COURSE PREPARATION

Dr. S. M. K. Quadri

Department of Computer Sciences
University of Kashmir

Wasim Ahmad Bhat

Department of Computer Sciences
University of Kashmir

COURSE COORDINATION & FORMAT EDITING

Wasim Ahmad Bhat

Department of Computer Sciences
University of Kashmir

Published By

Prof (Dr.) Neelofar Khan

Director

Directorate of Distance Education

University of Kashmir

Srinagar

Year of Publication: 2012

Srinagar

CONTENTS

UNIT I

Lesson 1: Introduction to C Programming Language

Lesson 2: C Language Constructs

Lesson 3: Variables and Data Types

Lesson 4: C Statements

Lesson 5: Operators and Expressions

UNIT II

Lesson 6: Control Flow – Decision Making

Lesson 7: Control Flow – Looping

Lesson 8: Control Flow – Nesting

Lesson 9: Arrays

Lesson 10: Structures & Unions

UNIT III

Lesson 11: Pointers

Lesson 12: Functions

Lesson 13: Function Parameters

Lesson 14: Pass by Value/Address

Lesson 15: Recursion

UNIT IV

Lesson 16: Introduction to Header Files

Lesson 17: C Pre-Processor

Lesson 18: Macros

Lesson 19: File Processing in C

Unit I

CONTENTS

Lesson 1

Introduction to C Programming Language

Lesson 2

C Language Constructs

Lesson 3

Variables and Data Types

Lesson 4

C Statements

Lesson 5

Operators and Expressions

Lesson 1

Introduction to C Programming Language

Structure

3.0. Introduction

3.1. Why learn C?

3.2. History of C language

3.3. Features of C language

3.4. Advantages of C language

3.5. Disadvantages of C language

3.6. Turbo C Compiler

3.7. Summary

3.8. Model Questions

1.0 Introduction

In order to communicate any idea, thought, instruction or information, humans make use of spoken language. The fact is that you have just understood the very first sentence of this chapter all due to that. However, spoken languages are not understood by machines. Not only machines but also other living creatures of this planet do not understand spoken languages. Hence, they need some different kind of a language. As an example, a ring master in circus uses the movement and sound of a whip to control the beast. Similarly, machines like cars understand mechanical motions such as the movement of steering wheel, brake pedal and so on, to perform specific actions. This means whenever some instruction is to be communicated to any living or non-living thing, we need some kind of a specific language that is understood by it. As such, computer is not an exception.

In order to communicate instructions to a computer, humans need a language which we call a programming language. Like we have a variety of spoken languages, we also have a variety of programming languages. However, computers nowadays are digital and only understand the language of 0’s and 1’s. Therefore, to instruct a computer to perform some specific task we have to put these 0’s and 1’s in a particular sequence. This kind of a programming language is called Machine Language. A set of these instructions is called a Program like a set of dialogs in a spoken language is called a Conversation. Communicating instructions to a digital computer in machine language is difficult, error prone, non-portable and so on. To overcome some of its drawbacks (specifically difficulty) an assembly language can be used which actually uses mnemonics for these strings of 1’s and 0’s. Indeed assembly language is not understood directly by a computer and hence a translator (which is called assembler) is required to convert these mnemonics into binary strings. A translator converts a source program written in some programming language into executable machine code. Machine and Assembly language, both called low-level programming languages, are highly machine dependent, error-prone, difficult to understand and learn, and so on. A high-level programming language is machine independent, less error-prone, easy to understand and learn, and so on. In this language, the statements (or instructions) are written using English words and a set of familiar mathematical symbols which makes it easier to understand and learn, and is thus less error-prone. Furthermore, the language is machine independent and hence can be ported to other machines. The high-level programming languages can be further classified into procedural, non-procedural and problem oriented languages. This text discusses the vocabulary, grammatical rules and technical aspects of a high-level procedural language called C. It must be noted that all high-level languages also need a translator to convert it into machine language. Depending upon whether the translation is made to actual machine code or some intermediate code, the translator is called a Compiler or an Interpreter respectively. C programming language uses a compiler to convert the instructions into actual machine code. Some researchers also classify C language as a middle level language. The reason behind this is that assembly code can also be mixed with C code.

This lesson will look into the motivation behind learning C language, unveil its history of development, highlight its features, and point out advantages and disadvantages of C programming language.

1.1 Why learn C?

One may argue that if there is plethora of programming languages available, then what makes C language so special? There are two answers to this question. First, C language has been used by programmers for past 30-40 years to develop every kind of utility. This means the language is well understood, the issues with the language have been completed eradicated, a lot of the principles used in C show up in a lot of other languages, and so on. Second, C combines portability across various computer architectures as provided by any other high-level language while retaining most of the control of the hardware as provided by assembly language. It is a stable and mature language whose features are unlikely to disappear for a long time

1.2 History of C language

In 1965, Bell Telephone Laboratories, General Electric Company and Massachusetts Institute of Technology where working together to develop a new operating system called MULTICS.

In 1969, Bell Laboratories ended its participation in the project. However, the participating members of Bell Labs, mainly Ken Thompson and Dennis Ritchie, still had an intention to create such kind of an OS which finally matured into UNIX operating system.

After it early success in 1970, Ken Thompson set out to implement a FORTRAN compiler for the new system, but instead came up with the language B. B Language was influenced by Richard Martins BCPL language which was type-less and interpretive. Hence, both BCPL and B language had some performance drawbacks.

In 1972, in an effort to add “types” and definition of data structures to B language and use compiler instead of interpreter, Dennis Ritchie came up with a new B language called C language.

Just like natural languages, programming languages also change. The original specification of the C language (as devised by Dennis Richie) together with close variations is sometimes known as Old-style C or

Traditional C. However, in 1988 ANSI (American National Standards Institute) published a new specification of the language which has become an international standard which is accepted by all compilers. It is known as ANSI C. ANSI C is mostly a superset (i.e. provides additional functionality) over Old-style C.

1.3 Features of C language

C language has a rich set of features. These include:

1. There are a small, fixed number of keywords, including a full set of control flow primitives. This means there is not much vocabulary to learn.
2. C is a powerful, flexible language that provides fast program execution and imposes few constraints on the programmer.
3. It allows low level access to information and commands while still retaining the portability and syntax of a high level language. These qualities make it a useful language for both systems programming and general purpose programs.
4. Another strong point of C is its use of modularity. Sections of code can be stored in libraries for re-use in future programs.
5. There are a large number of arithmetical, relational, bitwise and logical operators.
6. All data has a type, but implicit conversions can be performed. User-defined and compound data types are possible.
7. Complex functionality such as I/O, string manipulation, and mathematical functions are consistently delegated to library routines.
8. C is a Structured Compiled language widely used in the development of system software.

1.4 Advantages of C language

C language has in-numerous advantages. The important advantages of C language are described as follows:

1. Easy to Understand: C language is a structured programming language. The program written in C language is easy to understand and modify.

2. Middle Level Language: C language has combined features of low level language (such as assembly language) and some of high level language. It is, therefore, sometimes C language is also referred to as middle level language. Most of the problems that can be solved using assembly language can also be solved in C language.
3. Machine Independent: Program written in C language is machine independent. It means that a program written on one type of Computer system can be executed on another type of Computer.
4. Built in Functions: C language has a large number of built in functions. The programmer uses most of these built in functions for writing source program, instead of writing its own code.
5. Hardware Control: Like assembly language, the programmer can write programs in C to directly access or control the hardware components of the computer system.
6. Easy to learn and use: C language is easy to learn and to write program as compared to low level languages such as assembly language.
7. Basis for C++: Today, the most popular programming is C+. C is the basis for C++. The program structure of C is similar to C++. The statements, commands (or functions) and methodologies used in C are also available in C++. Thus learning C is a first step towards learning C++.
8. Modularity: C is a structured programming language. The programmer can divide the logic of program into smaller units or modules. These modules can be written and translated independently. Including

1.5 Disadvantages of C language

Though C language has rich set of features and have enormous advantages, however it suffers from some disadvantages. These include:

1. C does not have OOPS feature.
2. There is no runtime checking in C language.
3. There is no strict type checking.
4. C doesn't have the concept of namespace.

5. C imposes few constraints on the programmer. The main area this shows up is in C's lack of type checking. This can be a powerful advantage to an experienced programmer but a dangerous disadvantage to a novice.

1.6 Turbo C Compiler

Though there exist many C compilers which follow standard specification of C language, the most and commonly used C compiler is Turbo C compiler (TC). Turbo C compiler is the product of Borland International Inc. and runs on DOS and Windows platform. In addition to being widely used, the TC has been fully understood and most of the C books have been written keeping TC in view. This text also discusses the C programming language keeping in view its implementation in Turbo C. Specifically; the text considers TC++ 3.0.

TC++3.0 can be downloaded from web. By default, it gets installed in C: drive of your computer under the name TC. Within this folder, many sub-folders and files exist. The most important sub-folder called BIN contains all the necessary binaries (executable) of the TC. To start TC, we have to execute TC.EXE file in BIN subfolder. After execution, TC IDE pops up. The IDE (Integrated Development Environment) is a tool that allows a C programmer to write a program, compile it, execute and debug it. Figure 1.1 shows the screen shot of TC++3.0 IDE.

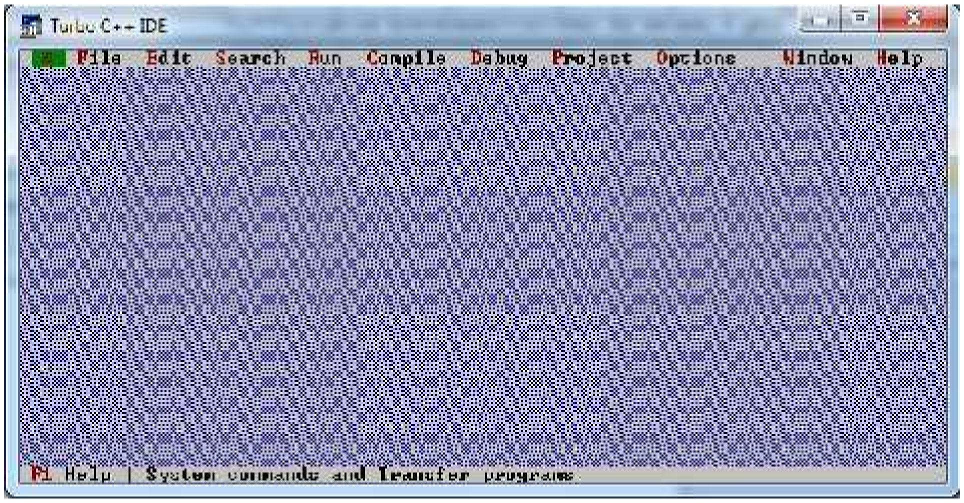

Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
F1 Help | System commands and Transfer programs

Figure 1.1: Screenshot of TC++3.0 IDE

We will assume that you know how to create, open, save, edit, etc. a file using an editor. However, it must be noted that C programs should be given extension **.c** while saving. IDE provide various commands to compile, run and debug the program. Table 1.1 shows the most important commands that you will be using.

Command	**Shortcut Key**
Compile a program	ALT + F9
Run a program	CTRL + F9
Check previous output	ALT + F5
Save changes	F2
Trace Execution	F7

Table 1.1: Some commonly used commands of IDE.

1.7 Summary

- Programming languages are means to instruct a computer to perform some desired task.
- A program is a set of instructions which are converted into equivalent machine instructions understood by machine.
- All programming languages require a translator to convert the program into machine code.
- C language is a structured compiled programming language which is widely used in the development of application and system software.
- C language was developed by Dennis Ritchie in order to port UNIX across various platforms.
- C language has rich set of features, innumerous advantages and few disadvantages.
- Turbo C is the most widely used and popular C compiler on DOS and Windows platform.

1.8 Model Questions

Q 1. Write a short note on classification of programming languages.

Q 2. Discuss history of C programming C language.

Q 3. Explain features of C programming language.

Q 4. Discuss advantages and dis-advantages of C programming language.

Q 5. Write a short note on Turbo C compiler.

Lesson 2

C Language Constructs

Structure

3.9. Introduction

3.10. Character Set

3.11. Tokens

3.12. Keywords

3.13. Identifiers

3.14. Constants

3.15. Variables

3.16. Summary

3.17. Model Questions

2.0 Introduction

Every language has a set of symbols that are used to create tokens or words. These tokens are used to create statements. A statement is a meaningful dialog. Finally, these statements are used to create a conversation_(Program). However, every language has a different set of symbols and has some different set of rules to create tokens, statements and conversation. As an example, English language has 26 lower-case and 26 upper-case symbols called Alphabets. These alphabets are used to create tokens like nouns, verbs and so on. These tokens are used to create statements called Sentences using some grammatical rules. Similarly, every programming language like C has a set of characters that are used to create tokens which in turn are used to create meaningful statements. These statements constitute a program. Note that certain rules are to be followed to create tokens and compose statements using these tokens. These rules are called Syntax Rules or Grammar. The character set, tokens and syntax rules of a programming language together are called Language Constructs.

In this lesson, we will discuss the language constructs of C programming language.

2.1 Character Set

The character set of C comprises of wide range of symbols that can be used to create tokens. These symbols are available on all modern personal computers. These characters can be grouped into following categories:

1. Letters
2. Digits
3. Special Characters
4. White Spaces

Table 2.1 shows the complete set of characters supported by C language.

Letters	**Digits**	**White Spaces**
Upper Case A to Z	0 to 9	Blank Space
Lower Case a to z		Tab
		Carriage Return
		Line Feed
	Special Characters	
, Comma	. Period	; Semicolon
: Colon	? Question Mark	‘ Apostrophe
“ Quotation Mark	! Exclamation Mark	| Vertical Bar
/ Slash	\ Backslash	~ Tilde
_ Underscore	$ Dollar Sign	% Percent Sign
# Number Sign	& Ampersand	^ Caret

* Asterisk	- Minus Sign	+ Plus Sign
< Opening Angle	> Closing Angle	(Left Parenthesis
) Right Parenthesis	[Left Bracket	] Right Bracket
{ Left Brace	} Right Brace	

Table 2.1 C Character Set

2.2 Tokens

In C language the smallest unit of a program is called Token. Tokens are created using the C character set. These tokens are delimited from each other by white spaces just like words in English language are delimited by white spaces. The Syntax Rule of the C language dictates how to create tokens in C and how to use these tokens to create meaningful statements. C language has 5 types of tokens as shown in Figure 2.1.

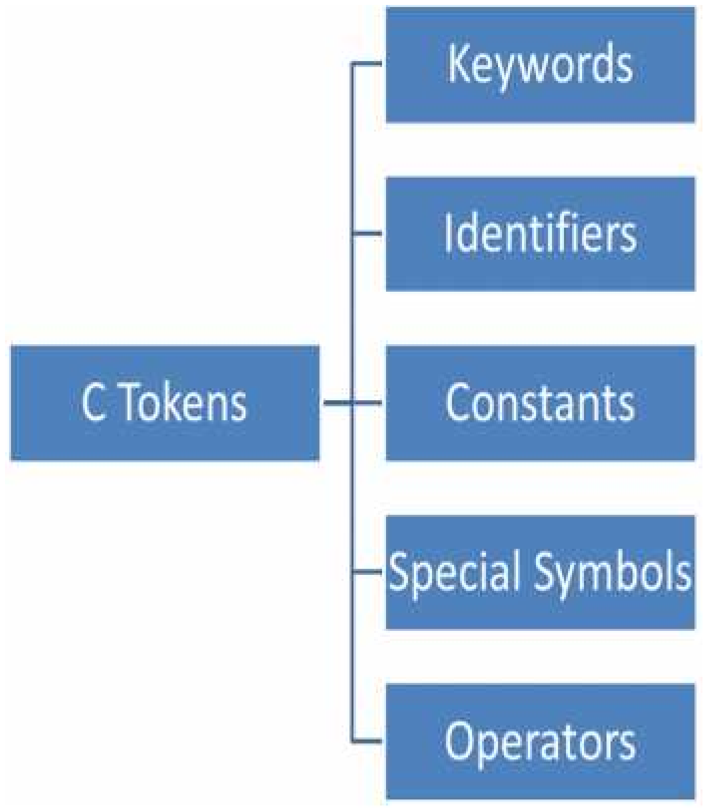


Figure 2.1: Classification of Tokens in C Language

2.3 Keywords

Keywords are those tokens or words whose meaning are already defined within the C language and cannot be changed. As an analogy, in English language the meaning of verbs, adjectives, common nouns, and so on is pre-defined and can’t be changed. These tokens of English language are its keywords. Similarly, in C language some tokens are reserved and have some pre-defined meaning. These keywords serve as the basic building blocks for program statements. The number of keywords present in C language is 32 as per ANSI standard. Table 2.2 shows this list of keywords.

auto	double	int	struct
break	Else	long	switch
case	Enum	register	typedef
char	Extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 2.2: Keywords in C language

2.4 Identifiers

Identifiers are user-defined tokens used to name variables, functions, etc. An identifier consists of a sequence of letters and digits. Underscore is also used. However, the first character of the identifier should a letter or an underscore. Though there is logically no restriction on the length of the number of characters in an identifier, however for some compilers only 31 characters are significant to differentiate between 2 or more identifiers. It should be noted that identifier is case sensitive. This means an identifier "test" is different from another identifier "teSt". Table 2.3 shows some valid and invalid identifiers along with the reason.

Identifier Name	Valid	Reason
First_name	Yes	
extern	No	extern is a keyword
Money$	No	Only Alphabets, Digits and Underscore are valid
extern_name	Yes	
First name	No	No space is allowed

Table 2.3: Examples of some valid and invalid identifiers

2.5 Constants

Constants are those tokens in C language whose value is fixed and does not change during the execution of a program. There are various types of constants in C.

2.5.1 Integer Constants

These constants refer to a sequence of digits. Any representation can be used to specify an integer constant. Table 2.4 shows ways to specify an integer constant, whose value is 1234 in decimal number system, in different number systems.

Representation	Value
Decimal	1234
Hexadecimal	0x4D2
Octal	02322

Table 2.4: Representation of an integer in different number systems

2.5.2 Real Constants

These constants refer to those numbers which contain fractional parts like 12.34. Such numbers are called real or floating point numbers. A real constant can also be expressed in exponential or scientific notation. As an example, 12.34 can also be expressed as 1234E-2. In this form, the real number is expressed in terms of mantissa and exponent. The mantissa is either a real number or an integer and exponent is an integer number. Both can have an optional plus or minus sign. The letter E separating the mantissa and the exponent can be written in lowercase also.

2.5.3 Character Constants

These constants refer to those tokens which contain a single character enclosed within a pair of single quotes like ‘W’. The character may be a letter, number, special character or blank space. As an example, ‘5’ is a character constant while as 5 is an integer constant. Similarly, 5.0 can be treated as real constant.

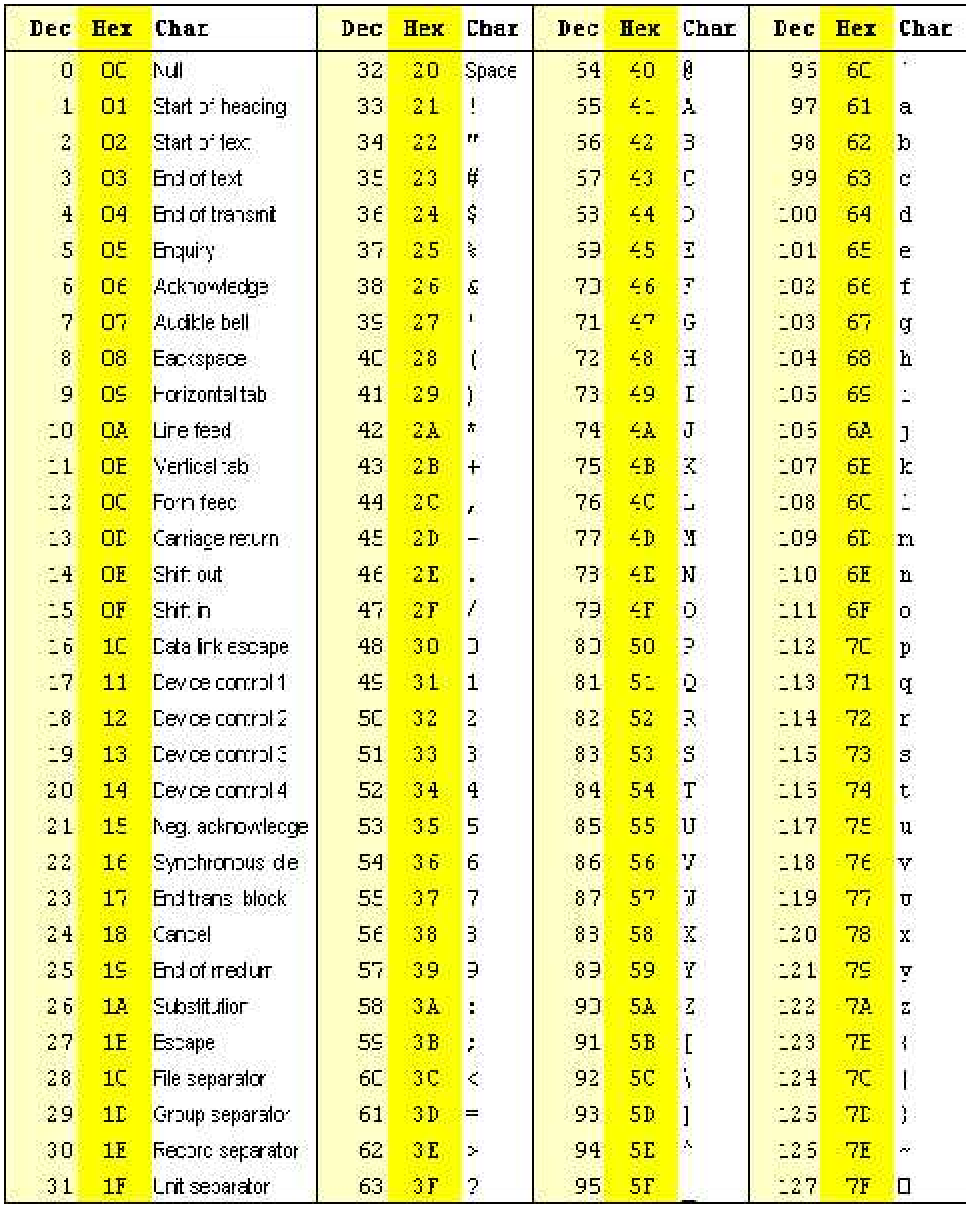
Character constants are printable characters which are stored in memory as their ASCII (American Standard Code for Information Interchange) code. Figure 2.2 shows the complete ASCII table.

