echnology; *Dr Sanjeev* tment; *Dr Vijay Gupta,* Engineering College; and *Naveen Aggarwal*

of the manuscript: *Mo-d, Amandeep Jakhu* and d efforts in proof read-

rajan, Jennifer Sargunar, s of Pearson Education *fer,* who has taken per- thank *Showick Thorpe,*

Sh. T.L. Mittal and *Smt.* moral support and pa- *Mittal* was my inspira- mpletion of the book.

Ajay Mittal

1

DATA TYPES, VARIABLES AND CONSTANTS

Learning Objectives

In this chapter, you will learn about:

- Various features of C language
- Various C's standards
- C's character set
- Identifiers and Keywords
- Rules to write identifier names in C
- Data types, type qualifiers and type modifiers
- Declaration statement
- Difference between declaration and definition
- Length and Range of various data types
- l-value and r-value concept
- Variables and constants
- Classification of constants
- Structure of a C program
- Process of compiling and executing a C program
- Writing simple C programs
- Using printf and scanf functions
- Use of sizeof operator

1.1 Introduction

C is a general-purpose, block-structured, procedural, case-sensitive, free-flow, portable and high-level programming language developed by Dennis Ritchie at the Bell Telephone Laboratories. The selection of 'C' as the name of a programming language seems to be an odd choice but it was named C because it evolved from earlier languages Basic Combined Programming Language (**BCPL**) and **B**.

In 1967, Martin Richards developed BCPL for writing system software (i.e. operating systems and compilers). Ken Thompson in 1970 developed a stripped version of BCPL and named it B. The language B was used to create early versions of UNIX operating system. Both the languages BCPL and B were 'typeless', and every data object occupied one word in the memory. In 1972, Dennis Ritchie developed C programming language by retaining the important features of BCPL and B programming languages and adding data types and other powerful features to the retained feature set of BCPL and B. The language C was initially designed as a system implementation language for developing system software for the UNIX operating system. Thus, it was widely known as the development language of the UNIX operating system. However, after its popularity, it has spread over many other platforms and is used for creating many other applications in addition to the system software. Thus, nowadays, C is known as a general-purpose language and not only as a system implementation language.

1.2 C Standards

The rapid expansion of C to various platforms led to many variations that were similar but were often incompatible. This was a serious problem for programmers who wanted to develop code that could run on several platforms. This problem led to the realization of the need for a standard. This section lists the formulation of various C standards in the chronological order:

1.2.1 Kernighan & Ritchie (K&R) C Standard

The first edition of 'The C Programming Language' book by Brian Kernighan and Dennis Ritchie was published in 1978. This book was one of the most successful computer science books and has served as an informal standard for the C language for many years. This informal standard was known as '**K&R C**'.

1.2.2 ANSI C/Standard C/C89 Standard

In 1983, a technical committee was created under the American National Standards Institute (ANSI) committee to establish a standard specification of C. In 1989, the standard proposed by the committee was formally approved and is often referred to as **ANSI C**, **Standard C** or sometimes **C89**.

1.2.3 ISO C/C90 Standard

In 1990, the International Organization for Standardization (ISO) adopted the ANSI C standard after minor modifications. This version of the standard is called **ISO C** or sometimes **C90**.

1.2.4 C99 Standard

After the adoption of the ANSI standard, the C language specifications remained unchanged for sometime, whereas the language C++ continued to evolve. To accommodate this evolution of C++, a new standard of C language that corrected some details of ANSI C standard

and added more e
in 1999 and is kno
ported by many p

The text
tested on
and Micro

1.3 Learning
An Analo

Writing a C progra
have undergone ir
have told you:

1. How to crea
2. How to forn
3. How to orga
4. How to arra

In this book, yo

1. How to crea
   guage. This
2. How to use
   ments, the s
   ing a senten
3. How to use
   paragraph.
4. How to use
   paragraphs.

The above learn

1. Creating ide
2. Creating exp
3. Creating fun

Since, I do not w
forward jumps in t
gram writing in thi
a bit patient. The co
be clear by the end

1.4 C Charact

A **character set** defi
when a program is
is called a **source ch**
executed is called a
different from the e
the two character se

and added more extensive support to it was introduced in 1995. The standard was published in 1999 and is known as **C99**. The C99 standard has not been widely adopted and is not supported by many popular C compilers.

