Programming In C

DECAP010

Edited by
Balraj Kumar

LOVELY
PROFESSIONAL
UNIVERSITY

LOVELY
PROFESSIONAL
UNIVERSITY

Programming In C

Edited By:
Balraj Kumar

Title: PROGRAMMING IN C

Author's Name: Dr. Prikshat Kumar Angra

Published By : Lovely Professional University

Publisher Address: Lovely Professional University, Jalandhar Delhi GT road, Phagwara - 144411

Printer Detail: Lovely Professional University

Edition Detail: (I)

ISBN: 978-93-94068-23-0

ISBN 978-93-94068-23-0

9 789394 068230

Copyrights@ Lovely Professional University

CONTENT

Unit 1: **Introduction** 1

Prikshat Kumar Angra, Lovely Professional University

Unit 2: **Data Types** 18

Prikshat Kumar Angra, Lovely Professional University

Unit 3: **Input/Output in C** 30

Prikshat Kumar Angra, Lovely Professional University

Unit 4: **Operations** 46

Prikshat Kumar Angra, Lovely Professional University

Unit 5: **Decision making statements** 58

Prikshat Kumar Angra, Lovely Professional University

Unit 6: **Loop Statements** 84

Prikshat Kumar Angra, Lovely Professional University

Unit 7: **Functions** 105

Prikshat Kumar Angra, Lovely Professional University

Unit 8: **Storage Classes** 123

Prikshat Kumar Angra, Lovely Professional University

Unit 9: **Arrays** 137

Prikshat Kumar Angra, Lovely Professional University

Unit 10: **Pointers** 152

Prikshat Kumar Angra, Lovely Professional University

Unit 11: **Strings** 168

Prikshat Kumar Angra, Lovely Professional University

Unit 12: **Dynamic Memory Management** 184

Prikshat Kumar Angra, Lovely Professional University

Unit 13: **Structure and Union** 196

Prikshat Kumar Angra, Lovely Professional University

Unit 14: **Structures** 219

Prikshat Kumar Angra, Lovely Professional University

Unit 01: Introduction

CONTENTS

Objectives
Introduction
1.1 Programming Language
1.2 Machine Level Language
1.3 Advantages and Limitations of Machine Language
1.4 Assembly Language
1.5 Assembly Program Execution
1.6 High Level Languages
1.7 Introduction to C Programming
1.8 Origin and Development of C Language
1.9 Applications of C Language
1.10 Evolution of C
1.11 Compiler and Interpreter
1.12 Program Development in C
1.13 Writing a C Program
1.14 C Character set
1.15 Identifiers and keywords
Summary
Keywords
Self-Assessment
Answer for Self Assessment
Further Readings

Objectives

After studying this unit, you will be able to:

- Understand Programming language
- Discuss C Programming
- Different stages in program developmentusing codeblocks IDE
- Programing methodologies

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 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 Machine Level Language

Computer language, also known as machine code, is a low-level programming language made up of binary digits (ones and zeros). Before a computer can run code written in high-level languages like Swift and C++, the code must be converted into machine language.

Since computers are digital devices, they only recognize binary data. Every program, video, image, and character of text is represented in binary. This binary data, or machine code, is processed as input by the CPU. The resulting output is sent to the operating system or an application, which displays the data visually. For example, the ASCII value for the letter "A" is 01000001 in machine code, but this data is displayed as "A" on the screen. An image may have thousands or even millions of binary values that determine the color of each pixel.

While computers can be programmed to understand a variety of computer languages, there is only one language that the computer understands without the use of a translation program; this language is known as the computer's machine language or machine code.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. The first part is command or operation, and it tells the computer what function to perform. Every computer has an operation code or op-code 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.

Discuss Example of Machine Level Language

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.

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

101100111111010011101100110000111001

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

0010000000000001100111001

00111100000001111111000111

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 of using 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.

1.3 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. *Difficult 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 for 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.

4. *Difficult 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.

1.4 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.

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.

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.

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
```

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

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

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.

1.5 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.

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

1. Manual assembly

2. By using an assembler.

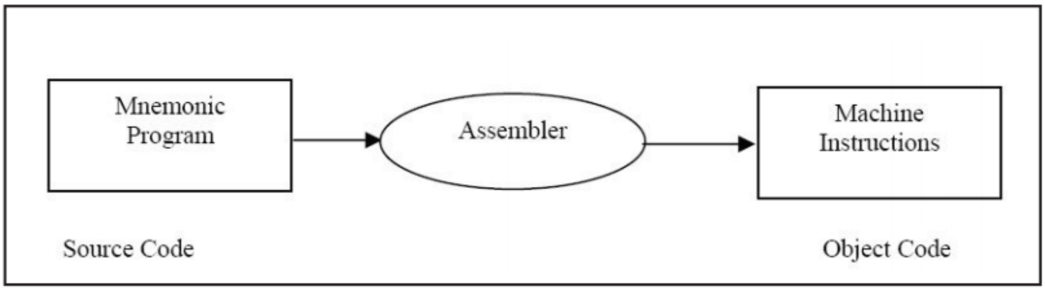
Manual Assembly

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.

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.

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.

Differences between Machine-Level language and Assembly language

Machine-level language	Assembly language
The machine-level language comes at the lowest level in the hierarchy, so it has zero abstraction level from the hardware.	The assembly language comes above the machine language means that it has less abstraction level from the hardware.
It cannot be easily understood by humans.	It is easy to read, write, and maintain.
The machine-level language is written in binary digits, i.e., 0 and 1.	The assembly language is written in simple English language, so it is easily understandable by the users.
It does not require any translator as the machine code is directly executed by the	In assembly language, the assembler is used to convert the assembly code into machine code.

computer.	
It is a first-generation programming language.	It is a second-generation programming language.

1.6 High 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.

When writing a program in a high-level language, then the whole attention needs to be paid to the logic of the problem. A compiler is required to translate a high-level language into a low-level language.

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

Advantages of a high-level language

1. Readability: Programs written in these languages are more readable than assembly andmachine 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 ofprogramming language are similar to natural languages (ENGLISH).

Differences between Low-Level language and High-Level language

Low-level language	High-level language
It is a machine-friendly language, i.e., the computer understands the machine language,	It is a user-friendly language as this language is written in simple English words, which can

which is represented in 0 or 1.	be easily understood by humans.
The low-level language takes more time to execute.	It executes at a faster pace.
It requires the assembler to convert the assembly code into machine code.	It requires the compiler to convert the high-level language instructions into machine code.
The machine code cannot run on all machines, so it is not a portable language.	The high-level code can run all the platforms, so it is a portable language.
It is memory efficient.	It is less memory efficient.
Debugging and maintenance are not easier in a low-level language.	Debugging and maintenance are easier in a high-level language.

1.7 Introduction to C Programming

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,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.

1.8 Origin and Development of C Language

C is a general-purpose, structured programming language. Structured Languages have a characteristic program 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 standardized 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.

1.9 Applications of C Language

Mainly C Language is used for Develop Desktop application and system software. Some application of C language are given below.

- C programming language can be used to design the system software like operating system and Compiler.
- To develop application software like database and spread sheets.
- For Develop Graphical related application like computer and mobile games.
- To evaluate any kind of mathematical equation use c language.
- C programming language can be used to design the compilers.
- UNIX Kernel is completely developed in C Language.
- For Creating Compilers of different Languages which can take input from other language and convert it into lower level machine dependent language.
- C programming language can be used to design Operating System.

- C programming language can be used to design Network Devices.
- To design GUI Applications. Adobe Photoshop, one of the most popularly used photo editors since olden times, was created with the help of C.

1.10 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 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.

Give two examples of high level languages.

1.11 Compiler and Interpreter

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.

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.

A number of different compilers are available these days for C language. GCC, 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).

Code blocks is an IDE for running C and C++ programs on different operating systems like Windows, Linux and Mac OS.

1.12 Program Development in C

The development of a "C" program involves the use of the following programs in the order of their usage.

Editor

This program is used for writing the Source Code, the first thing that any programmer writing a program in any language would be doing.

Debugger

This program helps us identify syntax errors in the source code.

Pre Processor

There are certain special instructions within the source code identified by the # symbol that are carried on by a special program called a preprocessor.

Compiler

The process of converting the C source code to machine code and is done by a program called Compiler.

Linker

The machine code relating to the source code you have written is combined with some other machine code to derive the complete program in an executable file. This is done by a program called the linker.

1.13 Writing a C Program

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 (;)

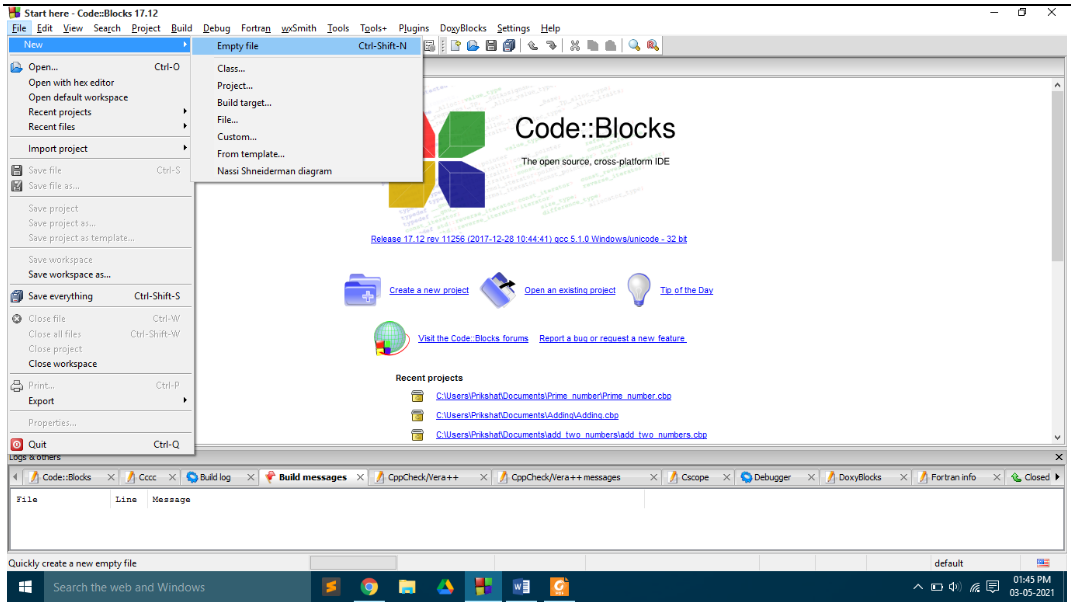
5. Every C program contains one main() function.

C Program Code

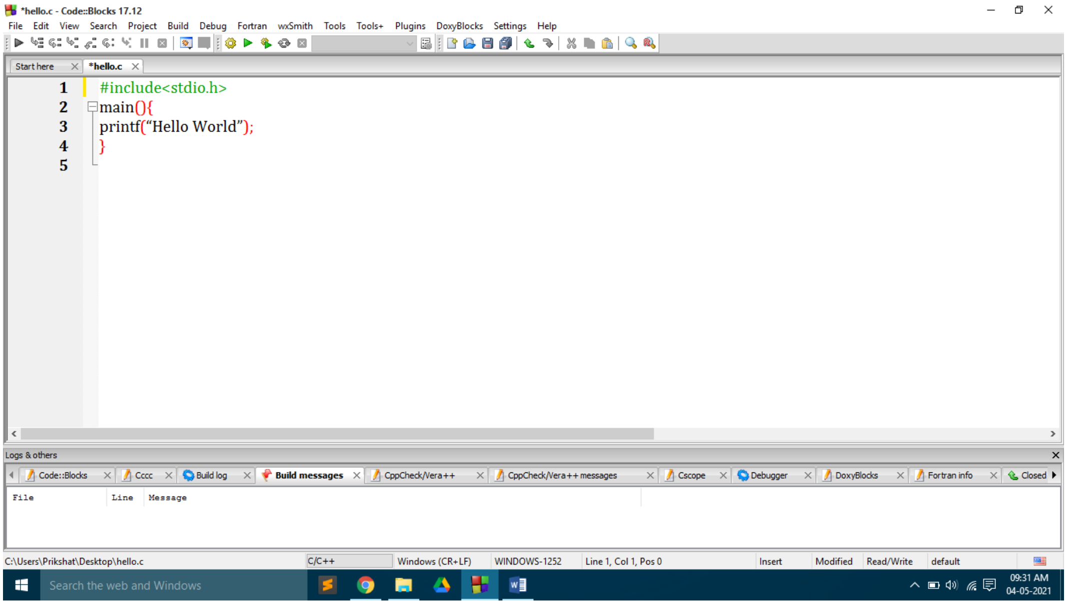
```
#include<stdio.h>
main(){
printf(“Hello World”);
}
```

Creating and Compiling a C Program

Creating a compiling a C program on operating system use compiler and an integrated development environment. Code blocks is used for create and execute program of C language. File name is hello.c and save in windows operating system using code blocks IDE.


Start here - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
New
Empty file Ctrl-Shift-N
Class...
Project...
Build target...
File...
Custom...
From template...
Nassi Shneiderman diagram
Open... Ctrl-O
Open with hex editor
Open default workspace
Recent projects
Recent files
Import project
Save file Ctrl-S
Save file as...
Save project
Save project as...
Save project as template...
Save workspace
Save workspace as...
Save everything Ctrl-Shift-S
Close file Ctrl-W
Close all files Ctrl-Shift-W
Close project
Close workspace
Print... Ctrl-P
Export
Properties...
Quit Ctrl-Q
Code::Blocks
The open source, cross-platform IDE
Release 17.12 rev 11256 (2017-12-28 10:44:41) gcc 5.1.0 Windows/unicode - 32 bit
Create a new project
Open an existing project
Tip of the Day
Visit the Code::Blocks forums
Report a bug or request a new feature
Recent projects
C:\Users\Prikshat\Documents\Prime_number\Prime_number.cbp
C:\Users\Prikshat\Documents\Adding\Adding.cbp
C:\Users\Prikshat\Documents\add_two_numbers\add_two_numbers.cbp
Logs & others
Code::Blocks
Cccc
Build log
Build messages
CppCheck/Vera++
CppCheck/Vera++ messages
Cscope
Debugger
DoxyBlocks
Fortran info
Closed
File
Line
Message
Quickly create a new empty file
default
Search the web and Windows
01:45 PM
03-05-2021

Click on empty file link and save that file with name hello.c and write code.


*hello.c - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Start here
*hello.c
Logs & others
Code::Blocks
Cccc
Build log
Build messages
CppCheck/Vera++
CppCheck/Vera++ messages
Cscope
Debugger
DoxyBlocks
Fortran info
Closed
File
Line
Message
C:\Users\Prikshat\Desktop\hello.c
C/C++
Windows (CR+LF)
WINDOWS-1252
Line 1, Col 1, Pos 0
Insert
Modified
Read/Write
default
Search the web and Windows
09:31 AM
04-05-2021

```
#include<stdio.h>
main(){
printf(“Hello World”);
}
```

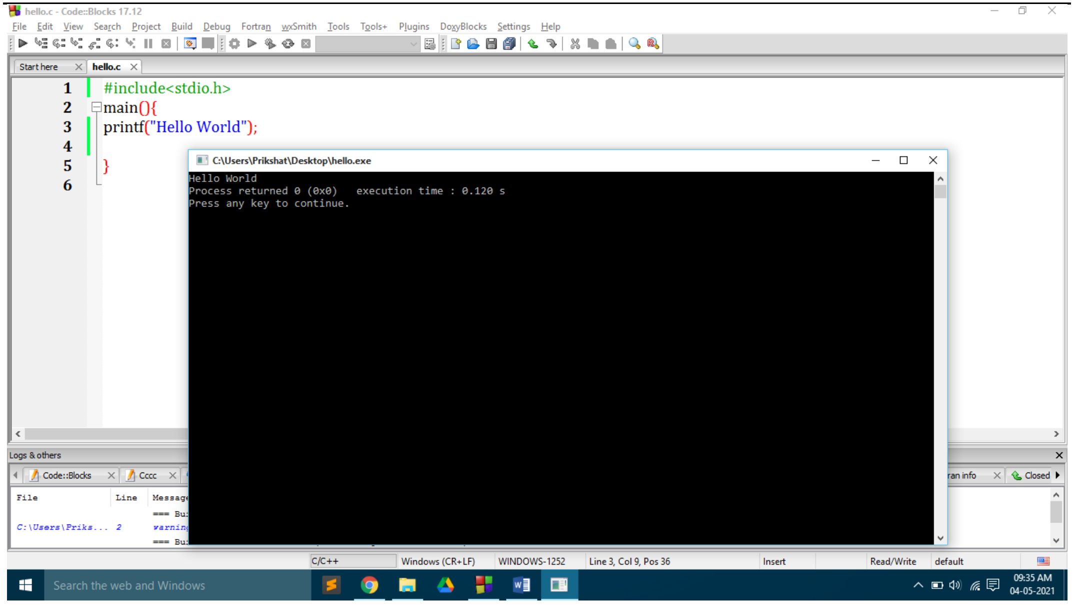
After write code in code blocks,

Gcc compiler is used for compiling code using code blocks

For compile press CTRL+F9 or click on build option and click on build

To run C program in code blocks after write code press first compile program than run (CTRL+F10).

Output will be


hello.c - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Start here
hello.c
Logs & others
Code::Blocks
Cccc
ran info
Closed
C/C++
Windows (CR+LF)
WINDOWS-1252
Line 3, Col 9, Pos 36
Insert
Read/Write
default
Search the web and Windows
09:35 AM
04-05-2021

```c
#include<stdio.h>
main(){
printf("Hello World");

}
```

C:\Users\Prikshat\Desktop\hello.exe

```
Hello World
Process returned 0 (0x0)   execution time : 0.120 s
Press any key to continue.
```

```
File                Line   Message
                           === Bu
C:\Users\Priks...   2      warning
                           === Bu
```

1.14 C Character set

Like each other language, 'C' additionally has its own character set. A program is a bunch of directions that, when executed, produce a yield. The information that is prepared by a program comprises of different characters and images. The yield produced is additionally a mix of characters and images.

A character set in 'C' is divided into

- Letters
- Numbers
- Special characters
- White spaces (blank spaces)

Letters

- Uppercase characters (A-Z)
- Lowercase characters (a-z)

Numbers

- All the digits from 0 to 9

White spaces

- Blank space
- New line
- Carriage return
- Horizontal tab

Special characters

- Special characters in 'C' are shown in the given table,

, (comma)	{ (opening curly bracket)
. (period)	} (closing curly bracket)
; (semi-colon)	[(left bracket)
: (colon)	] (right bracket)
? (question mark)	((opening left parenthesis)
' (apostrophe)	) (closing right parenthesis)
" (double quotation mark)	& (ampersand)
! (exclamation mark)	^ (caret)
| (vertical bar)	+ (addition)
/ (forward slash)	- (subtraction)
\ (backward slash)	* (multiplication)
~ (tilde)	/ (division)
_ (underscore)	> (greater than or closing angle bracket)
$ (dollar sign)	< (less than or opening angle bracket)
% (percentage sign)	# (hash sign)
, (comma)	{ (opening curly bracket)

1.15 Identifiers and keywords

Keywords have fixed meanings, and the meaning cannot be changed. They act as a building block of a 'C' program. There are a total of 32 keywords in 'C'. Keywords are written in lowercase letters.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	short	float	unsigned
continue	for	signed	void

An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc.

Summary

- 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 Computer Programming is the art of making a computer do what you want it to do.
- Machine language, or machine code, is a low-level language comprised of binary digits (ones and zeros).
- Assembly languages are also known as second generation languages. These languages substitute alphabetic symbols for the binary codes of machine language.
- High-level languages include FORTRAN, COBOL, BASIC, PASCAL, C, C++ and JAVA.
- C is a general-purpose, structured programming language. Structured Languages have a characteristic program 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.
- To develop application software like database and spread sheets.
- For Develop Graphical related application like computer and mobile games.
- Keywords have fixed meanings, and the meaning cannot be changed. They act as a building block of a 'C' program. There are a total of 32 keywords in 'C'
- Code blocks is an IDE for running C and C++ programs on different operating systems like Windows, Linux and Mac OS.
- An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc.

Keywords

Programming:Computer programming is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task.

Machine Level Language:Machine language, or machine code, is a low-level language comprised of binary digits (ones and zeros).

Assembly Level Language:Assembly language is a low-level programming language.

High Level Language:high-level programming language is a programming language with strong abstraction from the details of the computer.

Self-Assessment

1. Which of the following language instructions execute very fast?

A. Machine language Instructions
B. Hardwired Instructions
C. Procedural language Instructions
D. Assembly language Instructions

2. Program written in ____________ language is difficult to understand.

A. Procedural language
B. Machine Language
C. Assembly Language
D. High Level Language.

3. Which of the following is part of computer system.

A. Hardware
B. Programming
C. Software
D. Hardware & Software

4. _____________ is a computer language.

A. Binary Language
B. Low Level Language
C. High Level Language
D. All of the above

5. Compiling is a process.

A. To convert source code to machine independent code.
B. To make program.
C. To complete coding.
D. All of the above.

6. Who developed C language?

A. Dennis M. Ritchie
B. Steve Jobs
C. Steve Smith
D. James Gosling

7.What will this program print?

