-character using the getchar function and then written out to a data file character-by-character using putc.

```
#include <ctype.h>

#include <stdio.h>

/* read in a line of lower case text and store its upper case equivalent
within a data file */

main( )

{

        FILE *fpt;              /* define a pointer to pre-defined structure
type FILE */

        char c;

        /* open a new data file for writing only */
```

Notes

```
        fpt = fopen("sample.dat", "w");
        /* read each character and write its upper case equivalent to the
data file
    */
        do
        putc (toupper ( c = getchar() ), fpt);
        while (c != '\n');
        fclose (fpt);          /* close the data file */
    }
```

(a) A data file that has been created in this manner can be viewed in several different ways.

For example, the data file can be viewed directly, using an operating system command such as print or type.

(b) Another approach is to write a program that will read the data file and display its contents. Such a program will, in a sense, be a mirror image of the one described above, i.e., the library function getc will read the individual characters from the data file and putchar will display them on the screen.

2. The following program will read a line of text from a data file character-by-character and display the text on the screen. The program makes use of the library functions getc and putchar to read and display the data. It complements the program presented in the previous example.

```
#include <stdio.h>
#define NULL 0
/* read a line of text  from a data file and display it on the screen */
main()
{
        FILE *fpt; /* define a pointer to pre-defined structure type FILE
*/
        char c;
        /* open the data file for reading only */
        if ((fpt = fopen("sample.dat", "r")) == NULL)
                printf("\nERROR - Cannot open the designated file\n");
        else    /* read and display each character from the data file */
        do
                putchar(c = getc(fpt));
                while (c != '\n');
        /* close the data file */
        fclose(fpt);
}
```

Notes

(a) The logic is directly analogous to that of the program shown in previous example. However, this program opens the data file sample.dat as a read-only file. An error message is generated if sample.dat cannot be opened.

getc requires that the stream pointer fpt be specified as an argument.

(b) Data files consisting entirely of strings can often be created and read more easily with programs that utilize special string-oriented library functions. Some commonly used functions of this type are gets, puts, fgets and fputs. The functions gets and puts read or write strings to or from the standard output devices, whereas fgets and fputs exchange strings with data files.



Case Study

A direct access file is a file in which any single component may be accessed at random. Every component has a key value associated with it. A write operation takes a component and its key value, writes the component into the file, and stores both the key and the location of the record in a file, an index. A read operation takes the key of the desired component, searches the index to find the location of the component, and retrieves the component from the file.

Program:

A complete C program implementing a direct access file is given below:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 50
typedef struct
{
    char name[10];
    int key;
}   file_record;
/* this function adds the relative address to the index for a key */
void create_index(long index[], int key, long rel_add )
{
    index[key] = rel_add;
}
/* this function writes a record to the file */
void write_rec(FILE *fp, file_record rec)
{
    fwrite(&rec,sizeof(rec),1,fp);
}
```

Contd...

Notes

```
void main()
{
    long rel_add;
    int key;
    file_record frec;
    long index[MAX];/* an index list*/
    int n,i;
    FILE *recfile=NULL,*ifile=NULL;
    /* this initializes the index list to all -1 */
    for(i=0; i< MAX; i++)
        index[i]= (-1);

    recfile=fopen("mfile","w");
    if(recfile == NULL)
    {
        printf("Error in opening file mfile\n");
        exit(0);
    }
    rel_add = 0 ;
    do
    {
        printf(
" Enter the data value and the key of the record to be added to file mfile\n");
        scanf("%s %d",frec.name,&frec.key);
        while(index[frec.key] != (-1))
        {
              printf(
" A record with this key value already exist in a file enter record key value\n");
              scanf("%s %d",frec.name,&frec.key);
        }
        create_index(index,frec.key,rel_add);
        write_rec(recfile,frec);
        rel_add = ftell(recfile);
        /* this sets the relative address for the next record to be
       the value of current file position pointer in bytes from
       the beginning of the file */
        printf("Enter 1 to continue adding records to the file\n");
```

Contd...

Notes

```
        scanf("%d",&n);
    }while(n == 1);
    ifile=fopen("index_file","w");
    if(ifile == NULL)
    {
        printf("Error in opening file index_file\n");
        exit(0);
    }
   fwrite(index,sizeof(index),1,ifile);/*writes the complete index into the
index_file */
    fclose(recfile);
    fclose(ifile);
    printf("Enter 1 if you want to retrieve a record\n");
    scanf("%d",&n);
    if( n == 1)
    {
        ifile=fopen("index_file","r");
               if(ifile == NULL)
               {
                      printf("Error in opening file index_file\n");
                      exit(0);
               }
               fread(index,sizeof(index),1,ifile);
               /*reads the complete index into the index list from the
        index_file*/
               fclose(ifile);
          recfile=fopen("mfile","r");
          if(recfile == NULL)
               {
                      printf("Error in opening file mfile\n");
                      exit(0);
               }
    }
    printf("THE CONTENTS OF FILE IS \n");
    while( (fread(&frec,sizeof(frec),1,recfile)) != 0)
    printf("%s %d\n",frec.name,frec.key);
    do
    {
```

Contd...

Notes