The text and questions in this book are in accordance to ANSI/ISO standards and are tested on Borland Turbo C (TC) 3.0 compiler for DOS, Borland TC 4.5 compiler for Windows and Microsoft VC++ 6.0 compiler for Windows.

1.3 Learning Programming Language and Natural Language: An Analogy

Writing a C program is analogous to writing an essay. Recall all the stages through which you have undergone in the process of learning how to write an essay in English. Your teacher must have told you:

1. How to create words from letters.
2. How to form sentences using words and grammar.
3. How to organize sentences and create paragraphs.
4. How to arrange paragraphs and write an essay.

In this book, you will learn about:

1. How to create identifiers using the characters available in the character set of C language. This is analogous to creating words in a natural language.
2. How to use identifiers to form expressions, which can be further converted to statements, the smallest logical unit of a program. Forming a statement is analogous to forming a sentence.
3. How to use statements to write functions. Writing a function is analogous to writing a paragraph.
4. How to use functions to create a program. This is analogous to creating an essay from paragraphs.

The above learning objectives are organized in this book as follows:

1. Creating identifier names: Chapter 1
2. Creating expressions and statements: Chapters 2 and 3
3. Creating functions: Chapter 5

Since, I do not want to restrain you from writing programs till Chapter 5, I will make some forward jumps in the flow of learning C programming language. I will introduce you to program writing in this chapter itself, but if something does not seem obvious, I advise you to be a bit patient. The concepts will be clearer when you go through the first few chapters and will be clear by the end of Chapter 5.

1.4 C Character Set

A **character set** defines the valid characters that can be used in a source program or interpreted when a program is running. The set of characters that can be used to write a source program is called a **source character set**, and the set of characters available when the program is being executed is called an **execution character set**. It is possible that the source character set is different from the execution character set, but in most of the implementations of C language, the two character sets are identical.

The basic source character set of C language includes:

1. Letters:
   a. Uppercase letters: A, B, C, ..., Z
   b. Lowercase letters: a, b, c, ..., z
2. Digits: 0, 1, 2, ..., 9
3. Special characters: , . : ; ! " ^ # % ^ & * ( ) { } [ ] < > | \ / _ ~ etc.
4. White space characters:
   a. Blank space character
   b. Horizontal tab space character
   c. Carriage return
   d. New line character
   e. Form feed character

1.5 Identifiers and Keywords

If you know C's source character set, the next step is to write identifiers. This is analogous to writing words in a natural language.

1.5.1 Identifiers

An **identifier** refers to the name of an object. It can be a variable name, a label name,➲ a function name,➲ a typedef name,➲ a macro name➲ or a macro parameter, a tag or a member of a structure, a union or an enumeration.➲

The syntactic rules to write an identifier name in C are as follows:

1. Identifier name in C can have letters, digits or underscores.
2. The first character of an identifier name must be a letter (either uppercase or lowercase) or an underscore. The first character of an identifier name cannot be a digit.
3. No special character (except underscore), blank space and comma can be used in an identifier name.
4. Keywords or reserved words cannot form a valid identifier name.
5. The maximum number of characters allowed in an identifier name is compiler dependent, but the limit imposed by all the compilers provides enough flexibility to create meaningful identifier names.

The following identifier names are valid in C:

Student_Name, StudentName, student_name, student1, _student

The following identifier names are not valid in C:

Student Name (due to blank space), Name&Rollno (due to special character &), 1st_student (first character being a digit), for (for being a keyword).

It is always advisable to create meaningful identifier names. Meaningful identifier names are easier to read and increase the maintainability of a program. For example, it is better to create an identifier name as student_name instead of snam.

Forward Reference: Label name (Chapter 3), function (Chapter 5), typedef name (Chapter 7), macro name (Chapter 8), structure, union, enumeration (Chapter 9).

1.5.2 Keyword

Keyword is a rese
meaning of a key
language. There a
C language.