2.5.4 String Constants

These constants refer to those tokens which contain a string of characters enclosed within a pair of double quotes. The characters may be letters, numbers, special characters and blank space. Examples include “Hello World!”, “123”, and so on. It should be noted “123” is a string constant while 123 is integer constant.

2.5.4 Backslash Character Constants

These constants refer to those special character constants which contain a backslash and some character enclosed within a pair of single quotes. These constants are used in output functions and are known as escape sequences. Table 2.5 shows the list of backslash character constants along with their interpretation.



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Figure 2.2: ASCII Table

Backslash Character	**Meaning**
\a	alert (bell) character
\b	backspace
\f	form feed
\n	newline

\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\?	question mark
\'	single quote
\"	double quote

Table 2.5: List of backslash character constants in C

2.6 Variables

A Variable represents some memory location which is used to store some value. Unlike constants, a variable may take different values at during times during execution. In other words, the value within a variable can be changed; hence the name. A Variable-name is an identifier token that follows the syntax rules for creating identifier (refer to section 2.4), however programmers choose variable names in such way so as to reflect its function or nature in program. As an example, if length and breadth of a rectangle is required to be stored in some variable, one can use srinagar and kashmir as variable names. Though the names are legitimate but it would be better to use length_rect and breadth_rect.

2.7 Summary

- Every language has a grammar, so does C programming language.
- The character set of C language consists of alphabets, digits and special characters.
- The tokens in C language are delimited by white space.
- The tokens can be classified as keywords, identifiers, constants, special symbols and operators.

- C language has small vocabulary, just 32 keywords whose meaning is fixed and predefined.
- C language can create a wide variety of identifiers which are used to name variables, function, array, etc.
- C language supports integer constants, floating point constants, character constants, string constants and backslash character constants.
- Every character is stored as some ASCII code.
- A variable is a name given to some memory location that holds some data. The contents of variable can be changed.
- In C, variable-names are identifier tokens and the syntax rules to create variable names is same as that of identifiers.

2.8 Model Questions

Q 6. Enumerate the set of characters available in C language.

Q 7. Discuss classification on tokens in C programming language.

Q 8. Enumerate various keywords available in C programming language.

Q 9. Differentiate between Keywords and Identifiers in C programming language.

Q 10. Explain various types of constants available in C programming language.

Lesson 3

Variables and Data Types

Structure

3.18. Introduction

3.19. Variables and Data Types

3.20. Classification of Data Types

3.21. Integer Data Type

3.22. Floating Point Data Type

3.23. Character Data Type

3.24. Declaration of a Variable

3.25. Initialization of a Variable

3.26. User Defined Data Types

3.27. Summary

3.28. Model Questions

3.0 Introduction

A computer processes data according to the instructions specified in a program and produces the output. This processing is done by Central Processing Unit (CPU) of the computer which performs arithmetic operations such as addition, subtraction, multiplication and division, and logical operations such as comparing two numbers. The data and programs are stored in primary memory (RAM) of a computer where from it is fetched by CPU. This computer memory is made up of cells which are capable of holding information in the form of binary digits 0 and 1 (bits). This is the way information is stored in a memory just like we memorize numbers using digits 0 to 9. Every such cell of computer memory has a unique memory address. In order to instruct CPU to process some data located in its memory we have to specify the address of that location. However, it is very difficult for humans to remember the memory addresses in numbers like

316976. So instead of using some number to address any memory location which contains data, we label that location by a Variable. This means using variables a programmer no more needs to be concerned about the actual location in memory where some data is stored, rather he/she can simply use the label i.e. Variable name, to access it.

In addition, the CPU does not access memory by individual bits; instead it accesses data in a collection of bits, typically 8 bits, 16 bit, 32 bit or 64 bit. Furthermore, the data within that memory location may be of any type like integer, floating point, character, and so on. To further qualify that memory location, we associate the type of data it contains, which also includes the width of data. C programming language provides a rich set of Data Types which in association with variables can be used to label some memory location, specify the type of data it contains and width of that data.

In this lesson, we will look at various data types supported by C language, syntax rules to declare, initialize and access a variable, classify data types supported by C language, and so on.

3.1 Variables and Data Types

Variables are a way of reserving memory to hold some data and assign names to them so that we don't have to remember the numbers like 316976 and instead we can use the memory location by simply referring to the variable name. Every variable is mapped to a unique memory address. However, this is not enough. A variable can contain different types of data and each type may have different space requirement. Thus as mentioned earlier we also specify the data type of a variable which signifies the type of data it contains and the space it requires to hold it. Having a variety of data types at the disposal of programmer allows him to make appropriate data type selection as per the nature of computation and type of machine. In addition, the operations legitimate for one type of data may be invalid for some other type. Having specified the data type of a variable allows the compiler to make compile time type checking in order to avoid run time failures.

3.2 Classification of Data Types

The wide variety of data types supported by C language can be classified into 3 classes:

1. Primary Data Types
2. User-defined Data Types
3. Derived Data Types

C supports 4 primary data types which include **integer, character, floating point** and **double-precision floating point**.

C also supports creating user-defined data types using keywords **typedef** and **enum.**

Both classes of these data-types will be discussed in this lesson however derived data types will be discussed in lesson 9 and lesson 10.

3.3 Integer Data Type

The Integer data type is used to specify a variable that will contain only integer values, i.e. any positive or negative number without fractional part. The keyword used by C language to specify some variable as integer is int. The width of memory allocated to an integer variable is generally one word. However, the word length varies from machine to machine. For a 16-bit machine, the word length is 16-bits. Also, Turbo C 3.0 is a 16-bit compiler which means the size of an integer variable will be 2 bytes. Please note that because we will be implementing all our programs on TC3.0, so we will consider a 16-bit machine for here onwards. Furthermore, to represent both positive and negative integers, one bit is reserved for sign. This means, only 15-bits are used to store the magnitude of data while the remaining one bit is used to store the sign of the data. Therefore, the range of integers that can be stored in an integer variable can have a minimum value of -32768 and a maximum value of 32767 (-2^{15} to $+2^{15}-1$). The highest positive value is one less than the highest negative value, because 0 is treated as positive integer.

By default int specifies the variable as signed integer variable with width equal to machine word. However, C also provides certain qualifiers to further qualify the integer data type. These qualifiers can be classified into 2 classes.

1. One class specifies whether the integer variable will hold signed numbers (both positive and negative) or unsigned numbers (only positive numbers). The keyword signed if prepended to int keyword specifies that the integer variable will hold both positive and negative numbers. Indeed, by default int is signed. As mentioned earlier, the range of such a variable will be -32786 to +32767. If the keyword unsigned is prepended to int keyword, the variable so declared will only hold positive integers. This means no sign bit will be reserved and full word (16-bit in our case) will contain the magnitude. This means the variable can contain a range of integers with a minimum value of 0 to a maximum value of 65535 (0 to 2^{16}-1).

2. The other class specifies the size of integer variable. By default its size is one word (16-bits in our case). However, if short keyword is prepended to int keyword then the variable occupies only 8 bits. In contrast, if long keyword is prepended to int keyword then the variable occupies double word (32-bits in our case).

It should be noted that both classes of qualifiers can be used simultaneously. Table 3.1 shows various variants of integer data type along with their width and range on 16-bit machine.

Data Type	Width (in bits)	Range
int or signed int	16	-32768 to 32767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255

signed long int or long int	32	-2147483648 to 2147483647
unsigned long int	32	0 to 4294967295

Table 3.1: Various forms of Integer data type in C

3.4 Floating Point Data Type

The Floating Point data type is used to specify a variable that will contain real numbers, i.e. any positive or negative number with fractional part. The keyword used by C language to specify some variable as real is float. The width of memory allocated to floating point variable is 32 bits irrespective of the word size of the machine. The float data type can store real numbers with a precision of 6 digits. In order to store larger width floating point numbers with more precision, C provides double and long double floating point data types. The variable declared as double data type occupies 64 bits while the one declared as long double occupies 80 bits. Both also provide more precision as compared to float. It should be noted that float, double and long double only store signed real numbers. Table 3.2 shows various variants of floating point data type along with their width and range.

Data Type	Width (in bits)	Range
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 1.1E+4932

Table 3.2: Various forms of Floating Point data type in C

3.5 Character Data Type

The Character data type is used to specify a variable that will contain characters from ASCII table like ‘A’, ‘%’, and so on. The keyword used by C language to specify some variable as character is char. The width of memory allocated to a character variable is 8 bits. The char variable stores the character internally as an integer which corresponds to its equivalent ASCII code. This means a direct assignment of some valid 8-bit positive integer is also possible. However, it will be interpreted as its equivalent ASCII character in character manipulating functions. Consequently, negative integer values are also possible though they do not have any representation in ASCII code. By default, a char variable is signed. This means, only 7-bits are used to store the magnitude of data while the remaining one bit is used to store the sign of the data. Therefore, the range of character codes that can be stored in character variable can have a minimum value of -128 and a maximum value of 127 (-2^7 to $+2^7 -1$). The highest positive value is one less than the highest negative value, because 0 is treated as positive integer. Furthermore, C also provides keywords signed and unsigned to be prepended to the char keyword to further qualify the variable so created as one that will hold signed or unsigned integer values respectively. Table 3.3 shows various variants of character data type along with their width and range.

Data Type	Width (in bits)	Range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255

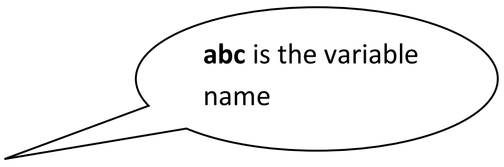
Table 3.3: Various forms of Character data type in C

3.6 Declaration of a Variable

All the variables that are to be used in a program must be declared first. The declaration of variable means specifying the identifier (i.e. variable name), the type and width of data it will hold (i.e. data type) to the compiler. The general syntax of declaring a variable in C is as follows:

Data-type Variable-name;

```
int x;

float y;

int abc;

float def;
```



Also, multiple variable declaration of same data type in a single line is possible. The general syntax for this is as follows:

Data-type Variable-name1[, ... Variable-nameN];

This means the above 4 statements can written in 2 statements as follows:

```
int x, abc;

float y, def;
```

Please note that as every sentence in English language ends with a period (.), every statement in C language ends with a semi-colon (;) called terminator. Table 3.4 shows some valid and invalid examples of variable declaration in C language.

Declaration	Status
int x, y, z, abc, def, pgdca, qwert;	Valid

float x;
int y;
	Valid

char c;	

float x, y;
 	
int x, y;
	Invalid; because a variable can’t be of two types

int x; int x, y, z;	Invalid; because a single variable can’t be declared twice
int 123x;	Invalid variable-name

Table 3.4: Some valid and invalid examples of variable declaration in C language

3.7 Initialization of a Variable

After a variable is declared it may be required to initialize it. Initialization of a variable means assigning some value to it. However, technically there is no problem with using a variable without initializing it. But logically it is wrong. When a variable is declared but not initialized it contains some random bogus but valid value. As an example, if a variable x is declared as int, then x may contain say 1278 value before its initialization. The value is bogus but valid. There are generally 2 ways to initialize a variable.

1. First, during declaration we can assign some value to it. As an example,

   ```
   int x = 1234;
   ```

   This way variable x is initialized with value 1234 when it is declared as integer.

2. Second, after the variable is declared we can assign it some value in a separate statement. As an example,

```
int x;

x = 1234;
```

3.8 User Defined Data Types

C language allows a programmer to create user defined data types. It provides 2 keywords to do so; typedef and enum.

3.8.1 typedef

typedef allows a programmer to define an identifier that would represent an existing data type. Therefore, the identifier so defined can be later used to declare variables. The general syntax to do so is as follows:

```
typedef data-type identifier;
```

The existing data type can belong to any class of data types; primary, derived and even user-defined. It should be noted that the user-defined data type so created does not actually create a new data type, rather it creates a new name for the data type. As an example,

```
typedef int rollno;
```

This will create another name, rollno, for int data type. Therefore, the new name can also be used to declare variables and it is as good as the existing data type. It is worth mentioning here that int keyword still exists and thus can be used to declare variables.

Hence, the statement

```
rollno student1, student2;
```

is as good as statement

```
int student1, student2;
```

One may argue what for can this be used? The answer is using this mechanism the readability of a program is enhanced.

3.8.2 enum

enum is an enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces. The general syntax to do so is as follows:

```
        enum identifier {value1, value2, .... valuen};
```

Now, the identifier can be used to declare variables and these so declared variables can only hold one of the values within the curly braces. As an example,

```
        enum pgdca { sem1, sem2, sem3, sem4};
```

declares a user-defined enumerated data type pgdca which can only hold values as sem1, sem2, sem3 and sem4.

```
        pgdca student1=sem1, student2=sem2;
```

The above statement declares 2 variables of type pgdca and initializes them to some legitimate values.

3.9 Summary

- Variables need to be qualified with the type of data it is going to hold and size of memory it will take. Data Types allow a programmer to specify this to a compiler.
- There are 3 classes of data types; primary, derived and user-defined.
- The primary data types include integer, floating point and character. The keywords used to qualify a variable as integer, real or character are **int, float** and **char** respectively.
- Integer and character variables can be further qualified as **signed** or **unsigned**.
- Integer variables can also be qualified as **short** and **long.**
- Floating point variables can also be qualified as **double** and **long double.**
- A variable needs to be declared and initialized before use.
- The keywords used to create user defined data types are **typedef** and **enum**.

3.10 Model Questions

Q 11. Explain the concept of a variable and a data type in C programming language.

Q 12. Discuss the classification of data types in C programming language.

Q 13. Enumerate the basic/primary data types in C programming language.

Q 14. Explain why the highest positive value is one less than the highest negative value in case of **signed int** and in case of **signed char**?

Q 15. Discuss **short** and **long** keywords in C programming language.

Q 16. Discuss **signed** and **unsigned** keywords in C programming language.

Q 17. Differentiate between **float, double** and **long double** data types in C language.

Q 18. What are the various variants of an integer data type in C language?

Q 19. What are the various variants of a floating point data type in C language?

Q 20. What are the various variants of a character data type in C language?

Q 21. Discuss how a variable can be declared and initialized in C language.

Q 22. Explain how user-defined data types can be created in C programming language?

Lesson 4

C Statements

Structure

5.0. Introduction

5.1. Structure of a C program

5.2. Errors in a C program

5.3. Simple and Compound statements

5.4. Input/Output statements

5.5. Formatted I/O

5.6. Non-Formatted I/O

5.7. Storage Classes

5.8. Summary

5.9. Model Questions

4.0 Introduction

The meaningful sentences are at the heart of every spoken language. Similarly, meaningful statements are at the heart of C programming language. In addition, as in English language the idea that is being conveyed by the text should follow a well-defined template. Consider the case of an application which has a well-defined structure as follows:

1. To whom it is addressed
2. Subject
3. Reason for application
4. Justification for approval
5. Request for approval
6. Anticipatory thanks, and
7. The details of applicant

Similarly, C programs have a well-defined structure. Also, programs do I/O with devices, like reading key presses, displaying data on screen and so on.

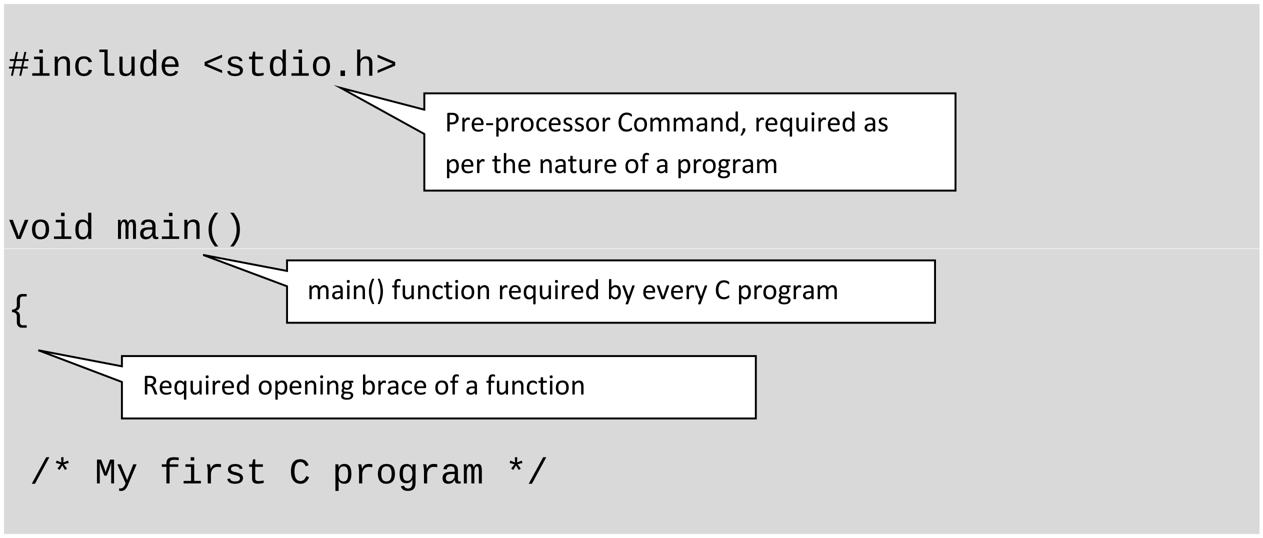
In this lesson, we will look at the structure of a C program, the types of C statements, I/O facility in C language, and storage classes supported by it. It should be noted that this lesson is very important and should be consulted in following lessons to better understand the text.

4.1 Structure of a C program

C is a case sensitive programming language which means in C two variables named x and X will have different meanings. However, it is a free-form language which means any C statement can start in any column but end of each C statement must be marked with a semicolon (;). In addition, multiple statements can be on the same line, statements can continue over multiple lines and white spaces (i.e. TAB and SPACE BAR) are ignored. In general, a C program basically contains following elements:

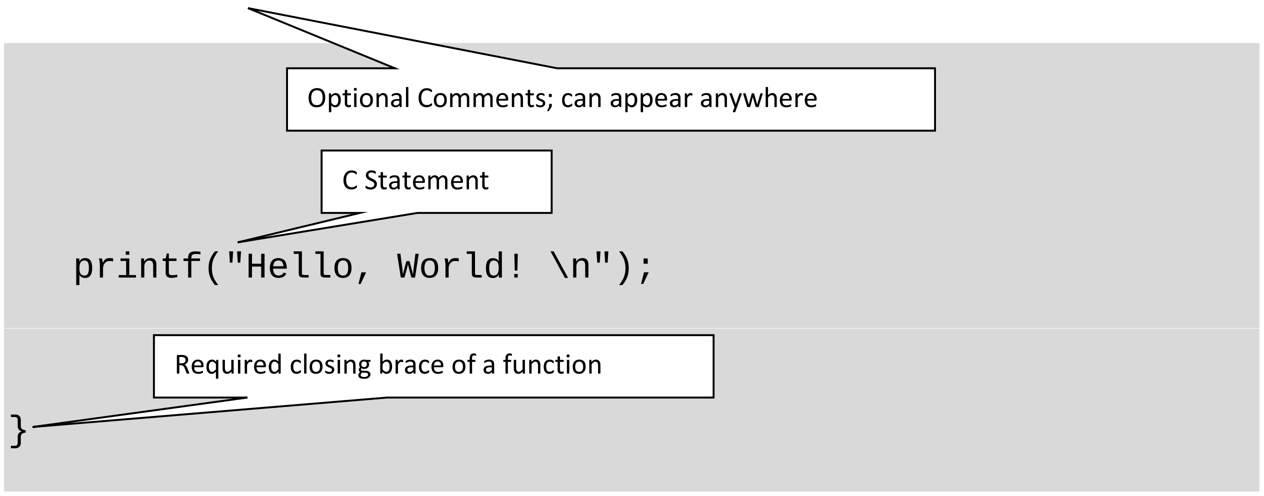
1. Preprocessor Commands
2. Functions
3. Variables
4. Statements & Expressions
5. Comments

Consider a simple program below:



```
#include <stdio.h>

void main()
{
 /* My first C program */
```

Pre-processor Command, required as per the nature of a program
main() function required by every C program
Required opening brace of a function


Optional Comments; can appear anywhere
C Statement
Required closing brace of a function

```
   printf("Hello, World! \n");

}
```

4.1.1 Pre-Processor Commands

These commands tell the compiler to do some processing before doing actual compilation. Like #include <stdio.h> is a preprocessor command which tells a C compiler to include stdio.h header file before going to actual compilation. stdio.h refers to a file supplied along with the C compiler which contains definitions of many functions that perform input-output with standard devices like keyword and screen. For example, one of the functions in the file stdio.h is printf() which accepts a string of character enclosed in double quotes to be displayed on the screen. This header file must be included in every program that reads some data from keyboard or displays some data on screen. stdio.h is not the only header file available in C language and to include other header files in a C program the same pre-processor command with different header file name can be used.