```
#include<stdio.h>
int main()
{
/* my first program in C */
printf("Hello, World! \n");
return 0;
}
```

A. Hello, World!
B. Hello, Programming
C. Error Message

D. All of the above

8. Which of the following is an extension of C language code?

A. .c
B. .x
C. .z
D. None of the above

9. Which of the following language is used to write Unix kernel?

A. C#
B. Java
C. C
D. SQL

10. Which of the following statement is true about comments?

A. Comments are C++ statements.
B. Comments can be inserted anywhere in C Program.
C. Comments are required in every program.
D. All of the above.

11. Which is not keyword in C language?

A. complex
B. near
C. typed ef
D. const

12. We can use keywords as Identifier.

A. True
B. False

13. Which of the following statement is true about identifier?

A. An identifier is used to identify elements of a program
B. An identifier acts as comment
C. An identifier is similar to keywords
D. All of the above

14. Which of the following statement is true about keyword?

A. Keywords are same as identifier
B. Keywords are not part of C language
C. Reserve words in C language are known as keywords.
D. All of the above

15. C character set includes

A. Letters & numbers
B. Letters, numbers and white spaces
C. Special characters only
D. Letters, numbers, white spaces and special characters

Answer for Self Assessment

1.	A	2.	B	3.	D	4.	A	5.	A
6.	A	7.	A	8.	A	9.	C	10.	B
11.	B	12.	B	13.	A	14.	C	15.	D

Further Readings

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.

YashvantKanetkar, Let us C

Web Links

https://www.tutorialspoint.com/index.htm

www.webopedia.com

www.web-source.net

Unit 02: Data Types

CONTENTS

Objectives
Introduction
2.1 Constants in C
2.2 Data Types
2.3 Variables
Summary
Keywords
Self Assessment
Answers for Self Assessment
Further Readings

Objectives

After studying this unit, you will be able to:

- Understand constant and variables
- Explain data type in C
- Describe C program structure
- Describe how to declare variables in C

Introduction

Data items (variables and constants) can be manipulated arithmetically in a C programme using operators. An operator is a symbol that instructs the machine to perform mathematical or logical operations on data held in variables. Operands are the variables that are being worked on.

2.1 Constants in C

A constant is a non-changing token with a fixed value. It can be stored in a place in the computer's memory and accessed using that memory address. In C, constants are divided into four categories: integer constants, floating-point constants, character constants, and string constants. Constants can be present in composite forms.

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:

<Data type > <variable_name> = <value>

Example: float pi = 3.14

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 with in a pair of single inverted commas. The maximum length of a character constant is one character.

Example: ‘a’ is a character constant

Example: ‘d’ is a character constant

Example: ‘P’ is a character constant

Example: ‘7’ is a character constant

Example: ‘*’ 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.

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.

Note: that 8 is an integer while 8. or 8.0 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. " Hello World"
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

The value of a logical constant may be true or false. In C, a non-zero value is considered true, while 0 is considered false.

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

int n=512.225;

2.2 Data Types

One of the factors that determines the strength of a programming language is the number of different types of data that it can handle. In this regard, the C programming language is extremely strong. In a C programme, almost every form of data can be interpreted and manipulated.

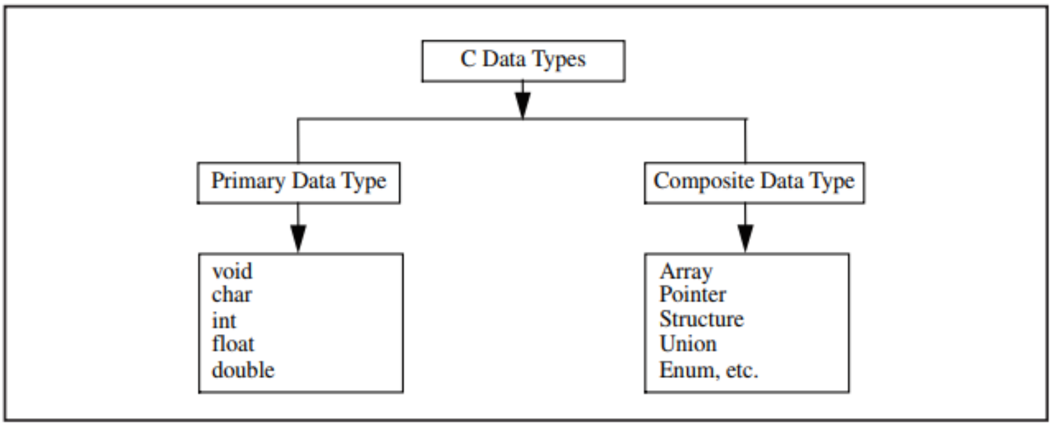
Inside a digital computer, at the lowest level, all data and instructions are stored using only binary digits (0 and 1). Thus, decimal number 65 is stored as its binary equivalent: 0100 0001. Also the character "A" is stored, as binary equivalent of 65(A's ASCII): 0100 0001. Both the stored values are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

Example: char a;

2. **int**: stores signed integers. e.g., positive or negative integers.

Example: int b;

3. **float**: stores real numbers with single precision (precision of six digits after decimal points)

Example: float c;

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

Example: double d;

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	$– 1.7 \times 10_{308}$ to $+1.7*10_{308}$
void	valueless or empty	0 byte	–	–

Additional Data Types

Primary C data types can come in a variety of sizes, allowing them to store a wide range of values. This is implied in a programme by appending a keyword (data type qualifier) before the data type.

For example, data type qualifiers for the int basic type include short, long, signed, and unsigned. In C, an integer data type can be specified as a short int, int, unsigned int, or long int. Implementation determines the set of values and size of these eligible data-types. Short, on the other hand, is smaller than or equal to int, which is smaller than long. Since it does not store negative integers, an unsigned int has a wider range.

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.

Example: int *ptr.

Note: - Asterisk (*) symbol is used to declare pointer variable.

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;

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.

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

2.3 Variables

A variable is an object whose value can change while a programme is running. A variable is a symbolic representation of the address in memory space where values can be stored, accessed, and updated. Each variable has a memory location or address assigned to it, and the value of that variable is stored there.

Each variable has its own name, data type, size, and value. All variables must have a type so that the compiler can record all necessary information about them, generate the appropriate code during translation, and allocate the necessary memory space.

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

Notes: C language is a case-sensitive language which differentiates between lower case and upper case. Thus, JOB is different from Job, jOB, JoB. 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.

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

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: int i, j, k; //creates integer variables i,j and K

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 a , b;
a = 100;
b = 20;
```

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;

Notes: 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:

```
1.  int i = 10, j = 15;
2. 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)

Did you know?

Allocating a variable implies reserving some memory for that variable. In some programming languages (like C) if a variable is allocated but not assigned, it is said to have a "garbage value".

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

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

Notes: Write a program in C to show the variable declaration.

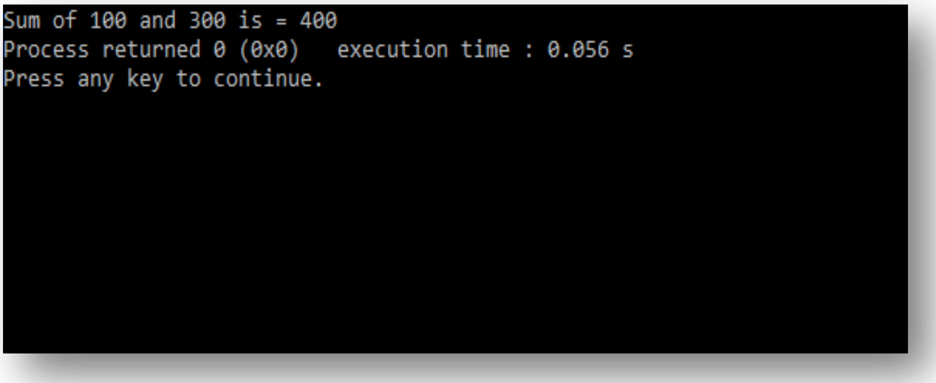
Example:

Write a program to find sum of two numbers.

Solution : -

```
#include<stdio.h>
int main(){
int num1,num2,sum;
num1=100;
num2=300;
sum=num1+num2;
printf("Sum of %d and %d is = %d",num1,num2,sum);
return 0;
}
```

Output:-



```
Sum of 100 and 300 is = 400
Process returned 0 (0x0)   execution time : 0.056 s
Press any key to continue.
```

Summary

- A constant is a value that doesn't change throughout the execution of a program.
- A variable is an identifier which is used to store a value.

- There are four commonly used data types such as int, float, char and a void.
- Each data type differs in size and range from one another.
- 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.

Keywords

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

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

Data type: A data type specifies the type of data that a variable can store such as integer, floating, character, etc.

Character constant: A "character constant" is formed by enclosing a single character from the representable character set within single quotation marks (' ').

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

Self Assessment

1. Which of the following is true for variable names in C?

A. Variable can be of any length

B. They can contain alphanumeric characters as well as special characters

C. Reserved Word can be used as variable name

D. Variable names cannot start with a digit

2. What are the entities whose values can be changed called?

A. Constants

B. Variables

C. Modules

D. Tokens

3. Which of the following is not a constant type in C language?

A. Real

B. Integer

C. Main

D. Character

4. Which is not a valid C variable name?

A. int number;

B. float rate;

C. int variable_count;

D. int $reg_no;

5. The format identifier '%d' is used for _____ data type?

A. Char

B. Int

C. Double

D. Float

6. The format identifier '%f' is used for _____ data type?

A. Char

B. float

C. Double

D. Int

7. Constant is ____________.

A. A value that is same as keyword

B. A value that doesn't change throughout the execution of a program.

C. A value that change throughout the execution of a program.

D. All of the above

8. Which of the following is not a valid format specifier?

A. %d

B. %\d

C. %f

D. %c

9. Which of the following is type of variable in C language?

A. Local Variable

B. Global Variable

C. All of the above

D. None of the above

10. Which is the only function all C programs must contain?

A. start()

B. system()

C. main()

D. printf()

11. Which of the following special symbol allowed in a variable name?

A. * (asterisk)

B. _ (underscore)

C. | (pipeline)

D. – (hyphen)

12. What are the different types of real data type in C?

A. float, double

B. short int, double, long int

C. float, double, long double

D. double, long int, float

13. What will this program print?

```
#include<stdio.h>
int main()
{
int a,b,sum;
a=100,b=200;
sum=a+b;
/* my first program in C */
printf(sum);
return 0;
}
```

A. Error

B. sum

C. 300

D. All of the above

14. Which is false?

A. Global constant variables are initialized to zero

B. const keyword is used to define constant values

C. Constant variables need not be defined as they are declared and can be defined later

D. You cannot reassign a value to a constant variable

15. Which of the following symbol is used in format specifier?

A. !

B. @

C. $

D. %

Answers for Self Assessment

1.	D	2.	B	3.	C	4.	D	5.	B
6.	B	7.	B	8.	B	9.	C	10.	C
11.	B	12.	C	13.	A	14.	C	15.	D

Further Readings

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

Web Links

https://www.tutorialspoint.com/index.htm

www.webopedia.com

www.web-source.net

Unit 03: Input / Output in C

CONTENTS

Objectives
Introduction
3.1 Input/output Function
3.2 Formatted Input/Formatted Output
3.3 printf()
3.4 Escape Sequences
3.5 scanf()
3.6 sign (*)
3.7 Reading and Writing a Character
3.8 gets() and puts() Functions
Summary
Keywords
Self-Assessment
Answer for Self Assessment
Review Questions
Further Readings

Objectives

After studying this unit, you will be able to:

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

Introduction

If a user is unable to communicate with a computer programme, it is unlikely that the programme would be useful. The software must read input values from the user console and generate the expected useful output to the user in most programming assignments.

C is unique in that, unlike other high-level languages, it does not have any built-in I/O statements in its syntax. We'll look at how C handles different I/O operations in this unit. We'll also hear about the standard library functions in C, as well as the header files that go with them.. 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.

3.1 Input/output Function

Data is transferred to peripheral devices such as a computer, keyboard, printer, or secondary storage through I/O operations. Since C has no built-in support for receiving data from input devices (such as a keyboard) or sending data to output devices (such as a monitor), all I/O operations must depend on library functions like printf (). A library is a set of predefined functions contained in one or more files. The creators of the C compiler created a library named "C standard library" that contains many standard I/O functions.

Depending on the compiler and necessary functions, various methods may be used to access these library functions from the standard library. Some compilers scan libraries for named functions automatically. Most of the time, the programmer must specifically state the library file name during the linking process, resulting in the executable programme only containing the necessary functions.

These functions are stored in a special file with the extension.h (such as stdio.h), which is referred to as a header file. The #include compiler directive can be used to insert a header file into a C programme file, as shown below.

#include<stdio.h>

The #include directive tells the compiler to read the stdio.h (standard input output header file) file and substitute this line with the contents of that file. Similarly, in order to use other predefined functions, their respective header files must be included in a programme such that the function declaration can be accessed by the programme.

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

1. Port I/O functions

2. Disk I/O function

3. Console I/O function

The Port I/O function handles various I/O operators on various ports, such as a mouse port or a printer port. The scope of this text does not include a thorough examination of the port I/O functions. The disc I/O functions are used to manipulate files on secondary storage devices such as a floppy disc or a hard disc. Since files are stored in secondary storage, such as discs, disc I/O functions are nothing more than file handling functions.

The term "console" refers to the regular input and output devices in its broadest sense. These basic input or output devices, which are also known as the keyboard and display by default, are dealt with by console I/O functions. These functions take keyboard input and display the results on the computer.

C takes all input and output as stream of characters. A stream is nothing,

but a series of bytes. Clanguage treats all streams equally i.e., whether a program gets input from the keyboard, a diskfile 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 streamsassociated with console I/O operations.

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

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.

For conducting console I/O operations, C offers a number of functions. These functions allow information to be transferred between the computer's standard input and output devices (i.e., keyboard and monitor). Only a few of them allow you to format input and output operations. Some of them, on the other hand, do not allow you to monitor 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

It is important to include the standard I/O library header file in order to use these functions. The declarations for these functions can be found in the stdio.h header file. As a result, before using these console I/O features, always include the header file stdio.h in your C programme.

3.2 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.

Printf() and scanf() are the two most commonly used functions for formatted I/O. (). The printf() function is used to view formatted data items on a standard output device, such as a monitor, while the scanf() function is used to read formatted data input from a standard input device, such as a keyboard. However, due to their greater complexity, both functions are slower than the previous classes of functions. These functions are specified in the stdio.h header file and return EOF if an error or the end of the file occurs. Let's take a closer look at each of these functions one by one.

3.3 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.

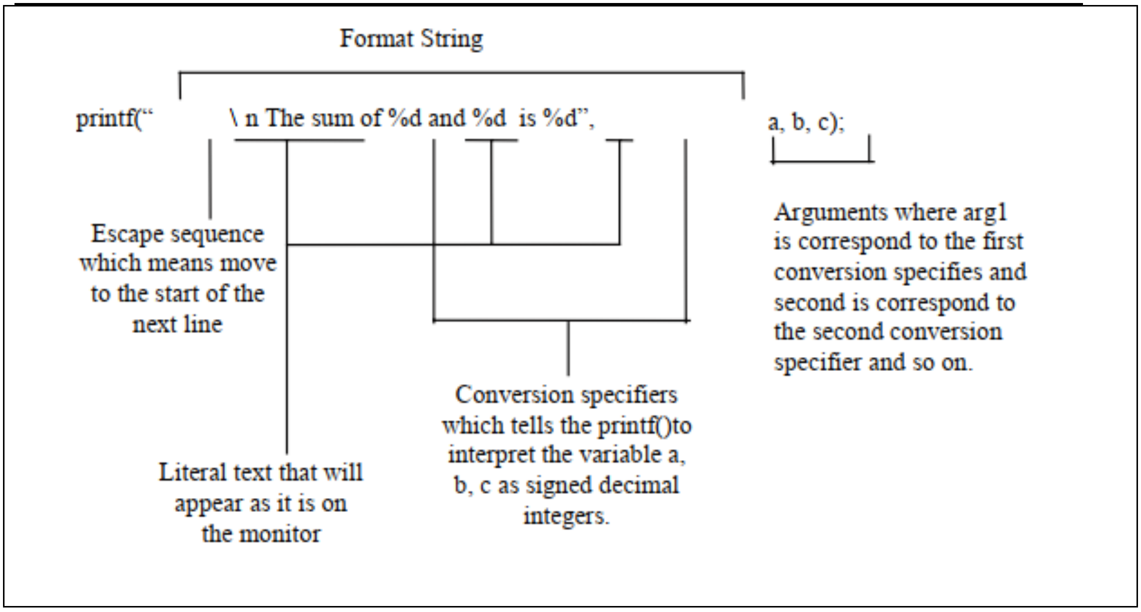
2. An escape sequence that begins with a \ (backslash) sign, provides special formatting

control.

3. A conversion specifier that begins with a % sign and followed by a single character, that

tellsprintf() how to interpret the arguments being used. To understand the various sections

of previously used printf() statement,


Format String
printf(“
\ n The sum of %d and %d is %d”,
a, b, c);
Escape sequence which means move to the start of the next line
Arguments where arg1 is correspond to the first conversion specifies and second is correspond to the second conversion specifier and so on.
Conversion specifiers which tells the printf()to interpret the variable a, b, c as signed decimal integers.
Literal text that will appear as it is on the monitor

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() function reads the format string from left to right and sends the characters that follow to the standard output unit. It takes action as soon as it encounters the (backslash) (that is, the signal of the existence of an escape sequence). When it comes across the percent (conversion specifier) symbol, it grabs the corresponding argument and prints its value in the format defined. When the closing pair of double quotes is encountered, the process comes to an end.

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.

Program:

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

Format Specifiers

%d -- displays a decimal (base 10) integer

%i -- used with other specifiers to indicate a "long"

%e -- displays a floating point value in exponential notation

%f -- displays a floating point value

%c -- displays a single character

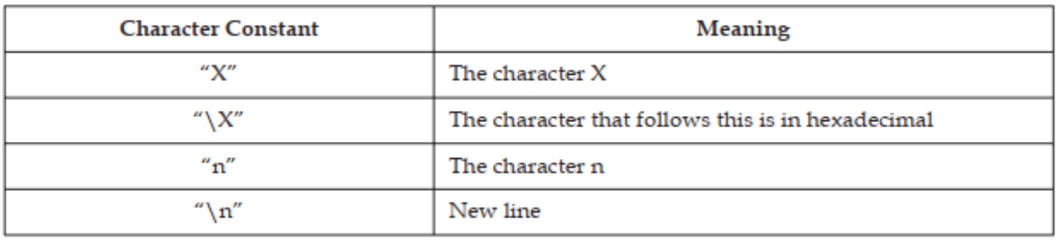
%s -- displays a string of characters

Example:Formatted Output with printf ( )

```
Int a;
Float b;
scanf("%d %f", &a, &b);
printf("the value of a= %d and b= %f \n", a, b);
```

3.4 Escape Sequences

Escape sequences, as previously mentioned, are used to control the position of output by shifting the monitor's cursor. Any character preceded by a backslash is meant to be treated as an escape sequence. The backslash informs the compiler that this is a special character constant of different significance than printf ().



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

Program shows the usage by some of the frequently used escape sequences

```
#include<stdio.h>
void main()
{
printf("\n 1..\t2..\t3....\n");
printf("The question is, \" said Humpty Dumpty,\"which is to be masterthat\'s all.\"");
}
```

Output:

```
1.. 2.. 3....
"The questions is, "said Humpty Dumpty," which is to be master-that's all".
```

3.5 scanf()

As previously stated, in order to write an interactive application, certain statements in the programme must be able to collect data from the user. We provided a few functions in this sense, such as getchar(), gets(), and getch(), among others. A programmer, on the other hand, 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);`

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:

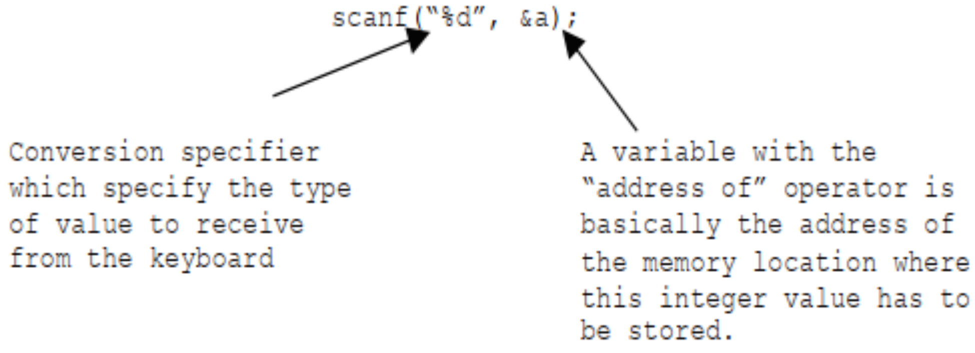
1. Conversion specifiers as in the printf() functions.

2. White space characters i.e., tabs, blanks, and newlines.

3. Other characters than white spaces, that are matching characters and asteric

3.6 sign (*)

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



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

Conversion specifier which specify the type of value to receive from the keyboard

A variable with the "address of" operator is basically the address of the memory location where this integer value has to be stored.

In layman's terms, the preceding statement is an instruction to the compiler to obtain an integer value from the keyboard and store it in a variable called a. Where an is an integer variable that was previously declared. Following this argument, the value stored in the variable can be used anywhere for any reason. Let's take a closer look at the different components of this sentence.

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.

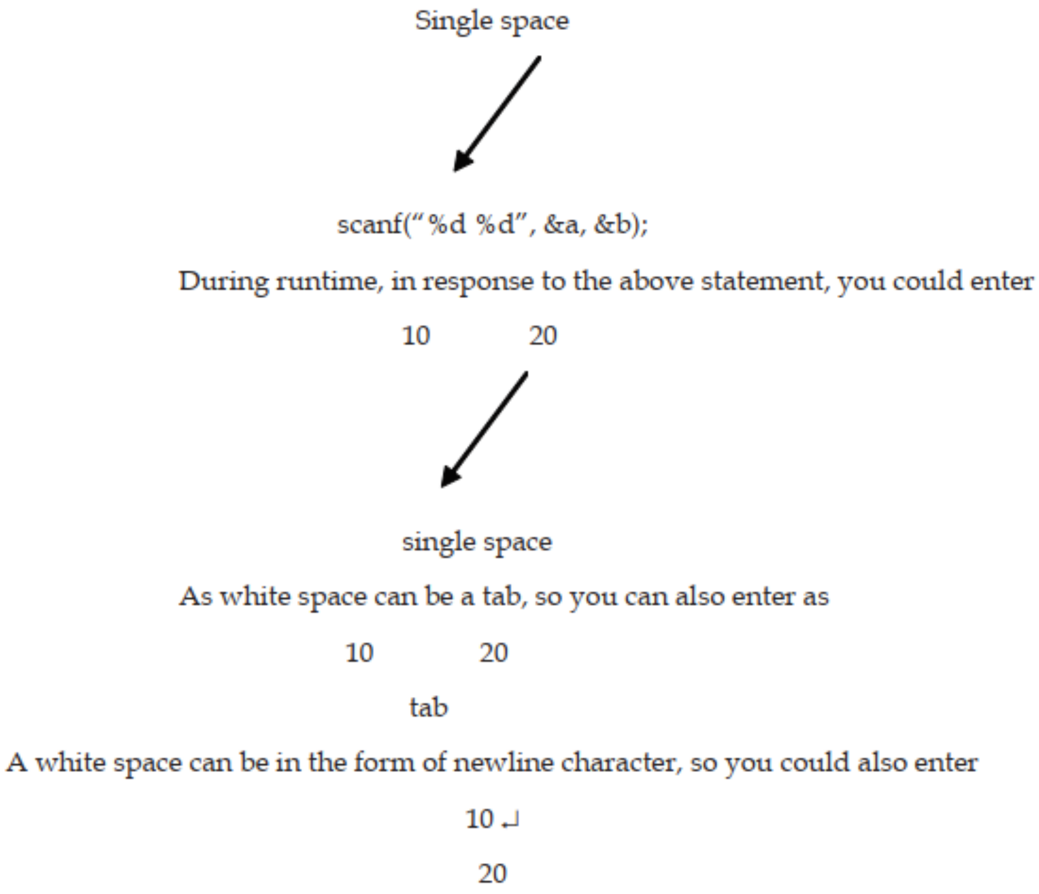
Example: The statement.

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 several variables are entered in a single scanf(), the white space character may be used to distinguish them (i.e., blank space, tabs or new line character). White spaces in the format specifier are not taken into account. As a result, they will be read in the input data but ignored. Example: Consider the following statement once again:



scanf(“ %d %d”, &a, &b);

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