```
        printf("Enter the key of the record to be retrieved\n");
        scanf("%d",&key);
         rel_add = index[key]; /* gets the relative address of the record
from index list */
        if( (fseek(recfile,rel_add,SEEK_SET))!= 0)
        {
               printf("Error\n");
               exit(0);
        }
        fread(&frec,sizeof(frec),1,recfile);
        printf("The data value of the retrieved record is %s\n",
        frec.name);
        printf("Enter 1 if you want to retrieve a record\n");
        scanf("%d",&n);
    } while(n == 1);
  fclose(recfile);
}
```

Explanation

1. This program writes the names in the file. A unique integer value is assigned to every name as a key value.
2. The program takes the name to be stored in the file along with its key value, writes the name and key value in the file, obtains the relative address of that record, and stores it in a list called an index. The index is organized by key value.
3. When the addition process ends, it writes the complete index into an index file.
4. When retrieval of names is requested, the following occurs:
   (a) The complete index is loaded into a list index from the index file.
   (b) The index file is used to find the relative address of the record whose key value is given.
   (c) The current position pointer is moved to that address.
   (d) The record is read from the file.

13.7 Summary

- File is a collection of data or set of characters may be a text or program.
- There are two types of files used in the C language: sequential file and random access file.
- While creating a file, fopen() function is used which opens a stream for use and link it with program and file. fopen() function has two string arguments which represent the name of the file and the type of I/O to be performed. fclose() function closes a stream that was opened by a call to fopen(). putc() is used to transfer one character into file. getc() functions allows you to read data from a file.

Notes

- Similarly to read or write strings, fgets and fputs functions are used.
- The fgets() function reads a string from the file and copies it in a string variable lying in memory.
- fputs() is used to write a string in a data file.
- getw() function is used to read an integer from a file.
- putw() is used to write as integer value in a file. fprintf() and fscanf() are similar to printf() and scanf().

13.8 Keywords

Data Structure: Data structure of a file is defined as file in the library of standard I/O function definitions.

NULL: A system-defined value (not 0) that indicates various exceptional conditions such as end of a string, a pointer referencing nothing, etc.

Random access f le: A file, which allows accessing its records without restriction on the order of access. You can access 60th record without accessing the 59th record, and so on.

Sequential f le: A file, which allows accessing its records only from the first record onwards. You cannot access 60th record without accessing the 59th record, and so on.

stderr: The default output stream where the errors are reported, usually the monitor.

stdin: The default input stream, usually the keyboard.

stdout: The default output stream, usually the monitor.

13.9 Self Assessment

Choose the appropriate answers:

1. File is a collection of
   (a) Data and set of characters
   (b) Data and set of text
   (c) Data and program
   (d) All of the above
2. Which one is not the basic operation of files?
   (a) Opening a file
   (b) Reading data from a book
   (c) Writing data to a file
   (d) Closing a file

Fill in the blanks:

3. Low level I/O operations in C use ..... calls.
4. All files should be declared as of type ..... before they are used.

Notes

5. The pointer which contains all the information about the file is used as a communication link between ..... and .....
6. ..... function can be used to test for an end of file condition.

State whether the following statements are true or false:

7. ferror() function returns a zero when it encounters an error.
8. rewind() sets the file pointer to its beginning.
9. getw and putw are character-oriented function.

13.10 Review Questions

1. What do you mean by the "File Handling"? Explain the concept of file handling in C.
2. Explain the various type of files and their access mechanisms.
3. Write in detail about any five file-handling functions in C.
4. What do mean by random file access? How C implements the concept of random file access?
5. Differentiate the followings:
   (a) stdin and stdout
   (b) fgets() and fputs()
   (c) fseek() and ftell()
   (d) getw() and getc()
6. Write a program to read a file and display contents with its line numbers.
7. Write a program to copy one file to another. While doing so replace all lowercase characters to their equivalent uppercase characters.
8. Write a program to read a text file "EXCEL.TXT" consisting of a maximum of 90 lines of text, each line with a maximum of 110 characters.
9. Write a program to find the size of a text file without traversing it character by character.
10. Suppose a file contains student's records with each record containing name and age of a student. Write a program to read these records and display them in sorted order by name.
11. Write a program that merges lines alternately from two files and writes the results to new file. If one file has less number of lines than the other, the remaining lines from the larger file should be simply copied into the target file.
12. Write a program to add the contents of one file at the end of another.

Answers: Self Assessment

1.	(d)	2.	(b)	3.	operating system	4.	FILE
5.	system, program	6.	feof()	7.	False	8.	True
9.	False						

13.11 Further Readings

Notes



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication, 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming with C*", Tata McGraw Hill Publishing Company Limited, New Delhi

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer,* Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 14: Additional in C

CONTENTS

Objectives

Introduction

14.1 Dynamic Memory Allocation

14.1.1 malloc, sizeof, and free

14.1.2 calloc and realloc

14.2 Memory Models

14.3 Concept of Linked Lists

14.4 Representation of Linked List

14.5 Inserting a Node using Recursive Programs

14.6 Deleting the Specified Node in Singly Linked List

14.7 Inserting a Node after the Specified Node in a Singly Linked List

14.8 Circular Linked List

14.9 Summary

14.10 Keywords

14.11 Self Assessment

14.12 Review Questions

14.13 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain dynamic memory allocation
- Discuss the concept of linked lists
- Know how to insert a node using recursive programs
- Realise how to delete the specified node in singly linked list

Introduction

As Static representation of linear ordered list through Array leads to wastage of memory and in some cases overflows. Now we don't want to assign memory to any linear list in advance instead we want to allocate memory to elements as they are inserted in list. This requires Dynamic Allocation of memory.

Notes

14.1 Dynamic Memory Allocation

Dynamic allocation is a pretty unique feature to C (amongst high level languages). It enables us to create data types and structures of any size and length to suit our programs need within the program.

14.1.1 malloc, sizeof, and free

The Function malloc is most commonly used to attempt to "grab'' a continuous portion of memory. It is defined by:

```
void *malloc(size_t number_of_bytes)
```

That is to say it returns a pointer of type void * that is the start in memory of the reserved portion of size number_of_bytes. If memory cannot be allocated a NULL pointer is returned.

Since a void * is returned the C standard states that this pointer can be converted to any type. The size_t argument type is defined in stdlib.h and is an unsigned type.

So:

```
    char *cp;
                cp = malloc(100);
```

attempts to get 100 bytes and assigns the start address to cp.

Also it is usual to use the sizeof() function to specify the number of bytes:

```
int *ip;
         ip = (int *) malloc(100*sizeof(int));
```

Some C compilers may require to cast the type of conversion. The (int *) means coercion to an integer pointer. Coercion to the correct pointer type is very important to ensure pointer arithmetic is performed correctly.

It is good practice to use sizeof() even if you know the actual size you want – it makes for device independent (portable) code.

sizeof can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function.

So:

```
   int i;
         struct COORD {float x,y,z};
         typedef struct COORD PT;
         sizeof(int), sizeof(i),
         sizeof(struct COORD) and
         sizeof(PT) are all ACCEPTABLE
```

In the above, we can use the link between pointers and arrays to treat the reserved memory like an array, i.e, we can do things like:

```
ip[0] = 100;
```

or

```
   for(i=0;i<100;++i) scanf("%d",ip++);
```

Notes

When you have finished using a portion of memory you should always free() it. This allows the memory freed to be available again, possibly for further malloc() calls.

The function free() takes a pointer as an argument and frees the memory to which the pointer refers.


Task

Write a program to print the elements of the linked list.

14.1.2 calloc and realloc

There are two additional memory allocation functions, calloc() and realloc(). Their prototypes are given below:

```
void *calloc(size_t num_elements, size_t element_size);

void *realloc( void *ptr, size_t new_size);
```

malloc does not initialise memory (to zero) in any way. If you wish to initialise memory then use calloc. calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between calloc and malloc in that calloc takes the number of desired elements, num_elements, and element_size, element_size, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

```
int *ip;

        ip = (int *) calloc(100, sizeof(int));
```

realloc is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged.

If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You must use this new value. If new memory cannot be reallocated then realloc returns NULL.

Thus to change the size of memory allocated to the *ip pointer above to an array block of 50 integers instead of 100, simply do:

```
   ip = (int *) calloc( ip, 50);
```

C language requires the number of elements in an array to be specified at compile time. But it is not practically possible with arrays. In arrays we allocate the memory first and then start using it. This may result in failure of a program or wastage of memory space.

The concept of dynamic memory location can be used to eradicate this problem. In this technique, the allocation of memory is done at run time. C language provides four library functions known as memory management functions that can be used for allocating and freeing memory during program execution. These functions help us to build complex application programs that use the available memory intelligently.

Notes

Table 14.1: Dynamic Memory Management Functions

Function	Task
malloc	allocates memory and return a pointer to the first byte of allocated space. e.g., ptr = (cast.type*) malloc (byte_size);
calloc	allocates the memory spaces, initializes them to zero and returns pointer to first byte. e.g., ptr = (cast_type*) calloc (n, elem_size);
free	frees previously allocated space. e.g., free (ptr);
realloc	modifies the size of previously assigned space. e.g., ptr = realloc (ptr, newsize);

14.2 Memory Models

Memory models in the C programming language are a way to specify assumptions that the compiler should make when generating code for segmented memory or paged memory platforms.



Example: On the 16-bit x86 platform, six memory models exist. They control what assumptions are made regarding the segment registers, and the default size of pointers.

Memory Segmentation

Four registers are used to refer to four segments on the 16-bit x86 segmented memory architecture. DS (data segment), CS (code segment), SS (stack segment), and ES (extra segment). A logical address on this platform is written segment:offset, in hexadecimal. In real mode, in order to calculate the physical address of a byte of memory, one left-shifts the contents of the appropriate register 4 bits, and then adds the offset.



Example: The logical address 7522:F139 yields the 20-bit physical address:

75220 + F139 = 84359

Note that this process leads to aliasing of memory, such that any given physical address may have multiple logical representations. This makes comparison of pointers difficult.

In protected mode, the GDT and LDT are used for this purpose.

Pointer Sizes

Pointers can either be near, far, or huge. Near pointers refer to the current segment, so neither DS nor CS must be modified to dereference the pointer. They are the fastest pointers, but are limited to point to 64 kilobytes of memory (the current segment).

Far pointers contain the new value of DS or CS within them. To use them the register must be changed, the memory dereferenced, and then the register restored. They may reference up to 1 megabyte of memory. Note that pointer arithmetic (addition and subtraction) does not modify

Notes

the segment portion of the pointer, only its offset. Operations which exceed the bounds of zero of 65355 (0xFFFF) will undergo modulo 64K operation just as any normal 16 bit operation.



Example: The code below will wrap around and overwrite itself:

```
char far* myfarptr = (char far*) 0x50000000L;

unsigned long counter;

for(counter=0; counter<128*1024; counter++) // access 128K memory

*(ptr+counter) = 7; // write all 7s into it
```

The moment counter becomes (0x10000), the resulting absolute address will roll over to 0x5000:0000.

Huge pointers are essentially far pointers, but are normalized every time they are modified so that they have the highest possible segment for that address. This is very slow but allows the pointer to point to multiple segments, and allows for accurate pointer comparisons, as if the platform were a flat memory model.

Memory Models

The memory models are:

Model	Data	Code
Small	near	near
Medium	near	far
Compact	far	near
Large	far	far
Huge	huge	huge
Tiny*	near	near

* In the Tiny model, all four segment registers point to the same segment. In all models with near data pointers, SS equals DS.



Task Malloc does not initialise memory in any way. If you wish to initialise memory which function to use?

14.3 Concept of Linked Lists

An array is represented in memory using sequential mapping, which has the property that elements are fixed distance apart. But this has the following disadvantage: It makes insertion or deletion at any arbitrary position in an array a costly operation, because this involves the movement of some of the existing elements.

When we want to represent several lists by using arrays of varying size, either we have to represent each list using a separate array of maximum size or we have to represent each of the lists using one single array. The first one will lead to wastage of storage, and the second will involve a lot of data movement.

So we have to use an alternative representation to overcome these disadvantages. One alternative is a linked representation. In a linked representation, it is not necessary that the elements be at a fixed distance apart. Instead, we can place elements anywhere in memory, but to make it a part of the same list, an element is required to be linked with a previous element of the list. This can be done by storing the address of the next element in the previous element itself. This requires

Notes

that every element be capable of holding the data as well as the address of the next element. Thus every element must be a structure with a minimum of two fields, one for holding the data value, which we call a data field, and the other for holding the address of the next element, which we call link field

Therefore, a linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.



Lab Exercise Program: Here is a program for building and printing the elements of the linked list:

```
# include <stdio.h>
   # include <stdlib.h>
   struct node
   {
   int data;
   struct node *link;
   };
   struct node *insert(struct node *p, int n)
   {
   struct node *temp;
   /* if the existing list is empty then insert a new node as the
starting node */
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node)); /* creates new node
data value passes
   as parameter */
         if(p==NULL)
         {
      printf("Error\n");
            exit(0);
         }
         p-> data = n;
         p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
      }
      else
      {
      temp = p;
```

Notes

```
/* traverses the existing list to get the pointer to the last node of
it */
   while (temp-> link != p)
      temp = temp-> link;
         temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using
          data value passes as
            parameter and puts its
          address in the link field
          of last node of the
          existing list*/
        if(temp -> link == NULL)
        {
      printf("Error\n");
           exit(0);
        }
        temp = temp-> link;
        temp-> data = n;
        temp-> link = p;
       }
       return (p);
}
void printlist ( struct node *p )
{
struct node *temp;
 temp = p;
printf("The data values in the list are\n");
   if(p!= NULL)
   {
   do
         {
         printf("%d\t",temp->data);
         temp=temp->link;
         } while (temp!= p);
   }
   else
      printf("The list is empty\n");
}
```

Notes

```
void main()
{
   int n;
   int x;
   struct node *start = NULL ;
   printf("Enter the nodes to be created \n");
   scanf("%d",&n);
   while ( n -- > 0 )
   {
printf( "Enter the data values to be placed in a node\n");
      scanf("%d",&x);
      start = insert ( start, x );
   }
   printf("The created list is\n");
   printlist ( start );
}
```

This program uses a strategy of inserting a node in an existing list to get the list created. An insert function is used for this. The insert function takes a pointer to an existing list as the first parameter, and a data value with which the new node is to be created as a second parameter, creates a new node by using the data value, appends it to the end of the list, and returns a pointer to the first node of the list. Initially the list is empty, so the pointer to the starting node is NULL. Therefore, when insert is called first time, the new node created by the insert becomes the start node. Subsequently, the insert traverses the list to get the pointer to the last node of the existing list, and puts the address of the newly created node in the link field of the last node, thereby appending the new node to the existing list. The main function reads the value of the number of nodes in the list. Calls iterate that many times by going in a while loop to create the links with the specified number of nodes.

Advantages of Linked Lists

The various advantages of linked list are:

1. They can grow or shrink in size during the execution of a program.
2. They do not waste memory space.
3. They provide flexibility in allowing the items to be rearranged efficiently.

Basic List Operations

We can perform the following basic operations on the linked lists:

1. Creating the list
2. Traversing the list
3. Printing the list
4. Deleting a node
5. Concatenating two lists

Notes

14.4 Representation of Linked List

Because each node of an element contains two parts, we have to represent each node through a structure.

While defining linked list we must have recursive definitions:

```
struct node
{
        int data;
        struct node * link;
}
```

Here, link is a pointer of struct node type i.e. it can hold the address of variable of struct node type. Pointers permit the referencing of structures in a uniform way, regardless of the organization of the structure being referenced. Pointers are capable of representing a much more complex relationship between elements of a structure than a linear order.

Initialization:

```
main()
{
        struct node *p, *list, *temp;
        list = p = temp = NULL;
        .
        .
        .
}
```

14.5 Inserting a Node using Recursive Programs

A linked list is a recursive data structure. A recursive data structure is a data structure that has the same form regardless of the size of the data. You can easily write recursive programs for such data structures.



Lab Exercise

```
Program:
# include <stdio.h>
   # include <stdlib.h>
   struct node
   {
   int data;
   struct node *link;
   };
   struct node *insert(struct node *p, int n)
   {
```

Notes

```
      struct node *temp;
      if(p==NULL)
      {
         p=(struct node *)malloc(sizeof(struct node));
         if(p==NULL)
         {
      printf("Error\n");
             exit(0);
      }
      p-> data = n;
      p-> link = NULL;
   }
   else
         p->link = insert(p->link,n);/* the while loop replaced by
recursive call */
      return (p);
   }
   void printlist ( struct node *p )
   {
         printf("The data values in the list are\n");
         while (p!= NULL)
         {
      printf("%d\t",p-> data);
            p = p-> link;
         }
   }
   void main()
   {
         int n;
         int x;
         struct node *start = NULL ;
         printf("Enter the nodes to be created \n");
         scanf("%d",&n);
         while ( n- > 0 )
         {
      printf( "Enter the data values to be placed in a node\n");
            scanf("%d",&x);
            start = insert ( start, x );
         }
```

Notes

```
        printf("The created list is\n");
        printlist ( start );
    }
```

This recursive version also uses a strategy of inserting a node in an existing list to create the list. An insert function is used to create the list. The insert function takes a pointer to an existing list as the first parameter, and a data value with which the new node is to be created as the second parameter. It creates the new node by using the data value, then appends it to the end of the list. It then returns a pointer to the first node of the list. Initially, the list is empty, so the pointer to the starting node is NULL. Therefore, when insert is called the first time, the new node created by the insert function becomes the start node. Subsequently, the insert function traverses the list by recursively calling itself. The recursion terminates when it creates a new node with the supplied data value and appends it to the end of the list.



Task Write a program to delete a node using recursive program.

14.6 Deleting the Specified Node in Singly Linked List

To delete a node, first we determine the node number to be deleted (this is based on the assumption that the nodes of the list are numbered serially from 1 to n). The list is then traversed to get a pointer to the node whose number is given, as well as a pointer to a node that appears before the node to be deleted. Then the link field of the node that appears before the node to be deleted is made to point to the node that appears after the node to be deleted, and the node to be deleted is freed. Figures 14.1 and 14.2 show the list before and after deletion, respectively.



Lab Exercise

Program:

```
# include <stdio.h>
# include <stdlib.h>
struct node *delet ( struct node *, int );
int length ( struct node * );
struct node
{
   int data;
   struct node *link;
};
struct node *insert(struct node *p, int n)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
```

Notes

```
      if(p==NULL)
      {
            printf("Error\n");
         exit(0);
      }
      p-> data = n;
      p-> link = NULL;
   }
   else
   {
      temp = p;
      while (temp-> link != NULL)
       temp = temp-> link;
       temp-> link = (struct node *)malloc(sizeof(struct node));
      if(temp -> link == NULL)
      {
            printf("Error\n");
         exit(0);
      }
      temp = temp-> link;
      temp-> data = n;
      temp-> link = NULL;
      }
      return (p);
}

void printlist ( struct node *p )
{
      printf("The data values in the list are\n");
      while (p!= NULL)
      {
            printf("%d\t",p-> data);
         p = p-> link;
      }
}

void main()
{
```

Notes

```
    int n;
    int x;
    struct node *start = NULL;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n- > 0 )
    {
        printf( "Enter the data values to be placed in a node\n");
       scanf("%d",&x);
       start = insert ( start, x );
    }
    printf(" The list before deletion id\n");
    printlist ( start );
    printf("% \n Enter the node no \n");
    scanf ( " %d",&n);
    start = delet (start , n );
    printf(" The list after deletion is\n");
    printlist ( start );
}
    /* a function to delete the specified node*/
struct node *delet ( struct node *p, int node_no )
{
    struct node *prev, *curr ;
    int i;

    if (p == NULL )
    {
       printf("There is no node to be deleted \n");
    }
    else
    {
       if ( node_no > length (p))
       {
             printf("Error\n");
       }
       else
       {
             prev = NULL;
```

Notes