4.1.2 Functions

Functions are main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called main() function. The main() function is the most important function and must be present in every C program. The execution of a C program begins in the main() function. The opening brace { indicates the beginning of the function. The closing brace } indicates the end of the function. This function is prefixed with keyword void which means this function returns nothings when it exits. However, it can also be prefixed with int in which case it means it will return some value. And to return that value return keyword is used.

The C Programming language provides a set of built-in functions. In the above example printf() is a C built-in function which is used to print anything on the screen.

4.1.3 Variables

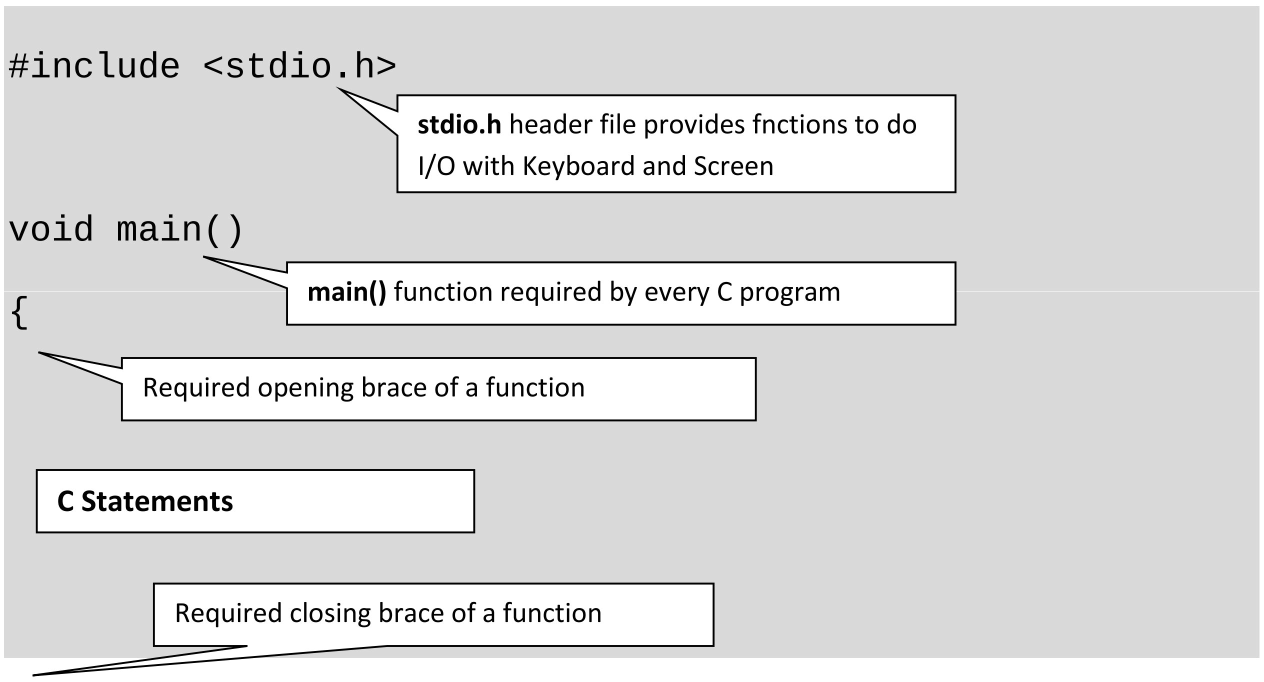
Variables are used to hold numbers, strings and complex data for manipulation.

4.1.4 Statements & Expressions

Expressions combine operators, variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

4.1.5 Comments

Comments are used to give additional useful information inside a C Program. Comments are optional and are just used to increase the readability of the program. All the comments are put inside /*...*/ as shown in the previous example. However, this comment can span through multiple lines and may appear anywhere in a C program. Single line comments can also entered by just prefixing them with //. The program 4.1. shows the general template of a C program that will used to create and test our programs.



```
#include <stdio.h>

void main()
{
```

stdio.h header file provides fnctions to do I/O with Keyboard and Screen
main() function required by every C program
Required opening brace of a function
C Statements
Required closing brace of a function

```
}
```

Program 4.1: General template of a C program

4.2 Errors in a C program

Error is a condition either during compilation or execution of a program which if not fixed can halt compilation or execution. There are three types of errors — syntax, logical, and run-time errors.

4.2.1 Syntax Error

These errors occur because of wrongly typed statements, which are not according to the syntax or grammatical rules of the language. For example, in C, if you don't place a semi-colon after the statement it results in a syntax error.

```
/* Syntatically wrong */

   printf("Hello, World! \n")
/* Syntatically Correct */

   printf("Hello, World! \n");
```

These errors are caught by compiler during compilation and can be fixed by correcting the syntax of the statement.

4.2.2 Logical Error

These errors occur because of logically incorrect instructions in the program. Let us assume that in a program which was supposed to do addition of 2 numbers, it was wrongly written to perform subtraction. This logically incorrect instruction will produce wrong results. Detecting such errors is difficult as they only surface during run-time that too occasionally.

4.2.3 Runtime Error

These errors occur during the execution of the programs though the program is free from syntax and logical errors. Some of the most common reasons for these errors are

1. When you instruct your computer to divide a number by zero.
2. When you instruct your computer to find logarithm of a negative number.
3. When you instruct your computer to find the square root of a negative integer.

4.3 Simple and Compound Statements

A Statement is the smallest standalone element of a programming language which in itself is collection of tokens of the language put in some proper sequence as per the syntax rules or grammar of the language. A sequence of one or more statements gives rise to a program. C language makes a distinction between statements and definitions, with a statement only containing executable code and a definition declaring an identifier.

The body of C functions (including the main() function) are made up of statements. These can either be Simple Statements or Compound Statements.

Simple Statements do not contain other statements and end with a semi-colon called Statement terminator. The simplest kind of statement in C is an expression. The following statements are simple statements and most of the statements in a typical C program are Simple Statements of this form.

```
    x = 2;
/* an assignment statement */

    x = 2+3;
/* another assignment statement */

    2+3;
/* has no effect & will be discarded */

    puts("hello, world");
```

```
/* a statement containing a function call */

     root2 = sqrt(2);
/* an assignment statement with a function call */
```

Compound Statements have other statements inside them; may be simple or other compound statements. The term Compound Statement (or Block) refers to a collection of statements (both Simple and Compound) that are enclosed in braces to form a single unit. Compound statements are not terminated with semicolons. The statement shown below is a compound statement. Notice that it contains three simple statements enclosed in a pair of curly braces. Also, though individual simple statements end with semi-colon, however the compound statement does not have one.

```
    {
    x = 2;
    y = 3;
    z = x + y;
    }
```

Compound statements generally form the body of functions, loops, conditional statements, and so on.

4.4 Input / Output Statements

Input/output is used for interfacing with outside world. Every programming language must provide a means to input some data into a program. This can be achieved by reading some input device, file, network, and so on. Similarly, every programming language must provide a means to output some data of a program onto screen, printer, file, and so on. However, C language does not provide any built-in input-output statements as part of its syntax. Therefore, programmers rely extensively on libraries for support in developing the user interface for a program. These libraries contain functions to perform I/O and thus a call to some specific I/O function is made within the main C program to accomplish I/O.

To use any of the C library function, that library must be first included in our program. Recall program 4.1 which shows the general template of a C program. The first line of that program contains a pre-processor command to include stdio.h header file. This command directs the compiler to include all the functions within the library to be included in the program. Thus, with the information of all the functions provided by it, one can now simply call those functions within the main body of program.

In fact, the header file stdio.h defines the standard input and output operations. This file forms part of the larger C standard library defined in the ANSI standard for the language. The standard input/output library is, by its very nature, standard and must therefore be suited to any hardware platform. Consequently the functions that it provides do not support color monitors because one cannot guarantee that the hardware platform has a color monitor. Similarly, the functions do not support mouse input, high resolution graphics output, or other “non-standard” input and output devices.

Nevertheless, the I/O functionality provided by stdio.h is standard, well documented, reliable and widely used. There are generally two classes of input/output functions provided by stdio.h header file. These include formatted I/O and non-formatted I/O functions. This is worth mentioning here that to fully understand the functions within formatted and non-formatted class, the rest of text should be read again in context of lesson 12.

4.5 Formatted I/O

The functions which fall in this category provide a convenient way to format the input and/or output data as per the requirements. The two main functions within this category are printf() and scanf().

4.5.1 printf()

printf() function writes to the standard output (monitor) a sequence of data, formatted as per the format specifier. The general syntax of printf() function is as follows

```
int printf (
    char * formatString
    [,argument, ...]
    );
```

where formatString contains a list of format specifiers indicating the format and type of data to be written to the screen. The data is stored in the corresponding argument. The argument can be numeric values, strings of characters, variables, expressions, and so on. formatString can contain any mix of the following

1. Literal text to be displayed on screen,
2. Escape sequences used as carriage control,
3. And, format specifiers for conversion of data in the arguments to a display format.

It should be noted that printf() function returns the number of characters displayed. Also, after the formatString parameter, depending on the formatString, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each format specifier in the formatString parameter.

The format specifier follows this prototype:

```
%[flags][width][.precision][length]specifier
```

Where specifier is the most important which defines the type and the interpretation of the value of the corresponding argument. Table 4.1 enumerates various specifiers supported by printf() function along with the interpretation and display format of the corresponding argument.

Specifier	**Interpretation and Display Format**
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character

f	Decimal floating point
g	Use the shorter of %e or %f
G	Use the shorter of %E or %f
o	Unsigned octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed. The argument must be a pointer to a signed int. where the number of characters written so
%	A % followed by another % character will write % to stdout.

Table 4.1: List of various specifiers supported by printf() function.

The format Specifier can also contain flags, width, .precision and length fields which are optional and follow these specifications:

Flags	**Description**
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.

#	Used with o, x or X *specifiers* The value is preceded with 0, 0x or 0X respectively for values different than zero.
	Used with e, E and f. It forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written.
	Used with g or G. The result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see *width* sub-specifier).
Width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format string*, but as an additional integer value argument preceding the argument that has to be formatted.
.Precision	**Description**

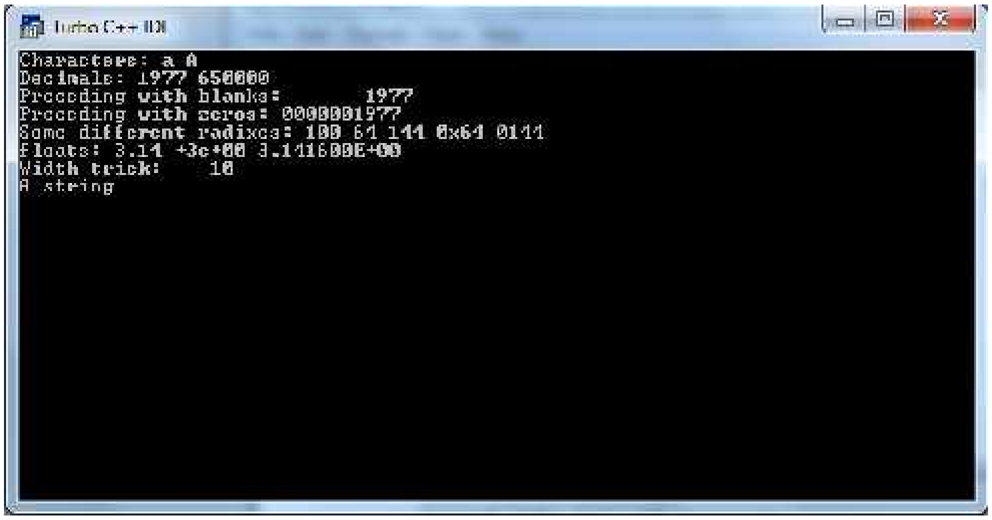
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For e, E and f specifiers: This is the number of digits to be printed after the decimal point.

Table 4.2: Lists and discusses the meaning of various fields of format-specifier

Program 4.2 shows some of the format specifiers for printf() function in action followed by the screenshot of the output.

```
#include <stdio.h>

void main()
{

printf ("Characters: %c %c \n", 'a', 65);

printf ("Decimals: %d %ld\n", 1977, 650000L);

printf ("Preceding with blanks: %10d \n", 1977);

printf ("Preceding with zeros: %010d \n", 1977);
printf ("Some different radixes: %d %x %o %#x %#o \n",
100, 100, 100, 100, 100);

printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416,
3.1416);

printf ("Width trick: %*d \n", 5, 10);

printf ("%s \n", "A string");

}
```

Progam 4.2: Program shows some of the format specifiers of printf() function in action



```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radixes: 100 64 144 0x64 0144
floats: 3.14 +3e+00 3.141600E+00
Width trick:    10
A string
```

4.5.2 scanf()

The scanf() function scans the data input through the keyboard and by default delimits values by whitespace. Whitespace is defined as being a TAB, a blank or the newline character ('\n'). Therefore, data that is input with the intention of having embedded blanks as part of the data value will be broken into several values and distributed among the input variables specified in the scanf() statement. The general syntax of the function is as follows

```
int scanf (
    char * formatString
    [,address, ...]
    );
```

where formatString is a list of format specifiers indicating the format and type of data to be read from the keyboard and stored in the corresponding address. There must be the same number of format specifiers and addresses. scanf() returns the number of input fields successfully scanned, converted, and stored. The return value does not include scanned fields that were not stored.

formatString contains one or more of the following items:

1. Whitespace character: The function will read and ignore any whitespace characters (this includes blank spaces and the newline and tab characters) which are encountered before the next non-whitespace character. This includes any quantity of whitespace characters, or none.
2. Non-whitespace character, except % sign: Any character that is not either a whitespace character (blank, newline or tab) or part of a format specifier (which begin with a % character) causes the function to read the next character from stdin, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of format. If the character does not match, the function fails, returning and leaving subsequent characters of stdin unread.
3. Format specifiers: A sequence formed by an initial percentage sign (%) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from stdin and stored in the locations pointed by the additional arguments.

A format specifier follows this prototype:

```
    %[*][width][modifiers]type
```

where

*	An optional starting asterisk indicates that the data is to be retrieved from keyboard but ignored, i.e. it is not stored in the corresponding argument.
width	Specifies the maximum number of characters to be read in the current reading operation
modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument:
	h : short int (for d, i and n), or unsigned short int (for o, u and x)
	l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g)
	L : long double (for e, f and g)
type	A character specifying the type of data to be read and how it is expected to be read.

The type field can have following values:

type	**Qualifying Input**	**Type of argument**

c	**Single character:** Reads the next character. If a *width* different from 1 is specified, the function reads *width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	**Decimal integer:** Number optionally preceded with a + or - sign.	int *
e,E,f,g,G	**Floating point:** Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally folowed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	**Octal integer**.	int *
s	**String of characters**. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank,	char *

	newline and tab).	
u	Unsigned decimal integer.	unsigned int *
x,X	Hexadecimal integer.	int *

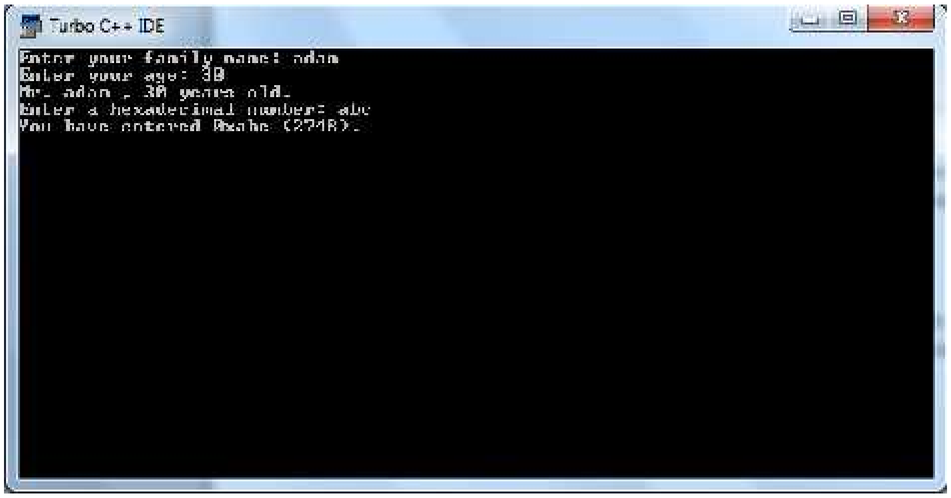
The function expects a sequence of references as additional arguments, each one pointing to an object of the type specified by their corresponding format-specifier within the formatString, in the same order. For each format specifier in the formatString that retrieves data, an additional argument should be specified. These arguments are expected to be references (pointers); so to store data in a regular variable, it should be precede with the reference operator, i.e. an ampersand sign (&), like as follows:

```
    int n;
    scanf ("%d",&n);
```

Program 4.3 shows some of the format specifiers of scanf() function in action followed by the screenshot of the output.

```

#include <stdio.h>

void main ()
{

  char str [80];
  int i;

  printf ("Enter your family name: ");

  scanf ("%s",str);

  printf ("Enter your age: ");

  scanf ("%d",&i);

  printf ("Mr. %s , %d years old.\n",str,i);

  printf ("Enter a hexadecimal number: ");

  scanf ("%x",&i);

  printf ("You have entered %#x(%d).\n",
  i,i);

}
```

Program 4.3: Program shows some of the format specifiers of scanf() function in action.



```
Enter your family name: adam
Enter your age: 30
Mr. adam , 30 years old.
Enter a hexadecimal number: abc
You have entered 0xabc (2748).
```

4.6 Non-Formatted I/O

The functions which fall in this category just provide bare bone functionality to read and write a character or string from or to the standard input or output respectively. There are four main functions within this category.

4.6.1 gets()

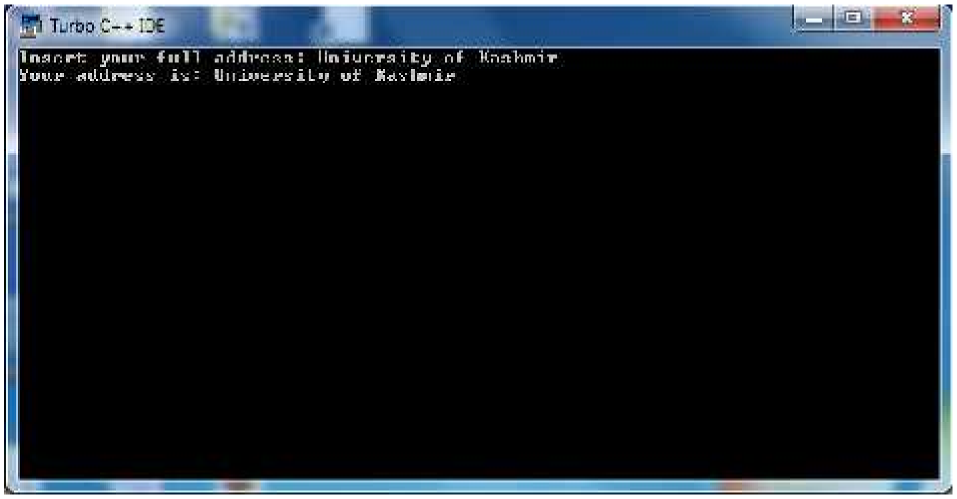
gets reads characters from keyboard and stores them as a string into the argument until a newline character (’\n’) is reached. The ending newline character (’\n’) is not included in the string. A null character (’\0’) is automatically appended after the last character copied to the argument to signal the end of the string. The general syntax of gets() function is as follows:

```
    char * gets ( char * argument );
```

where argument is a pointer to an array of chars where the C string is stored. Program 4.4 shows gets() function in action followed by screenshot of the output.

```
#include <stdio.h>

void main()
 {

 char string [256];

 printf ("Insert your full address: ");

 gets (string);

 printf ("Your address is: %s\n",string);

}
```

Program 4.4: Program shows gets() function in action.



```
Insert your full address: University of Kashmir
Your address is: University of Kashmir
```

4.6.2 puts()

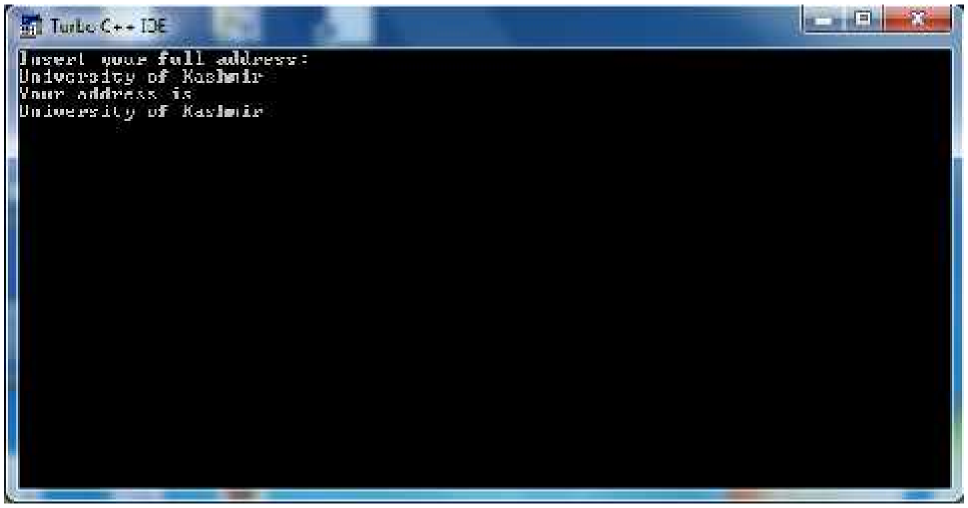
puts writes characters to screen which are stored in some string (argument) until a null character ('\0') is reached. The function begins copying from the address specified (argument) until it reaches the terminating null character ('\0') which is not copied to screen. The general syntax of puts() function is as follows:

```
    int puts ( char * argument );
```

where argument is a pointer to an array of chars where the C string is stored. Program 4.5 shows puts() function in action followed by screenshot of the output.

```
#include <stdio.h>

void main()
 {

 char string [256];

 puts ("Insert your full address: ");

 gets (string);

 puts ("Your address is");

 puts (string);

}
```