```
10          20
```

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

```
10          20
```

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

Program: input function

```
#include<stdio.h>
int main()
{
int var1;
printf(“Enter value of var1”);
scanf(“%d”,&var1);
printf(“Value of Var1 is = %d”,var1);
return 0;
}
```

Program: To find radius of circle

```
#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);
Printf(“\n circumference = %f”, c);
}
```

Program : lower Case to upper case

```
#include<stdio.h>
void main()
{
charch;
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: h
The typed character in upper case is: H
```

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

3.7 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

Character I/O functions are functions that programme input/output of one character at a time. Since they deal with individual character values, these are the most basic I/O functions. To enter a character from the keyboard, use the following functions:

1. getchar()
2. getch()
3. getche()

Whereas 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.

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

```
#include <stdio.h>
void main( )
{
int c;
printf("Enter a character");
 c = getchar();
putchar(c);
}
```

3.8 gets() and puts() Functions

Character I/O functions have a drawback in that they can only handle one character at a time. Strings, on the other hand, are commonly used in real-world programmes. A string is nothing more than a set of characters. String I/O functions are functions that make it easier to move strings between a computer and regular I/O devices. 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.

gets() function

The gets() function allows the user to type a string of characters and then press the enter key. A character list is used to store all of the characters entered by the user. To render the array a string, the null character is inserted. The user will enter space-separated strings using the gets() method. It returns the string that the user typed in. The gets() function is problematic to use because it doesn't search for array bounds and keeps reading characters until the new line (enter) is encountered.

Declaration

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

Program for gets ()

```
#include<stdio.h>
void main ()
{
char s[25];
printf("Enter the string? ");
gets(s);
printf("You entered %s",s);
}
```

puts() function

The puts() function looks a lot like the printf() function. The puts() function is used to print the string that was previously read using the gets() or scanf() functions on the console. The integer value returned by the puts() function represents the number of characters printed on the console. Since it prints an extra newline character with the string, which transfers the cursor to a new line on the console, the integer value returned by puts() will always be the number of characters in the string plus 1.

Declaration

```
int puts(char[])
```

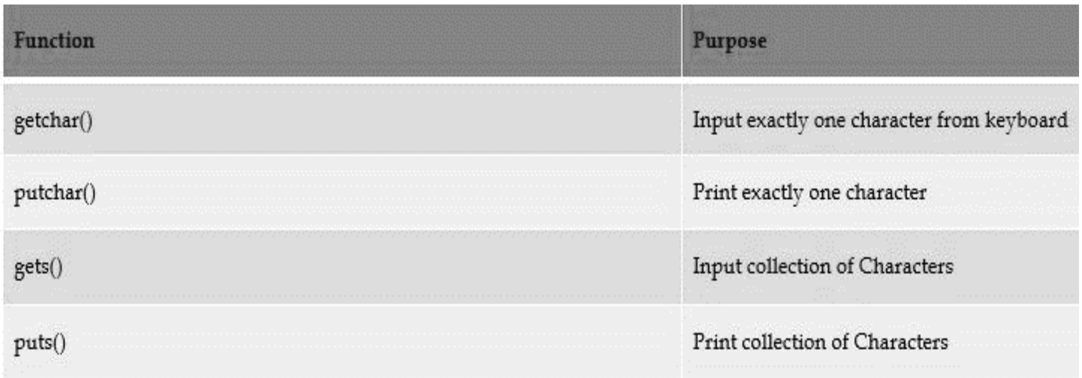
```
#include<stdio.h>
#include <string.h>
int main(){
char name[30];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name);  //displays string
return 0;
}
```

Program For gets() and puts()

```
#include <stdio.h>
void main( )
{
char name[10];
printf("Enter Name");
gets(name);
printf("Entered name is");
puts(name);
return 0;
}
```

Difference between Formatted and Unformatted Functions

- Formatted input and output functions contain format specifier in their syntax.
- Unformatted input and output functions do not contain format specifier in their syntax.
- printf() and scanf() are examples for formatted input and output functions.
- getchar(), gets(), puts(), putchar() etc. are examples of unformatted input output functions.



Function	Purpose
getchar()	Input exactly one character from keyboard
putchar()	Print exactly one character
gets()	Input collection of Characters
puts()	Print collection of Characters

Program

```
#include<stdio.h>
void main()
{
charstr [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): GoodDay
The entered string is: GoodDay
Thank you
```

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.
- 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 within 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 supposed to be treated as an escape sequence.
- Conversion's specifiers always begin with the percent sign (%) and immediately followed by one or more conversions characters
-

Keywords

printf()

scanf()

getchar()	putchar()
gets()	puts()

Header files: 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

Self-Assessment

1. Which format specifier is used for integer

A. % c

B. % d

C. % f

D. % h

2. Which is correct file pointer for standard input

A. Stdout

B. Stdin

C. Stderr

D. Stdab

3. Which function is used for output on screen

A. Main ()

B. Scanf ()

C. Printf ()

D. Get ()

4. What is extension of header file?

A. .c

B. .b

C. .e

D..h

5. main ( ) is a

A. Keyword

B. Function

C. Integer

D. Pointer

6. Which format specifier is used for float?

A. % c

B. % d

C. % h

D. % f

7. Input is taken using function….

A. Printf ()

B. Fprintf ()

C. Scanf ()

D. Main ()

8. Chose the odd one.

A. Put()

B. Printf()

C. Putchar()

D. Scanf()

9. Putchar( ) is used for..

A. As input function for string

B. display exactly one character as output.

C. Display one string on screen

D. Input from user

10. Getchar( ) having similarities with..

A. Puts ( )

B. Gets ( )

C. Main ( )

D. Putchar ( )

11. Default return-type of getchar( )?

A. Int

B. float

C. char

D. reading character doesn't require a return-type

12. What is the return value of putchar( )?

A. EOF if an error occurs

B. Nothing

C. Both character written & EOF if an error occurs

D. All of the above

13. Puts() function is used for

A. Print collection of Characters

B. Input collection of Characters

C. Print exactly one character

D. All of above

14. Which statement is incorrect

A. char ch;

B. ch = getchar ( ) ;

C. putchar (ch) ;

D. main ( ) :

15. Which one is true about formatted functions?

A. getchar(), gets(), puts(), putchar() etc. are examples of formatted input output functions.

B. printf() and scanf() are examples for formatted input and output functions.

C. All of the above

D. None of the above

Answer for Self Assessment

1.	B	2.	B	3.	C	4.	D	5.	B
6.	D	7.	C	8.	D	9.	B	10.	B
11.	A	12.	C	13.	A	14.	D	15.	B

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

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( )?

Further Readings

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.

YashvantKanetkar, Let us C

Web Links

https://www.tutorialspoint.com/index.htm

www.webopedia.com

www.web-source.net

Unit 04: Operations

CONTENTS

Objectives

Introduction

4.1 Operators

4.2 Unary operator

4.3 Binary operator

4.4 Arithmetic Operators

4.5 Relational Operators

4.6 Logical Operators

4.7 Assignment Operators

4.8 Increment and Decrement Operators

4.9 Conditional Operators

4.10 Bitwise Operators

4.11 Special Operators

4.12 Operator Precedence and Associativity

Summary

Keywords

Self-Assessment

Answer for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Various Data types
- Explain arithmetic operators
- Describe conditional, logical and relational operators
- Describe arithmetic expression

Introduction

The term "data types" refers to a comprehensive framework for defining variables and functions of various types. The type of a variable decides how much storage space it takes up and how the stored bit pattern is interpreted. An operator is a symbol that instructs the machine to perform mathematical or logical operations on data held in variables. A value's data form identifies a set of properties that it shares.

An expression is a set of constants, variables, and operators that follow the grammatical rules of the language C and evaluate to a valid value. Activity refers to the impact that operators have on their operands.

Valid or well-formed expression refers to an expression that follows the grammar rules of the C programming language. An expression that is true or well evaluates to a single value of a valid C data form. 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

An operator is a symbol that instructs the machine to perform mathematical or logical operations on data held in variables. Data items (variables and constants) can be manipulated arithmetically in a C program using operators.

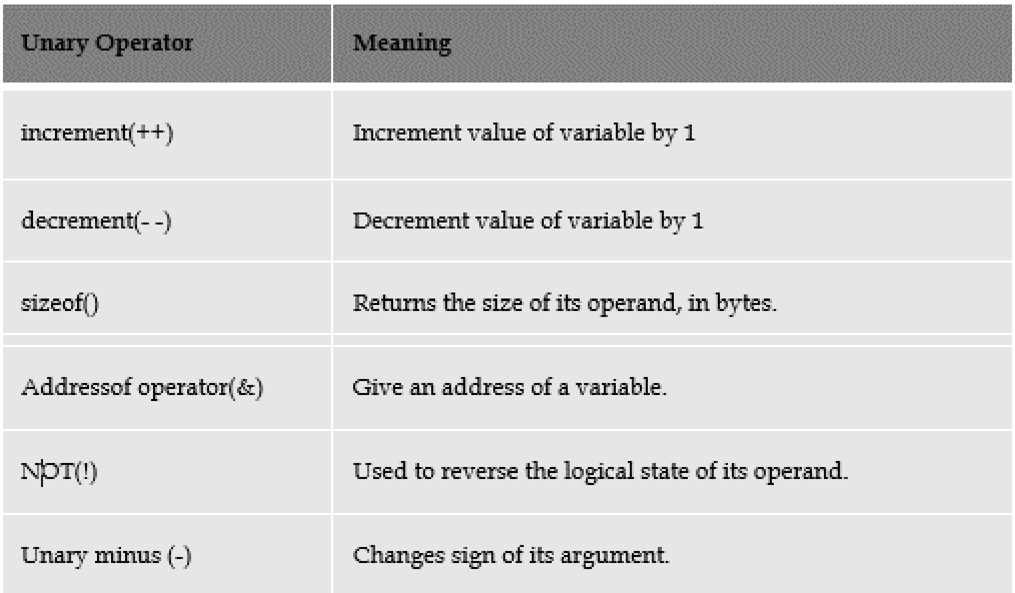
C operators can be classified into a number of categories.

- Unary operator
- Binary operator
- Arithmetic operators
- Relational operators
- Logical operators
- Conditional operators
- Assignment operators
- Bitwise operators
- Increment and decrement operators
- Special operators

4.2 Unary operator

Unary operators are operators that act upon a single operand to produce a new value.

There are following types unary operators in C



Unary Operator	Meaning
increment(++)	Increment value of variable by 1
decrement(- -)	Decrement value of variable by 1
sizeof()	Returns the size of its operand, in bytes.
Addressof operator(&)	Give an address of a variable.
NOT(!)	Used to reverse the logical state of its operand.
Unary minus (-)	Changes sign of its argument.

4.3 Binary operator

Binary operators are those that operate on two operands. For example, a + b—the addition operator (+) surrounded by two operands—is a popular binary expression. Arithmetic, relational, logical, and assignment operators are all subsets of binary operators.

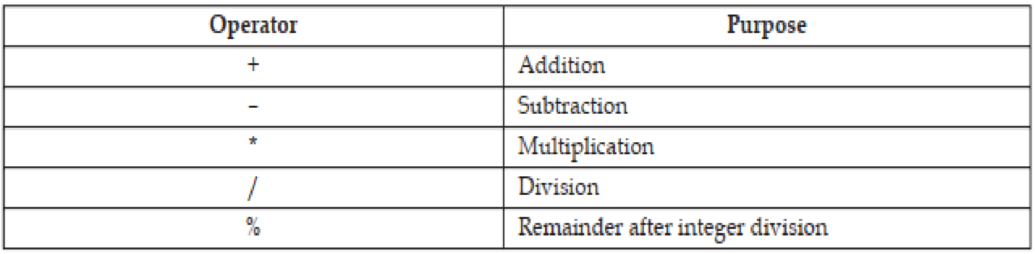
The binary operators are categorized as follow:

- Multiplicative operators: multiplication (*), remainder (%), and division (/)
- Additive operators: addition (+) and subtraction (-)
- Shift operators: left shift (<<) and right shift (>>)
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=)
- Equality operators: equality (==) and inequality (!=)
- Bitwise operators: AND (&), OR (|), and XOR (^)
- Logical operators: AND (&&) and OR (||)

4.4 Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations include addition, subtraction, multiplication and division on one or more operands.

It is used to performed operations on numeric data types like int, float and double 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;
```

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.

4.5 Relational Operators

The relational operator compares two operands to see if they are equivalent, unequal, or if one is greater or inferior to the other.

The consequence is always a numeric value equal to true or false, regardless of whether the operands are variables, constants, or expressions. As previously stated, a non-zero result indicates fact, while a zero result indicates false. Six relational operators are available in the C programming language.

= =	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 is0.

Example:

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

4.6 Logical Operators

Using logical operators, several relational expressions may be combined to create a single compound relational expression. To combine two or more relational statements, logical operators are used. Three logical operators are available in C. A compound relation behaves in the same way as a single relation generating a real 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

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.

4.7 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>.

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.

```
int averylongvariablename;

averylongvariablename = 2;

while (averylongvariablename < 20)

{

averylongvariablename*= averylongvariablename;

}
```

4.8 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
```

4.9 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

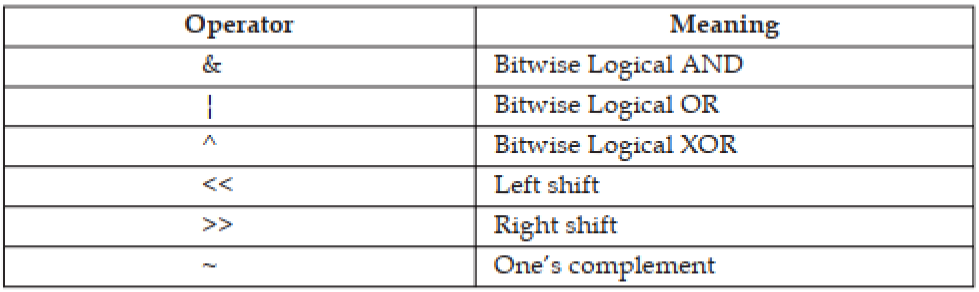
value generated by this operator in the expression depends on the value of the conditional

expression "A". If the value of "A" is true then the expression evaluates to "B" otherwise it

results in "C".

4.10 Bitwise Operators

You're aware that a numeric value is stored in binary form in a variable. Data is manipulated at the bit level using bitwise operators. These operators are used to measure bits or to move them left or right. Only integer data types are supported by bitwise operators. The following is a list of the various bit wise operators available in the C language, along with their corresponding meanings.



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]

>> (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]

4.11 Special Operators

The C programming language also includes a variety of special operators that are not found in other languages. The comma operator, size of operator, pointer operators (& and *), and member selection operators (. and ->) are among these operators. When we introduce pointers, we'll talk about pointer operators, and when we talk about structures and unions, we'll talk about member selection operators.

We will discuss comma operator and size of 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.

Size of Operator

The size of 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

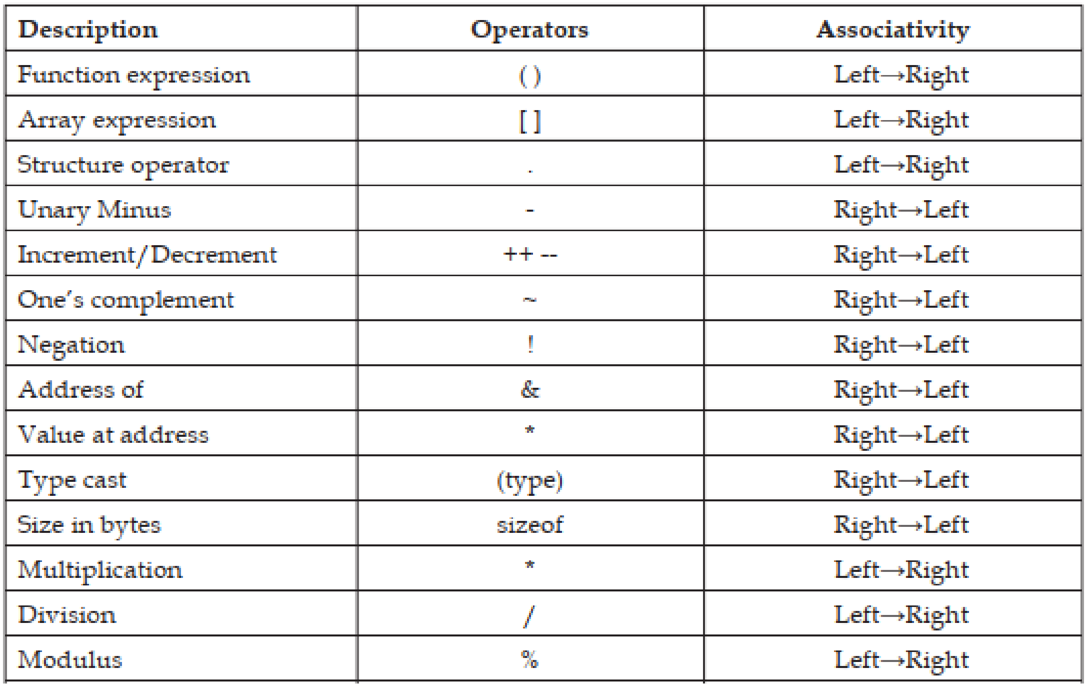
//float type variable

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 size of operator returns the number of bytes occupied by its operand on that particular machine.

4.12 Operator Precedence and Associativity

More than one operator can be used in a single sentence. The order in which operators are executed is determined by their precedence. When evaluating expressions involving multiple operators, precedence specifies the order in which operators should be added to the operands.

Depending on the level, operators of the same precedence are evaluated from left to right or right to left. This is referred to as an operator's associativity property. The following is a full list of operator precedence in the C language.



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 Less than or equal to	< < =	Left→Right 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

Summary

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

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.

Self-Assessment

1.Choose the correct statements for unary operator

A. Unary operators are operators that act upon a single operand to produce a new value.
B. Unary operator acts like constant
C. Unary operator is used for logic operations
D. Above all
E.

2.Choose the correct statements for operator

A. constant expressions are evaluated at compile time

B. An operator perform predefined action on operand.
C. Main( ) function is a operator
D. All of the above

3.Which is correct basic data type

A. Int
B. Char
C. double
D. Above all

4.In C programming language, which of the following type of operators have the highest precedence

A. Relational operators
B. Equality operators
C. Logical operators
D. Arithmetic operators

5.Which is not unary operator

A. ++
B. --
C. &
D. +

6.Relational operator is used with

A. 1 operand.
B. 2 operands
C. Both 1 operand and 2 operand
D. none of above

7.Which among the following is NOT a logical operator

A. &&
B. | |
C. !
D. @

8.Find out the wrong statement

A. a>b
B. b>=c
C. a<=> c
D. a++

9.Which among the following is NOT a relational operator

A. >=
B. !=
C. ==
D. | |

10. Find out the wrong statement

A. if (a>b)
B. if(x=10 && y=20)

C. if (a<2 | | b> 30)
D. if (c=10 ! d>20)

11. Which one is not assignment operator

A. +=
B. *=
C. &
D. >>=

12. (&=) operator is used for…..

A. Bitwise exclusive OR and assignment operator
B. Bitwise inclusive OR and assignment operator.
C. Bitwise AND assignment operator
D. None of these

13. Which one is Modulus AND assignment operator

A. <<=
B. %=
C. /=
D. None of these

14. In c which operators having highest priority

A. Logical operator
B. Relational operator
C. Arithmetic operator
D. Conditional operator

15. Conditional operator in c is also called

A. Relational operator
B. Logical operator
C. Ternary operator
D. None of these

Answer for Self Assessment

1.	A	2.	B	3.	D	4.	D	5.	D
6.	C	7.	D	8.	C	9.	D	10.	C
11.	C	12.	C	13.	B	13.	C	14.	C

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. Write in detail about the various data type available in C.

Further Readings

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

Web Links

https://www.tutorialspoint.com/index.htm

www.webopedia.com

www.web-source.net

Unit 05: Decision making statements

CONTENTS

Objectives

Introduction

5.1 Designing Structured Programs in C

5.2 Top Down Design

5.3 Type Conversion in C

5.4 Type Modifiers

5.5 Decision-Making Control Statement

5.6 Simple if Statement

5.7 if-else statement

5.8 nested if statement

5.9 Nested else-if statement

5.10 Switch statement

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

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
- Understand Type Conversion and Type Modifiers

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.

In the same 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.

5.1 Designing Structured Programs in C

A C program is divided into different sections. There are six main sections to a basic c program.

- Documentation
- Link
- Definition
- Global Declarations
- Main functions
- Subprograms

Documentation Section

The documentation section is the part of the program where the programmer gives the details associated with the program. He usually gives the name of the program, the details of the author and other details like the time of coding and description. It gives anyone reading the code the overview of the code.

Example:

```
**

* File Name: Helloworld.c

* Author: Name

* date: DD/MM/YYYY

* description: a program to display hello world

*           no input needed

*/
```

Link Section

This part of the code is used to declare all the header files that will be used in the program. This leads to the compiler being told to link the header files to the system libraries.

Example: `#include<stdio.h>`

Definition Section

In this section, we define different constants. The keyword define is used in this part.

Example: #define PI=3.14

Global Declaration Section

The declaration of global variables takes place in this section of the code. This section declares all of the global variables that are used. In this section of the code, the user-defined functions are also declared.

Example:
```
float area(float r);
int a=5;
```

Main Function Section

The main role is required in all C programmes. There are two sections of each key role. There are two parts of this: a declaration part and an execution part. All of the variables are declared in the declaration section. The curly brackets are used to start the execution and the curly close bracket is used to end it. The curly braces contain both the declaration and execution sections.