```
            curr = p;
            i = 1 ;
            while ( i < node_no )
            {
                  prev = curr;
                  curr = curr-> link;
                  i = i+1;
            }
            if ( prev == NULL )
            {
                  p = curr -> link;
                  free ( curr );
            }
            else
            {
               prev -> link = curr -> link ;
               free ( curr );
            }
      }
   }
   return(p);
}
/* a function to compute the length of a linked list */
int length ( struct node *p )
{
   int count = 0 ;
   while ( p != NULL )
   {
      count++;
      p = p->link;
   }
   return ( count ) ;
}
```

Notes

Figure 14.1: Before Deletion

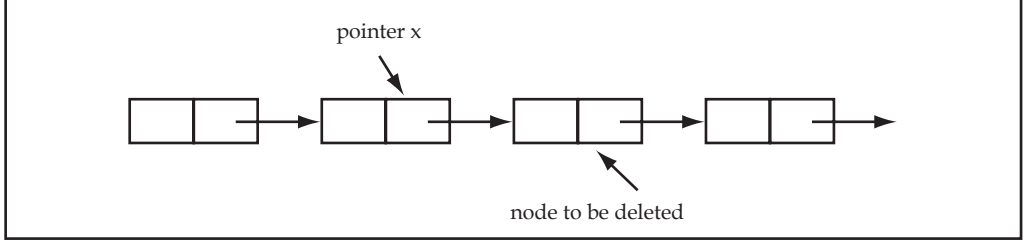
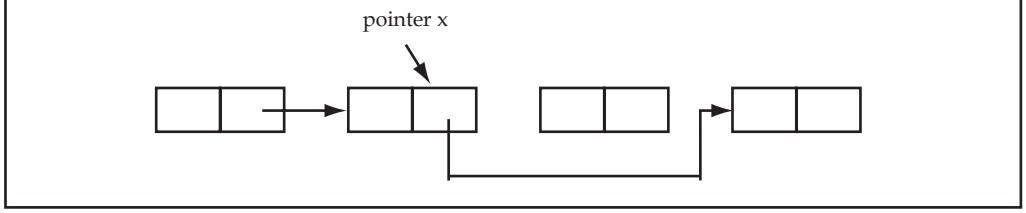


Figure 14.2: After Deletion



14.7 Inserting a Node after the Specified Node in a Singly Linked List

To insert a new node after the specified node, first we get the number of the node in an existing list after which the new node is to be inserted. This is based on the assumption that the nodes of the list are numbered serially from 1 to n. The list is then traversed to get a pointer to the node, whose number is given. If this pointer is x, then the link field of the new node is made to point to the node pointed to by x, and the link field of the node pointed to by x is made to point to the new node. Figures 14.3 and 14.4 show the list before and after the insertion of the node, respectively.



Lab Exercise

```
Program:
# include <stdio.h>
   # include <stdlib.h>
   int length ( struct node * );
   struct node
   {
      int data;
      struct node *link;
   };

   /* a function which appends a new node to an existing list used for
building a list */
   struct node *insert(struct node *p, int n)
   {
      struct node *temp;
```

Notes

```
      if(p==NULL)
      {
          p=(struct node *)malloc(sizeof(struct node));
         if(p==NULL)
         {
                 printf("Error\n");
            exit(0);
         }
         p-> data = n;
         p-> link = NULL;
      }
      else
      {
         temp = p;
         while (temp-> link != NULL)
                temp = temp-> link;
         temp-> link = (struct node *)malloc(sizeof(struct node));
        if(temp -> link == NULL)
         {
                 printf("Error\n");
            exit(0);
         }
         temp = temp-> link;
         temp-> data = n;
         temp-> link= NULL;
      }
      return (p);
   }
   /* a function which inserts a newly created node after the specified
node */
   struct node * newinsert ( struct node *p, int node_no, int value )
   {
      struct node *temp, * temp1;
      int i;
      if ( node_no <= 0 || node_no > length (p))
      {
              printf("Error! the specified node does not exist\n");
         exit(0);
```

Notes

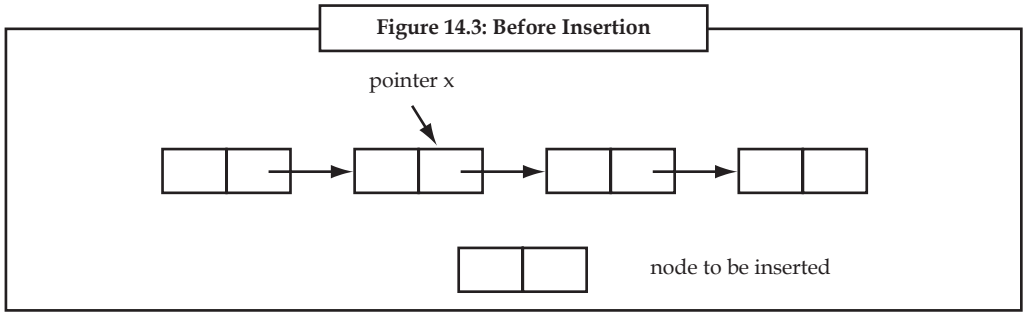
```
    }
    if ( node_no == 0)
    {
       temp = ( struct node * )malloc ( sizeof ( struct node ));
       if ( temp == NULL )
       {
               printf( " Cannot allocate \n");
               exit (0);
       }
       temp -> data = value;
       temp -> link = p;
       p = temp ;
}
else
{
    temp = p ;
   i = 1;
   while ( i < node_no )
   {
         i = i+1;
         temp = temp-> link ;
   }
   temp1 = ( struct node * )malloc ( sizeof(struct node));
   if ( temp == NULL )
   {
         printf ("Cannot allocate \n");
         exit(0)
   }
   temp1 -> data = value ;
   temp1 -> link = temp -> link;
   temp -> link = temp1;
}
return (p);
}