Program 4.5: Program shows puts() function in action.



```
Insert your full address:
University of Kashmir
Your address is
University of Kashmir
```

4.6.3 putchar()

putchar() function writes a single character to the current position in the standard output (screen). The general syntax of putchar() function is as follows:

```
    int putchar (char character );
```

where character is the character to be outputted. The following program displays W on screen.

```
    #include <stdio.h>

    void main ()
    {
      char c='W';

      putchar (c);
    }
```

4.6.4 getchar()

getchar() function reads a single character from keyboard. The general syntax of getchar() function is as follows:

```
    char getchar ();
```

The following program displays what it reads from keyboard on screen.

```
#include <stdio.h>

void main ()
{
  char c;

  c = getchar();

  putchar (c);

}
```

4.7 Storage Classes

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program. These specifiers precede the type that they modify. It is worth mentioning here that the following text should be read again in the context of functions discussed in lesson 13 to better understand it. There are following storage classes which can be used in a C program:

1. auto
2. register
3. static
4. extern

The auto storage class is the default storage class for all local variables. The keyword used to specify a variable as as auto storage class is auto.

```
{
int w;
auto int w;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

The register storage class is used to define local variables that should be stored in a register instead of RAM. The keyword used to specify a variable as within register storage class is register. This means that the variable has a maximum size equal to the register size (usually one word). The register should only be used for variables that require quick access such as counters. It should also be noted that defining register does not guarantee that the variable will be stored in a register. It will be stored in a register depending on hardware and implementation restrictions.

The static storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The keyword used to specify a variable as within static storage class is static. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

The extern storage class is used to give a reference of a global variable that is visible to all the program files. The keyword used to specify a variable as within extern storage class is extern. When you use extern the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined. When you have multiple files and you define a global variable or function which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another file. The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

4.8 Summary

- C programs have a well-defined structure.

- C programming language is a case sensitive programming language.
- C programming language is a free-form language.
- C program basically contains following elements:
    - Preprocessor Commands
    - Functions
    - Variables
    - Statements & Expressions
    - Comments
- There are three types of errors in C programming language — syntax, logical, and run-time errors.
- Simple Statements do not contain other statements and end with a semi-colon called Statement terminator.
- The term Compound Statement (or Block) both refers to a collection of statements (both Simple and Compound) that are enclosed in braces to form a single unit. Compound statements are not terminated with semicolons.
- C language does not provide any built-in input-output statements as part of its syntax. Therefore, programmers rely extensively on libraries for support in developing the user interface for a program.
- The header file **stdio.h** defines the standard input and output operations.
- The formatted I/O functions provide a convenient way to format the input and/or output data as per the requirements. The two main functions within this category are **printf()** and **scanf()**.
- The non-formatted functions just provide bare bone functionality to read and write a character or string from or to the standard input or output respectively. There are four main functions within this category which include **gets(), puts(), getchar()** and **putchar().**
- A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.
- There are following storage classes which can be used in a C program:
    - auto
    - register

- static
- extern

4.9 Model Questions

Q 23. Discuss the basic structure of a program in C programming language.

Q 24. What are pre-processor commands? Explain the concept with the help of a logical example.

Q 25. Discuss various types of errors possible in a C program.

Q 26. Compare Syntax errors, Logical errors and Run time errors.

Q 27. Differentiate between Simple and Compound statements in a C programming language.

Q 28. Discuss how I/O is accomplished in C programming language. Explain the significance and usage of **stdio.h** header file.

Q 29. Differentiate between formatted and non-formatted I/O in C language.

Q 30. With the help of a program explain the working of **printf()** function.

Q 31. With the help of a program explain the working of **scanf()** function.

Q 32. With the help of a program explain the working of **gets()** function.

Q 33. With the help of a program explain the working of **puts()** function.

Q 34. With the help of a program explain the working of **getchar()** function.

Q 35. With the help of a program explain the working of **putchar()** function.

Q 36. Differentiate between various storage classes supported by C programming language.

Lesson 5

Operators and Expressions

Structure

8.0. Introduction

8.1. Arithmetic Operators

8.2. Relational Operators

8.3. Logical Operators

8.4. Bitwise Operators

8.5. Assignment Operators

8.6. Miscellaneous Operators

8.7. Operator Precedence and Associativity

8.8. Type Conversions in Expressions

8.9. Summary

8.10. Model Questions

5.0 Introduction

An Operator is a symbol in C language that performs some mathematical, logical, etc. operations on some data. The data items on which operators act upon are called Operands. Some operators require two operands while others require only one operand. A few operators permit only single variable as operand. Operators form Expressions by joining individual operands which can be constants, variables, etc. Most operators allow the individual operands to be expressions.

C programming language provides several operators to perform different kind to operations. There are operators for assignment, arithmetic functions, logical functions and many more. These operators generally work on many types of variables or constants, though some are restricted to work on certain data types. Most operators are binary, meaning they take two operands. A few are unary and only take one operand. There is a single ternary operator which operates on three operands. Operators can be classified based on the type of operations they perform as follows:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Bitwise Operators
6. Miscellaneous Operators

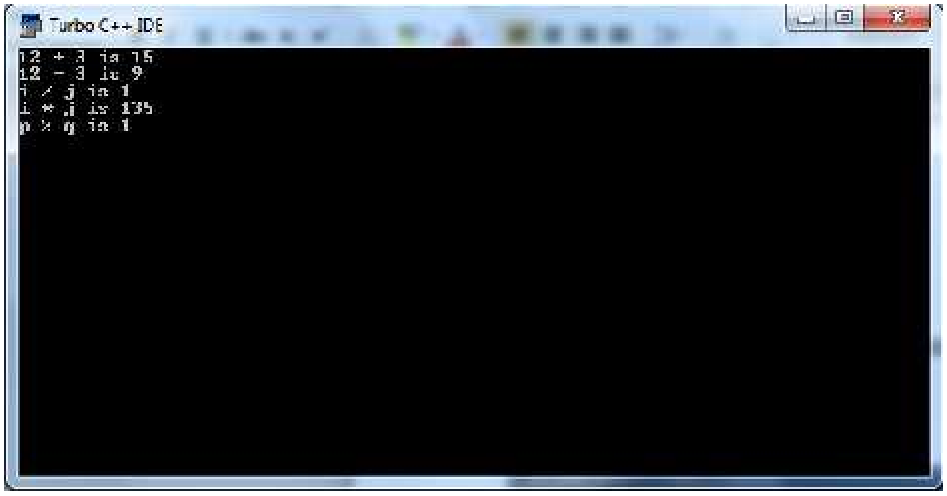
In this lesson, we will discuss all the above enumerated classes of operators available in C language. In addition, we will discuss the precedence and associativity of these operators during expression evaluation. Furthermore, data type conversions in expressions will be analyzed.

5.1 Arithmetic Operators

C language provides five main arithmetic operators which include the addition (+), subtraction (-), multiplication (*) and division (/) operator. These operators are used to perform arithmetic operations on numeric values. All these operators perform the function what is signified by their name. In addition there is a modulus operator (%) which gives the remainder left over from a division operation. For exponentiation there is no specific operator in C. Operands can be constants or variables containing integer quantities, floating-point quantities or characters. However, the modulus operator requires that both operands be integers and the second operand be non-zero. Similarly, the division operator (/) requires that the second operand be non-zero, though the operands need not be integers. Division of one integer quantity by another is referred to as integer division. With this division the decimal portion of the quotient will be dropped. An arithmetic operator along with its operand(s) gives rise to Arithmetic Expression. The output of an arithmetic expression is a numeric value decided by the nature of operation. Program 5.1 shows these operators in action which is followed by the screenshot of the output.

```
#include <stdio.h>

void main()
     {
     int i, j, p, q, r;

     i = 12 + 3;
     j = 12 – 3;
     p = i / j;
     q = i * j;
     r = p % q;

     printf(“12 + 3 is %d\n”,i);

     printf(“12 - 3 is %d\n”,j);

     printf(“i / j is %d\n”,p);

     printf(“i * j is %d\n”,q);

     printf(“p %% q is %d\n”,r);

     }
```

Program 5.1: Program shows binary arithmetic operators in action



```
12 + 3 is 15
12 - 3 is 9
i / j is 1
i * j is 135
p % q is 1
```

C also provides two unusual arithmetic operators called increment (++) and decrement (--) operators. The operator ++ adds 1 to the operand while –- subtracts 1. It should be noted that all arithmetic operators are binary operators, however these 2 operators (++ and --) are unary (Note that addition [+] and subtraction [-] are both binary as well as unary operators). This means, they only operate on one operand. However, the operand can only be a variable but no constant. To understand the working of these operators, let’s consider an arithmetic expression in which ++ operator is used on some variable x. As such, the value of x after operation will be 1 greater than that what was before the operation. Therefore, the two statements shown below will have same affect individually.

```
X ++;

X = X + 1;
```

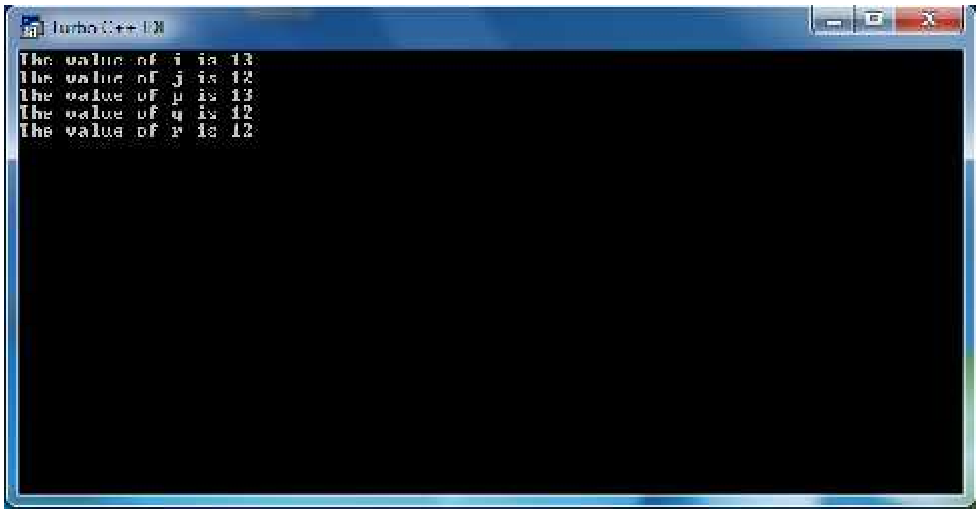
However, both unary operators take two forms as shown below.

```
X ++; or ++ X;

X --; or ––X;
```

Though ++X and X++ mean same thing, they behave differently when they are used in an expression wherein their value is further processed. Specifically, ++X means increment the value of X by 1 and then substitute the expression ++X by that value. In contrast, X++ means substitute the expression by the current value of X and then increment the value by 1. Similarly postfix and prefix decrement operators behave. Program 5.2 shows the difference in postfix and prefix operators followed by screenshot of the output.

```
#include <stdio.h>

void main()
    {
    int i, j, p, q, r;

    i = 12;
    j = i++; // Postfix Increment
    p = ++j; // Prefix Increment
    q = j--; // Postfix Decrement
    r = --q; // Prefix Decrement

    printf(“The value of i is %d\n”,i);
    printf(“The value of j is %d\n”,j);

    printf(“The value of p is %d\n”,p);

    printf(“The value of q is %d\n”,q);

    printf(“The value of r is %d\n”,r);

    }
```

Program 5.2: Program shows postfix and prefix operators in action



```
The value of i is 13
The value of j is 12
The value of p is 13
The value of q is 12
The value of r is 12
```

5.2 Relational Operators

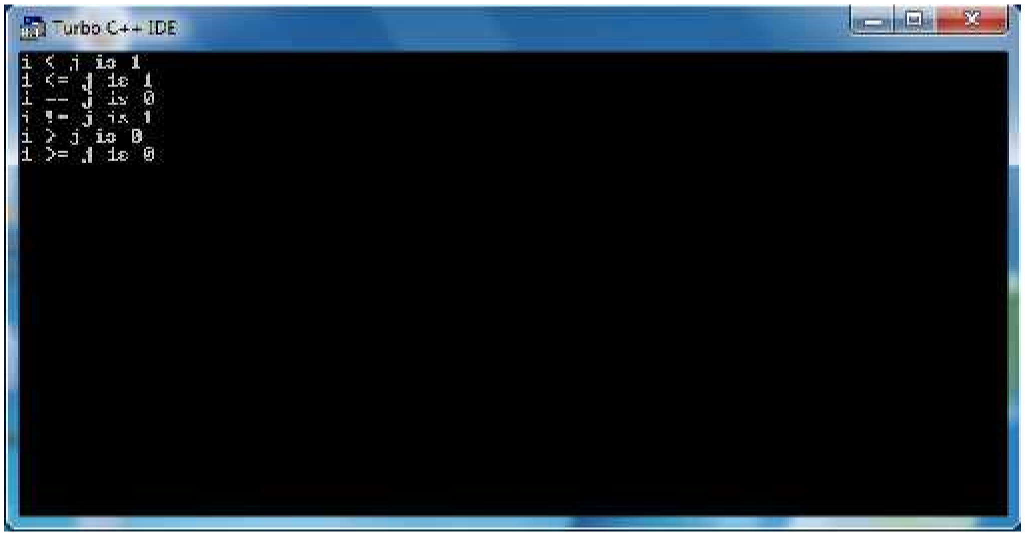
C language provides six relational operators which include the greater than (>), greater or equal (>=), lesser than (<), lesser or equal (<=), equal (==), and not equal (!=) operator. These operators are used to compare two numeric values and thus are binary operators. All these operators perform the function what is signified by their name. Operands can be constants or variables containing integer quantities, floating-point quantities or characters. In addition, the operands can be arithmetic expressions. A relational operator along with its operand(s) gives rise to Relational Expression. It is worth mentioning here that the equality operator is “==” and not “=”, which is an assignment operator. The output of a relational expression is a Boolean value (true=non-zero or false=0) decided by the nature of operation. Table 5.1 enumerates the working of each relational operator.

Operator	Example	Meaning
>	3 > 0	True; output 1
>=	3 >= 0	True; output 1
<	3 < 0	False; output 0
<=	3 <= 0	False; output 0
==	3 == 0	False; output 0
!=	3 != 0	True; output 1

Table 5.1: Usage and meaning of Relational operators

Program 5.3 shows these operators in action which is followed by the screenshot of the output.

```
#include <stdio.h>

void main()
    {
    int i, j, k;

    i = 10;
    j = 20;

    k = i < j;
    printf(“i < j is %d\n”,k);

    k = i <= j;
    printf(“i <= j is %d\n”,k);

    k = i == j;
    printf(“i == j is %d\n”,k);

    k = i != j;
    printf(“i != j is %d\n”,k);


    k = i > j;
    printf(“i > j is %d\n”,k);

    k = i >= j;
    printf(“i >= j is %d\n”,k);
    }
```
Program 5.3: Program shows binary relational operators in action



```
i < j is 1
i <= j is 1
i == j is 0
i != j is 1
i > j is 0
i >= j is 0
```

5.3 Logical Operators

C language provides three logical operators which include the logical AND (&&), logical OR (||) and logical NOT (!) operator. These operators are used to perform logical operations on two operands and thus are binary operators. However, logical NOT (!) has only one operand and hence is a unary operator. All these operators perform the function what is signified by their name. Operands can be constants or variables containing integer quantities, floating-point quantities or characters. In addition, the operands can be relational expressions and/or arithmetic expressions. A logical operator along with its operand(s) gives rise to Logical Expression. The output of a logical expression is a Boolean value (true=non-zero or false=0) decided by the nature of operation. Table 5.2 enumerates the working of each logical operator.

Operator	Example	Meaning
&&	3 && 0	Both operands are not non-zero i.e. true. Hence the output will be false i.e. 0.
||	Let x=1	One of the operand is true (3>x will be true), hence the output is true i.e. 1.

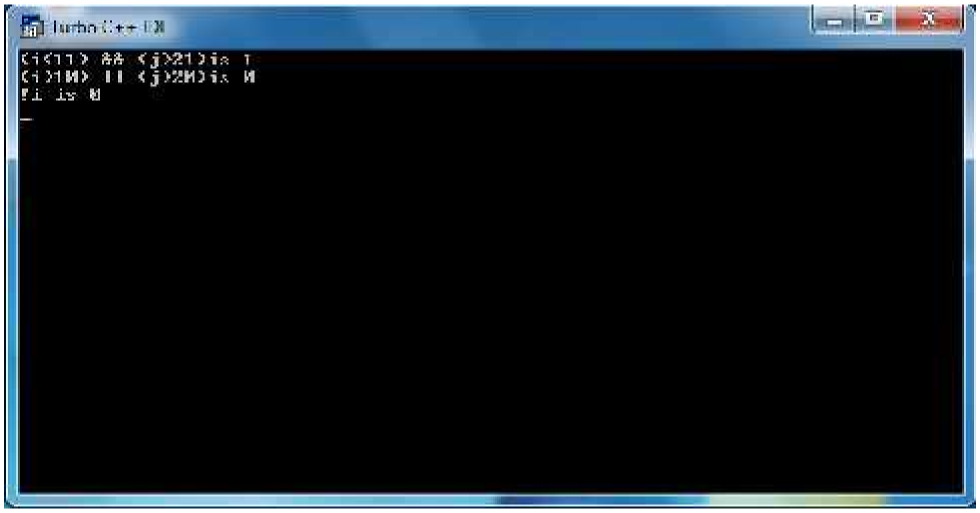
	(3>x) | y	
!	Let x=121 !(x < 1)	(x < 1) is false i.e. zero, and its negation will be true i.e. 1.

Table 5.2: Usage and meaning of Logical operators

Program 5.4 shows these operators in action which is followed by the screenshot of the output.

```
#include <stdio.h>

void main()
    {
    int i, j, k;

    i = 10;
    j = 20;

    k = (i < 11) && (j < 21);
    printf("(i<11) && (j<21)is %d\n",k);

    k = (i > 10) || (j > 20);
    printf("(i>10) || (j>20)is %d\n",k);

    k = !i;
    printf("!i is %d\n",k);

    }
```

Program 5.4: Program shows logical operators in action



```
(i<11) && (j>21)is 1
(i>10) || (j>20)is 0
!i is 0
_
```

5.4 Bitwise Operators

C language provides six bitwise operators which for manipulation of data at bit level. These operators are used to perform shifting, complementing, ANDing, ORing and so on, al at bit level. These include the bitwise AND (&), bitwise OR (|), bitwise exclusive OR (^), shift left (<<), shift right (>>) and one's complement (~) operator. All these operators perform the function what is signified by their name. All bitwise operators are binary operators except One's Complement (~) operator which is a unary operator. Operands can be constants or variables containing integer quantities or characters but not floating point. In addition, the operands can be logical expressions, relational expressions and/or arithmetic expressions. The output of an expression having bitwise operator is numeric. Table 5.3 enumerates the working of each bitwise operator.

Operator	Example	Meaning
&	3 & 2	Converts 3 and 2 to binary number, and then performs AND between corresponding bits. The resulting binary number is the outcome.
|	3 | 2	Converts 3 and 2 to binary number, and then performs OR between corresponding bits. The resulting binary number is the

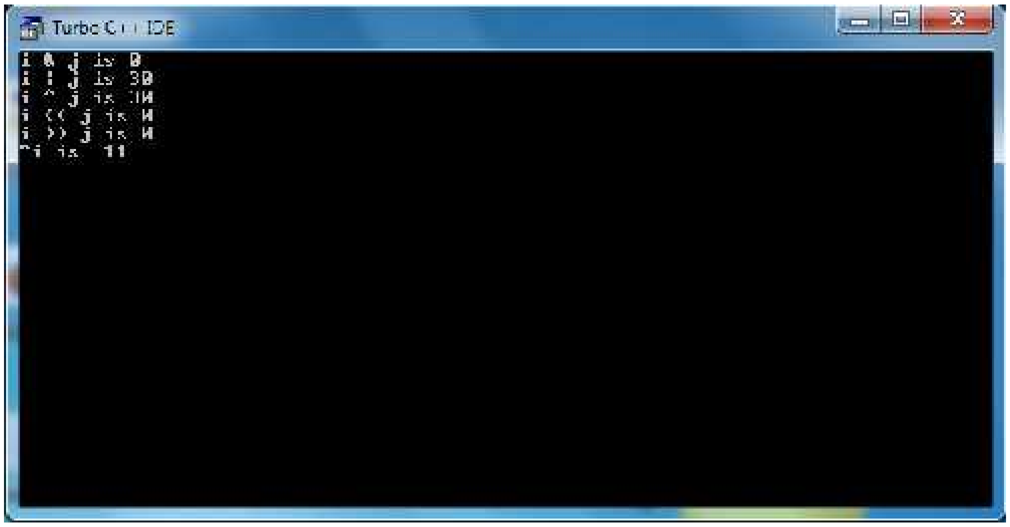
		outcome.
^	3 ^ 2	Converts 3 and 2 to binary number, and then performs Exclusive OR between corresponding bits. The resulting binary number is the outcome.
<<	3 << 2	Converts 3 to binary number and shifts the bits towards left 2 times. The resulting binary number is the outcome.
>>	3 >> 2	Converts 3 to binary number and shifts the bits towards right 2 times. The resulting binary number is the outcome.
~	~3	Converts 3 to binary number and complements each bit. The resulting binary number is the outcome.

Table 5.3: Usage and meaning of Bitwise operators

Program 5.5 shows these operators in action which is followed by the screenshot of the output.