```
int main(void)
{
int a=500;
printf(" %d", a);
return 0;
}
```

Sub Program Section

This section of the software defines all user-defined functions.

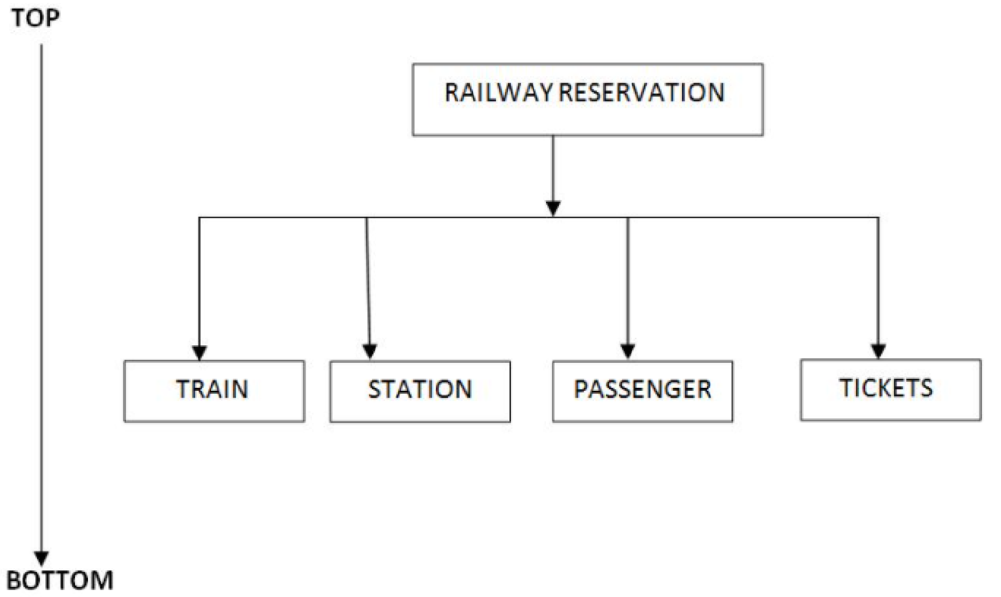
```
int add(int a, int b)
{
return a+b;
}
```

5.2 Top Down Design

A large project is divided into small projects, which are referred to as modules, in this approach. This method of project development is supported by the C programming language. Decomposing a solution into modules in a hierarchical fashion is often a good idea. A top-down approach's basic task is to divide the problem into tasks, which are then divided into smaller sub-tasks, and so on. In this method, the main module is built first, followed by the next level modules. This process is repeated until all of the modules have been developed.

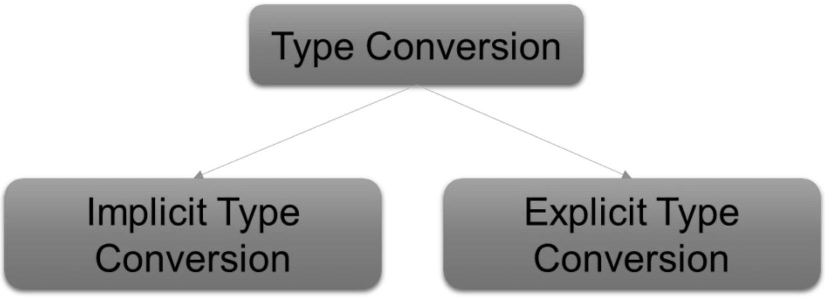
Advantages of top-down approach:

1. In this approach, first, we develop and test most important module.
2. This approach is easy to see the progress of the project by developer or customer.
3. Using this approach, we can utilize computer resources in a proper manner according to the project.
4. Testing and debugging is easier and efficient.
5. In this approach, project implementation is smoother and shorter.
6. This approach is good for detecting and correcting time delays.


TOP
RAILWAY RESERVATION
TRAIN
STATION
PASSENGER
TICKETS
BOTTOM

5.3 Type Conversion in C

Converting one type of data type to other to perform some operations. Conversion is done on those data types where conversion is possible.


Type Conversion
Implicit Type Conversion
Explicit Type Conversion

Implicit Type Conversion

This type of conversion is done by compiler whenever necessary automatically without any commands by user. This conversion is performed by compiler when a particular expression contains more than one data type.

```
Example: int a = 10;
            double b = 50.5,sum;
            sum=a + b;
```

Lab Exercise:

```
 #include<stdio.h>
int main(){
int a=10;
double b=9.34,sum;
sum=a+b;
printf("%lf",sum);
return 0;
}
```

Explicit Type Conversion

Without the use of compiler when one data type is changed to another data type, user explicitly defines within the program the data type of the operands in the expression.

Example:

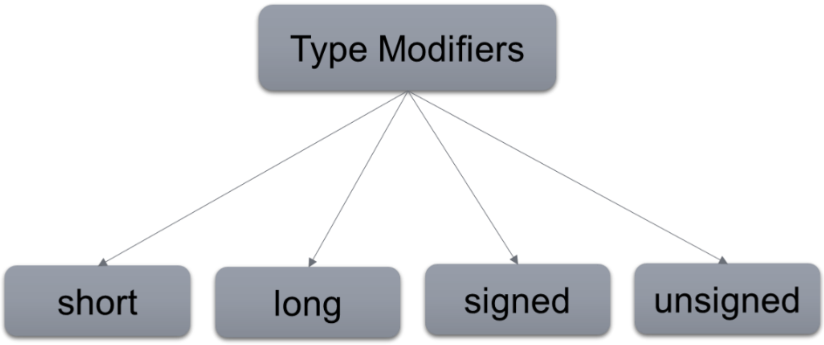
```
double a = 4.5,b=3.5;
 //explicitly defined by user
 int sum=(int)a+(int)b;
```

Lab Exercise:

```
#include<stdio.h>
int main(){
double a = 10.5,b=5.5;
//explicitly defined by user
int sum=(int)a+(int)b;
printf("%d",sum);
return 0;
}
```

5.4 Type Modifiers

In c language Data Type Modifiers are keywords used to change the current properties of data type.



Long

This can be used to increased size of the current data.it is applied on int or double data type.

Example: long double a;

Lab Exercise:

```
#include<stdio.h>
int main(){
long double  a ;
printf("%d",sizeof(a));
return 0;
}
```

Short

In general int data type occupies different memory spaces for a different operating system; to allocate fixed memory space short keyword can be used.

Example: short int a;

Lab Exercise:

```
#include<stdio.h>
int main(){
short int a ;
printf("%d",sizeof(a));
return 0;
}
```

Unsigned

This keyword can be used to make the accepting values of a data type is positive data type.

```
Example: unsigned int a=10;          //right
         unsigned int a= -50;        //wrong
```

Lab Exercise:

```
#include<stdio.h>
int main(){
unsigned int  a=-10 ;
printf("%u", a);
return 0;
}
```

Signed

This keyword accepts both positive and negative value.

Example: unsigned int a=10; //right
unsigned int a= -50; //right

Lab Exercise:

```
#include<stdio.h>
int main(){
signed int a=-10 ;
printf("%d", a);
return 0;
}
```

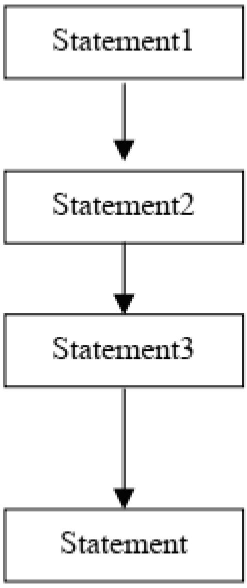
5.5 Decision-Making Control Statement

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

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.

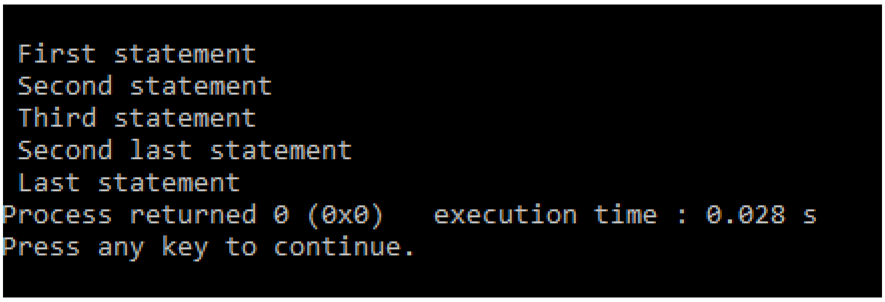


You probably noticed in the previous unit that a C programme is executed top down, that is, execution begins with the beginning of the main() function and continues, statement by statement, until the end of the main (). In a C programme, the following programme illustrates the sequential execution of statements.

Lab Exercise:

```
#include<stdio.h>
main(){
printf("\n First statement");
printf("\n Second statement");
printf("\n Third statement");
printf("\n Second last statement");
printf("\n Last statement");
}
```

Output : -

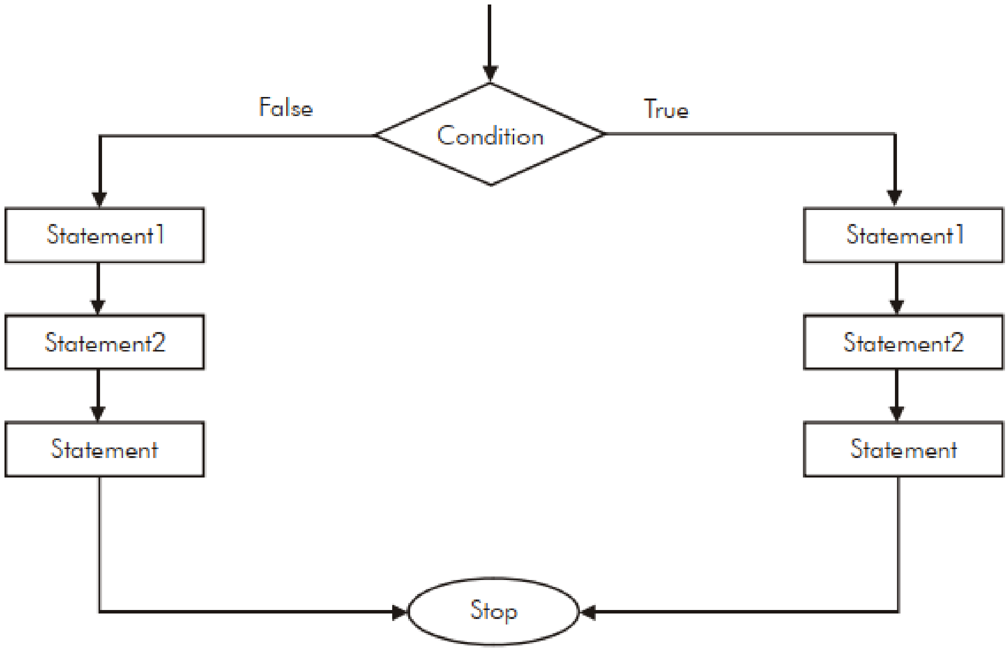


```
 First statement
 Second statement
 Third statement
 Second last statement
 Last statement
Process returned 0 (0x0)   execution time : 0.028 s
Press any key to continue.
```

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

Selection

Depending on the input and the situation involved, only appropriate statements are often executed instead of all of them. The execution of statements in the selection construct is based on a condition test. If the test returns true, you instruct the programme to take one course of action; otherwise, you instruct the programme to take a different course of action. Two or more sets of statements are written in the selection construct, but only one of these sets is executed, as shown in the Figure.

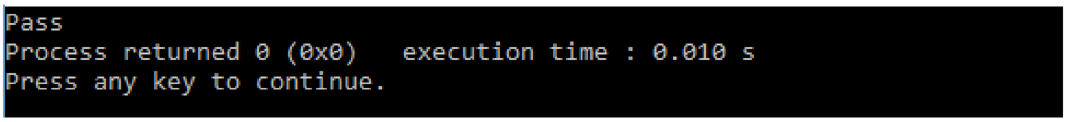

False
Condition
True
Statement1
Statement2
Statement
Statement1
Statement2
Statement
Stop

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:

Lab Exercise:

```
#include<stdio.h>
main(){
int marks=50;
if(marks>40)
   printf("Pass");
else
   printf("Fail");
}
```

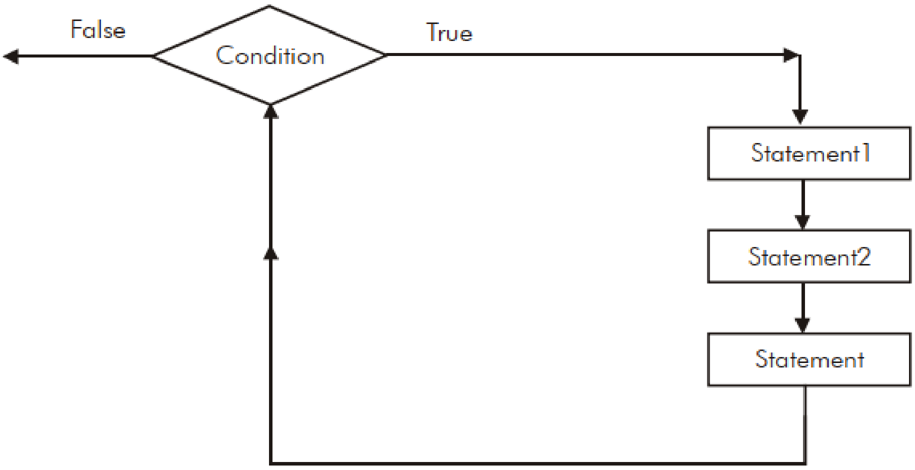
Output: -



```
Pass
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

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.


False
Condition
True
Statement1
Statement2
Statement

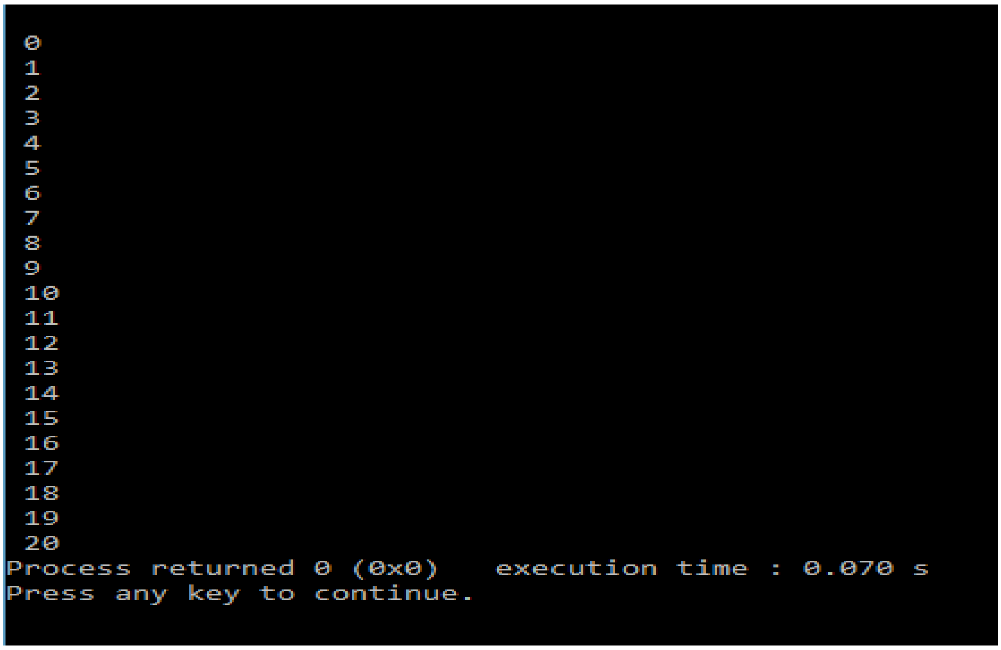
The iteration construct is also called as loop. The statements that are to be executed is called

as body of the loop and the condition on which a loop terminates is called as exit condition as demonstrated by the program given below:

Lab Exercise:

```
#include<stdio.h>
main(){
int i=0;
while(i<=20){
  printf("\n %d ",i);
  i++;
}
}
```

Output: -



```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
 12
 13
 14
 15
 16
 17
 18
 19
 20
Process returned 0 (0x0)   execution time : 0.070 s
Press any key to continue.
```

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.

Notes: What would be the output of following program?

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

5.6 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)
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:

```
#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
```

Notes: “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()
{
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
```

Notes: 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”.

5.7 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.

```
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:

```
#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
{
printf (“\n Fail”);
printf (“\n Sorry. Good luck next time …”);
} }
```

```
Output: run1 → run2 →
Enter your marks: 77→ Enter your marks : 30→
Pass Fail
Congratulations…. Sorry. Good luck next time…
```

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

5.8 nested if statement

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
{
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 ()
{
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
```

Notes: 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 evaluates to true.

5.9 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
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
```

Notes: 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.

5.10 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>
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
```

Notes: 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.

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.

Keywords

Conditional statement: A statement that evaluates to either true or false.

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.

Self Assessment

1. Every C program must have

A. Data type
B. Constant
C. Main Function
D. None of the above

2. Choose the right statement

A. User defined functions is known as sub-program.
B. Sub program is a constant in C program
C. Sub program and main function is same
D. All of above

3. Global declaration is __________

A. A sub program
B. A data type
C. A part of C program where we can declare global variables.
D. All of the above

4. How many type of conversion are there in c?

A. 4
B. 2
C. 3
D. 1

5. Which conversion also called Automatic Type Conversion?

A. Implicit Type Conversion
B. Explicit Type Conversion
C. All of above
D. None of the above

6. What will be output for the following code?

```
#include<stdio.h>
int main()
{
  double x = 2.2;
  int sum = (int)x + 5;
 printf("sum = %d", sum);
  return 0;
}
```

A. 2
B. 2.2
C. 5
D. 7

7. Which type of conversion is NOT accepted?

A. From char to int
B. From float to char pointer

C. From negative int to char
D. From double to char

8. Type modifiers in C are

A. Int, main
B. Sub program
C. Short, long, signed and unsigned
D. All of above

9. What will be output for the following code?

```
 #include <stdio.h>
 int x;
 void main()
 {
    if (x)
       printf("hi");
    else
       printf("how are u");
 }
```

A. hi
B. how are u
C. compile time error
D. error

10. What will be output for the following code?

```
#include <stdio.h>
 void main()
 {
    int x = 5;
    if (true);
       printf("hello");
 }
```

A. It will display hello
B. It will throw an error
C. Nothing will be displayed
D. Compiler dependent

11. What will be the output of following program?

```
 #include <stdio.h>
void main()
{
    char val=1;
    if(val--==0)
        printf("TRUE");
    else
        printf("FALSE");
}
```

A. FALSE
B. Error
C. TRUE
D. None

12. Which of the following is an invalid if-else statement?

A. if (if (a == 1)){}
B. if (func1 (a)){}
C. if (a){}
D. if ((char) a){}

13. The label in goto statement is same like

A. Case in switch statement
B. Initialization in for loop
C. Continuation condition in for loop
D. All of them

14. What will be the output of following program?

```
#include <stdio.h>
void main()
{
    int a=10;
    switch(a){
        case 5+5:
            printf("Hello ");
        default:
            printf("World\n");
    }
}
```

A. Hello
B. World
C. Hello World
D. Error

15. What will be the output of following program?

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

```
void main()
{
   short day=2;
   switch(day)
   {
      case 2: || case 22:
         printf("%d nd",day);
      break;
      default:
         printf("%d th",day);
      break;
   }
}
```

A. 2 nd
B. 22 nd
C. Error
D. 2 nd 22 nd

16. What will be the output of following program?

```
#include <stdio.h>
int main()
{
int x = 2;
switch (x)
{
case 1: printf("x =1");
break;
case 2: printf("x= 2");
break;
  case 3: printf("x= 3");
   break;
default: printf("x other than 1, 2 and 3");
break;
}
return 0;
}
```

A. x=1
B. x=2
C. x=3
D. All of above

Answers for Self Assessment

1. C 2. A 3. C 4. B 5. A

6. D 7. B 8. C 9. B 10. B

11. A 12. A 13. A 14. C 15. C

16. B

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.

Further Readings

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

Web Links

https://www.tutorialspoint.com/index.htm

www.webopedia.com

www.web-source.net

Unit 06: Loop Statements

CONTENTS

Objectives
Introduction
6.1 Looping
6.2 for loop
6.3 While Loop
6.4 do-while loop
6.5 Jump and Break Statement
6.6 goto Statement
Summary
Keywords
Self-Assessment
Answer for Self Assessment
Review Questions
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.

6.1 Looping

Iteration statements are also known 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

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 whereas 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 evaluates 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 evaluated 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. For, while and do-while statements of C, comprises of all these essential components, hence referred to as loop statements.

6.2 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 whereas 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;

Consider the following statement:

```
for ( i =1 ; i<= 10; ++i)
printf(" \n Hello World!");
```

wherei 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.

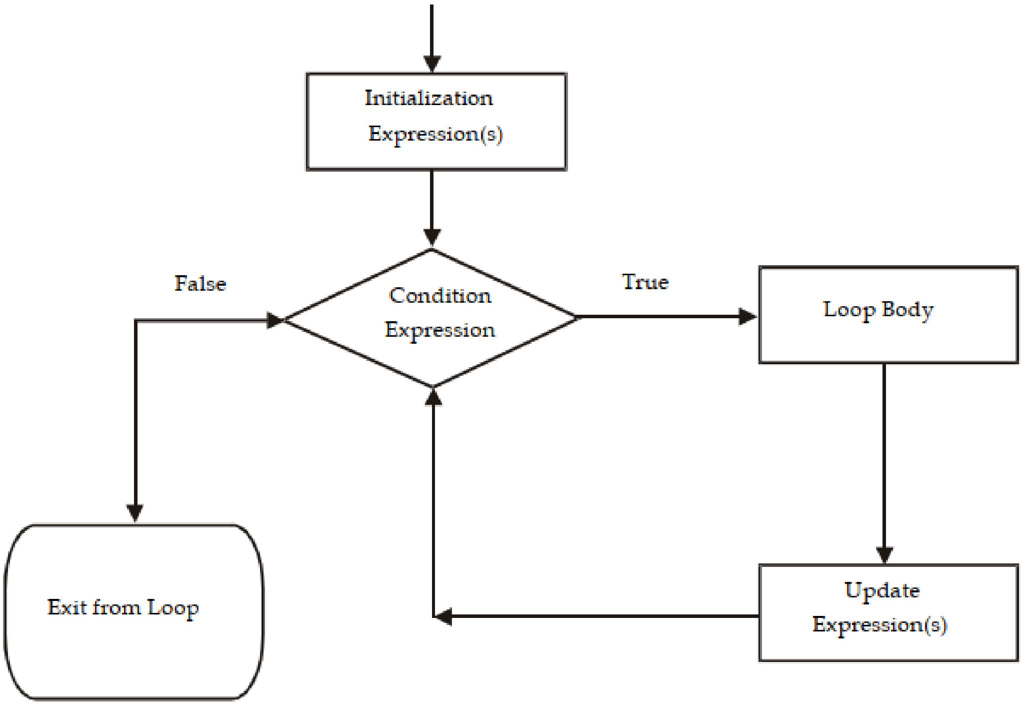
5. After the execution of the update expression the condition expression is again evaluated. If it evaluates to true the sequence is repeated from step no. 3, otherwise the loop terminates.

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 shows the operation of a for loop.

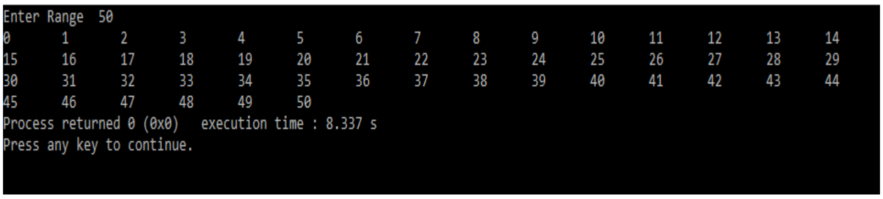


Program :-

#include<stdio.h>