void printlist ( struct node *p )
{
   printf("The data values in the list are\n");
```

Notes

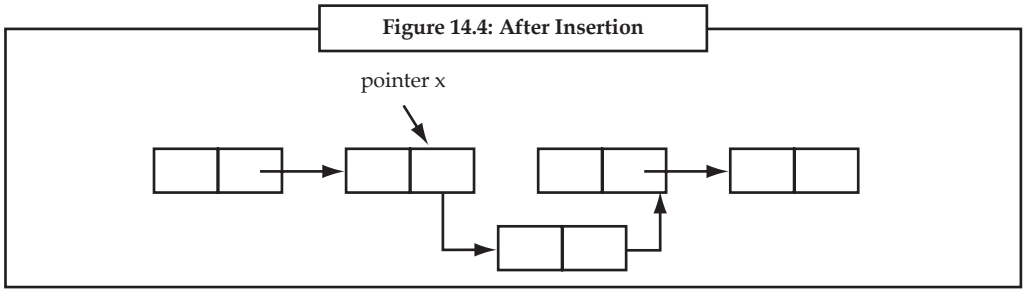
```
    while (p!= NULL)
    {
       printf("%d\t",p-> data);
       p = p-> link;
    }
}
void main ()
{
    int n;
    int x;
    struct node *start = NULL;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n- > 0 )
    {
       printf( "Enter the data values to be placed in a node\n");
       scanf("%d",&x);
       start = insert ( start, x );
    }
    printf(" The list before deletion is\n");
    printlist ( start );
    printf(" \n Enter the node no after which the insertion is to be
done\n");
    scanf ( " %d",&n);
    printf("Enter the value of the node\n");
    scanf("%d",&x);
    start = newinsert(start,n,x);
    printf("The list after insertion is \n");
    printlist(start);
}
```

Figure 14.3: Before Insertion



Notes

Figure 14.4: After Insertion



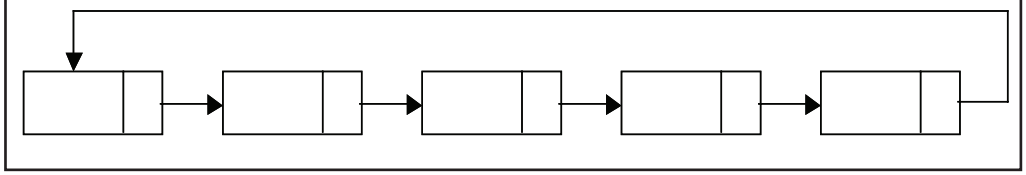


Task Write a program to insert the specified node in linked list?

14.8 Circular Linked List

Circular Linked List is another remedy for the drawbacks of the Single Linked List besides Doubly Linked List. A slight change to the structure of a linear list is made to convert it to circular linked list; link field in the last node contains a pointer back to the first node rather than a NULL. See Figure 14.5.

Figure 14.5: Circular Linked List



From any point in such a list it is possible to reach any other point in the list. If we begin at a given node and traverse the entire list, we ultimately end up at the starting point.



Lab Exercise

```
Program: Here is a program for building and printing the elements of the
circular linked list.

# include <stdio.h>

# include <stdlib.h>

struct node

{

   int data;

   struct node *link;

};

struct node *insert(struct node *p, int n)

{

   struct node *temp;
```

Notes

```
   /* if the existing list is empty then insert a new node as the
starting node */
       if(p==NULL)
       {
         p=(struct node *)malloc(sizeof(struct node)); /* creates new
node data value passes
   as parameter */
        if(p==NULL)
        {
                 printf("Error\n");
                 exit(0);
         }
         p-> data = n;
          p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
      }
      else
      {
         temp = p;
   /* traverses the existing list to get the pointer to the last node of
it */
         while (temp-> link != p)
                temp = temp-> link;
         temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using
         data value passes as
         parameter and puts its
         address in the link field
         of last node of the
         existing list*/
         if(temp -> link == NULL)
         {
                  printf("Error\n");
         }
            exit(0);
            temp = temp-> link;
            temp-> data = n;
            temp-> link = p;
```

Notes

```
        }
        return (p);
    }
    void printlist ( struct node *p )
    {
       struct node *temp;
       temp = p;
       printf("The data values in the list are\n");
       if(p!= NULL)
       {
       do
       {
                 printf(%d\t",temp->data);
                 temp=temp->link;
       } while (temp!= p)
    }
    else
       printf("The list is empty\n");
    }

    void main()
    {
       int n;
       int x;
       struct node *start = NULL ;
       printf("Enter the nodes to be created \n");
       scanf("%d",&n);
       while ( n- > 0 )
       {
              printf( "Enter the data values to be placed in a
node\n");
          scanf("%d",&x);
          start = insert ( start, x );
       }
       printf("The created list is\n");
       printlist ( start );
    }
```

This program appends a new node to the existing list (that is, it inserts a new node in the existing list at the end), and it makes the link field of the newly inserted node point to the start or first

node of the list. This ensures that the link field of the last node always points to the starting node of the list.

Notes



Case Study

Write programs with structures by using modular programming.