```
#include <stdio.h>

void main()
    {
    int i, j, k;

    i = 10;
    j = 20;

    k = i & j;
    printf("i & j is %d\n",k);

    k = i | j;
    printf("i | j is %d\n",k);
    k = i ^ j;
    printf("i ^ j is %d\n",k);

    k = i << j;
    printf("i << j is %d\n",k);

    k = i >> j;
    printf("i >> j is %d\n",k);

    k = ~ i;
    printf("~i is %d\n",k);
    }
```

Program 5.5: Program shows bitwise operators in action



```
i & j is 0
i | j is 30
i ^ j is [illegible]
i << j is [illegible]
i >> j is [illegible]
~i is  11
```

5.5 Assignment Operators

C language provides one main assignment operator (=) which assigns to a variable on its LHS the outcome of some expression on its RHS. We have been using this operator since program 5.1. It is a binary operator. In addition, C language provides a set of shorthand assignment operators of the form

variable arithmetic/bitwise-Operator = expression;

These operators are very useful when in an expression a variable on LHS is repeated immediately on RHS. As an example, the statement below

```
x = x + 22;
```

can be written as

```
x += 22;
```

Table 5.4 lists all possible shorthand assignment operators along their equivalent expressions.

Shorthand Operator	Example	Equivalent expression
+=	x += 2;	x = x + 2;
-=	x -= 2;	x = x - 2;
*=	x *= 2;	x = x * 2;
/=	x /= 2;	x = x / 2;
%=	x %= 2;	x = x % 2;
<<=	x <<= 2;	x = x << 2;
>>=	x >>= 2;	x = x >> 2;
&=	x &= 2;	x = x & 2;
|=	x |= 2;	x = x | 2;
^=	x ^= 2;	x = x ^ 2;

Table 5.4: Usage and meaning of Shorthand operators

5.6 Miscellaneous Operators

C language some non-conventional operators like ternary conditional operator (?:), sizeof operator, comma operator (,), pointer operators (& and *) and member selection operators (. and ->). Here we will discuss first 3 operators while rest will be discussed later.

Conditional operator (?:) is ternary operator whose first operand is the conditional expression outcome of which will decide whether the second operand expression will be executed or the third operand expression. As an example, consider the following code snippet in which we want to assign value of x to a if and only x > y; otherwise we will assign it some other value.

```
x = 10;

y = 20;

a = ( x > y ) ? x : 0;

// This code will assign 0 to a
```

The sizeof() operator which return the number of bytes that are occupied by the operand. The operand can be a constant, data type or variable.

```
printf("The size of int = %d", sizeof(int));

// This code will display 2 on the screen
```

The comma operator (,) is used to link related expressions together. The expression so formed is evaluated from left-to-right and the value of right-most expression is the value of the combined expression.

```
x = 10;
y = 20;

z = (x = y + 1, y = x + 1, x + y);
//What is the value of z;
```

There are two more operators in C language; unary plus (+) and unary minus (-). Unary Plus is a unary operator which simply multiplies +1 to its operand which can be any expression. Same way unary minus multiplies -1 to its operand. These operators are used to explicitly assign some sign to the expression.

5.7 Operator Precedence and Associativity

Consider an expression below

```
z = 5 + 4 * 3;
```

What will be the value of z? 27 or 17? To avoid the ambiguity during the evaluation of expressions involving 2 or more operators in a C program, C language has given precedence to each and every operator supported by it. Therefore, in C language the operators at higher level of precedence are evaluated first. Table 5.5 shows the various precedence levels (Rank 1 being the highest level) along with the operators which fall under this level. Using this table we can solve the above mentioned problem. Because * has rank 3 and + has rank 4, so * has higher precedence than +. This means 4 * 3 will be evaluated first yielding 12 which will be added to 5 to yield 17. Hence, z will be assigned value 17.

However, certain operators have been assigned same level of precedence as shown in Table 5.5. Consider another expression below

```
z = 8 * 3 / 2;
```

What will be the value of z? 12 or 8? To solve this problem, to every precedence level associativity is assigned. The associativity of a precedence level signifies the order in which same precedence level operators will be evaluated. Means they will either be evaluated from left to right or from right to left. So in the problem mentioned above, though both * and / have same precedence level but they should be evaluated from left-to-right as per the table 5.5. This means we have to start from left of the expression and whichever operator is encountered first will be evaluated first. Hence, in this case * will be evaluated first which will yield 24 and this will be divided by 2 to yield 12. Hence, z will be assigned 12.

Rank	**Operator Type**	**Associativity**

1	`() []`
	member operators `(. ->)`
	`expr++ expr--`
		left-to-right

2	pointer operators `(* &)`	right-to-left

	unary plus (+) unary minus (-) ! ~ ++expr --expr (typecast) sizeof	
3	* / %	left-to-right
4	+ -	left-to-right
5	>> <<	left-to-right
6	< > <= >=	left-to-right
7	== !=	left-to-right
8	&	left-to-right
9	^	left-to-right
10	|	left-to-right
11	&&	left-to-right
12	||	left-to-right
13	Ternary Operator	right-to-left
14	Assignment Operators	right-to-left
15	Comma	left-to-right

Table 5.5: C Precedence and Associativity Chart

Table 5.6 lists some expressions and their evaluation in C programming language.

Expression	**C Evaluation**

`1+2*2`	`1+(2*2)`
`1+2*2*4`	`1+((2*2)*4)`
`(1+2)*2*4`	`((1+2)*2)*4`
`1+4,c=2|3+5`	`(1+4),(c=(2|(3+5)))`
`1 + 5&4 == 3`	`(1 + 5) & (4 == 3)`
`c=1,99`	`(c=1),99`
`!a++ + ~f`	`(!(a++)) + (~f)`
`s == ‘w’ || i < 9`	`(s == ‘w’) || (i < 9)`
`r = s == ‘w’`	`r = (s == ‘w’)`

Table 5.5: Expression evaluation in C

5.8 Type Conversions in Expressions

C expressions may contain many variables and constants which may be having different data types. However, when they are evaluated they should be raised or grounded to same common type for the sake of correct evaluation. Certain automatic type conversions are done by C compiler. In general, a narrower operand is converted into a wider one without losing information if expression involves a wider variable or constant. Consider the example below,

```
int x = 10;
float y = 5.5;

double z;

z = x + y;
```

In this example, integer variable x will be first converted to float (temporarily) and then the contents will added to real variable y. The result will be raised to double that will be assigned to z. However, x will still be an integer and y will still be a floating point. It should be noted that if z would have be integer type, then the result would have been grounded to integer.

Sometimes, due the complexity of an expression and nature of computation required, explicit type conversions are needed. Consider an example below

```
int x = 10, y = 3;
float z;

z = x / y;
```

Because both x and y are integers, the fractional part will be dropped during division. However, because z is floating point variable, the result (3) will be raised to float (3.000). In explicit type conversions, any variable or constant can be forced for a type conversion that is different from automatic conversion. In order to achieve this, C language provides type-casting operator which takes first argument as the data type to which the second argument should be type casted temporarily. The general syntax of type casting is as follows

(data-type) Expression;

So using explicit type casting, we can solve the above mentioned problem as shown below.

```
int x = 10, y = 3;
float z;

z = (float) x / y;
```

In this case, x is type-casted to float. Thus, to evaluate the expression y is also raised to float. Hence, the result is accurate and is assigned to z.

5.9 Summary

- An Operator is a symbol in C language that performs some mathematical, logical, etc. operations on some data.
- Operators form Expressions by joining individual operands which can be constants, variables, etc. Most operators allow the individual operands to be expressions.
- C programming language provides several operators to perform different kind to operations.
- Most operators are binary, meaning they take two operands. A few are unary and only take one operand. There is a single ternary operator which operates on three operands.
- C language provides five main arithmetic operators which include the addition (+), subtraction (-), multiplication (*), division (/) operator and modulus (%) operator.
- C language provides six relational operators which include the greater than (>), greater or equal (>=), lesser than (<), lesser or equal (<=), equal (==), and not equal (!=) operator.
- C language provides three logical operators which include the logical AND (&&), logical OR (||) and logical NOT (!) operator.
- C language provides six bitwise operators which for manipulation of data at bit level. These include the bitwise AND (&), bitwise OR (|), bitwise exclusive OR (^), shift left (<<), shift right (>>) and one's complement (~) operator.
- C language provides one main assignment operator (=) which assigns to a variable on its LHS the outcome of some expression on its RHS. In addition, C language provides a set of shorthand assignment operators.
- C language some non-conventional operators like ternary conditional operator (?:), sizeof operator, comma operator (,), pointer operators (& and *) and member selection operators (. and ->).
- There are two more operators in C language; unary plus (+) and unary minus (-).
- C language has given precedence to each and every operator supported by it so as to avoid the ambiguity during the evaluation of expressions. In C language the operators at higher level of precedence are evaluated first.
- Certain operators have been assigned same level of precedence. The associativity of a precedence level signifies the order in which same precedence level operators will be evaluated.

- Certain automatic type conversions are done by C compiler.
- Sometimes, due the complexity of an expression and nature of computation required, explicit type conversions are needed for which C language provides type-casting operator.

5.10 Model Questions

Q 37. Discuss the classification operators in C programming language.

Q 38. Explain various arithmetic operators in C programming language.

Q 39. Write a simple program to explain usage of arithmetic operators in C language.

Q 40. Explain various relational operators in C programming language.

Q 41. Write a simple program to explain usage of relational operators in C language.

Q 42. Explain various logical operators in C programming language.

Q 43. Write a simple program to explain usage of logical operators in C language.

Q 44. Explain various bitwise operators in C programming language.

Q 45. Write a simple program to explain usage of bitwise operators in C language.

Q 46. Explain various miscellaneous operators in C programming language.

Q 47. Write a simple program to explain usage of miscellaneous operators in C language.

Q 48. Write a short note on the following.

  a. Arithmetic Expression
  b. Relational Expression
  c. Logical Expression

Q 49. Differentiate between Unary and Binary Operators in C programming language.

Q 50. How is Unary plus and Unary minus different from Binary plus and Binary minus.

Q 51. What is the difference between postfix increment operator and prefix increment operator. Explain with the help of a program.

Q 52. What is the difference between postfix decrement operator and prefix increment operator. Explain with the help of a program.

Q 53. Explain the concept of Operator Precedence in C programming language.

Q 54. Explain the concept of Operator Associativity in C programming language.

Q 55. Why is type casting required? Explain the difference between automatic and explicit type casting in C programming language.

Unit II

CONTENTS

Lesson 6

Control Flow – Decision Making

Lesson 7

Control Flow – Looping

Lesson 8

Control Flow – Nesting

Lesson 9

Arrays

Lesson 10

Structures & Unions

Lesson 6

Control Flow – Decision Making

Structure

12.0. Introduction

12.1. Types of Flow Control Statements

12.2. if Statement

12.3. if…else Statement

12.4. if…else if Ladder

12.5. switch case Statement

12.6. Summary

12.7. Model Questions

6.0 Introduction

Control Flow (or flow of control) refers to the order in which the individual statements of a program are executed. In C language, statements are executed sequentially i.e. one after another. However, sometimes the nature of computation demands that the statements be executed out of order based on some condition. In our daily life we make such decisions every second. As an example, when a student leaves for the institute he makes a decision whether to carry an umbrella or not based on the weather conditions. So, if it is raining or is expected to rain, he may pick up one else he won't. Right now, while you are reading this text, based on your comprehension you will make a decision whether to read further or not. In both cases, you will make a decision and will do things out of sequence. Same is true with computer programs. This is very common in programs to execute certain statements based on some condition. As an example, suppose you are asked to write a simple program that will read a number from keyboard and will display on screen whether the number is even or odd. How will you do that? The answer is very simple. You will declare some variable as integer or float, call scanf() function to read from keyboard, use some logic to decide whether the number is even or odd. The logic to decide whether a number is even or odd is simple: Divide the number by 2 and if there is some remainder then it is odd else it is even. You can use modulus operator (%) to do that. Finally, you will make a decision on run time whether to display “The number is even.” or “The number is odd.”. But what should you use in your C program to make a decision? In lesson 5 we have discussed one such operator (?:). However, this operator is limited in its capability and thus, C language provides certain Decision Making Flow Control statements.

In this lesson, we will discuss types of flow control statements support by C language and will discuss all the decision making flow control statements provided by C programming language. The looping flow control statements will be discussed in next lesson.

6.1 Types of Flow Control Statements

In general, C language supports 2 types of flow control statements:

1. Decision Making
2. Looping

Decision Making statements are those statements which execute a set of other statements if and only some condition is true. In contrast, Looping statements execute a specific set of statements multiple times as long as some condition is true.

The Branching statements supported by C language include

1. if statement
2. if…else statement
3. if…else if ladder
4. switch case statement

The Looping statements supported by C language include

1. while statement
2. do while statement

3. for statement

6.2 if Statement

if statement is one of the basic and primitive decision making statement supported by C language. The general syntax of if statement is as follows:

```
if ( condition )
    statement;
```

This flow control construct first checks the condition and if it evaluates to True then executes the statement. The condition can be any expression that evaluates to True or False. It must be recalled that any non-zero numeric value in C language is treated as True while as zero is treated as False. This means, condition can be any of the following expressions:

1. Any constant,
2. Any variable, and
3. Any expression.

However, in general Relational expressions and Logical expressions are used to test any condition. Also, the statement can be a simple statement or a compound statement. Recall that a compound statement is a set of simple statements enclosed within a pair of curly braces. The following code snippets shows various legitimate if statements.

```

int x, y;

x = 10;

y = 20;

/* Using Constant values */
if ( 10 )
    printf(“I will be displayed\n”);

if ( 0 )
    printf(“I will NOT be displayed\n”);
```

```
/* Using Variables values */
if ( x )
    printf(“I will be displayed\n”);

/* Using Arithmetic Expressions */
if ( x + y )
    printf(“I will be displayed\n”);

/* Using Relational Expressions */
if ( x < y )
    printf(“I will be displayed\n”);

if ( x > y )
    printf(“I will NOT be displayed\n”);

/* Using Logical Expressions values */
if ( x + y > 10 && y > 10)
    printf(“I will be displayed\n”);

if ( x + y > 10 && 0 )
    printf(“I will NOT be displayed\n”);
```

It should be noted that no matter whether a condition evaluates to True or False, the rest of the statements which immediately follow the construct will be executed normally. The following code snippet explains this.

```
if (0 )
    printf(“I will NOT be displayed\n”);

printf(“I will be displayed\n”);
```

It is sometimes required that execution of multiple statements be controlled by the outcome of a condition. In that case, the statements are grouped into a single compound statement. The following code snippet explains how a compound statement can be used with if construct.

```
if ( 0 )
    {
    printf(“I will NOT be displayed\n”);
    printf(“Me too NOT displayed\n”);
    printf(“Same here, NOT displayed\n”);
    }

printf(“Oh! I will be displayed\n”);
```

6.3 if…else Statement

if…else statement is an extension of if statement supported by C language. The statement is a two-way decision making statement in which the condition is evaluated first and if it evaluates to true one set of statements is executed and other set is not. Conversely, if the condition evaluates to false then the other set will be executed but the first set won’t. The general syntax of if else statement is as follows:

```
if ( condition )
    statement1;

else
    statement2;
```

The condition can be any expression that evaluates to True or False. Also, statement1 and statement2, both can be a simple statement or compound statement. The following code snippet shows various legitimate if..else statements.

```
int x, y;

x = 10;

y = 20;

/* Using Constant values */
```

```
if ( 10 )
    printf(“I will be displayed\n”);
else
    printf(“I will NOT be displayed\n”);

if ( 0 )
    printf(“I will NOT be displayed\n”);
else
    printf(“I will be displayed\n”);

/* Using Variables values */
if ( x )
    printf(“I will be displayed\n”);
else
    printf(“I will NOT be displayed\n”);

/* Using Arithmetic Expressions */
if ( x + y )
    printf(“I will be displayed\n”);
else
    printf(“I will NOT be displayed\n”);

/* Using Relational Expressions */
if ( x < y )
    printf(“I will be displayed\n”);
else
    printf(“I will NOT be displayed\n”);

if ( x > y )
    printf(“I will NOT be displayed\n”);

    printf(“I will be displayed\n”);

/* Using Logical Expressions values */
if ( x + y > 10 && y > 10)
    printf(“I will be displayed\n”);
else
    printf(“I will NOT be displayed\n”);

if ( x + y > 10 && 0 )
    printf(“I will NOT be displayed\n”);
else
```

```
    printf(“I will be displayed\n”);
```

In addition, no matter whether a condition evaluates to True or False, the rest of the statements which immediately follow the construct will be executed normally. The following code snippet explains this.

```
if (0 )
    printf(“I will NOT be displayed\n”);
else
    printf(“I will be displayed\n”);

printf(“I will be displayed\n”);
```

The following code snippet explains how a compound statement can be used with if..else construct. However, it should be noted that it is not necessary that compound statement when used should be used with both if and else.

```
if ( 0 )
    {
    printf(“I will NOT be displayed\n”);

    printf(“Me too NOT displayed\n”);

    printf(“Same her, NOT displayed\n”);
    }
else
    printf(“I will be displayed\n”);

printf(“Oh! I will be displayed\n”);
```

Now, let us write a simple program to check whether a number is even or odd. Program 6.1 shows the code of program.

```
#include <stdio.h>

void main()
```

```
{
int number;

printf(“Please enter the number “);

scanf(“%d”, &number);

if (number % 2 == 0)
printf(“The number %d is Even.”, number);

else
printf(“The number %d is Odd.”, number);
}
```

Program 6.1: Program to check whether a number is even or odd

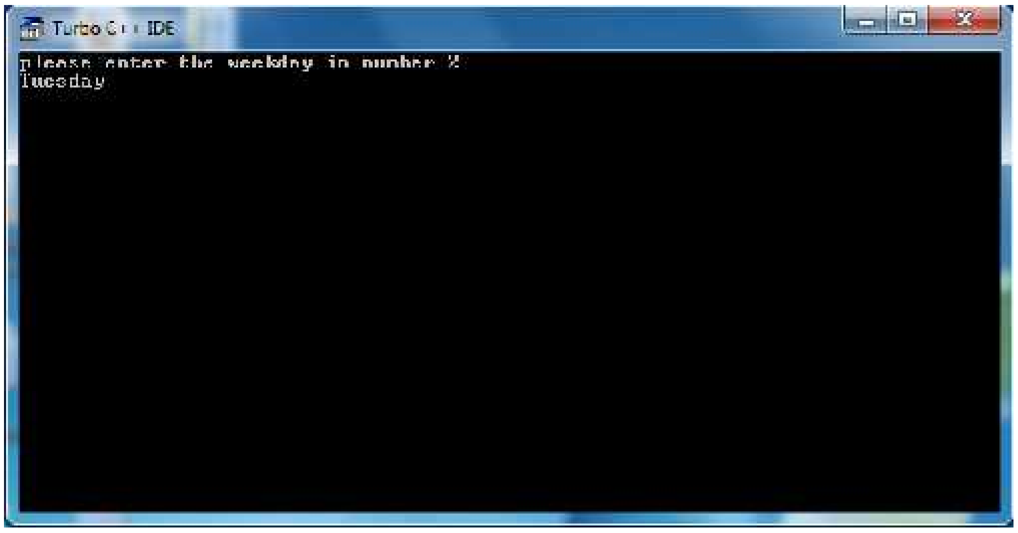
6.4 if…else if Ladder

if…else if ladder is another extension of if statement supported by C programming language. if statement checks a single condition and if True executes one single statement (simple or compound). if else statement also checks a single condition, however it executes one of the two statements (both of which can be simple or compound). Sometimes, it is required to check multiple conditions and accordingly decide the execution of a specific statement (simple or compound) among multiple possible statements (simple or compound).

One way to achieve this is to have multiple if statements. Consider a simple problem in which depending upon the value of some integer we have to display the day of week. The value can be entered by a user using a keyboard and we have to display the corresponding weekday. The logic of the program is very simple. We have to check whether the value of variable is 1 then display “Monday”, or if 2 then display “Tuesday”, and so on. The program 6.1 shows how this can be done using multiple if statements which followed by the output. The program assumes that the user will only enter number between 0 to 7.

```
#include <stdio.h>

void main()
     {
     int weekday;

     printf(“please enter the weekday in number “);

     scanf(“%d”, &weekday);

     if (weekday == 1)
          puts(“Monday”);

     if (weekday == 2)
          puts(“Tuesday”);

     if (weekday == 3)
          puts(“Wednesday”);

     if (weekday == 4)
          puts(“Thursday”);

     if (weekday == 5)
          puts(“Friday”);

     if (weekday == 6)
          puts(“Saturday”);

     if (weekday == 7)
          puts(“Sunday”);

     }
```

Program 6.2: Program that uses multiple if statements



```
please enter the weekday in number 2
Tuesday
```

This program will run fine. However, this program has an efficiency problem. If the user enters value 1 then as soon as "Monday" will be displayed, the next if statement will also checked; in this case weekday == 2. Indeed this condition will fail but it will consume CPU time. The worst part of the scenario is that all following conditions will be tested. To stop as soon checking other condition as soon as some condition is met, we can use if...else if statement. The general syntax of if..else if statement is as follows:

```
if ( condition1 )
    statement1;

else if ( condition2 )
    statement2;
.
.
.
else
    statementn;
```

The condition can be any expression that evaluates to True or False. Also, all statements can be a simple statement or compound statement. Furthermore, there is no limit on number of else if constructs that can be used. Also, else construct is not mandatory. Program 6.2 recodes the program 6.1 using else..if construct to achieve efficiency.