```
int main(){


inti,range;
printf("Enter Range");
scanf("%d",&range);
for(i=0;i<=range;i++)
{


  //  printf("%d\n",i);
printf("%d\t",i);
}
return 0;
}
```

Output : -

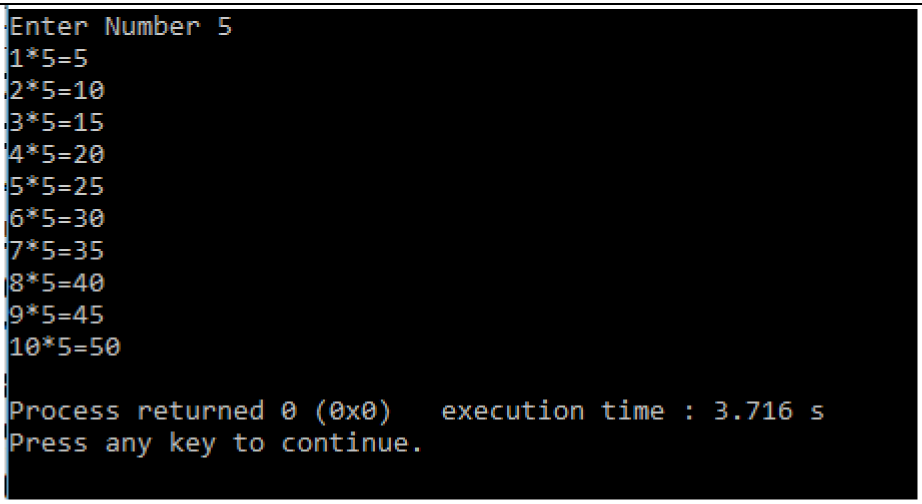


```
Enter Range  50
0       1       2       3       4       5       6       7       8       9       10      11      12      13      14
15      16      17      18      19      20      21      22      23      24      25      26      27      28      29
30      31      32      33      34      35      36      37      38      39      40      41      42      43      44
45      46      47      48      49      50
Process returned 0 (0x0)   execution time : 8.337 s
Press any key to continue.
```

Program :

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


inti,n;
printf("Enter Number");
scanf("%d",&n);
for(i=1;i<=10;i++)
{
printf("%d*%d=%d\n",i,n,i*n);


}
return 0;


}
```

Output : -



```
Enter Number 5
1*5=5
2*5=10
3*5=15
4*5=20
5*5=25
6*5=30
7*5=35
8*5=40
9*5=45
10*5=50

Process returned 0 (0x0)   execution time : 3.716 s
Press any key to continue.
```

6.3 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 iterations. The statements repeat 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 contain 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 contain 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 updating 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;

}

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

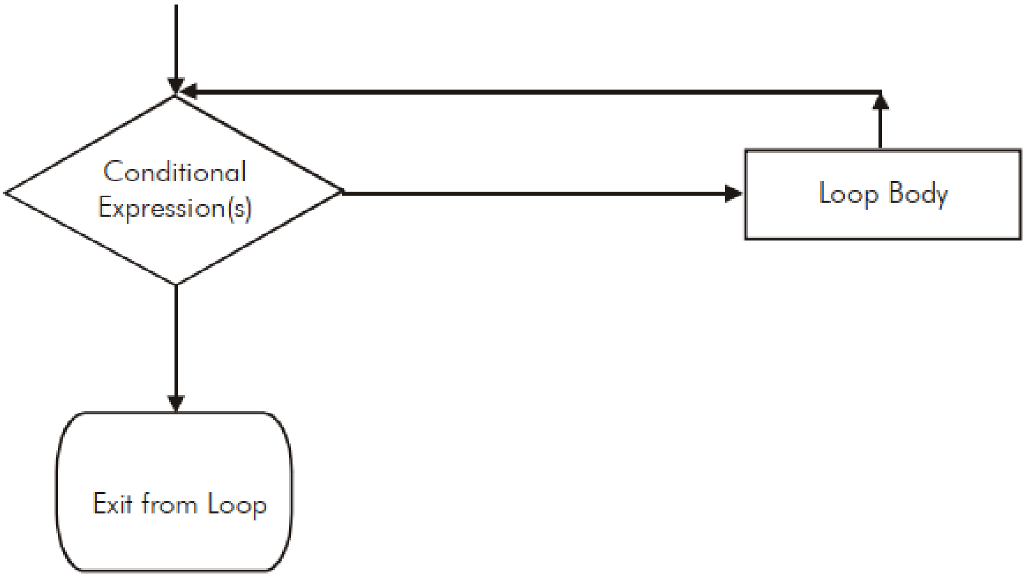
When the program execution readers a while statement, the following events occur:

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 evaluates 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 print statement and the updating 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.

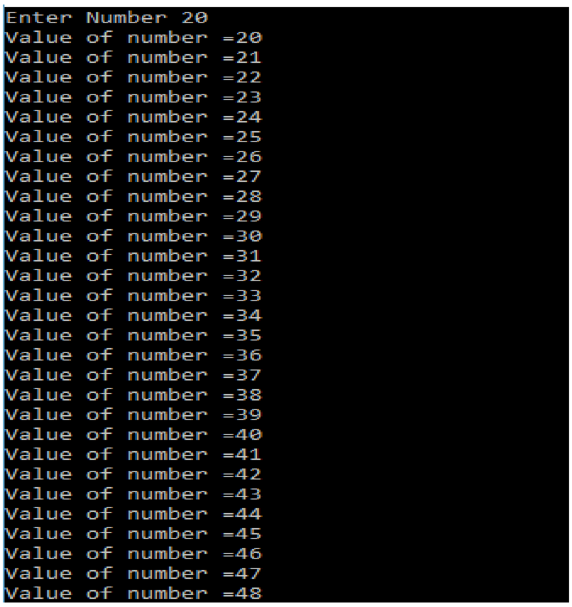


Program : -

```
#include<stdio.h>
int main(){
int number;
printf("Enter Number");
scanf("%d",&number);
while(number<=50)
{
```

```
printf("Value of number =%d\n",number);
  number++;
}
return 0;
}
```

Output : -

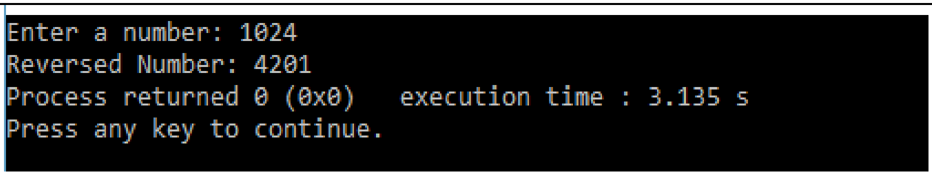


```
Enter Number 20
Value of number =20
Value of number =21
Value of number =22
Value of number =23
Value of number =24
Value of number =25
Value of number =26
Value of number =27
Value of number =28
Value of number =29
Value of number =30
Value of number =31
Value of number =32
Value of number =33
Value of number =34
Value of number =35
Value of number =36
Value of number =37
Value of number =38
Value of number =39
Value of number =40
Value of number =41
Value of number =42
Value of number =43
Value of number =44
Value of number =45
Value of number =46
Value of number =47
Value of number =48
```

Program : -

```
#include<stdio.h>
int main()
{
int n, reverse=0, rem;
printf("Enter a number: ");
scanf("%d", &n);
 while(n!=0)
 {
   rem=n%10;
   reverse=reverse*10+rem;
   n/=10;
 }
printf("Reversed Number: %d",reverse);
return 0;
}
```

Output : -



```
Enter a number: 1024
Reversed Number: 4201
Process returned 0 (0x0)   execution time : 3.135 s
Press any key to continue.
```

6.4 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

{

loop-boody ;

}while (conditional expression) ;

The braces { } can be discarded when the loop-boody contains a single statement. The do-whileloop 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 whileloop.

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));

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 (initially=1).

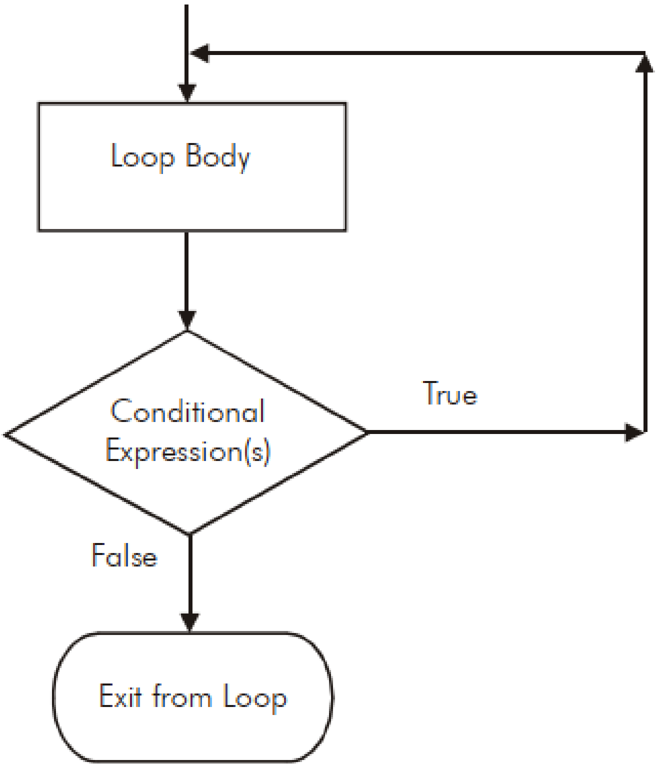
4. Since the condition expression is true, the control will move back to execute the loop-bodyonce again.

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.

The output of the above code may looks likes as:

```
Hello World !

Hello World !

:

Hello World ! (10 times)
```

Figure demonstrate working of do-while loop



The do-while loop is well suited for the problems where number of repetitions is not known inadvance. But this is sure that the loop will be executed at least once. The following segment ofcodes 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
```

```
i=11;
do
{
printf(“\n% d”, i);
++i;
} while (i<=10);
printf(“\n Thank you”);
```

Output:

```
11
Thank you
```

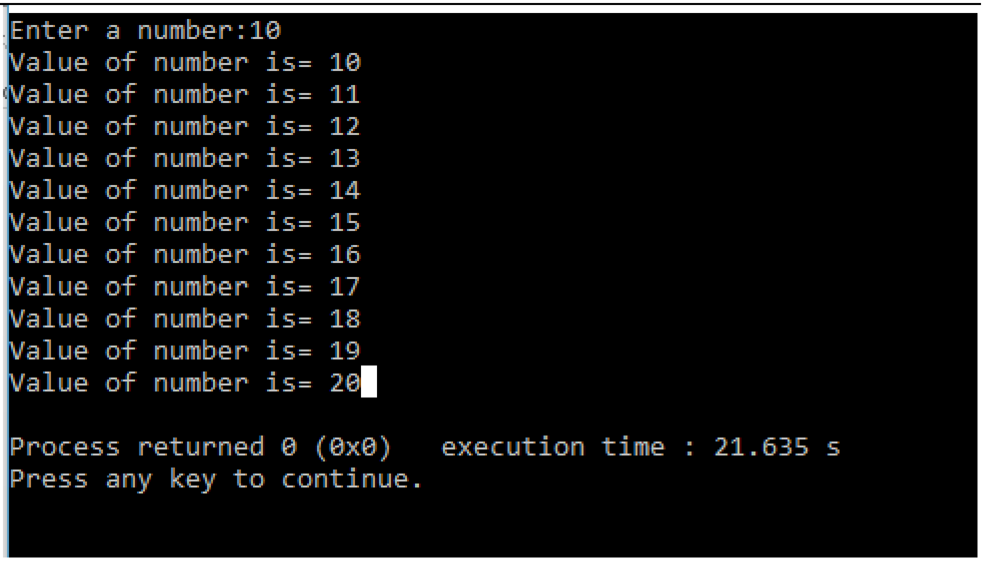
It's obvious from the code above that a do-while loop is always executed at least once, regardless of the condition's outcome. Since the condition expression is evaluated at the end of the loop rather than at the beginning.

Program to print series of number after enter first number

Program :-

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

int number;
printf("Enter a number:");
scanf("%d",&number);
do{

printf("Value of number is= %d\n",number);
++number;

}while(number<=20);
return 0;

}
```

Output:-



```
Enter a number:10
Value of number is= 10
Value of number is= 11
Value of number is= 12
Value of number is= 13
Value of number is= 14
Value of number is= 15
Value of number is= 16
Value of number is= 17
Value of number is= 18
Value of number is= 19
Value of number is= 20

Process returned 0 (0x0)   execution time : 21.635 s
Press any key to continue.
```

Let’s write another interesting program using do-while loop. This program will printf thealphabet A to Z along with their ASCII codes.

Program : -

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

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

6.5 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 froma switch (discussed later). When break is encountered inside any C loop, the loop is immediatelyexited without testing the loop condition and control automatically passes to the first statementafter the loop. It can be used within a while, a do-while, a for or a switch statement. The breakstatement 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;
}
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 transferof control out of the immediate enclosing statement, but not out of the outer surroundingstatements.

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.

Write a program to print all prime numbers from 1 to 300. (Use nested loops, break and continue)

6.6 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:

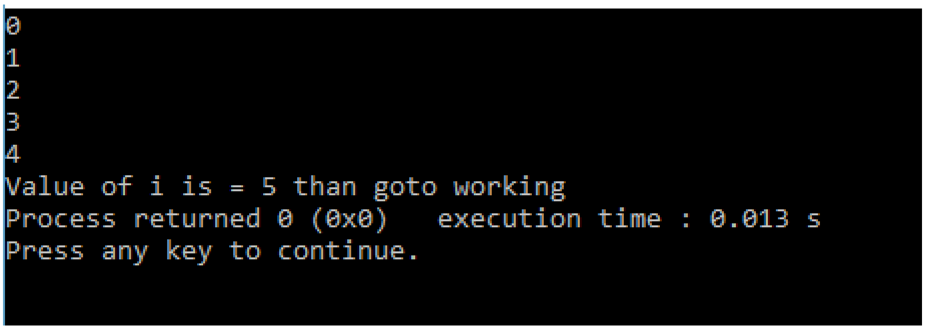
```
            goto label;
…..
…..
label: statements;
            . . . . . . . .
            statement;
```

The label: can by anywhere in the program either before or after the goto label; statement.

There is no built-in mechanism for the flow of execution to come back from where it branched.

```
/*Program to demonstrate working of goto statement*/
#include<stdio.h>
int main(){

inti;

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

{
   if(i==5)
   {
goto STATUS;

   }
printf("%d\n",i);

}
STATUS:
printf("Value of i is = 5 than goto working");
return 0;

}
```

Output :-



```
0
1
2
3
4
Value of i is = 5 than goto working
Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

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 Updating 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 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.

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.

goto statement: The goto statement is known as jump statement in C.

Self-Assessment

1. Choose a right C Statement.

   A. Loops or Repetition block executes a group of statements repeatedly.

   B. Loop is usually executed as long as a condition is met.

   C. Loops usually take advantage of Loop Counter

   D. All the above.

2. Choose a correct C for loop syntax.

   A. for(initialization; condition; increment operation)

   ```
   {
    //statements
   }
   ```

   B. for(declaration; condition; increment operation)

   ```
   {
     //statements
   }
   ```

   C. for(declaration; increment operation; condition)

   ```
   {
     //statements
   }
   ```

   D. for(initialization; condition; increment operation;)

   ```
   {
   //statements
   }
   ```

3. How many times Hello be printed in the following program?

```
#include <stdio.h>
int main()
{
inti;
for (i=0; i<=10; i++)
printf("Hello");
return 0;
}
```

A. 100

B. 10

C. 11

D. Infinite

4. What will be the output of the following C code?

```
  #include <stdio.h>
void main()
  {
int k = 0;
for (k)
Printf ("Hello");
  }
```

A. Compile time error

B. Hello

C. Nothing

D. Varies

5. What is the output of this program?

```
void main()
{
int x=0;
for(;;)
  {
if(x==3)
break;
printf("%d ",++x);
  }

}
```

A. 1 2 3

B. 0 1 2 3

C. 1 2

D. Compilation Error

6. Which loop is guaranteed to execute at least one time.

A. for

B. while

C. do while

D. None of the above

7. do-while loop terminates when conditional expression returns?

A. One

B. Zero

C. Non - zero

D. None of the above

8. Which of the following statement about for loop is true?

A. Index value is retained outside the loop

B. Index value can be changed from within the loop

C. Goto can be used to jump, out of the loop

D. All of these

9. Which one is correct syntax for do-while loop?

A. do

```
{
  //Statements

}while(condition test);
```

B. do,

```
{
  //Statements

}while(condition test);
```

C. do,

```
{
  //Statements

}while(do);
```

D. do

```
{
  //Statements

}while(do);
```

10. How many times Programming will be printed?

```
#include <stdio.h>
int main()
{
inti = -5;
while (i<= 5)
  {
if (i>= 0)
break;
else
    {
i += 1;
continue;
    }
printf("Programming");
  }
return 0;
}
```

A. 5 times

B. 10 times

C. 0 times

D. Infinite times

11. The break statement is used to exit from:

A. a DO loop.

B. a FOR loop.

C. a SWITCH statement.

D. all of above.

12. The GOTO statement is used:

A. to permit two different expressions to appear in situation where only one expression would ordinarily used.

B. to terminate loops or to exit from a switch.

C. an unconditional transfer of control to a named label.

D. to carry out a logical test and then take one of two possible actions, depending upon the outcome of a test.

13. Destination is specified by a label within the statement, is called

A. While loop

B. Goto statement

C. For loop

D. For each loop

14. Which from the following is a jumping statement?

A. Break statement

B. Continue statement

C. Switch statement

D. All of them

15. Which keyword is used to come out of a loop only for that iteration?

A. break

B. continue

C. return

D. none of the mentioned

Answer for Self Assessment

1.	D	2.	A	3.	C	4.	A	5.	A
6.	C	7.	B	8.	D	9.	A	10.	C
11.	D	12.	C	13.	B	14.	D	15.	B

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.

6. Write a program to print all the ASCII values and their equivalent characters using a whileloop. 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.

Further Readings

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.

YashvantKanetkar, Let us C

Web Links

https://www.tutorialspoint.com/index.htm

www.webopedia.com

www.web-source.net

Unit 07: Functions

CONTENTS

Objectives
Introduction
7.1 Need for User-defined Function
7.2 A Multifunction Program
7.3 Elements of User-defined Functions
7.4 Definition of Functions
7.5 Return Value and their Types
7.6 Function Calls
7.7 Function Declaration
7.8 Category of Functions
7.9 Argument but no Return Values
7.10 Arguments with Return Values
7.11 Functions that Return Multiple Values
7.12 Function Prototype
7.13 Recursive Functions
7.14 Library functions
Summary
Keywords
Self-Assessment
Answers for Self Assessment
Review Questions
Further Readings

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

Introduction

A function is a programming unit with a unique name that may be identified. It can be invoked by a programme once it has been defined. When called, it may accept zero or more inputs. The code placed inside the function specification determines what should be done with the incoming input(s). The function generates a single output after doing the given transformation. The caller of the function receives this output.

7.1 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.

7.2 A Multifunction Program

The use of a function is one of the advantages of the C programming language. Always behave like a standard function or method in C. One function may call another, and so on. In C, there are no limitations on the number of functions that can be called in a programme. It is preferable to decompose the complex problem into tiny, easily manageable parts and develop a function. The control will be passed from the calling programme part to the called function block. If the called function is successfully executed, control will be returned to the programme segment that called it. There is always overhead of a transfer of the control between calling portion and a called function block.

Example: The multifunction program segment is shown below

```
function1 ()
{
        --------------
        --------------
function2 ();
        --------------
        --------------
function4 ();
}
function2 ()
{
        --------------
        --------------
function3 ();
        --------------
        --------------
}
function3 ();
{
        --------------
        --------------
}
function4 ()
{
        --------------
        --------------
}
```

: A program to demonstrate the transfer of control between the multifunction program.

```
Main()
{
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 ()
{
inti,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”);
}
```

7.3 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 there 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.

7.4 Definition of Functions

A function is a standalone piece of executable code that can be called from any other function. The idea of functions comes to mind in many systems when a group of statements must be executed repeatedly at various points in the programme and possibly with different sets of data. Those repeating statements are stored in a function and called as needed. When a function is called, control is sent to the called function, which is then run, before being returned 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 c

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.ommas: (data type1 var1,data type2 var2,…….. data type n var n) for example (int x, float y, char z).

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
```

7.5 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

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)
{
intsq;
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:

```
            charfunc_char( …….. )
{
            char c;
            ……………
            ……………
            ……………
}
```

(b) Code Fragment – 1:

```
            floatfunc_float (……..)
            { fl
            oat 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 callingfunction with the value associated with it.

In the above example, if we take x = 5 and y = 3, then the control will be transferred tothe 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 beexecuted after this returning.

7.6 Function Calls

A function can be called by supplying its name followed by a list of arguments separated by commas and surrounded in parentheses. If a function call does not require any parameters, it must be followed by an empty pair of parenthesis.

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.

e.g.: /* Program to find square of given number */

```
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
```

7.7 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.

```
floatgetsq(x)
float x;
{
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.

```
intaddthem(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.

What would be the output of thus program?

```
Main( )
{
       printf("\nOnly stupids use C?") ;
      Display( );
}
display( )
 {
      Printf("\nFools too use C!") ;
      Main( ) ;
}
```

7.8 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.

```
: /* 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
```

7.9 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.

7.10 Arguments with Return Values

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
```

7.11 Functions that Return Multiple Values

The compiler will raise a compilation error if a function is not called with the required number of properly typed arguments. When calling a function, there are two methods for passing arguments to it: call by value and call by reference.

Call by Value

Call by value means sending the values of the arguments to functions. When a single value ispassed to a function via an actual argument, the value of the actual argument is copied into thefunction. Therefore, the value of the corresponding formal argument can be altered within thefunction, 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.

e.g.:

```
/* A simple C program containing a function that alters the
value of its argument. */
#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.

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.

7.12 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_typefunction_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.

7.13 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;
longint fact (int);
printf (“\n n = “);
scanf (“%d”, &n);
printf (“\n n! = % ld” fact (n));
}
longint fact (int n)
{
if (n < = 1)
return 1;
else
return (n * factorial (n-1));
}
```

7.14 Library functions

The C programming language comes with a set of standard library functions that perform a variety of useful tasks. Library functions implement all input and output operations (e.g., writing to the terminal) as well as all math operations (e.g., sine and cosine evaluation).