```
Program:
struct student
{
    name char[30];
    marks float;
}
main ( )
{
    struct student student1;
    student1 = read_student ( )
    print_student( student1);
    read_student_p(student1);
    print_student (student1);
}
struct student read_student( )     \\ A
{
    struct student student2;
    gets(student2.name);
    scanf("%d",&student2.marks);
    return (student2);
}
void print_student (struct student student2)     \\ B
{
    printf( "name is %s\n", student2.name);
    printf( "marks are%d\n", student2.marks);
}
void read_student_p(struct student student2)     \\ C
{
    gets(student2.name);
    scanf("%d",&student2.marks);
}
```

Contd...

Notes

Explanation

1. The function read_student reads values in structures and returns the structure.
2. The function print_student takes the structure variable as input and prints the content in the structure.
3. The function read_student_p reads the data in the structure similarly to read_student. It takes the structure student as an argument and puts the data in the structure. Since the data of a member of the structure is modified, you need not pass the structure as a pointer even though structure members are modified. Here you are not modifying the structure, but you are modifying the structure members through the structure.

Question

Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent location. The individual elements are stored "somewhere" in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. Thus, a linked list is a collection of elements called nodes, each of which stores two item of information—an element of the list, and a link, i.e., a pointer or an address that indicates explicitly the location of the node containing the successor of this list element.

Write a program to build a linked list by adding new nodes at the beginning, at the end or in the middle of the linked list. Also write a function display( ) which display all the nodes present in the linked list.

14.9 Summary

- In the single linked list each node provides information about where the next node is in the list.
- It faces difficulty if we are pointing to a specific node, then we can move only in the direction of the links.
- A Circular Linked List has no beginning and no end. Linked lists find many applications in implementing other data structures like stacks and queues.

14.10 Keywords

Circular Linked List: A linear linked list in which the last element points to the first element, thus, forming a circle.

Doubly Linked List: A linear linked list in which each element is connected to the two nearest.

Linear List: A one-dimensional list of items.

Linked List: A dynamic list in which the elements are connected by a pointer to another element.

NULL: A constant value that indicates the end of a list.

14.11 Self Assessment

Notes

Choose the appropriate answers:

1. ..... enables us to create data types and structures of any size and length to suit our programs need within the program.
   (a) Static memory
   (b) Dynamic memory
   (c) Initialization
   (d) Valuation
2. The Function ..... is most commonly used to attempt to ``grab'' a continuous portion of memory.
   (a) calloc
   (b) valloc
   (c) malloc
   (d) none

Fill in the blanks:

3. ..... is a function which attempts to change the size of a previous allocated block of memory.
4. C language provides ..... library functions known as memory management functions that can be used for allocating and freeing memory during program execution.
5. An array is represented in memory using .....
6. ..... are capable of representing a much more complex relationship between elements of a structure than a linear order.
7. A linked list is a ..... data structure.
8. An ..... function is used to create the list.
9. A ..... Linked List has no beginning and no end.
10. Initially the list is empty, so the pointer to the starting node is .....

14.12 Review Questions

1. Write a program to sort a linked list.
2. Write a program to implement a linked list using recursion.
3. Write a program to traverse a linked list and display the contents in reverse order.
4. Write a C program to sort the elements of a linked list.
5. Why are linked list better than arrays? Compare giving examples.
6. Insert a node at the nth position where n is accepted as an input from the keyboard.
7. Delete a node from the nth position where n is accepted as an input from the keyboard.
8. Shift the node at the nth position to the position p where n and p are accepted as inputs from the keyboard.

Notes

9. Explain circular linked list with the help of program.
10. Distinguish between calloc and realloc.

Answers: Self Assessment

1.	(b)	2.	(c)	3.	Realloc	4.	four
5.	sequential mapping			6.	Pointers		
7.	recursive	8.	insert	9.	Circular	10.	NULL

14.13 Further Readings



Books

Ashok N. Kamthane, "*Programming with ANCI & Turbo C*", Pearson Education, Year of Publication, 2008

B.W. Kernighan and D.M. Ritchie, "*The Programming Language*", Prentice Hall of India, New Delhi

Byron Gottfried, "*Programming with C*", Tata McGraw Hill Publishing Company Limited, New Delhi

R.G. Dromey, Englewood Cliffs, N.J., *How to Solve it by Computer*, Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C

Greg W Scragg, Genesco Suny, *Problem Solving with Computers,* Jones and Bartlett, 1997.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com