```
#include <stdio.h>

void main()
     {
     int weekday;

     printf(“please enter the weekday in number “);

     scanf(“%d”, &weekday);

     if (weekday == 1)
          puts(“Monday”);

     else if (weekday == 2)
          puts(“Tuesday”);

     else if (weekday == 3)
          puts(“Wednesday”);

     else if (weekday == 4)
          puts(“Thursday”);

     else if (weekday == 5)
          puts(“Friday”);

     else if (weekday == 6)
          puts(“Saturday”);

     else if (weekday == 7)
          puts(“Sunday”);

     }
```

Program 6.3: Program that uses else...if statements

Now, if the user enters value 1 then as soon as “Monday” will be displayed, the next and all subsequent else if statements will not be checked and control will come out of the construct. However, if the user enters value 2, then first two conditions will be checked and rest will be skipped. Though there will no difference in the output but this program will run faster as compared to other one. It should be noted that if else is to be used it should be the last statement.

6.5 switch case Statement

switch case statement is another control flow decision making statement supported by C programming language. The general syntax of switch case is as follows

```
switch ( expression )
     {
     case value1:
          statement1;
          break;

     case value2:
          statement2;
          break;

     .
     .
     .
     case valuen:
          statementn;
          break;

     default:
          statementx;
     }
```

The switch case construct first evaluates the expression and jumps to the case which has the same value to execute the statement. After the statement is executed, the break keyword transfers the control out from the construct. Note that if break keyword is not used the control will execute all the cases after matching ist choice.

It should be noted that expression can be any expression that evaluates to integer value. Therefore, the values associated with all cases should also be integer.

However, there may be a case wherein no case value matches to the expression outcome. In that case, the statements of default case will be executed and then control will be transferred out from the construct. It should be noted that default is optional and needs no break. In case, the outcome of expression matches no case value and there is no default case, the control will be directly transferred out from the construct without executing any statement.

One can visualize switch case as a faster version of if…else if construct. Recall the program 6.3. in which if the user enters value 2, then first two conditions will be checked and rest will be skipped. If the program is recoded using switch case then the control will directly jump to second case without evaluating first case. Program 6.4 shows how to recode it using switch case construct.

```
#include <stdio.h>

void main()
    {
    int weekday;

    printf(“please enter the weekday in number “);

    scanf(“%d”, &weekday);

    switch (weekday)
    {
    case 1:
        puts(“Monday”);

    case 2:
        puts(“Tuesday”);
    case 3:
        puts(“Wednesday”);

    case 4:
        puts(“Thursday”);

    case 5:
        puts(“Friday”);

    case 6:
        puts(“Saturday”);

    case 7:
        puts(“Sunday”);

    default:
        puts(“Wrong weekday”);
    }
```

Program 6.3: Program that uses switch case statement

6.6 Summary

- Control Flow (or flow of control) refers to the order in which the individual statements of a program are executed.
- In general, C language supports 2 types of flow control statements:
    - Decision Making
    - Looping
- Decision Making statements are those statements which execute a set of other statements if and only some condition is true.
- Looping statements execute a specific set of statements multiple times as long as some condition is true.
- The Branching statements supported by C language include
    - if statement

- if…else statement
- if…else if ladder
- switch case statement

6.7 Model Questions

Q 56. Discuss the need and significance of Flow Control statements.

Q 57. Classy the Flow Control statements in C programming language.

Q 58. Explain the if decision making flow control statements in C language.

Q 59. Explain the if..else decision making flow control statements in C language.

Q 60. Explain the if…else if decision making flow control statements in C language.

Q 61. Explain the switch case decision making flow control statements in C language.

Q 62. Write a program to check whether a number is even or odd using ternary operator provided by C programming language.

Q 63. Write a program to check whether a number is even or odd using switch case statement provided by C programming language.

Q 64. Write a program in C programming language to get a number as month and display the month name accordingly.

Q 65. Write a program in C programming language to get a number as weekday and display the day name accordingly.

Q 66. Write a program in C programming language to get a number as year and display whether the year is leap year or not.

Q 67. Write a program in C programming language to get a number as age and display whether the person is minor, young or adult.

Lesson 7

Control Flow - Looping

Structure

17.0. Introduction

17.1. while Statement

17.2. do while Statement

17.3. for Statement

17.4. break Statement

17.5. continue Statement

17.6. goto Statement

17.7. Summary

17.8. Model Questions

7.0 Introduction

Looping flow control statements in C programming language execute a specific set of statements multiple times as long as some condition is true. The Looping statements supported by C language include

1. while statement
2. do while statement
3. for statement

In a program written in any programming language it is common to repeat a specific set of statements multiple times. As an example, consider that we want to get a number from user, increment it by 1 and display it. We can accomplish this by using `scanf()` function that will populate some variable, increment it and then display it using `printf()`. Now, consider that we have to perform this task 10 times. What should we do? Till now the only way we can achieve this is to write these statements 10 times. Or another way can be to execute the program 10 times. Well 10 is a very small number as compared to 10,000 or so. You can argue that doing copy paste can solve this. Indeed, yes.

Let us modify the above program to display multiplication table of any number entered by user. In this case, we will get some number from user using scanf(), multiply it by 1, and display the result using printf(). Then, multiply the same number by 2 and display the result. The procedure needs to be repeated while the number is not multiplied by 10. You can see that we are repeating a specific set of statements 10 times. These statements include multiplying by an integer and displaying the result. This task can be achieved easily and efficiently by using Looping statements in C language.

In this lesson, we will discuss all the looping flow control statements available in C language. In addition, we will discuss the goto, break and continue statements.

7.1 while Statement

The while-loop is an entry-controlled loop in which first the condition is checked, and if it is True then the statements are executed. However, if the condition is False, the control is transferred to the statement immediately after the body of loop. The condition can be any constant, variable or expression that evaluates to True or False.

However, the general syntax of while-loop requires a counter variable to be initialized before while statement. Then, within the condition, this counter variable needs to be checked against some value. If the condition evaluates to True, the statements within the body of loop will be executed. After this, the control is transferred back to condition. If this time the condition is again True, then the body of loop will be executed same way and control will be transferred back to the condition. However, if the condition is False, then the control will be transferred to the statement immediately following the loop body. The general flow of an entry-controlled loop is shown in figure 7.1.

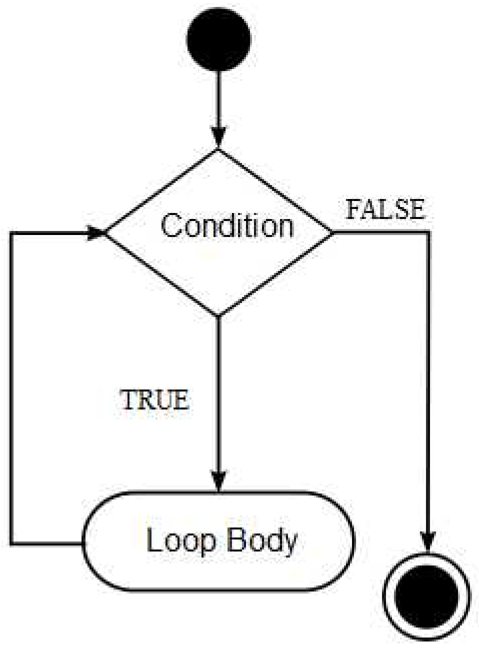


Figure 7.1: General flow of an entry-controlled loop

This means the while construct has a condition statement that decides whether the statements of loop body will be executed or not. Also, it has a counter variable to keep the count of number of times the statements within body of loop are executed. However, this counter variable needs to be modified during iteration within the body of loop. The counter modification statement should be the last statement of the body of the loop. The general syntax of while looping statement is as follows

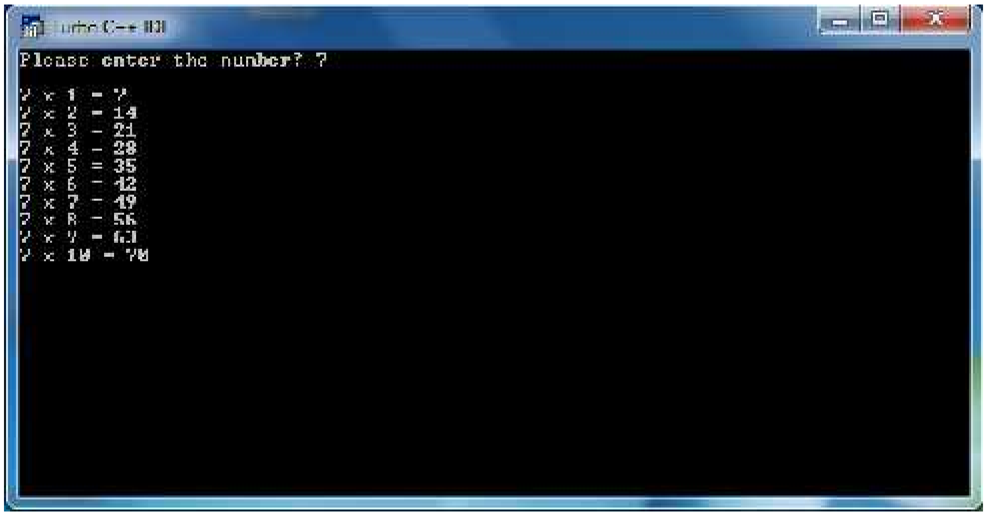
```
counter-initialisation;

while ( condition )
     {

     statements;

     counter-modification;
     }
```

It should be noted that if the initialization of counter variable is skipped, then there will be no syntax error but a logical error. This is because of the fact that un-initialized variables contain bogus value which will be checked against some value in condition. Depending upon bogus value, the condition will evaluate to True or False.

It is worth mentioning here that if the statement to modify counter variable is skipped, there will be no syntax error. However, this may result into an infinite loop. An infinite loop is a loop in which statements are executed infinite number of times. This is true for all types of loops. Program 7.1 shows how to display a multiplication table of any number entered by a user using while loop. It is followed by the output of the program.

```
#include <stdio.h>

void main()
    {

    int number, count, mul;

    printf(“Please enter the number? ”);

    scanf(“%d”, &number);

    count = 1;

    while ( count <= 10 )
        {

        mul = number * count;

        printf(“\n%d x %d = %d”,
                number, count, mul);

        count ++;
        }
    }
```

Program 7.1: Program to display multiplication table of a number using while loop.



```
Please enter the number? 7

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

7.2 do while Statement

The do-while loop is an exit-controlled loop in which first the body of a loop is executed and then the condition is checked. Now, if the condition is true then the statements are executed again. However, if the condition is False, the control is transferred to statement immediately after the body of loop. The condition can be any constant, variable or expression that evaluates to True or False.

The general syntax of do-while requires a counter variable to be initialized before do-while statement. Then within the condition, this counter variable needs to be checked against some value. The general flow of an exit-controlled loop is shown in figure 7.2.

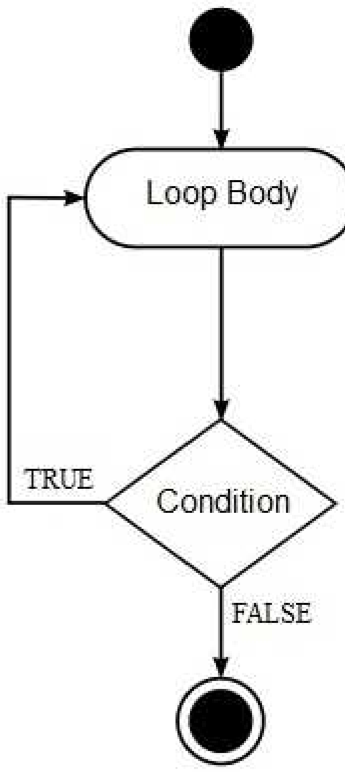


Figure 7.2: General flow of an exit-controlled loop

This means the do-while construct has a condition part that decides whether the statements will be executed or not. Also, it has a counter variable to keep the count of number of times the statements are executed. However, this counter variable needs to be modified during iteration within the body of loop. The counter modification statement should be last statement of the body of the loop. The general syntax of while looping statement is as follows:

```
counter-initialisation;

do
    {

    statements;

    counter-modification;
    }
```

```
while ( condition );
```

One may argue why we should use such kind of a loop. The answer is simple. There are certain situations wherein we need it. For example, we want to get a number from user that is greater than 10. And if the number is less than or equal to 10, we should again ask for the number. The code snippet below shows how this can be done using a while loop.

```
    int num;

    printf(“Enter number? ”);
    scanf(“%d”, &num);

    while( num <= 10 )
    {
    printf(“Enter number? ”);
    scanf(“%d”, &num);
    }
```

In contrast, we can do this with lesser number of statements using do-while loop as shown in code snippet below.

```
    int num;

    do
    {
    printf(“Enter number? ”);
    scanf(“%d”, &num);
     }
    while( num <= 10 )
```

It should be noted that the body of exit-controlled loops is executed at-least once. Hence, even if the condition is False in first iteration, the body of loop will be still executed at least once. In contrast, in entry controlled loops the body in every iteration will only be executed if condition is True. The code snippet below explains this clearly.

```
int x = 1;

do
    {

    printf("I will be printed");

    }

while ( x < 0);
```

7.3 for Statement

The for loop is an entry-controlled loop (like while) in which first the condition is checked, and if it is True then the statements are executed. However, if the condition is False, the control will be transferred to the statement immediately after the body of loop. The condition can be any constant, variable or expression that evaluates to True or False.

The general syntax of for loop requires a counter variable to be initialized within the for statement. Then within the condition, this counter variable needs to be checked against some value. If the condition evaluates to True, the statements within the body of loop will be executed. After this, the control is transferred to counter-modification part of the for loop, from where the control is transferred back to condition. If this time the condition is again True, the body of loop will be executed same way and control will be transferred back to counter-modification and then to the condition. However, if the condition is False, the control will be transferred to the statement immediately following the loop body.

```
for (counter-initialisation;
    condition;
    counter-modification;
    )
    {

    statements;

    }
```

One may argue that if we have one entry controlled loop (while) then what for we need another (for)? There are many explanations for existence of for in C language.

1. Some argue that for is faster than while.
2. Most claim that for is used when we know in advance the number of iterations that need to be performed.
3. It is very common that initialization and modification of counter variable is missed by a programmer in while loop. In lieu of this observation, people believe that for-loop is more programmer friendly as compared to while-loop.

We believe first and last argument is acceptable.

It should be noted that if the initialization of counter variable is skipped, then there will be no syntax error but a logical error. Also, if the statement to modify counter variable is skipped, there will be no syntax error. Program 7.2 shows how to display a multiplication table of any number entered by a user using for loop.

```
#include <stdio.h>

void main()
    {

    int number, count, mul;

    printf(“Please enter the number? ”);
    scanf(“%d”, &number);

    for (count = 1; count <= 10; count ++)
        {

        mul = number * count;

        printf(“\n%d x %d = %d”,
                number, count, mul);

        }
    }
```

Program 7.2: Program to display multiplication table of a number using for loop.

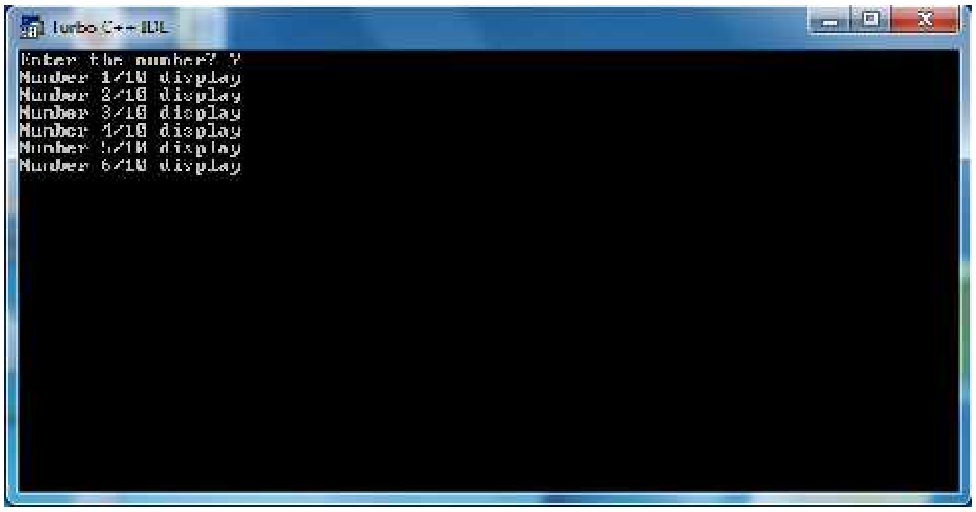
This program will produce same output as that of program 7.1.

7.4 break Statement

The break statement is used to break the execution of body of loop. When this statement is encountered, the control is directly transferred to the statement immediately after the loop body. break is also used in switch-case statement to transfer the control of execution from the statements of some case to the statement immediately after the switch-case construct. Program 7.3 shows the working of break statement followed by the output.

```
#include <stdio.h>

void main()
    {

    int num, count;

    printf(“Enter the number? ”);

    scanf(“%d”, &num);

    for( count=1; count < 10; count++)
        {

        if (num == count)
            break;

        printf(“Number %d/10 display\n”,
                count);

        }
    }
```

Program 7.3: Program to show the working of break statement.



```
Turbo C++ IDE
Enter the number? 7
Number 1/10 display
Number 2/10 display
Number 3/10 display
Number 4/10 display
Number 5/10 display
Number 6/10 display
```

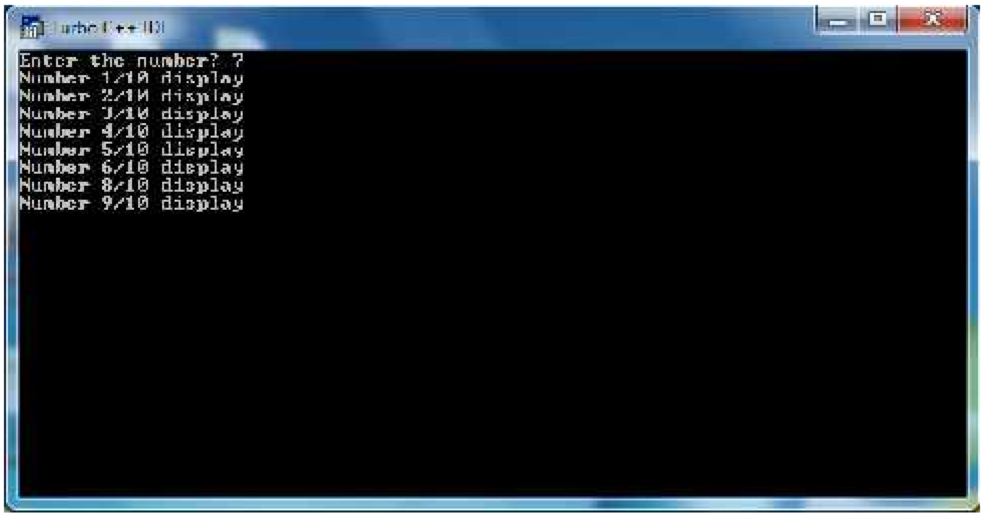
7.5 continue Statement

The continue statement when encountered, transfers the control to the condition of the while loop or counter-modification statement of for loop. This means all the statements of loop body after the continue statement are skipped and the control is directly transferred to the condition of the while loop or counter-modification statement of for loop. Program 7.4 shows the working of continue statement followed by the output.

```
#include <stdio.h>

void main()
```

```
   {

   int num, count;

   printf(“Enter the number? ”);

   scanf(“%d”, &num);

   for( count=1; count < 10; count++)
        {

        if (num == count)
             continue;

        printf(“Number %d/10 display\n”,
                count);

        }
 }
```

Program 7.4: Program to show the working of continue statement.



```
Enter the number? 7
Number 1/10 display
Number 2/10 display
Number 3/10 display
Number 4/10 display
Number 5/10 display
Number 6/10 display
Number 8/10 display
Number 9/10 display
```

7.6 goto Statement

The goto statement is used to unconditionally transfer the control to any location within the program whose address in the form of label follows the statement. It should be noted that break and continue statements transfer control to a fixed location; after loop body in case of break and to the condition of the while loop or counter-modification statement of for loop in case of continue. However, goto can transfer control to any location. Program 7.4 shows how to display a multiplication table of any number entered by a user using goto statement.

```
#include <stdio.h>

void main()
    {

    int number, count, mul;

    printf(“Please enter the number? ”);

    scanf(“%d”, &number);

    count = 1;

    myLoop: /* Label for if statement*/
    if (count <= 10)
        {

        mul = number * count;

        printf(“\n%d x %d = %d”,
                number, count, mul);

        count ++;

        goto myLoop;
        }
    }
```

Program 7.2: Program to display multiplication table of a number using goto statement.

This program will produce same output as that of program 7.1.