It is important to call the proper header file at the start of the programme in order to use a library function. All of the functions in the library in question have a header file that tells the programme their name, type, and number and type of arguments. The preprocessor statement calls a header file.#include<filename>

where filename represents the name of the header file.

A library function is accessed by simply writing the function name, followed by a list of arguments, which represent the information being passed to the function. The arguments must be enclosed in parentheses, and separated by commas: they can be constants, variables, or more complex expressions. Note that the parentheses must be present even when there are no arguments.

Library Functions in Different Header Files

C Header Files

<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions
<math.h>	Mathematics functions

<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

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'.

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.

Self Assessment

1. Which one is not a library function

A. printf ( )
B. scanf ( )
C. gets ( )
D. abc ( )

2. Function call is _____________

A. Print the statement
B. Use header file in program
C. calling a function whenever it is required in a program
D. None of these

3. Functions are used to __

A. Enhances the logical clarity of the program.
B. Helps to avoid repeated programming across programs.
C. Helps to avoid repeating a set of statements many times.
D. All of above

4. Scope of variable in C is______

A. local
B. global
C. intermediate
D. both a and b

5. Variable used inside function is called _____________

A. Global variable
B. Local variable
C. Both a and b
D. None of these

6. In program a and b is a_______

```
int sum()
{
int a=15, b =20;
returnx+y;
}
```

A. Global variable
B. Local variable
C. Intermediate variable
D. None of above

7. int x and int y is a______

```
add(intx,int y)
{
int z;
   z=x+y;
printf("result is= %d",z);
}
```

A. formal Parameter
B. actual Parameter
C. intermediate
D. both a and b

8. Actual parameter is_______

A. parameters that appear in function calls.
B. parameters that appear in function definition.

C. local to the function definition
D. above all

9. Call by value and call by reference is part of ______

A. pointers
B. array
C. functions
D. loops

10. In program we can modify original value in

A. Call by value
B. Call by reference
C. None of above
D. above all

11. A function is called indirect recursive ______

A. if it calls the same function.
B. if it calls the another function.
C. Execute other function
D. Above all

12. Function which call itself is called______

A. Static function
B. Auto function
C. Recursive function
D. above all

13. C Function can return ______ value at a time.

A. 2
B. 3
C. 1
D. 4

14. Scope of variable is______

A. local
B. global
C. intermediate
D. both local and global

15. Call by value and call by reference is part of ______

A. pointers
B. array
C. functions

D. loops

Answers for Self Assessment

1.	D	2.	C	3.	D	4.	D	5.	B
6.	C	7.	A	8.	A	9.	C	10.	B
11.	B	12.	C	13.	C	14.	D	15.	C

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: 12 + 22 + 32 + 42 + ……..n2
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(x1, y1), B(x2, y2), and C(x3, y3). 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.

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.

YashvantKanetkar, Let us C

Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

www.programiz.com

Unit 08: Storage Classes

CONTENTS

Objectives
Introduction
8.1 Storage Classes and their Usage
8.2 Automatic Variable
8.3 External Variable
8.4 External Declaration
8.5 Static Variable
8.6 Register Variable
8.7 Const Qualifier
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After studying this unit, you will be able to:

- storage classes
- Scope of a variable
- Auto, Static, Extern and Register

Introduction

Storage Classes are used to characterize a variable's or function's characteristics. These characteristics include scope, visibility, and life-time, which allow us to track the presence of a variable through the course of a program's execution. The following items are described by a storage class in C:

The variable scope.

The location where the variable will be stored.

The initialized value of a variable.

A lifetime of a variable.

Who can access a variable?

8.1 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

8.2 Automatic Variable

An automatic variable's scope is limited to the function in which it is declared. When the function is called, it is formed, and when the function is exited, it is automatically deleted. As a result, the name Automatic was chosen.

Local variables are variables defined with the auto storage class. The term "auto" refers to the automatic storage class. If a variable is not explicitly declared, it is in the auto storage class by default.

The scope of an auto variable is restricted to a single block. The access is destroyed once the control leaves the block. This means that the auto variable can only be accessed from the block in which it is declared.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

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.

```
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
```

8.3 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.

```
 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.

```
/* illustration of working of global variable int x;
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
x = 20
x = 1
x = 30
```

8.4 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.

```
main()
{
externint y; /* external declaration */
--------


}
fun1()
{
externint 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.

8.5 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.

```
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.

```
/* Illustration of static variable */
main()
{
inti;
for(i=1; i<=3; i++) stat();
}
stat()
{
staticint 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.

8.6 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:

```
registerint 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.

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 ( )
{
inti;
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 ( )
{
inti;
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>
void f1(int *k) // B
{
*k = *k + 10; // C
} .
main ( )
{
inti;
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

Program to demonstrate different storage classes

```
#include <stdio.h>


// declaring the variable which is to be made extern

// an initial value can also be initialized to x

int x;


voidautoStorageClass()

{


        printf("\nDemonstrating auto class\n\n");


        // declaring an auto variable (simply

        // writing "int a=32;" works as well)

        autoint a = 32;


        // printing the auto variable 'a'

        printf("Value of the variable 'a'"

                " declared as auto: %d\n",

                a);


        printf("--------------------------------");

}


voidregisterStorageClass()

{


        printf("\nDemonstrating register class\n\n");


        // declaring a register variable

        register char b = 'G';
        // printing the register variable 'b'
        printf("Value of the variable 'b'"
                " declared as register: %d\n",
                b);

        printf("--------------------------------");
}

Void extern Storage Class()
{

        printf("\n Demonstrating extern class\n\n");

        // telling the compiler that the variable
        // z is an extern variable and has been
        // defined elsewhere (above the main
        // function)
        externint x;

        // printing the extern variables 'x'
        printf("Value of the variable 'x'"
                " declared as extern: %d\n",
                x);

        // value of extern variable x modified
        x = 2;

        // printing the modified values of
        // extern variables 'x'
        printf("Modified value of the variable 'x'"
                " declared as extern: %d\n",
                x);

        printf("--------------------------------");
}

Void static Storage Class()
{
        inti = 0;
        printf("\n Demonstrating static class\n\n");

        // using a static variable 'y'
        printf("Declaring 'y' as static inside the loop.\n"
                "But this declaration will occur only"
                " once as 'y' is static.\n"
                "If not, then every time the value of 'y' "
                "will be the declared value 5"
                " as in the case of variable 'p'\n");

        printf("\n Loop started:\n");

        for (i = 1; i< 5; i++) {

                // Declaring the static variable 'y'
                staticint y = 5;

                // Declare a non-static variable 'p'
                int p = 10;

                // Incrementing the value of y and p by 1
                y++;
                p++;

                // printing value of y at each iteration
                printf("\nThe value of 'y', "
                        "declared as static, in %d "
                        "iteration is %d\n",
                        i, y);

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

        printf("\n Loop ended:\n");

        printf("--------------------------------");
}
```

```
int main()
{

          printf("A program to demonstrate"
                    " Storage Classes in C\n\n");

          // To demonstrate auto Storage Class
          Auto Storage Class();

          // To demonstrate register Storage Class
          Register Storage Class();

          // To demonstrate extern Storage Class
          Extern Storage Class();

          // To demonstrate static Storage Class
          Static Storage Class();

          // exiting
          printf("\n\n Storage Classes demonstrated");

          return 0;
}
```

8.7 Const Qualifier

To declare a variable constant, we use the const qualifier. That is, after the variable has been initialized, we cannot modify its value. Const offers a lot of advantages. If you have a constant value for PI, for example, you don't want any element of the programme to change that value. As a result, you should declare it as a const.

The compiler may store objects defined with const-qualified types in read-only memory, and if the address of a const object is never used in a programme, it may not be stored at all.

Summary

- Auto, extern, register, static are the four different storage classes in a C program.
- In this unit, we learnt about “storage classes”. A keyword auto is used to define an auto storage class.
- Extern storage class is used when we have global functions or variables which are shared between two or more files
- The static variables are used within function/ file as local static variables. They can also be used as a global variable
- Register is used to store the variable in CPU registers rather memory location for quick access.
- Storage class represents the scope and lifespan of a variable.

- It also tells who can access a variable and from where?

Keywords

Auto

Extern

Register

Static

Self Assessment

1. What is the initial value of register storage class specifier?

A. 0
B. Null
C. Garbage
D. Infinite

2. What is the scope of extern class specifier?

A. Within block
B. Within Program
C. Global Multiple files
D. None of the above

3. What is the scope of static class specifier?

A. Within block
B. Within Program
C. Global Multiple files
D. None of the above

4. Which is not a storage class?

A. Auto
B. Struct
C. Typedef
D. Static

5. What is the output of the following program?

```
#include<stdio.h>

int main()
{
staticint a = 3;
printf(“%d”, a --);
return 0;
}
```

A. 0
B. 1

C. 2

D. 3

6. What will be the output of the following program?

```
#include <stdio.h>
staticint y = 1;
int main()
{
staticint z;
printf(“%d %d”, y, z);
return 0;
}
```

A. Garbage value

B. 0 0

C. 1 0

D. 1 1

7. In case of a conflict between the names of a local and global variable what happens?

A. The global variable is given a priority.

B. The local variable is given a priority.

C. Which one will get a priority depends upon which one is defined first.

D. The compiler reports an error.

8. Where will the space be allocated for an automatic storage class variable?

A. In CPU register

B. In memory as well as in CPU register

C. In memory

D. On disk.

9. What is the output of the program?

```
staticint k;
int main()
{
    printf("%d", k);

    return 50;
}
```

A. -1

B. 50

C. 0

D. Compiler error

10. The statement below is a __________?

externint a;

A. Declaration
B. Definition
C. Initialization
D. None of the above

11. An external variable is one-

A. Which is globally accessible by all functions
B. Which is declared outside the body of any function
C. Which resides in the memory till the end of a program
D. All of the above

12. Functions in C are always _________

A. Internal
B. External
C. External and Internal are not valid terms for functions
D. Both Internal and External

13. Global variables are __________

A. External
B. Internal
C. Both internal and external
D. None of above

14. Which storage class is used for faster execution?

A. Register
B. Auto
C. Extern
D. Static

15. Which of the following is default storage class?

A. Register
B. Auto
C. Extern
D. Static

Answers for Self Assessment

1.	C	2.	C	3.	A	4.	B	5.	D
6.	C	7.	B	8.	C	9.	B	10.	A
11.	D	12.	B	13.	A	14.	A	15.	B

Review Questions

1. Write a program to demonstrate static storage classes
2. What is significance of storageclasses.
3. Write a program to demonstrate auto storage classes
4. Write a program to demonstrate extern storage classes
5. Write a program to demonstrate register storage classes

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.

Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 09: Arrays

CONTENTS

Objectives

Introduction

9.1 Arrays

9.2 Advantages of Arrays

9.3 Types of Arrays

9.4 One-dimensional Array

9.5 Two-dimensional and Multi-dimensional Array

9.6 Array Declaration

9.7 Array Initialization

9.8 Accessing Elements of an Array

9.9 Passing array as an argument to function

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

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 allotted 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.

9.1 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

9.2 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.

9.3 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

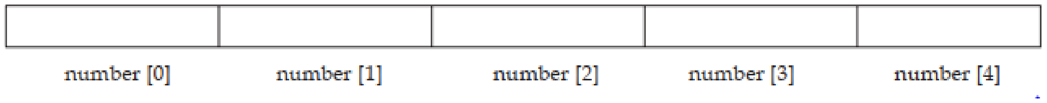
9.4 One-dimensional Array

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:



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.

9.5 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

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.

- 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]);
}
```

- To insert an element into an existing sorted array (Insertion Sort).

```
# 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]);
}
```

- 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);
```

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

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( ).

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", &rarr[i] ) ;
                         printf ( " \ n%d", arr[i] ) ;
}
}
```

9.6 Array Declaration

Arrays are defined in the same manner as ordinary variables, except that each array name must be accompanied by the size specification.

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.

: 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]

9.7 Array Initialization

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.

3. If the array is initialized where it is declared, mentioning the dimension of the array is optional.

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 };

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}, } };

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.

9.8 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.

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
}
}
```

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.

9.9 Passing array as an argument to function

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Method -1

Formal parameters as a pointer −

```
void myFunction(int *param) {
   .
   .
   .
}
```

Method -2

Formal parameters as a sized array −

```
void myFunction(int param[10]) {
   .
```

```
  .
  .
}
```

Method -3

Formal parameters as an unsized array −

```
void myFunction(int param[]) {
  .
  .
  .
}
```

: Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows −

```
double getAverage(int arr[], int size)
 {
  int i;
  double avg;
  double sum = 0;

  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }
  avg = sum / size;
  return avg;
}
```

Now, let us call the above function as follows −

```
#include <stdio.h>
 double getAverage(int arr[], int size);
int main () {
  int balance[5] = {1000, 2, 3, 17, 50};
  double avg;
  avg = getAverage( balance, 5 ) ;
  printf( "Average value is: %f ", avg );
     return 0;
}
```

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.

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

Self Assessment

1. What is an Array in C language?

A. A group of elements of same data type.
B. An array contains more than one element.
C. Array elements are stored in memory in continuous or contiguous locations.
D. All the above.

2. An array Index starts with?

A. 1
B. 0
C. -1
D. 2

3. Arrays can

A. store data elements of same data type at contiguous memory location.

B. be used for CPU scheduling.

C. be used for reverse data elements, sort data elements etc.

D. All of above

4. Two dimensional arrays in C

A. An array of arrays is known as two dimensional array.

B. An array of loops

C. An array of tokens

D. All of above

5. Choose the correct syntax for two dimensional array

A. data_type name_of_array;

B. data_type name_of_array[rows][columns];

C. data_type [rows][columns];

D. name_of_array[rows][columns];

6. What will be the output of the following C code?

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

A. 1 2 3 4 5 0

B. 1 2 3 4 5 junk

C. 1 2 3 4 5 5

D. Run time error

7. One-dimensional array of two-dimensional arrays is called

A. One-dimensional array

B. Multi-dimensional array

C. Two-dimensional array

D. Three-dimensional array

8. How many kinds of elements an array can have?

A. Char and int type

B. Only char type

C. Only int type

D. All of them have same type

9. Choose the correct statement

A. Array stores data of the same type

B. Array can be a part of a structure

C. Array of structure is allowed

D. All of the above

10. Choose correct declaration of function with as array parameter, let name of function is sum and array (arr) is of integer type

A. int sum(int)

B. int sum(int arr[])

C. int sum(arr[])

D. int sum(arr)

11. Choose the right statement

A. Arrays and function can't be used in C language.

B. Passing functions to array is possible in C Language

C. Passing array to function is possible in C Language

D. All of above

12. A character constant is enclosed by?

A. Left Single Quotes

B. Right Single Quotes

C. Double Quotes

D. None of the above

13. A character array can be initialized using

A. Floats value

B. Integer values

C. A string literal

D. None of them

14. 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

15. The general form of array declaration is

A. array_name [size];

B. data_type array_name [size];

C. data_type [size];

D. None

Answers for Self Assessment

1.	D	2.	B	3.	D	4.	A	5.	B
6.	A	7.	C	8.	D	9.	D	10.	B
11.	C	12.	B	13.	C	14.	A	15.	B

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.
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.

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

Web Links

www.en.wikipedia.org
www.web-source.net
www.webopedia.com

Unit 10: Pointers

CONTENTS

Objectives
Introduction
10.1 Pointers
10.2 Accessing the Address of a Variable
10.3 Pointer Declaration
10.4 Address Operator - &
10.5 Indirection Operation - *
10.6 Pointer Variables
10.7 Initialization of Pointer Variables
10.8 Accessing a Variable through its Pointer
10.9 Pointer Expression
10.10 Pointer arithmetic
10.11 Pointer and Arrays
10.12 Array of Pointers
10.13 Pointers and functions
10.14 NULL Pointer
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss the concepts of pointers
- Identify pointer increment and scale factors
- Pointer expressions
- Pointers and arrays
- NULL Pointer

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 specialcharacteristic to hold the address of some other memory location as its r-value. C programminglanguage provides ‘&’ operator to extract the address of any object. These addresses can be storedin 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>.

```
: # 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*/

}
```

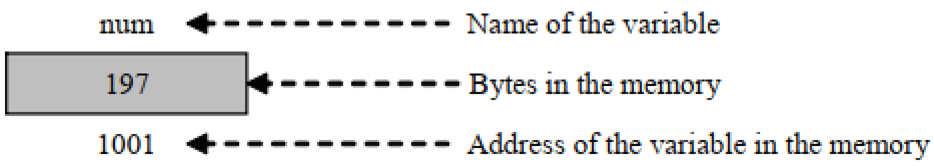
A null value can be assigned to a pointer when it does not point to any data or in the other words, as a good programming habit every pointer should be initialized with the null value. A pointer with a null value assigned to it is nothing but a pointer which contains the address zero.

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.

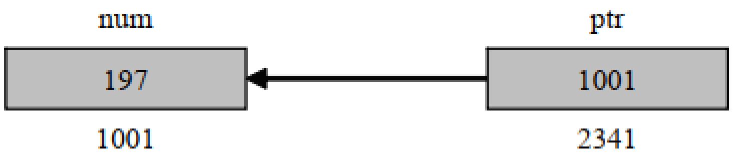
Pointer is a variable which can hold the address of a memory location. The value stored in a pointer type variable is interpreted as an address. Consider the following declarative statement:

intnum = 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:



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).

Intx[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.

: 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 *roll number;	Create a pointer variable roll number 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

10.4 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 = &num; 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.

10.5 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.

10.6 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( )
{
inti = 3;
printf ("\n Address of i: = %u", &i); /* returns the address * /
printf ("\t value i = %d", * (&i)); /* returns the value of address of i */
}
```

10.7 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.

10.8 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 assignedthe value at address contained in i to another variable n. Thus, indirectly we have accessed thevariable q through n. using pointer variable i.

10.9 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.

10.10 Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer

Following arithmetic operations are possible on the pointer in C language:

Increment

Decrement

Addition

Subtraction

Comparison

Increment:

It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

Example:

If an integer pointer that stores address 1000 is incremented, then it will increment by 2(size of an int) and the new address it will points to 1002. While if a float type pointer is incremented then it will increment by 4(size of a float) and the new address will be 1004.

Decrement:

It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores address 1000 is decremented, then it will decrement by 2(size of an int) and the new address it will points to 998. While if a float type pointer is decremented then it will decrement by 4(size of a float) and the new address will be 996.

Example:

Program to illustrate pointer increment/decrement

```
#include <stdio.h>

// Driver Code
int main()
{
        // Integer variable
        int N = 4;

        // Pointer to an integer
        int *ptr1, *ptr2;

        // Pointer stores
        // the address of N
        ptr1 = &N;
        ptr2 = &N;

        printf("Pointer ptr1 "
                "before Increment: ");
        printf("%p \n", ptr1);

        // Incrementing pointer ptr1;
        ptr1++;

        printf("Pointer ptr1 after"
                " Increment: ");
        printf("%p \n\n", ptr1);

        printf("Pointer ptr1 before"
                " Decrement: ");
        printf("%p \n", ptr1);

        // Decrementing pointer ptr1;
        ptr1--;

        printf("Pointer ptr1 after"
                " Decrement: ");
        printf("%p \n\n", ptr1);
        return 0;
}
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]

10.11 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:

staticint 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 arrayelements, 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.

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.

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.

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.

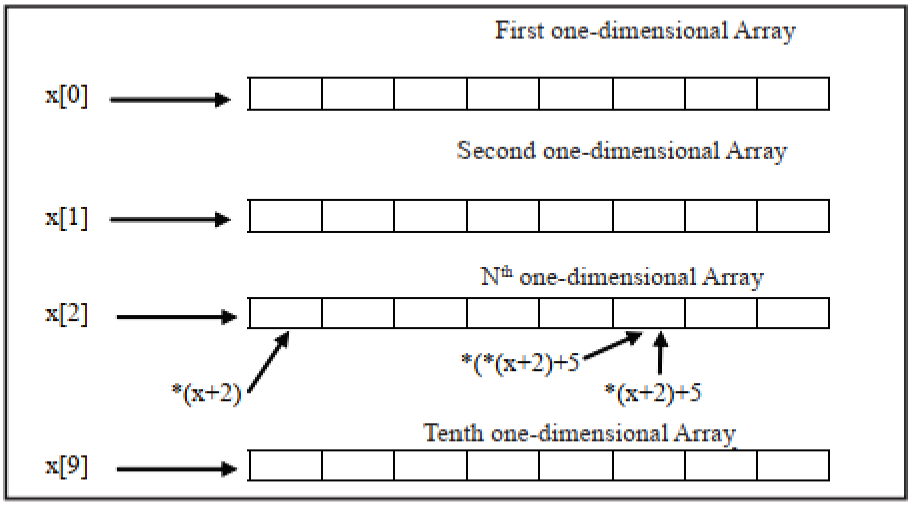
When a n dimensional array is expressed in this manner, an individual array element within the n dimensional array can be accessed by a single use of the indirection operator. The following example illustrates how this is done.

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:


First one-dimensional Array
x[0]
Second one-dimensional Array
x[1]
Nth one-dimensional Array
x[2]
*(x+2)
((x+2)+5
*(x+2)+5
Tenth one-dimensional Array
x[9]

10.13 Pointers and functions

We can use function pointers to avoid code redundancy. For example a simple qsort() function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type. Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function.

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

voidgetSeconds(unsigned long *par);

int main () {

unsigned long sec;
getSeconds(&sec );

   /* print the actual value */
printf("Number of seconds: %ld\n", sec );

return 0;
}

voidgetSeconds(unsigned long *par) {
   /* get the current number of seconds */
   *par = time( NULL );
return;
}
```

Lab Exercise

```
// Program to demonstrate working of Pointers


#include<stdio.h>
int main(){

int n=123;
int *ptr=&n;
int **nn=n;

//*ptr=&n;
printf("Original value of variable n is = %d\n",n);
printf("Address of n is = %p\n",ptr);
printf("Address of n in decimal number is = %d\n",ptr);
printf("Value of  %d",nn);
return 0;
}
```

Lab Exercise

```
// Program to access value using Pointers


#include<stdio.h>
int main(){
int x;
printf("Enter Value of x\n");
scanf("%d",&x);
int *p;
p=&x;
printf("value of x entered by user is %d\n",x);
printf("Address of  variable x is %p\n",p);
printf("Getting value from pointer variable  %d\n",*p);
printf("Address of variable x is %d\n",p);
printf("Address of variable x is %u\n",p);
*p=500;
printf("New value of x is %d\n",x);
return 0;

}
```

Lab Exercise

```
// Program for Pointer Arithmatics


#include<stdio.h>
int main(){
inta,b,sum,sub,mul,div;
int *ptr1,*ptr2;
printf("Enter first number");
scanf("%d",&a);
printf("Enter second number");
scanf("%d",&b);
ptr1=&a;
ptr2=&b;
sum=*ptr1+*ptr2;
sub=*ptr1-*ptr2;
mul=*ptr1* *ptr2;
div= *ptr1/ *ptr2;
printf("Using pointers, all arithmetic operations
performed\n");
printf("Sum of first and second number is =%d\n",sum);
printf("Subtraction of first and second number is %d\n",sub);
printf("Product of first and second number is %d\n",mul);
printf("Division of first and second number is %d\n",div);
printf("Address of first and second number a=%p, b=%p
",ptr1,ptr2);
return 0;
}
```

10.14 NULL Pointer

A Null Pointer is a pointer that does not point to any memory location. It stores the base address of the segment. The null pointer basically stores the Null value while void is the type of the pointer.