7.7 Summary

- Looping flow control statements in C programming language execute a specific set of statements multiple times as long as some condition is true.
- The Looping statements supported by C language include
  - while statement
  - do while statement
  - for statement
- The while-loop is an entry-controlled loop in which first the condition is evaluated, and if it evaluates to True then the statements are executed.
- The general syntax of while-loop and do-while requires a counter variable to be initialized before while statement, which needs to tested in condition and modified every time within the body of loop.
- An infinite loop is a loop in which statements are executed infinite number of times.
- The do-while loop is an exit-controlled loop in which first the body of a loop is executed and then the condition is evaluated.
- It should be noted that the body of exit-controlled loops is executed at-least once.
- The for loop is an entry-controlled loop (like while) in which first the condition is evaluated, and if it evaluates to True then the statements are executed.
- The general syntax of for loop requires a counter variable to be initialized within for statement.
- If the initialization of counter variable is skipped, then there will be no syntax error but a logical error.
- There are certain situations where we need exit controlled loops.
- In all loops, if the condition evaluates to False, the control is transferred to the statement immediately after the body of loop.
- The break statement is used to break the execution of body of loop and transfer the control directly to the statement immediately after the loop body.
- The continue statement when encountered, transfers the control to the condition of the while loop or counter-modification statement of for loop.

7.8 Model Questions

Q 68. Explain the working of while loop with the help of a program.

Q 69. Explain the working of do-while loop with the help of a program.

Q 70. Explain the working of for loop with the help of a program.

Q 71. Compare entry and exit controlled loops available in C language.

Q 72. How is for-loop similar and different than while-loop available in C language?

Q 73. What is the significance of do-while loop in C programming language?

Q 74. Explain the working of break statement with the help of a program.

Q 75. Explain the working of continue statement with the help of a program.

Q 76. Explain the working of goto statement with the help of a program.

Q 77. Write a program to add first 10 natural numbers using while loop.

Q 78. Write a program to calculate factorial of any number using for loop.

Q 79. Write a program to check whether a number is prime or not using a loop.

Q 80. With the help of a program show how goto can be used to implement a loop.

Q 81. Write a program in C language to display first 100 even numbers using loop.

Lesson 8

Control Flow - Nesting

Structure

23.0. Introduction

23.1. Nesting of Decision Making Statements

23.2. Nesting of Looping Statements

23.3. Breaking Nested Loops

23.4. Some Examples

23.5. Summary

23.6. Model Questions

8.0 Introduction

There may be a situation when you want to check for another condition within the body if or if-else statement. Also, it may be required to execute a loop inside the body of another loop. Placing one construct inside the body of another construct of same type is called Nesting.

In this lesson, we will discuss Nesting of control flow statements in C programming language.

8.1 Nesting of Decision Making Statements

Nesting if statement means placing if statement inside another if statement. The general syntax of nesting if statement is as follows:

```
if ( condition1 )
     {

     statement1;

     if ( conditionA )
```

```
          {
          statementA;
          }

     statement2;
     }
```

This means if condition1 evaluates to True, then statement1 will be executed. After this, conditionA will be checked, if it also evaluates to True then statementA will be executed, else not. However, in both cases statement2 will be executed.

It should be noted that same syntax applies to nesting of if-else construct. In fact, within the body of if statement or else part of if-else statement, we can have nested if statement or if-else statement as follows.

```
if ( condition1 )
     {
     statement1;

     if ( conditionA )
          {
          statementA;
          }
     else
          {
          statementB;
          }

     statement2;
     }
```

This means any variant of if statement can be nested inside another variant of if statement. In fact, the else-if ladder is actually nested if-else statement. Consider the code snippet below.

```
int x;
```

```
if ( x == 1 )
    printf(“First one”);

else if ( x == 2 )
    printf(“Second one”);

else if ( x == 3 )
    printf(“Third one”);

else
    printf(“Last one”);;

printf(“Done”);
```

In this code snippet we are using else-if ladder. Let us assume that x=1. Then, as the first condition evaluates to True, “First one” will be displayed. Now, no other condition will be checked and control will be transferred to “Done”. Now, consider that x=2, then first condition fails and second condition will be checked. Thus, this time “Second one” will be displayed as x==2 evaluates to True. Now, no more checks will be made and control will be transferred to “Done”.

Ok. This code snippet can re-arranged as follows.

```
int x;

if ( x == 1 )
    printf(“First one”);
else
    if ( x == 2 )
        printf(“Second one”);
    else
       if ( x == 3 )
           printf(“Third one”);
        else
           printf(“Last one”);;

printf(“Done”);
```

Yes. Actually, “Last one” and “Third one” are part of if-else statement which is nested inside the else part of another if-else statement “Second one”. And, this “Second one” if-else statement is actually nested into the else part of “First one” if-else statement. This is the only reason why all conditions before the True condition in else-if ladder are evaluated and all conditions after the True condition are never evaluated.

It is worth mentioning here that logically there is no limit on the level of nesting carried. However, if nesting is carried out to too deep a level and indenting is not consistent then deeply nested if or if-else statements can be confusing to read and interpret. Consider the code snippet below which is actually the above code without indentation.

```
int x;

if ( x == 1 )
    printf(“First one”);
else

if ( x == 2 )
    printf(“Second one”);
else

if ( x == 3 )
    printf(“Third one”);
else
    printf(“Last one”);;

printf(“Done”);
```

At first glance, it is very difficult to point out that this is nested if-else. Second, it is difficult to answer which construct is nested inside whom. One simple rule can solve this. It is important to note that an else always belongs to the closest if without an else.

As per this rule,

1. the “Last one” else belongs to “Third one” if,
2. this construct belongs is a statement that belongs to else above it.
3. this else belongs to “Second one” if statement.
4. this construct belongs is a statement that belongs to else above it.
5. this else belongs to “First one” if statement.

8.2 Nesting of Looping Statements

Nesting loops means placing one loop inside the body of another loop. Any variant of loop can be nested inside any other variant of loop just like any variant of if-statement can be nested inside any other variant of if-statement. Also, logically there is not limit on the level of nesting. However, to make code readable and easily understood, proper indentation should be done. While all types of loops may be nested, the most commonly nested loops are for loops. So, let us consider a for loop for explanation of nested loops.

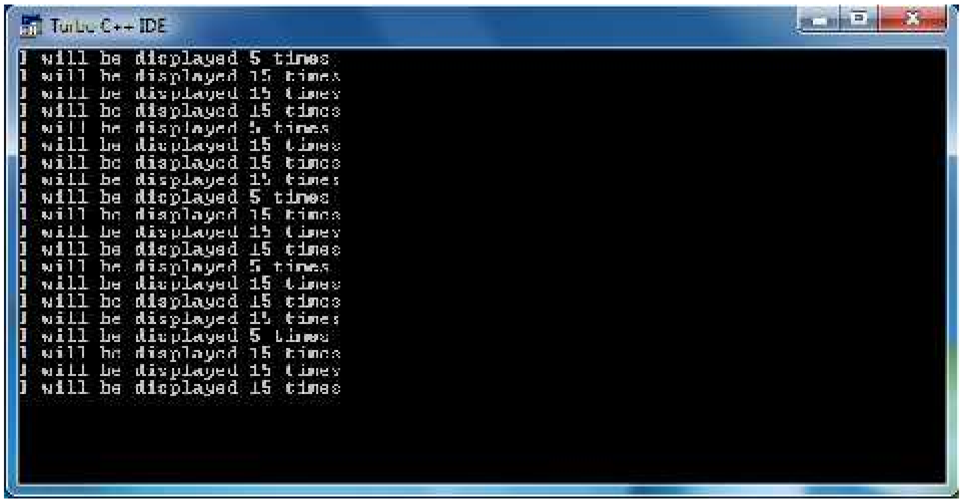
The general syntax of nested loops is as follows

```
for ( ; condition1 ; ) /* Outer Loop */
    {

    statement1;

    for ( ; conditionA ; ) /* Inner Loop */
        {
        statementA;
        }

    statement2;
    }
```

It is obvious from the syntax that during the first iteration of the outer loop, the inner loop gets trigger and executes to completion. Then, during the second iteration of the outer loop, the inner loop is again trigger and executed to its completion. This repeats until the outer loop finishes. This means the outer loop changes only after the inner loop is completely finished and the outer loop takes control of the number of complete executions of the inner loop. The program 8.1 explains the concept. In this program, the outer loop is executed 5 times while during every iteration of outer loop, the inner loop is also executed 3 times. This means, in total the inner loop will be executed 15 times.

```
#include <stdio.h>

void main()
    {

    int i, j;

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

        printf("I will be displayed 5
               times\n");

        for (j=1; j <=3; j++)
            {

            printf("I will be displayed
                   15 times\n");

            }
        }
    }
```

Program 8.1: Program to show working of nested loops



```
J will be displayed 5 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 5 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 5 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 5 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 5 times
J will be displayed 15 times
J will be displayed 15 times
J will be displayed 15 times
```

8.3 Breaking Nested Loops

It is sometimes required to break from the inner loop of nested loops. In this a break statement can be used. However, the break statement will only exit from the nested inner loop in which it is encountered and control will be transferred to outer loop. In case we want to transfer control to some statement after the outer most nested loop, we can use goto statement. The code snippet below shows how to do that.

```
for (i=1; i<=10; i++)
    {
    for (j=1; j <=10; j++)
        {
        if ( i==j )
            goto outside;
        }
    }
outside:
```

This code snippet transfers the control to the statement after the outermost loop when inside the innermost loop the value of i and j are same.

8.4 Some Examples

Program 8.2 shows how to create multiplication table of first 10 natural numbers using nested loops.

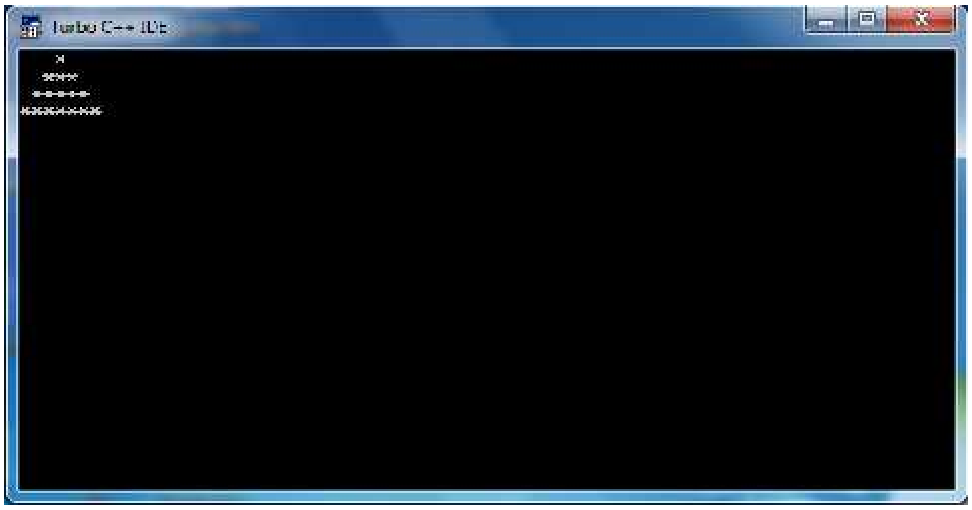
```
#include <stdio.h>

void main()
     {

     int i, j;

     for (i=1; i<=10; i++)
         for (j=1; j <=10; j++)
             printf(“%d x %d = %d \n”,
                     i, j, i*j);
     }
```

Program 8.2: Program to display multiplication table of first 10 natural numbers.

Program 8.3 shows how to print out pyramid of stars followed by screenshot of output.

```
#include <stiod.h>

void main()
{
int i, row, height=4;

for (row = 1; row <= height; row++)
     {

     for (i = 1; i <= height - row; i++)
         printf(" ");

     for (i = 1; i <= row * 2 - 1; i++)
         printf("*");

     printf("\n");
     }

}
```

Program 8.3: Program to display pyramid of stars



```
   *
  ***
 *****
*******
```

8.5 Summary

- Placing one construct inside the body of another construct of same type is called Nesting.
- Nesting if statement means placing if statement inside another if statement within a program written in C language.
- Any variant of if statement can be nested inside another variant of if statement within a program written in C language.
- The else-if ladder in C language is actually nested if-else statement.
- Nesting loops means placing one loop inside the body of another loop.
- Any variant of loop can be nested inside any other variant of loop.
- The outer loop changes only after the inner loop is completely finished.
- The outer loop takes control of the number of complete executions of the inner loop.
- Logically there is no limit on the level of nesting carried. However, if nesting is carried out to too deep a level, code should be properly indented.

8.6 Model Questions

Q 82. Explain nesting in decision making statements in C programming language.

Q 83. Write a program to explain nested-if statement in C language.

Q 84. Explain how else-if ladder is actually nested if-else statement.

Q 85. Explain nested loops in C language.

Q 86. Using a program show how outer loop control the number of times the inner loop is executed to its completion.

Q 87. Why break is not enough in nested loops? What is the remedy?

Q 88. Write a program to create pyramid of stars of height 8 using nested loops.

Lesson 9

Arrays

Structure

30.0. Introduction

30.1. Concept of Array

30.2. One Dimensional Array

30.3. Two Dimensional Array

30.4. Multi-Dimensional Array

30.5. Summary

30.6. Model Questions

9.0 Introduction

Imagine that you have to store and later retrieve the unique ID of all of the registered voters in your city. Yes, you can create and name hundreds of thousands of distinct variable names to store the information. However, that would scatter hundreds of thousands of names all over memory. In addition, there is a good probability that same operation needs to be performed on every variable. Let us assume that these variables are all populated with UID and we need to display them. In C language, we have to use `printf()` statement on every individual variable. Considering 100 voters and variables names as number0, number1, number2…number99 to store 100 UIDs, then the code to display them will be as follows:

```
printf(“Voter 1 has UID %d”, number0);
printf(“Voter 2 has UID %d”, number1);

printf(“Voter 3 has UID %d”, number2);
.
.
printf(“Voter 100 has UID %d”, number99);
```

It is obvious that we are simply performing same operation on a set of different data. This means it could be performed more efficiently by using a loop and an Array. Instead of declaring individual variables, such as number0, number1,...number99, we can declare one array variable such as numbers and use numbers[0], numbers[1],…numbers[99] to represent individual variables. So, using arrays the program can rewritten as follows:

```
int i;
for (i=0; i< 100; i++)
    printf(“Voter %d has UID %d”,
           i+1, number[i]);
```

An array has the property that items stored in it are stored contiguously and can be accessed randomly (by position or index). They are simple, fast and can be used as the basis for more advanced data structures. Almost all programming languages support the concept of Arrays.

In this lesson, we will discuss the concept of arrays in C programming language and will look at the various types of arrays supported by C language. In each type, we will discuss the declaration, initialization and accessing arrays.

9.1 Concept of Array

In C programming language, a derived data type which can store a collection of elements of the same data type is possible. Such a derived data type is called an Array. The data type of Array elements can be either primitive or user-defined. An array in C programing language can be defined as a name given to number of consecutive memory locations, each of which can store the data of same data type. Thus, one can visualize an array as a collection of variables of the same data type. All the individual variables (also called elements) of Array can be referenced through the same Array name.

However to make distinction between individual elements an index is used to refer to a particular element in an array.

The general syntax for declaring an array is as follows:

```
<data-type> <Array-name> [<Element-Count>];
```

This means to declare an array of integers with 20 elements the valid C statements will be

```
int myArrayName [20];
```

where myArrayName is the name of the Array. It should be noted that this will declare an integer array of size 20. Because the elements of an Array are placed at consecutive memory locations, every element can be uniquely accessed using the Array name and its index/location within the Array. The general syntax to name any array element is as follows:

```
<Array-name> [<index>]
```

where index can have any value between 0 to ArraySize -1. It should be noted that, the lowest address corresponds to the first element and the highest address to the last element. This means the first element of an integer array declared previously can be accessed by myArrayName[0] and last element by myArrayName[19]. All other elements will have index between 0 and 19. After naming an array element, it can be used in any C expression wherein any other variable of same data type can be used.

An array is a collective name given to a group of similar quantities. These similar quantities could be percentage marks of 100 students, number of chairs in home, or salaries of 300 employees or ages of 25 students. Thus an array is a collection of similar elements. These similar elements could be all integers or all floats or all characters etc. but mixture of different data types. All elements of any given array must be of the same type i.e we can’t have an array of 10 numbers, of which 5 are ints and 5 are floats. Usually, the array of characters is called a “string”.

Arrays are of two types; one dimension array and multi-dimension array.

9.2 One Dimensional Array

The array which is used to represent and store data in a linear form is called as Single or One Dimensional Array. To declare One Dimensional Array in C, the general syntax is as follows:

```
<data-type> <Array-name> [<Array-Size>];
```

The Array-Size must be an integer constant greater than zero and data-type can be any valid C data type; primary or user-defined. For example, to declare a 3 element one dimensional array called StudentsMarks of type double, we can use following statement:

```
double StudentMarks [3];
```

In order to initialize this array, we have two ways:

1. Initialize all elements at a time during declaration,
2. Initialize elements individually after declaration

The first method can be used to declare and assign values to all elements of 1D array as follows:

```
double StudentMarks [3] = {1.1, 2.2, 3.3};
```

This means element StudentMarks[0], which is the first element will be assigned value 1.1, StudentMarks[1] will be assigned value 2.2 and last element StudentMarks[2] will be assigned value 3.3. It should be noted that the number of values between braces { } can't be larger than the number of elements that we declare for the array between square brackets [ ]. This means the following statement is invalid.

```
double StudentMarks [2] = {1.1, 2.2, 3.3};
```

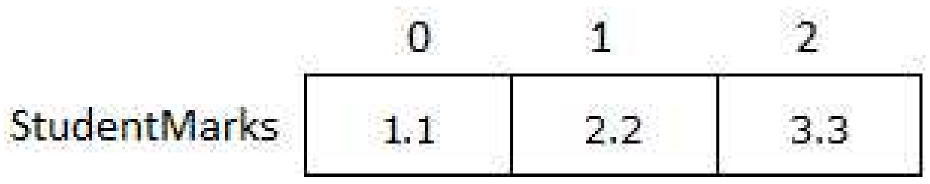
However, if we omit the size of the array during declaration, an array just big enough to hold the values of initialization, is declared. Therefore, the following statement has same effect as the first valid statement above has.

```
double StudentMarks [] = {1.1, 2.2, 3.3};
```

The second method is to assign values to every element individually. Recall that to access any individual array element, the array name along with index within square braces is used, and it can be used the same way any other variable of same data type can be. Therefore, after declaration of StudentMarks array, we can initialize it as follows:

```
double StudentMarks [3];

StudentMarks[0] = 1.1;
StudentMarks[1] = 2.2;
StudentMarks[2] = 3.3;
```

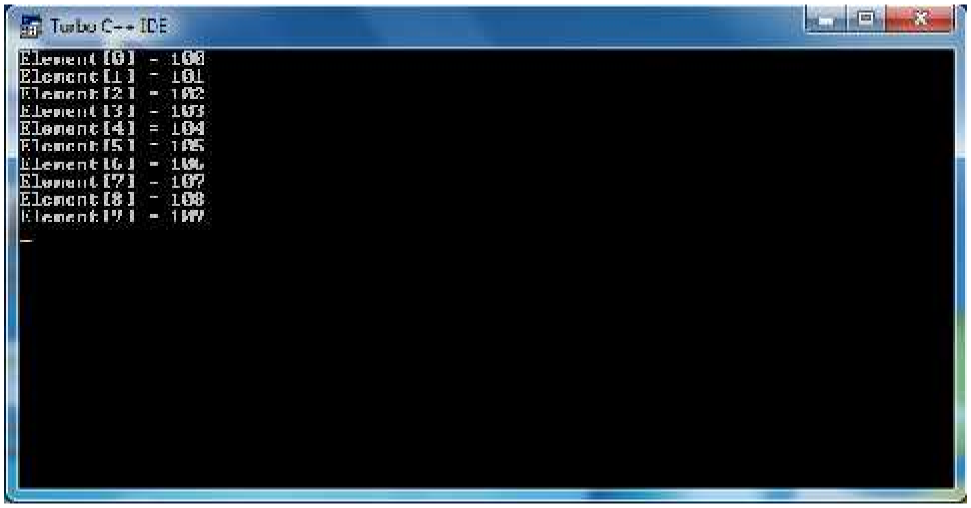
The last statement above assigns element number 3rd in the array a value of 3.3. Array with 2nd index will be 3rd i.e. last element because all arrays have 0 as the index of their first element. Following is the pictorial representation of the same array we discussed above:



Program 9.1 explains the declaration, assignment, access, usage and working of 1D array. It is followed by the output of the program.

```
#include <stdio.h>

void main ()
    {

    int n[ 10 ];
    /* n is an array of 10 integers */

    int i, j;

    for ( i = 0; i < 10; i++ )
        {
        n[ i ] = i + 100;

        /* set element at location i
            to i + 100 */
        }
    /* output each array element's value */
    for (j = 0; j < 10; j++ )
        {
        printf("Element[%d] = %d\n",
                j, n[j] );
        }
    }
```

Program 9.1: Program shows the usage and working of 1D array



```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
_
```

9.3 Two-Dimensional Array

The array which is used to represent and store data in a tabular form is called as Two Dimensional Array. Such type of array specially used to represent data in a matrix form. To declare Two Dimensional Array in C, the general syntax is as follows:

```
<data-type> <Array-name> [<rows>][<columns>];
```

The rows and columns, both must be integer constants greater than zero and data-type can be any valid C data type; primary or user-defined. For example, to declare a two dimensional array called StudentsMarks of type double and having 3 rows and 4 columns, we can use following statement:

```
double StudentMarks [3][4];
```

In order to initialize this array, we have two ways:

1. Initialize all elements at a time during declaration,

2. Initialize elements individually after declaration

The first method can be used to declare and assign values to all elements of 1D array as follows:

```
double StudentMarks [3][2] =
{
{0.0, 0.1},
{1.0, 1.1},
{2.0, 2.1}
};
```

Recall that to access any individual array element, the array name along with index within square braces is used, and it can be used the same way any other variable of same data type can be. However, in case of 2-D arrays we have use another dimensional also. This means element StudentMarks[0][0], which is the first element will be assigned value 0.0, second element StudentMarks[0][1] will be assigned value 0.1, third element StudentMarks[1][0] will be assigned value 1.0, fourth element StudentMarks[1][1] will be assigned value 1.1, and last element StudentMarks[2][1] will be assigned value 2.1.

It should be noted that the number of values between braces { } can't be larger than the number of elements (rows x columns). However, if we omit the size of the array during declaration, an array just big enough to hold the values of initialization, is declared. Therefore, the following statement has same effect as the first valid statement above has.

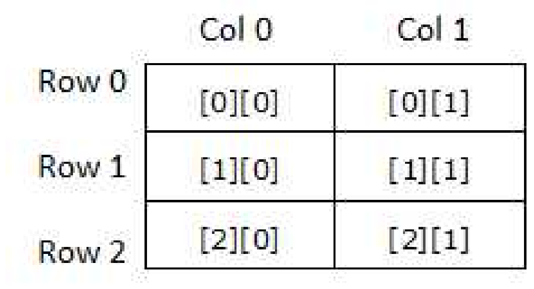
```
double StudentMarks [][] =
{
{0.0, 0.1},
{1.0, 1.1},
{2.0, 2.1}
};
```

The second method is to assign values to every element individually. After declaration of StudentMarks array, we can initialize it as follows:

```

double StudentMarks [3][2];

StudentMarks[0][0] = 0.0;
StudentMarks[0][1] = 0.1;
```

```
StudentMarks[1][0] = 1.0;
StudentMarks[1][1] = 1.1;
StudentMarks[2][0] = 2.0;
StudentMarks[2][1] = 2.1;
```

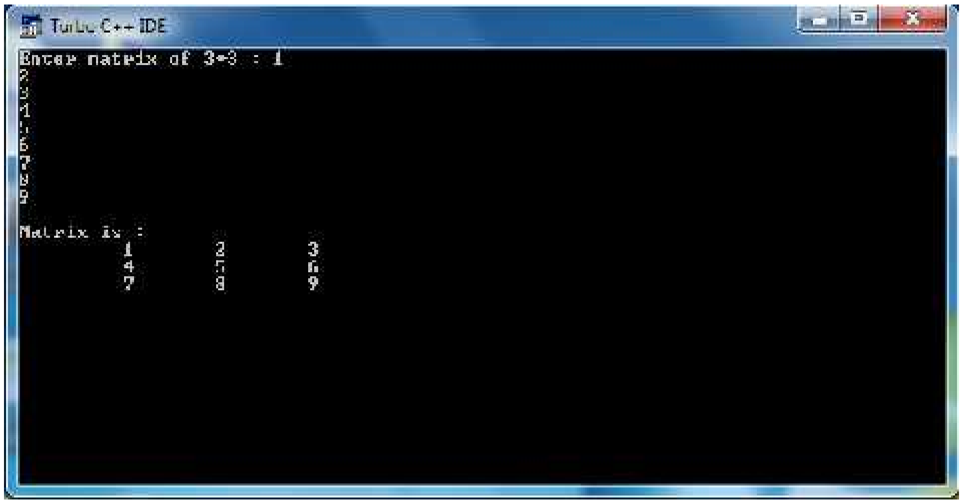
Following is the pictorial representation of the same array we discussed above:



Program 9.2 shows how to read and display a 3x3 matrix using 2D array. It is followed by the output of the program.

```
    #include <stdio.h>

    void main()
    {
        int a[3][3], i, j;

        printf("Enter matrix of 3*3 : ");

        for(i=0; i<3; i++)
        {
            for(j=0; j<3; j++)
            {
            scanf("%d",&a[i][j]);
            }
        }

        printf("\nMatrix is : \n");

        for(i=0; i<3; i++)
        {
            for(j=0; j<3; j++)
            {
                printf("\t %d",a[i][j]);
                }
              printf("\n");
         }
    }
```

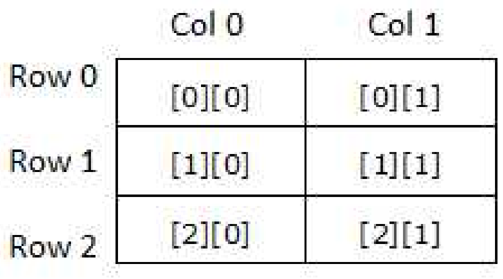
Program 9.2: Program to read and display a 3x3 matrix using 2D array



```
Enter matrix of 3*3 : 1
2
3
4
5
6
7
8
9

Matrix is :
        1       2       3
        4       5       6
        7       8       9
```

9.4 Multi-Dimensional Array

C does not have true multidimensional arrays. However, because of the generality of C's type system; we can have arrays of arrays. This is what a 2D Array is. Consider the example of 2D Array discussed in last section.



	Col 0	Col 1
Row 0	[0][0]	[0][1]
Row 1	[1][0]	[1][1]
Row 2	[2][0]	[2][1]

In this case, each row is one-dimensional array of 2 elements. Now, all these rows are treated as single variables and are placed next to each other. This gives rise to one dimensional Array of one-dimensional Arrays. In other words, we have a multi-dimensional array.

C allows array of two or more dimensions and maximum number of dimensions an Array in a C program can have depends upon the compiler. Logically, there is no limit on the maximum number of dimensions. So in C programming an array can have two or three or four or even ten or more dimensions. More dimensions in an array means more data it can hold and of course more difficulties to manage and understand these arrays. A multidimensional array has following syntax:

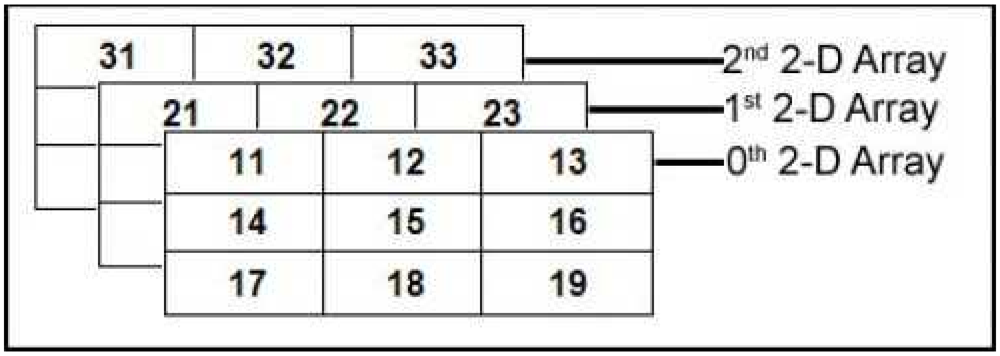
```
<data-type> <Array-Name> [d1][d2][d3]…[dn];
```

Where d_i is the size of ith dimension.

Consider a 3D array declared and initialized as follows:

```
int array_3d[3][3][3]=
        {
            {
            {11, 12, 13},
            {14, 15, 16},
            {17, 18, 19}
            },
            {
            {21, 22, 23},
            {24, 25, 26},
            {27, 28, 29}
            },
            {
            {31, 32, 33},
            {34, 35, 36},
            {37, 38, 39}
            },
        };
```

This means the array_3d is one dimensional array of 2D arrays. On other words, each element of this array is a 2D array. The pictorial representation of array_3d is as follows.



9.5 Summary

- An array in C programing language can be defined as a name given to number of consecutive memory locations, each of which can store the data of same data type.
- One can visualize an array as a collection of variables of the same data type.
- All the individual variables (also called elements) of Array can be referenced through the same Array name.
- To make distinction between individual elements an index is used to refer to a particular element in an array.
- Arrays are of two types;
  - one dimension array, and
  - multi-dimension array.
- The array which is used to represent and store data in a linear form is called as Single or One Dimensional Array.
- The array which is used to represent and store data in a tabular form is called as Two Dimensional Array.
- C does not have true multidimensional arrays. However, because of the generality of C's type system; we can have arrays of arrays.

9.6 Model Questions

Q 89. Explain the concept and significance of an Array.

Q 90. Discuss the declaration, initialization and accessing an array in C language.

Q 91. Explain the declaration, initialization and accessing one dimensional array in C language with the help of a suitable program.

Q 92. Explain the declaration, initialization and accessing two dimensional array in C language with the help of a suitable program.

Q 93. Explain the declaration, initialization and accessing three dimensional array in C language with the help of a suitable program.

Q 94. Explain various methods to initialize an array.

Q 95. Write a program to populate a 3x3 matrix in C language using 2D arrays.

Lesson 10

Structures & Unions

Structure

38.0. Introduction

38.1. Structure Definition

38.2. Structure Declaration

38.3. Initialization & Accessing Structure Members

38.4. Nesting of Structures

38.5. Union: Definition, Declaration & Initialization

38.6. Union: Accessing Union Members & Nesting

38.7. Arrays of Structure & Unions

38.8. Summary

38.9. Model Questions

10.0 Introduction

Arrays provide a mechanism to declare a derived data type which is a collection of homogenous elements. This means each element has same data type. However, it is sometimes required to create a collection of non-homogeneous elements in which elements can be of different type. Consider the automation of student records in University of Kashmir. Every record may contain student name, its registration number, address, and so on. It is obvious that the information to be stored is not homogenous. In order to create such kind of records programming languages provides various types of constructs. Specifically, C language provides user-defined data type called struct (Structure). In addition, it also provides another user-defined data type called union. These user-defined data types can be used to create various types of collection of elements in which elements can be of different data types.

In this lesson, we will discuss the struct (structure) and union user defined data types supported by C language. Specifically, we discuss how to define, declare, initialize and access these data types.

10.1 Structure Definition

In some programming contexts, you need to access multiple data types under a single name for easier data manipulation; for example you want to refer to address with multiple data like house number, street, zip code, country. C array is a derived data type allow you to define type of variables that can hold several data items of the same kind but structure is a user defined data type available in C programming, which allows you to combine data items of different kinds.

C supports structure which allows us to wrap one or more variables with different data types. A structure can contain any valid data types like int, char, float, arrays or even other structures. Each variable in structure is called a structure member.

To define a structure, you use struct keyword. Here is the common syntax of structure definition:

```
struct <structure-name>
    {
    <data-type> <variable-name>;
    .
    .
    <data-type> <array-name> [<size>];
    .
    .
    struct <other-struct-name>;
    .
    .
    };
```

It should be noted that the name of structure follows the rule of variable name. Here is an example of defining address structure:

```
struct address
    {

    int house_number;

    char street_name[50];

     int zip_code;

     char country[50];

     };
```

The address structure contains house number as an integer, street name as a string, zip code as an integer and country as a string.

10.2 Structure Declaration

The above example only defines an address structure without creating any structure variable. To create or declare a structure variable, there are two ways:

The first way is to declare a structure variable immediately after the structure definition, as follows:

```
struct address
    {

    int house_number;

    char street_name[50];

     int zip_code;

     char country[50];
    } struct_var1, struct_var2;
```

In the second way, you can declare the structure variable at a different location after structure definition. Here is structure declaration syntax:

```
struct address
    {

    int house_number;

    char street_name[50];

     int zip_code;

     char country[50];

    } ;

struct address struct_var1, struct_var2;
```

10.3 Initialization & Accessing Structure Members

C programming language treats a structure as a user-defined data type; therefore you can initialize a structure like a variable. Here is an example of initialize product structure:

```
struct product
    {
    char name[50];
    double price;
    }
    book = {
            "C programming language",
            40.5
            };
```

In above example, we defined product structure, then we declare and initialize book structure variable with its name and price.

To access structure members we can use dot operator (.) between structure variable and structure member name as follows:

```
<Structure-Variable-Name>.<MemberAddress>
```

For example to access street name of structure address we do as follows:

```
struct address myAddress;
myAddress.street_name = “Srinagar”;
```

One of major advantage of structure is you can copy it with = operator. The syntax as follows

```
     struct_var1 = struct_var2;
```

However, it should be noted that both variables should be of same structure type.

10.4 Nesting of Structures

A Structure can contain other Structure variables as members. This is called Nesting of Structures. For example address structure is a structure. We can define a nested structure called customer which contains address structure as follows:

```
struct address
     {

     int house_number;

     char street_name[50];

      int zip_code;
     char country[50];

    };

struct customer
     {
     char name[50];

     structure address billing_addr;

     structure address shipping_addr;
    };

struct customer cust1;
```

If the structure contains another structure, we can use dot operator to access nested structure and use dot operator again to access variables of nested structure.

```
cust1.billing_addr.house_number = 10;
```

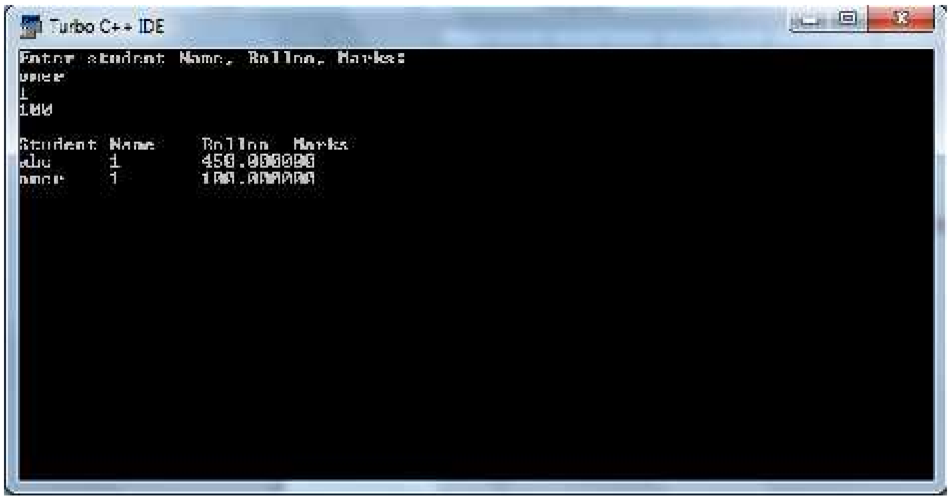
It should be noted that a structure definition can’t contain the variable of its own type as a data member. This means the following structure definition is illegal.

```
struct address
{

    int house_number;

    char street_name[50];

     int zip_code;

     char country[50];

    struct address old_address;
    /* The statement above is illegal*/
    };
```

Program 10.1 shows a simple program to define a structure, declare a structure variable, initialize and access members. It is followed by screenshot of the output.

```
#include<stdio.h>

struct student
    {
    char name[20];

    int rollno;

    float marks;
    };


void main()
    {
    struct student s1 = {"abc", 1, 450};

    struct student s2;

    printf("Enter student Name, Rollno,
    Marks:\n");

    scanf("%s%i%f", &s2.name, &s2.rollno,
    &s2.marks);

    printf("\nStudent Name\tRollno\tMarks
    \n");

    printf("%s\t%i\t%f", s1.name,
    s1.rollno, s1.marks);

    printf("\n");

    printf("%s\t%i\t%f",s2.name,
    s2.rollno,s2.marks);

    }
```

Program 10.1: Program to show usage and working of structure



```
Enter student Name, Rollno, Marks:
umer
1
100

Student Name    Rollno  Marks
abc      1       450.000000
umer     1       100.000000
```

10.5 Union: Definition, Declaration & Initialization

Union, is a collection of variables of different types, just like a structure, however a union can only store information in one field at one time. This means, a union can be viewed as a variable type that can contain many different variables as members (like a structure), but only actually holds one of them at a time (unlike a structure).

One can visualize union as a chunk of memory that is used to store variables of different types. So, once a new value is assigned to a field, the existing data is wiped over with the new data. This can save memory if you have a group of data where only one of the types is used at a time. Therefore, the size of a union is equal to the size of its largest data member. In other words, the C compiler allocates just enough space for the largest member. This is because only one member can be used at a time, so the size of the largest, is the most one will need. In contrast, the size of a structure is equal to the sum of sizes of its members because every member is allocated a unique memory chunk.

The syntax for defining a Union and declaring a variable of Union is same as that of Structure; except the keyword used is union.

```
union <union-name>
     {
     <data-type> <variable-name>;
     .
     .
     <data-type> <array-name> [<size>];
     .
     .
     union <other-union-name>;
     .
     .
     };
```

It should be noted that a structure can have union as a data member and union can have a structure as a data member.

The following code snippet defines a union student

```
union student
     {

    int marks;

    float percentage;

      };
```

Indeed if it would have been a structure then the size of this structure would have been 6 bytes. However, as it is a union its size is 4 bytes (largest data member is float).

The following code snippet defines, declares and initializes a union student.

```

union student
     {
```

```
    int marks;

    float percentage;

     } std1, std2 = {10};
/* Only one assignment is needed */
```

10.6 Union: Accessing Union Members & Nesting

To access union members we can use dot operator (.) between union variable and union member name as follows:

```
<Union-Variable-Name>.<MemberAddress>
```

For example, to access marks of union student we can do as follows:

```
union student std2;
std2.marks = 20;
```

Now, the data member percentage of union student also contains the same value 20 but in floating point format. As soon as, the other data member of union is assigned some value, the marks member is overwritten with that.

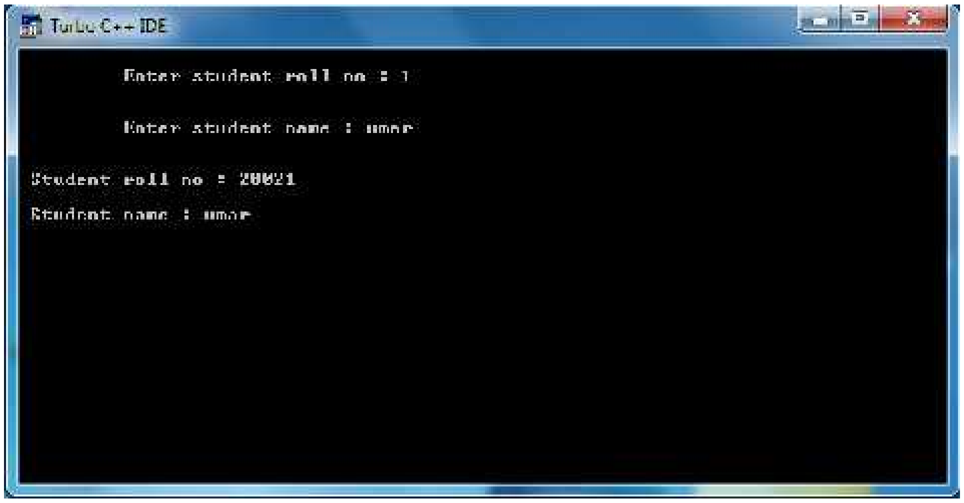
Also, two variables of same union type can be used on two sides of an assignment operator.

```
    union_var1 = union_var2;
```

As a Structure definition can contain other Structure variables as members, so can Union. This is called Nesting of Union. In addition, we can use dot operator to access nested union and use dot operator again to access variables of nested union as is in case of structures. However it should be noted that a union definition can’t contain the variable of its own type as a data member. The program 10.2 shows how union variable behaves which is followed by the screenshot of the output.

```
#include <stdio.h>

union student
{
    int rno;
    char name[50];
}
std;


void main()
    {

    printf("\n\t Enter student roll no: ");

    scanf("%d", &std.rno);

    printf("\n\n\t Enter student name : ");

    scanf("%s", std.name);

    printf("\n\n Student roll no : %d",
    std.rno);

    printf("\n\n Student name : %s",
    std.name);
}
```

Program 10.2: Program to show usage and working of union



```
        Enter student roll no : 1

        Enter student name : umar

Student roll no = 20021

Student name : umar
```

10.7 Arrays of Structures & Unions

Since structures are data types that are especially useful for creating collection items, why not make a collection of them using an array? Similarly, we can create array of Unions. The following code snippet shows how to create an array of structures and access a particular structure element's member.

```
struct student
{
    int rno;
    char name[50];
}
std1[10];

struct student std2[5];

std1[0].rno = 10;
std2[4].name = “umar”;
```

The following code snippet shows how to create an array of unions and access a particular union element's member.

```
union student
{
    int rno;
    char name[50];
}
std1[10];

union student std2[5];

std1[0].rno = 10;
std2[4].name = “umar”;
```

10.8 Summary

- C array is a derived data type that allows us to define type of variable
- Structure is a user defined data type available in C programming, which allows us to combine data items of different kinds.
- Each variable in structure is called a structure member.
- To define a structure, we use struct keyword.
- The name of structure follows the rule of variable name.
- C programming language treats a structure as a user-defined data type; therefore we can initialize a structure like a variable.
- To access structure members we can use dot operator (.) between structure variable and structure member name
- A Structure can contain other Structure variables as members. This is called Nesting of Structures.
- A structure definition can’t contain the variable of its own type as a data member.
- Union, is a collection of variables of different types, just like a structure, however a union can only store information in one field at one time.
- One can visualize union as a chunk of memory that is used to store variables of different types.
- The size of a union is equal to the size of its largest data member.
- The size of a structure is equal to the sum of sizes of its members.
- The syntax for defining a Union and declaring a variable of Union is same as that of Structure; except the keyword used is union.

10.9 Model Questions

Q 96. Explain why C Array is not enough?

Q 97. Explain the concept of a structure in C programming language.

Q 98. With the help of a program explain the definition of a structure data type, declaration and initialization of structure variables and member access in C programming language.

Q 99. Explain nesting of structures. What are the limitations?

Q 100. Explain with the help of a program structure array.

Q 101. Explain the concept of a union in C programming language.

Q 102. With the help of a program explain the definition of a union data type, declaration and initialization of union variables and member access in C programming language.

Q 103. Explain nesting of union. What are the limitations?

Q 104. Explain with the help of a program union array.

Q 105. Differentiate between Structure and Union in C programming language.