If we do not have any address which is to be assigned to the pointer, then it is known as a null pointer. When a NULL value is assigned to the pointer, then it is considered as a **Null pointer**.

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

        int *ptr;
        ptr=NULL;
        printf("Value of null pointer is %d",ptr);
        return 0;
        }
```

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.

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

Self Assessment

1. What are the correct statements about pointers?

A. pointer is a variable that stores the address of another variable
B. pointer can also be used to refer to another pointer function
C. Pointers assign and releases the memory as well
D. all of above.

2. What are the applications of pointers?

A. Implement data structure
B. Dynamic memory allocation
C. Accessing array and functions
D. Above all

3. Which symbol is used during pointer declaration?

A. +
B. -
C. *
D. /

4. What format specifier is used for pointers?

A. %c
B. %d
C. %p
D. %s

5. Which statements are true about NULL pointers

A. NULL pointer pointing to nothing
B. The value of null pointer is 0
C. Both a and b
D. None of above

6. What is the output of following program?

```
#include<stdio.h>
int main(){
int *ptr=NULL;
printf(“Value of null pointer is= %d”,ptr);
return 0;
}
```

A. 1
B. 2
C. 0
D. 4

7. Which one is incorrect statement?

A. int x=90;
B. int *ptr1,*ptr2;
C. ptr1=@x;
D. ptr2=ptr1;

8. What type of arithmetic operations can performed on pointers?

A. Addition
B. Subtraction
C. Multiply
D. Above all

9. What is the output of following program?

```
#include<stdio.h>
int main(){
int x=90,y=10,result;
int *ptr1,*ptr2;
    ptr1=&x;
    ptr2=&y;
result=*ptr1**ptr2;
printf("Product of x and y  using pointers  is %d\n",result);
return 0;
}
```

A. 100
B. 80
C. 900
D. 10

10. Which one is incorrect statement?

A. int a=10,*ptr;
B. ptr=/a;
C. ptr--;
D. above all

11. What are the different operations that can be performed on pointers?

A. Decrement
B. Addition
C. Subtraction

D. Above all

12. Before using a pointer variable

A. It should be declared
B. It should be initialized
C. It should be both declared and initialized
D. None of above

13. Which is correct statement for expression?

Int *ptr, p;

A. ptr is a pointer to integer, p is not an integer variable.
B. ptr and p, both are pointers to integer.
C. ptr is pointer to integer, p may or may not be.
D. ptr and p both are not pointers to integer.

14. Which is correct statement?

A. int ->ptr[3];
B. int *ptr[2];
C. intptr[*5];
D. None of above

15. Which is correct statement for array of pointers?

A. We can store pointers to an int or char or any other data type available.
B. We can also declare an array of pointers
C. Use a pointer to point to an array, and then we can use that pointer to access the array elements
D. Above all

Answers for Self Assessment

1.	D	2.	D	3.	C	4.	C	5.	C
6.	C	7.	C	8.	D	9.	C	10.	C
11.	D	12.	C	13.	A	14.	B	15.	D

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.

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. Write a function to calculate the factorial value of any integer entered through the keyboard.

8. Write a function power (a, b), to calculate the value of a raised to b.

9.Explain pointers and functions with suitable example.

10.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.

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.

YashvantKanetkar, Let us C

Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 11: Strings

CONTENTS

Objectives
Introduction
11.1 Strings
11.2 Declaring and Initializing String
11.3 Reading and Writing Strings
11.4 Build-in-Library Functions to Manipulate Strings
11.5 strlen( )
11.6 strcpy( )
11.7 strcat( )
11.8 strcmp( )
11.9 Putting String Together
11.10 Comparison of two String
11.11 String Handling Functions
11.12 Pointers and Strings
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After studying this unit, you will be able to:

- Explain strings
- Describe reading and writing strings
- Explain string handling functions

Introduction

Computers process a variety of data kinds in addition to numeric data. The data to be processed is frequently textual, such as words, names, and addresses. String type variables are used to store and process this type of data. There is no explicit string data type in C.

Character arrays, on the other hand, can be used to simulate the same thing. In this session, we'll look at strings and how to manipulate them in C.

11.1 Strings

In C, a string is defined as a collection of characters. The NULL character, which signifies the end of the string, is used to end each string. Any group of characters enclosed in double-quote marks is referred to as a string constant.

When characters in a string constant are stored, the NULL character is automatically appended to the end of them. The escape sequence '0' is used to represent the NULL character within a programme. A string constant is an array with a lower bound of 0 and an upper bound of the string's length in characters.

When choosing a data representation for a given data object, the cost of performing various operations with that representation must be considered. Furthermore, a hidden cost resulting from the required storage management procedures must be considered.

Strings are stored in three types of structures:

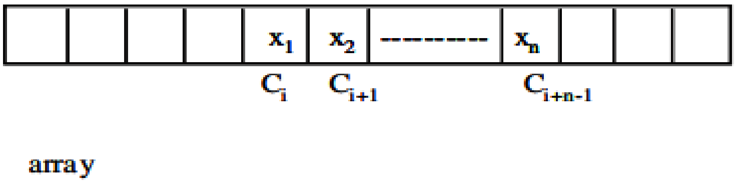
1. Fixed length structure

2. Variable length structure

3. Linked structure

9.2 Sequential Fixed Length Structure

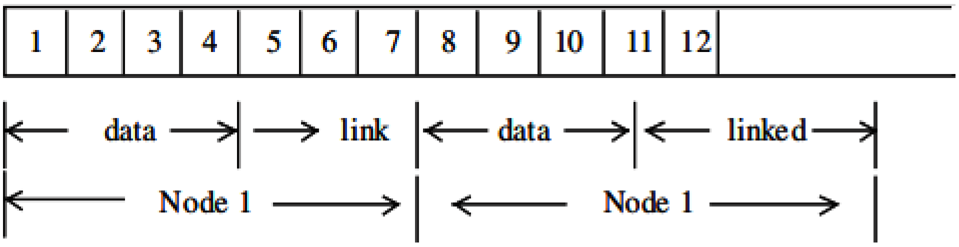
In this representation successive characters of a string will be placed in consecutive character positions. The string S = ‘x1---xn’ could then be represented as in Figure with s as a pointer to the first character.



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 andLink. The size of a node is number of characters that can be stored in the DATA fields.



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.

Each node in the purest version of a linked list representation of strings would be one in size. Normally, this would be considered a huge waste of space. With a two-character link field, this means that only 1/3 of the available Memory will be used to store string data, while the remaining 2/3 will be used exclusively for link data.

11.2 Declaring and Initializing String

In C, strings are represented as character arrays. The null character, which is just the character with the value 0, is used to indicate the end of the string. (The null character is unrelated to the null pointer except by name.) The null character is known as NUL in the ASCII character set.) Another character escape sequence, 0 represents the null or string-terminating character.

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.

: We can declare and define an array of characters, and initialize it with a stringconstant:

```
char string[] = “Hello, world!”;
```

In this situation, we may omit the array's dimension because the compiler will figure it out for us based on the size of the initializer. The compiler will only size a string array for us in this scenario; in all other circumstances, we will have to decide how big the arrays and other data structures we employ to house strings are

To do anything else with strings, we must typically call functions. The C library contains a fewbasic string manipulation functions, and to learn more about strings, we’ll be looking at howthese functions might be implemented.

Since C never lets us assign entire arrays, we use the strcpy function to copy one string toanother:

```
#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 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”);

elseprintf(“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”;
intlen = strlen(string7);
printf(“%d\n”, len);
```

Finally, you can print strings out with printf using the %s format specifier, as we’ve been doingin 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[])
{
inti = 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[])
{
inti = 0;
while(1)
{
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).

Finally, here is a version of strlen:

```
intmystrlen(char str[])
{
inti;
for(i = 0; str[i] != ‘\0’; i++)
{}
returni;
}
```

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++)
```

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 likethis:

```
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, andcopy the substring to string9. Just as with strcpy, it's our responsibility to declare the destinationstring (string9) big enough. Here is an implementation of substr. Not surprisingly, it's quite similar to strcpy:

```
substr(char dest[], char src[], int offset, intlen)
{
inti;
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 thatcharacters 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 comesfirst).

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 returnvalues, 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 betweencharacters 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 smallinteger, 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 tolearn these values, because the vast majority of the time, you use character constants to referto characters, and the compiler worries about the values for you. Using character constants inpreference 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, evena string which contains only one character (other than the \0).

‘A’ is not the same as “A”. To drive home this point, let's illustrate it with a fewexamples.

If you have a string:

```
char string[] = “hello, world!”;
```

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

```
inti = ‘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”;
inti = atoi(string);
int j = atoi(“456”);
```

11.3 Reading and Writing Strings

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.

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 thenprintf( ). It is faster than printf( ). It does not output numbers

or does format conversions as puts( ) outputs character string only.

```
# include <stdio.h>
# include <conio.h>
main( )
{
charstr [50]
gets (str);
puts (str);
}
```

11.4 Build-in-Library Functions to Manipulate Strings

With every C compiler a large set of useful string handling library functions are provided.

Table lists the more commonly used functions along with their purpose.

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.

11.5 strlen( )

This function counts the number of characters present in a string. Its usage is illustrated in the following program.

```
main( )
{
chararr[ ] = "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
```

11.6 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...

```
main( )
{
char source[ ] = “Sayonara” ;
char target[20] ;
strcpy ( target, source ) ;
printf ( “\nsource string = %s”, source ) ;
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( ).

```
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
```

11.7 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.

```
 #include<stdio.h>
 #include<string.h>
 int main(){
 char name[10];
 charlname[10];
 printf("Enter Name");
 gets(name);
 printf("Enter last name");
 gets(lname);
 strcat(name,lname);
 printf("Full name is %s",name);
 return 0;

 }
```

11.8 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.

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

 char str1[10]="aBC";
char str2[10]="ABC";
printf("Comparing Str1 and Str2  = %d",strcmp(str1,str2));
return 0;
 }
```

11.9 Putting String Together

```
        #include <string.h>
        #include <stdio.h>
        int main()
        {
        char first[100];
        char last[100];
        charfull_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);
}
```

11.10 Comparison of two String

```
#include <string.h>
i = strcmp( s1, s2 );
Where:
const char *s1, *s2;
are the strings to be compared.
inti;
```

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.

11.11 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
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. intstrcmp(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. intstrlen(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. intstrncmp(const char *s1, const char *s2, intmaxlen): 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

11.12 Pointers and Strings

A String is a sequence of characters stored in an array. A string always ends with null ('\0') character.

A pointer to array of characters or a string can hold value and address both.

Syntax

char *names[];

```
#include <stdio.h>
int main()
{
char *names[] = {"ABC", "DEF"};
inti;
for(i = 0; i< 2; i++)
printf("%s \n", names[i]);
return 0;
}
```

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.

Keywords

gets( ): A C library function used to read a character entered at the keyboard and to place it at theaddress pointed to by its character pointer argument

puts( ): A C library function that writes its string argument to the screen followed by the newline

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

Self Assessment

1. Which one is correct method for Initializing string?

A. char abc[ ] ="hello";
B. char abc[10]= {'h','e','l','l','o','\0'};
C. Char abc[ ] = {'h','e','l','l','o','\0'};
D. above all

2. Which method is use to read string ?

A. Puts ( )
B. Gets ( )
C. Print ( )
D. None of above

3. Which method is use to write string?

A. Scanf ( )
B. Gets ( )
C. Puts ( )
D. Printf ( )

4. String is an______________

A. an array of characters

B. sequence of characters terminated with a null character
C. both a and b
D. none of above

5. Which of the following function is more appropriate for reading in a multi-word string?

A. gets ( )
B. Puts ( )
C. Printf ( )
D. Sizeo ( )

6. Format specifier is used to print a string is ___________

A. % d
B. % c
C. % s
D. % f

7. Is there any difference between the two statements?

char *ch = "string ";

charch[] = "String";

A. Yes
B. No

8. Function of strcat ( ) is_______________

A. compares two strings
B. converts string to lowercase
C. concatenates (joins) two strings
D. none of above

9. If the two strings are identical, then strcmp( ) function returns

A. 1
B. 0
C. 2
D. -1

10. The library function used to computes string's length

A. strlwr ( )
B. strcpy ( )
C. strlen ( )
D. strpr ( )

11. What will strupr( ) function do?

A. compares two strings
B. converts string to uppercase

C. computes string's length

D. none of above

12. Function of strcpy ( ) is______________

A. converts string to uppercase

B. sequence of characters terminated with a null character

C. copies a string to another

D. none of above

13. What is the maximum length of a C String?

A. 16 characters

B. 8 characters

C. 32 characters

D. None of above

14. What will strlwr( ) function do?

A. converts string to lowercase

B. converts string to uppercase

C. computes string's length

D. none of above

15. Find incorrect statement _______

A. char input[100];

B. puts('Input string');

C. gets(input);

D. puts("Entered string is");

Answers for Self Assessment

1.	D	2.	B	3.	C	4.	C	5.	A
6.	C	7.	A	8.	C	9.	B	10.	C
11.	B	12.	C	13.	D	14.	A	15.	B

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.

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 takes a set of names of individuals and abbreviates the first, middle and other names except the last name by their first letter.

Further Readings

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.

YashvantKanetkar, Let us C

Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 12: Dynamic Memory Management

CONTENTS

Objectives
Introduction
12.1 Dynamic Memory Allocation
12.2 malloc() - memory allocation, size of, and free
12.3 calloc and realloc
12.4 free
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After studying this unit, you will be able to:

- Dynamic Memory Management functions
- malloc() and calloc()
- realloc() and free()

Introduction

The Dynamic Memory Allocation concept is utilized to solve the array problem. An array is a set of values with a fixed number of elements. You can't adjust the size of an array once it's been declared. Programmers can use Dynamic Memory Allocation to allocate memory during runtime.

Because the static representation of a linear ordered list using an array wastes resources and, in some situations, causes overflows. We no longer want to pre-allocate memory to any linear list; instead, we want to allocate memory to elements as they are added to the list. This necessitates memory allocation that is dynamic.

12.1 Dynamic Memory Allocation

There are several limitations in static memory allocation:

This is done in RAM dedicated solely to a programme, which is frequently limited in capacity. The size of a static array is fixed. We won't be able to expand it to accommodate situations that require more elements. As a result, we'll likely to declare larger arrays than necessary, resulting in memory waste. We also can't lower array size to conserve memory when fewer array elements are necessary. Advanced data structures such as linked lists, trees, and graphs, which are needed in most real-world programming situations, are not possible (or efficient) to develop.

C has a feature called dynamic allocation that is quite unique (amongst high level languages). It allows us to design data types and structures of any size and duration to meet the needs of our programmes. Dynamic memory allocation occurs when memory is allocated at runtime, that is, when a programme is running. Pointers and four common library functions are used in dynamic memory management.

Programming language (C) provides several functions for memory allocation and management, namely, malloc, calloc, realloc and free. Functions are defined in the <stdlib.h> header file.

Static Vs Dynamic memory allocation

Static Memory Allocation	**Dynamic Memory Allocation**
variables get allocated permanently	variables get allocated only if your program unit gets active
Allocation is done before program execution	Allocation is done during program execution
It uses the data structure called stack for implementing static allocation	It uses the data structure called heap for implementing dynamic allocation
Less efficient	More efficient
There is no memory reusability	There is memory reusability and memory can be freed when not required
Memory is allocated at compile time.	memory is allocated at run time

12.2 malloc() - memory allocation, size of, and free

It's used to allocate a single block of memory with the specified size dynamically. It returns a void pointer that can be cast into any type of pointer. It does not perform memory initialization during execution. Initially, it has garbage value. If memory is insufficient, it returns NULL.

The Function malloc is most commonly used to attempt to "grab" a continuous portion of

memory. It is defined by:

```
void *malloc(size_tnumber_of_bytes)
```

Syntax

```
ptr = (cast-type*) malloc(byte-size)
```

Example

```
ptr = (int*) malloc(20 * sizeof(int));
```

That is to say it returns a pointer of type void * that is the start in memory of the reserved portionof 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 ofthese as an argument to the function.

So:

```
inti;
        struct COORD {float x,y,z};
        typedefstruct 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++);
```

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.

Example:

```
#include <stdlib.h>
int main(){
int *ptr;
ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL) {
   *(ptr + 7) =80;

printf("Value of the 8th integer is %d\n",*(ptr + 7));
printf("Address of 8th integer is %d\n",(ptr+7));
        }
return 0;
}
```

Example: #include <stdio.h>

```
              #include <stdlib.h>

              int main()
              {

              int* ptr;
              int n, i;
              // Get the number of elements for the array
              n = 5;
            printf("Enter number of elements: %d\n", n);

              // Dynamically allocate memory using malloc()
            ptr = (int*)malloc(n * sizeof(int));

              // Check if the memory has been successfully
              // allocated by malloc or not
            if (ptr == NULL) {
            printf("Memory not allocated.\n");
            exit(0);
              }
else {

    // Memory has been successfully allocated
printf("Memory successfully allocated using malloc.\n");

    // Get the elements of the array
for (i = 0; i< n; ++i) {
ptr[i] = i + 1;
    }

    // Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i< n; ++i) {
printf("%d, ", ptr[i]);
    }
  }

return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

12.3 calloc and realloc

calloc

It is used to dynamically allocate the specified number of blocks of memory of the specified type. The malloc() function allocates memory and leaves the memory uninitialized. but, the calloc() function allocates memory and initializes all bits to zero.It returns NULL if memory is not sufficient.

realloc

If already allocated dynamically memory is insufficient or more than required, in program we can change the size of previously allocated memory using the realloc() function.

It is used to dynamically change the memory allocation of a previously allocated memoryre-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

There are two allocation functions, calloc() and realloc(). Their prototypes

are given below:

```
void *calloc(size_tnum_elements, size_telement_size};
```

```
void *realloc( void *ptr, size_tnew_size);
```

malloc does not initialise memory (to zero) in any way. If you wish to initialise memory then usecalloc. 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 individualarguments.

Thus to assign 100 integer elements that are all initially zero you would do:

```
int *ip;

ip = (int *) calloc(100, size of(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 remainunchanged 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 ofmemory and will copy the old block contents. Note a new pointer (of different value) willconsequently 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 50integers 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 isnot practically possible with arrays. In arrays we allocate the memory first and then start usingit. 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 knownas memory management functions that can be used for allocating and freeing memory duringprogram execution. These functions help us to build complex application programs that use theavailable memory intelligently.

Example: **Calloc**

```
#include <stdio.h>

int main() {

int * ptr;

ptr = calloc(10, sizeof(int));
if (ptr == NULL) {
printf("Error! memory not allocated.");
exit(0);
    }
printf("Dynamically Building  10 Blocks  ");
return 0;
  }
```

Example: realloc

```
#include<stdio.h>
int main()
{
char *ptr;
ptr = NULL;
ptr = realloc(ptr,10);
if(ptr != NULL)
printf("Memory created successfully\n");
return 0;
}
```

12.4 free

The memory allocated using functions malloc() and calloc() is not de-allocated on their ownIt is used to dynamically de-allocate the memory.

Syntax: free(ptr);

Example: free

```
#include <stdio.h>
int main() {
int* ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL){
 *(ptr + 4) = 50;
printf("Value of the 4th  integer is %d\n",*(ptr + 4));
}
free(ptr);
return 0;
}
```

Example

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

```
#include <stdlib.h>

int main()
{
int *ptr, *ptr1;
int n, i;

   // Get the number of elements for the array
   n = 5;
printf("Enter number of elements: %d\n", n);

   // Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));

   // Dynamically allocate memory using calloc()
   ptr1 = (int*)calloc(n, sizeof(int));

   // Check if the memory has been successfully
   // allocated by malloc or not
if (ptr == NULL || ptr1 == NULL) {
printf("Memory not allocated.\n");
exit(0);
   }
else {

      // Memory has been successfully allocated
printf("Memory successfully allocated using malloc.\n");

      // Free the memory
free(ptr);
printf("Malloc Memory successfully freed.\n");

      // Memory has been successfully allocated
printf("\nMemory successfully allocated using calloc.\n");

      // Free the memory
free(ptr1);
printf("Calloc Memory successfully freed.\n");
   }

return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.

Memory successfully allocated using calloc.
Calloc Memory successfully freed.
```

Dynamic Memory Management Functions

Function	Typical call	Description
malloc	malloc (sz)	Allocate a block of size *sz* bytes from memory heap and return a pointer to the allocated block e.g., ptr = (cast.type*) malloc (byte_size);
calloc	calloc *in (sz)*	Allocate a block of size *n* x *sz* bytes from memory heap, initialize it to zero and return a pointer to the allocated block e.g., ptr = (cast_type*) calloc (n, elem_size);
realloc	realloc *(bl,; sz)*	Adjust the size of the memory block *blk* allocated on the heap to *sz,* copy the contents to a new location if necessary and return a pointer to the allocated block e.g., ptr = realloc (ptr, newsize);
free	free *(blk)*	Free block of memory *blk* allocated from memory heap e.g., free (ptr);

Advantages of Dynamic Memory allocation

- When we don't know how much memory will be required for the software ahead of time.
- When we need data structures that don't have a memory restriction.
- When you wish to make better use of your memory space.
- For example, if you allocate memory space for a 1D array like array[20] and only use 10 memory spaces, the remaining 10 memory spaces will be squandered, and this wasted memory will be unavailable to other programme variables.
- Insertions and deletions in dynamically constructed lists may be done quickly and easily by manipulating addresses, whereas insertions and deletions in statically allocated memory result in additional movements and memory waste.
- Dynamic memory allocation is required when using the concepts of structures and linked lists in programming.

Summary

Dynamic memory allocation is to allocate memory at “run time”.

Dynamically allocated memory must be referred to by pointers.

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

Malloc method in C is used to dynamically allocate a single large block of memory with the specified size.

free method in C is used to dynamically de-allocate the memory.

realloc method in C is used to dynamically change the memory allocation of a previously allocated memory.

Keywords

Dynamic memory allocation: DMA perform manual memory allocation.

free: The free() function used to release space.

malloc: malloc() function reserves a block of memory of the specified number of bytes.

calloc: The calloc() function allocates memory and initializes all bits to zero.

Self Assessment

1. Among 4 header files, which should be included to use the memory allocation functions like malloc(), calloc(), realloc() and free()?

A. #include<string.h>
B. #include<stdlib.h>
C. #include<memory.h>
D. All of above

2. Which of the following statement is correct prototype of the malloc() function in c ?

A. int* malloc(int);
B. Char* malloc(char);
C. unsigned int* malloc(unsigned int);
D. void* malloc(size_t);

3. malloc() returns a float pointer if memory is allocated for storing float's and a double pointer if memory is allocated for storing double's. A.

A. True
B. False
C. May Be
D. Can't Say

4. DMA stands for

A. Dynamite Memory Access
B. Dynamic Memory Available
C. Direct Memory Access
D. None of Above

5. Which of the following is memory allocation technique

A. Static Memory Allocation
B. Dynamic Memory Allocation
C. All of above
D. None of above

6. There is no memory reusable in

A. Static Memory Allocation
B. Dynamic Memory Allocation
C. All of above
D. None of above

7. Dynamic memory allocation is

A. More efficient
B. Less efficient
C. Don't know
D. None of above

8. Which function is used to delete the allocated memory space?

A. Dealloc()
B. free()
C. Both A and B
D. None of the above

9. Which of the following is/are true

A. calloc() allocates the memory and also initializes the allocates memory to zero, while memory allocated using malloc() has random data.
B. malloc() and memset() can be used to get the same effect as calloc()
C. Both malloc() and calloc() return 'void *' pointer
D. All of the above

10. Specify the 2 library functions to dynamically allocate memory?

A. malloc() and memalloc()
B. alloc() and memalloc()
C. malloc() and calloc()
D. memalloc() and faralloc()

11. calloc() returns a storage that is initialized to.

A. Zero
B. Null
C. Nothing
D. One

12. Which one is used during memory deallocation in C?

A. remove(p);
B. delete(p);
C. free(p);
D. terminate(p);

13. In which technique memory allocated at run time

A. Static Memory Allocation
B. Dynamic Memory Allocation
C. All of above
D. None of above

14. Syntax of calloc( ) is

A. ptr = (castType*)calloc(n, size);
B. ptr =calloc(n, size);
C. ptr = (castType*) (n, size);
D. ptr = (castType*)calloc();

15. In this program the allocated memory block can store

```
#include<stdio.h>
   #include<stdlib.h>
int main()
   {
int *ptr;
ptr = malloc(10);
return 0;
   }
```

A. int
B. char
C. float
D. all of above

Answers for Self Assessment

1.	B	2.	D	3.	B	4.	C	5.	C
6.	A	7.	A	8.	B	9.	D	10.	C
11.	A	12.	A	13.	A	14.	A	15.	D

Review Questions

1. What are the disadvantages of static memory allocation?
2. Differentiate between static memory allocation and dynamic memory allocation.
3. Write a program to demonstrate the malloc() function.
4. Differentia between free and realloc() function.
5. What are the advantages of dynamic memory allocation?

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.

YashvantKanetkar, Let us C

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

Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

www.geeksforgeeks.org

Unit 13: Structure and Union

CONTENTS

Objectives

Introduction

13.1 Structure Definition

13.2 Structure Initialization

13.3 Comparison of Structure Variables

13.4 Arrays of Structures

13.5 Array within Structures

13.6 Structures within Structures / nested structures

13.7 Passing Structures to Functions

13.8 Structure Pointers

13.9 Union – Definition and Declaration

13.10 Accessing a Union Member

13.11 Union of Structures

13.12 Initialization of a Union Variable

13.13 Uses of Union

13.14 Differences between Union and Structure

Summary

Keywords

Self-Assessment

Answer for Self Assessment

Review Questions

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

Introduction

Arrays can be used to aggregate groups of variables of the same type, as you've seen. The difficulty now is how to combine data that isn't typed in the same way. The explanation is that C is a language that can be easily extended. It can be extended by defining data types that are built from the basic types. That example, you can use a data type called a structure to organize variables of various 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

13.1 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:

```
structtag_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:

```
structaddr

{

char name [30];

char street [20];

char city [15];

char state [15];

intpincode;

};
```

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;
```

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.

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);
}
```

13.2 Structure Initialization

A structure variable can be initialized as any other data type.

```
main( )
{
staticstruct
{
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( )
{
structst_record
{
int weight;
float height;
}
staticstructst_record student1 = {60, 180.75};
staticstructst_record student2 = {53, 170.60};
- - - - - -
- - - - - - -
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
structst_record / * No static word */
{
int weight;
int height;
}
student1 = {60, 180.75};
}
main( )
{
staticstructst_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.

13.3 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;
        staticstruct class student1 = {111, “Rao”, 72.50};
        staticstruct 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
```

13.4 Arrays of Structures

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( )
    {
    inti;
    staticstruct marks student [3] = {{45, 67, 81, 0}, {75,
    53, 69, 0},
    {75, 53, 69, 0}, {57, 36, 71, 0}};
    staticstruct 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,
    “Subject2”, total.sub2. “Subject3”, total.sub3);
    printf (“\n GRAND TOTAL = \t\t %d \n”, total.total.);
    }
```

13.5 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;
};
staticstruct marks student [3] = {45, 76, 81, 0, 75, 53, 69, 0, 57, 36,
71,
0};
staticstruct marks total;
inti, 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;
}
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
```

13.6 Structures within Structures / nested 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];
intbasic_pay;
intdearness_allowance;
inthouse_rent_allowance;
intcity_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;
inthouse_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.

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;
inthouse_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:

```
structpersonal_record
{
structname_part name;
struct date date_of_birth;
- - - - - -
- - - - - -
};
```

```
structpersonal_record person1;
```

The first member of the structure is name which is of the type structname_part. Similarly, other members have their structure types.

13.7 Passing Structures to Functions

The values of a structure can be passed from one function to another using one of three approaches.

The first way is to pass each element of the structure as a function call's actual argument. The actual arguments are then processed as ordinary variables on their own.

The second way entails providing the full structure to the called function as a copy. Changes to the original structure are not reflected in the called function since the function works on a copy of the complete structure (in the calling function). As a result, it is required that the function return the full structure 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.

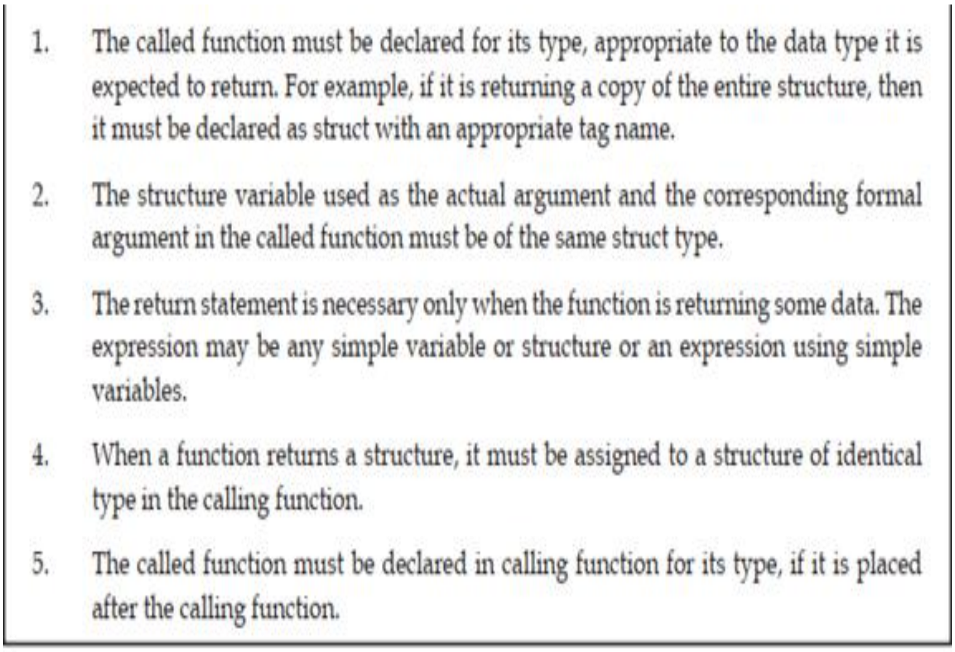
The general format of sending a copy of a structure to the called function is:

```
function_name (structure_variable_name)
```

The called function takes the following form:

```
data_typefunction_name (st_name)

struct_typest_name;

{

- - - - - -

- - - - - -

return (expression);

}
```

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.

Lab Exercise

/* Program showing passing of structure member as function parameters * /

```
struct stores
{
char name [20];
float price;
int quantity;
};
main( )
{
struct stores update( );
floatmul( ), p_increment, value;
intq_increment;
staticstruct 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);
}
struct stores update (struct stores product, float p, int q)
        {
        product.price += p;
        product.quantity += q;
        return (product);
        } fl
        oatmul (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

```
/* Program passing entire structure as function parameter. * /
        structemp
        {
        charempname [25];
        char company [25];
        intempno;
        }
        main( )
        {
        staticstructemp emp1 = {“Prashant”, SOCEM”, 101};
        display (emp1);
        }
        display (e)
        structemp e;
        {
        printf (“%s\n%s\n%d”, emp.empname, emp.company,
        emp.empno);
        }
        Output: Prashant
        SOCEM
        101
```

13.8 Structure Pointers

A complete structure can be transferred to a function by passing a structure-type pointer as anargument. 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 anargument. 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 berecognized outside the function.

Example:

```
#include <stdio.h>
typedefstruct
{
char *name;
intacct_no;
characcttype;
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)
{
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( )
{
structemp
{
charempname [25];
char company [25];
intempno;
};
staticstructemp emp1 = {“Prashant”, “SOCEM’, 101};
structemp *ptr;
ptr = &emp1;
printf (“%s %s %d\n”, emp1.empname,emp1.company,emp1.empno);
printf (“%s %s %d\n”, ptr->company, ptr->empno);
}
```

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.

```
structemp
{
charempname [25];
intempno;
}
main( )
{
staticstructemp emp1 = {Prashant”,”socem”, 101};
display (&emp1);
}
display (e)
structemp *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.

13.9 Union – Definition and Declaration

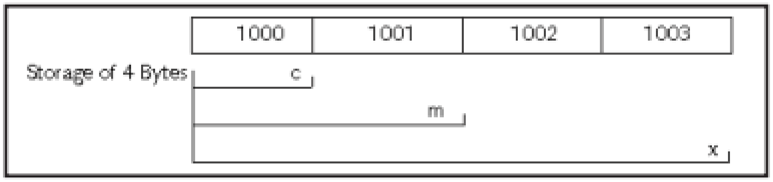
Unions follow the same syntax as structures but differ in terms of storage. In structures, eachmember has its own storage location, whereas all the members of a union use the same location.

This implies that, although a union may contain many members of different types, it can handleonly one member at a time.

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 factthat 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.

13.10 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.

13.11 Union of Structures

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
        {
        inti;
        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
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 originaldeclarations.

An individual union member can be accessed in the same manner as an individual structuremember, 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 aunion, then ptvar->member refers to a member of that union.

Example:

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

13.12 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;
        };
        staticstruct 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
```

13.13 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.

13.14 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.

3. Union is best in the environment where memory is less as it shares the memory allocated. But structure cannot implemented in shared memory.

4. As memory is shared, ambiguity are more in union, but less in structure.

5. Self referential union cannot be implemented in any data structure ,but self referential structure can be implemented.

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.
- 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.

Keywords

Random access file: A file, which allows accessing its records without restriction on the order of access.

Sequential file: 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 defined 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.

Self-Assessment

1. A structure is a __

A. collection of variables (different types) represented by single name
B. A structure is a user defined data type in C
C. Keyword 'struct' is used to define structure in C
D. All of above

2. Structure members are accessed using___

A. :
B. .
C. >
D. <

3. Which keyword is used to define structure in C__

A. structure
B. struct
C. structC
D. none of above

4. Passing structure in function using___

A. function by value
B. function by reference
C. both function by reference and function by value
D. none of above

5. How to access members using Pointer__

A. Using indirection (*) operator and dot (.) operator
B. Using arrow (->) operator or membership operator
C. Both using (.) (*) and (->)
D. none of above

6. Which of the following operation is illegal in structures?

A. Pointer to a variable of the same structure
B. Dynamic allocation of memory for structure
C. Typecasting of structure
D. All of the mentioned

7. Self-Referential Structures are_____

A. Have one or more pointers which point to the same type of structure, as their member.
B. Can dynamically be expanded or contracted
C. Both of them
D. None of above

8. What are the types of Self Referential Structures?

A. Structure with Single Link
B. Structure with Multiple Links
C. Both Structure with Single Link and Structure with Multiple Links
D. None of the above

9. Union is _____

A. user define data type available in C
B. only one member of the union can occupy the memory
C. the size of the union in any instance is equal to the size of its largest element
D. all of above

10. How to access union member?

A. (->)
B. (+)
C. (:)
D. (:)

11. Union differs from structure in the following way

A. All members are used at a time
B. Union cannot have more members
C. Only one member can be used at a time
D. Union initialized all members as structure

12. Which of the following cannot be a structure member?

A. Array
B. Structure
C. Function
D. None of the above

13. Which of the following is a collection of multiple data types in and of itself?

A. String
B. Structures
C. Char
D. None of the above

14. Point out the error in the program?

```
structemp
{
```

```
intecode;
structemp *e;
};
```

A. Error: in structure declaration
B. Linker Error
C. No Error
D. None of above

15. Which of the following is correct about structures?

A. Structure elements can be initialized at the time of declaration.
B. Structure members can not be initialized at the time of declaration
C. Only integer members of structure can be initialized at the time of declaration
D. None of the above

Answer for Self Assessment

1.	D	2.	B	3.	B	4.	C	5.	C
6.	C	7.	C	8.	C	9.	D	10.	A
11.	C	12.	C	13.	B	14.	C	15.	B

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. 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".
7. Explain the usefulness of structures and unions in C.

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.

YashvantKanetkar, Let us C

Web links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 14: Structures

CONTENTS

Objectives

Introduction

14.1 What is a File?

14.2 Defining and Opening a File

14.3 Input/output Operations on Files

14.4 Errors during Input/output

14.5 Types of Files: Text and Binary files

14.6 Reading, writing and append in file

14.7 Header files

Summary

Keywords

Self-Assessment

Answers for Self Assessment

Review Questions

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
- Appending in files
- Preprocessor Directives and Macros

Introduction

Data is only stored in variables and arrays for a limited time. When a programme ends, all of this data is lost. Files are used to store enormous volumes of data indefinitely. Secondary storage devices, particularly disc storage devices, are used by computers to store files. We'll go through how C programmes edit, update, and process data files in this unit.

A file is a set of data or characters that can be text or a programme. In the C programming language, there are two sorts of files: sequential and random access. Random access files are more difficult to create than sequential ones. The data or text will be stored or read back sequentially. In random access file, the data can be accessed and processed randomly

14.1 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

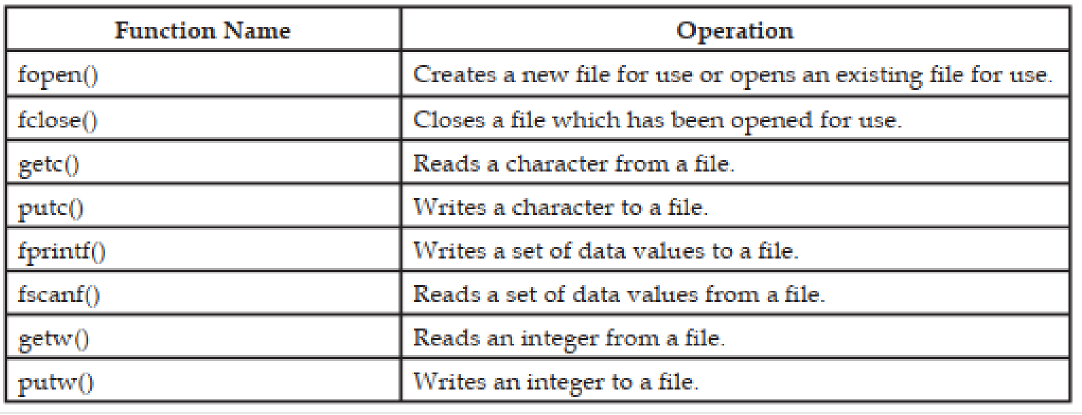
2. Opening a file

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)

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.

14.2 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

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

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.

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:

File opening modes

Modes	**Description**
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode

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.

Example:

```
#include<stdio.h>
int main(){
FILE *fp;
fp=fopen(“abc.txt",“r");
if(fp==NULL)
{
printf("Error, In opening file");
exit(1);
}
fclose(fp);
return 0;
}
```

14.3 Input/output Operations on Files

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.

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);

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);

wherefp 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.

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.

14.4 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.

The statement

```
if (ferror (fp)! = 0)
printf (“An error has occurred. \n”);
```

would print the error message, if the reading is not successful.

14.5 Types of Files: Text and Binary files

When dealing with files, there are two types of files you should know about:

Text files: - Text files are the normal .txt files. You can easily create text files using any simple text editors such as Notepad.

Binary files: -Binary files are mostly the .bin files in your computer.

Text file vs. binary file

Text File	**Binary File**
Bits represent character.	Bits represent a custom data.
Less prone to get corrupt as changes reflect as soon as the file is opened and can easily be undone.	Can easily get corrupted, even a single bit change may corrupt the file.
Can store only plain text in a file.	Can store different types of data (image, audio, text) in a single file.
Widely used file format and can be opened using any simple text editor.	Developed especially for an application and may not be understood by other applications.
Mostly .txt and .rtf are used as extensions to text files.	Can have any application defined extension.

14.6 Reading, writing and append in file

Reading in file

In C Programming reading data block from a file is process of reading from file.

Syntax

```
fp=fopen(“filename", “r");
while(fscanf(fp, "%s", str)!=EOF)
  {
```

```
printf("%s",str);
   }
```

Example:

```
#include<stdio.h>
int main()
{
charstr[50];
   FILE *fp;
fp=fopen(“abc.txt","r");
if(fp==NULL){
printf("-Error");
exit(1);
   }
while(fscanf(fp, "%s", str)!=EOF)
{
printf("%s",str);
}
fclose(fp);
return 0;
}
```

Writing in file

In C programming writing data block in file is possible.

Syntax

```
fp=fopen(“filename", “w");
```

Example:

```
#include<stdio.h>
int main()
{
char name[20]=“this is C";
FILE *fp;
fp=fopen("abc.txt","w");
fprintf(fp, "%s",name);
printf("Contents written");
fclose(fp);
return 0;
}
```

Append in file

In C Programming appends the content of file at the end of another is possible with append mode of file.

Syntax

```
fp=fopen(“filename", "a");
```

Example:

```
#include<stdio.h>
int main()
{
FILE *fp;
fp=fopen(“abc.txt","a");
fprintf(fp, "\n of programming");
printf("contents added in file");
fclose(fp);
return 0;
}
```

14.7 Header files

Header files provide predefined functions to make programming easier. header files contain the set of predefined standard library functions

A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process.

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol.

Preprocessor

Preprocessor	Syntax/Description
Macro	**Syntax:** #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	**Syntax:** #include <file_name> The source code of the file “file_name” is included in the main program at the specified place.
Conditional compilation	**Syntax:** #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.

Other directives	**Syntax:** #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

Header Files Types

Standard library

User-defined

Pre-existing header files: already available in C/C++ compiler user need to import them.

User-defined header files: defined by the user and can be imported using "#include".

Standard Header Files:

#include<stdio.h>: functions scanf() and printf().

#include<math.h>: sqrt(), pow(), etc.

#include<iostream>: cin and cout.

#include<string.h>: strlen(), strcmp(), strcpy(), size()

#include<time.h> : date() and time()

Syntax of Header File

For Standard library: #include<filename>

For user defined: #include"filename"

Example:

For Standard library

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

For user defined

```
#include"myheader.h"
```

Example:

```
#include<stdio.h>
#include"header.h"
int main(){
printf("%d",fact(5));
return 0;
}
```

Macros

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

C Macros

Predefined

User-defined

Example: Predefine

```
#include<stdio.h>
int main(){
printf("File name : %s\n",__FILE__);
printf("File name : %s\n",__DATE__);
printf("File name : %s\n",__TIME__);
return 0;
}
```

Example: User define

```
#include<stdio.h>
#define pi 3.14
int main(){
int r=10,area;
area=pi*r*r;
printf("Area of circle is %d",area);
return 0;
}
```

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.
- 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()
-

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 file: 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.

Macros

User Define

Predefine

Self-Assessment

1. Text files are______

A. Can store only plain text in a file.
B. Bits represent character
C. .txt and .rtf are used as extensions
D. All of above

2. What are the different file operations?

A. Creating a new file
B. Opening an existing file
C. Closing a file
D. Above all

3. Putc( ) is used for_______

A. reads a integer from a file
B. set the position to desire point
C. writes a character to a file
D. none of above

4. fseek( ) is used for_______

A. set the position to desire point
B. reads a integer from a file
C. writes a character to a file
D. none of above

5. r+ mode is used for

A. opens a binary file in reading mode
B. opens a text file in both reading and writing mode
C. opens or create a text file in writing mode.
D. All of above

6. EOF indicates ______

A. function closes the file and returns zero on success.
B. if there is an error in closing the file
C. gives current position in the file
D. none of above

7. Find incorrect statement

A. fp=fopen(“abc*txt","a");
B. fprintf(fp, "\n of programming");
C. printf("contents added in file");

D. fclose(fp);

8. Which one is used for append mode_______

A. r
B. a
C. rb
D. none of above

9. Header files are ______

A. A header file is a file with extension .h
B. Header files provide predefined functions to make programming easier
C. header files contain the set of predefined standard library functions
D. all of above

10. What are the part of Conditional Compilation?

A. #endif
B. #define
C. #undef
D. None of above

11. Extension of text file is

A. .com
B. .in
C. .txt
D. .ppt

12. Macro is defined by__________

A. #endif
B. #define
C. #else
D. #ifdef

13. How to import User-defined header files_______

A. using “#include”.
B. using “#ifdef”.
C. using “#define”.
D. None of above

14. File type can be represented by_______

A. File extension
B. File name
C. file identifier
D. none of above

15. A file is a sequence of __________

A. bytes
B. lines
C. bits
D. All of above

Answers for Self Assessment

1.	D	2.	D	3.	C	4.	A	5.	B
6.	B	7.	A	8.	B	9.	D	10.	A
11.	C	12.	B	13.	A	14.	A	15.	D

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. Write a program to read a file and display contents with its line numbers.

6. What is significance of Macros in programming?

7. Differentiate between user define and predefine Macros.

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.

YashvantKanetkar, Let us C

Web links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in

ISBN 978-93-94068-23-0