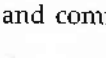
Table 1.1 | List of k

S.No	Keyword
1.	auto
2.	break
3.	case
4.	char
5.	const
6.	continue
7.	default
8.	do

1.6 Declaratio

If you have learnt h
(except label name)
An identifier can
declaration statemen
type) to the compile

[storage



The terms en
declaration st
the mandator

The following dec

```
int variable;
static int variable;

static unsigned int variable;

static const unsigned int varia

int variable=20;
int a=20, b=10;
```

† Refer Section 1.8.1 for a
‡ Refer Section 1.8.2 for a
§ Refer Section 1.7 for a de
¶ These are actually definiti

1.5.2 Keywords

Keyword is a reserved word that has a particular meaning in the programming language. The meaning of a keyword is predefined. A keyword cannot be used as an identifier name in C language. There are 32 keywords available in C. Table 1.1 gives a set of keywords present in C language.

Table 1.1 | List of keywords in C

S.No	Keyword	S.No	Keyword	S.No	Keyword	S.No	Keyword
1.	auto	9.	double	17.	int	25.	struct
2.	break	10.	else	18.	long	26.	switch
3.	case	11.	enum	19.	register	27.	typedef
4.	char	12.	extern	20.	return	28.	union
5.	const	13.	float	21.	short	29.	unsigned
6.	continue	14.	for	22.	signed	30.	void
7.	default	15.	goto	23.	sizeof	31.	volatile
8.	do	16.	if	24.	static	32.	while

1.6 Declaration Statement

If you have learnt how to create an identifier name, you should know that every identifier (except label name) needs to be declared before it is used.

An identifier can be declared by making use of the **declaration statement**. The **role** of a declaration statement is to introduce the name of an identifier along with its data type (or just type) to the compiler before its use. The general form of a declaration statement is:

[storage_class_specifier‡][type_qualifier†|type_modifier†] **type§ identifier** [=value[,...]];

The terms enclosed within square brackets (i.e. []) are optional and might not be present in a declaration statement. The type, identifier and the terminating semicolon (shown in bold) are the mandatory parts of a declaration statement.

The following declaration statements¶ are valid in C:

`int variable;`	(type int and identifier name variable present)
`static int variable;`	(Storage class specifier static, type int and identifier name variable present)
`static unsigned int variable;`	(Storage class specifier static, type modifier unsigned, type int and identifier name variable present)
`static const unsigned int variable;`	(Storage class specifier static, type qualifier const, type modifier unsigned, type int and identifier name variable present)
`int variable=20;`	(type int, identifier name variable and value 20 present)
`int a=20, b=10;`	(type int, identifier name a and its initial value 20 present, another identifier name b and its initial value 10 present)

† Refer Section 1.8.1 for a description on type qualifiers.
‡ Refer Section 1.8.2 for a description on type modifiers.
§ Refer Section 1.7 for a description on types.
¶ These are actually definition statements. Refer Section 1.9 for a description on declaration and definition.

A declaration statement in which more than one identifier is declared is known as a **shorthand declaration statement**. For example, int a=20, b=10; is a shorthand declaration statement. The corresponding **longhand declaration statements** equivalent to this shorthand declaration statement are int a=20; int b=10;. It is important to note that shorthand declaration can only be used to declare identifiers of the same type. In no way can it be used to declare identifiers of different types, e.g. int a=10, float b=2.3; is an invalid statement.

Forward Reference: Storage class specifier (Chapter 7).

1.7 Data Types

If you know how to write a declaration statement, you would probably know that the declaration statement is used to tell the data type (or just type) of an identifier to the compiler before its use.

Data type or just **type** is one of the most important attributes of an identifier. It determines the possible values that an identifier can have and the valid operations that can be applied on it.

In C language, data types are broadly classified as:

1. Basic data types (primitive data types)
2. Derived data types
3. User-defined data types

1.7.1 Basic/Primitive Data Types

The five basic data types and their corresponding keywords available in C are:

1. Character (char)
2. Integer (int)
3. Single-precision floating point (float)
4. Double-precision floating point (double)
5. No value available (void)

1.7.2 Derived Data Types

These data types are derived from the basic data types. Derived data types available in C are:

1. Array type➲ e.g. char[ ], int[ ], etc.
2. Pointer type➲ e.g. char*, int*, etc.
3. Function type➲ e.g. int(int,int), float(int), etc.

Forward Reference: Array type (Chapter 4), pointer type (Chapter 4), function type (Chapter 5).

1.7.3 User-defined Data Types

The C language provides flexibility to the user to create new data types. These newly created data types are called **user-defined data types**. The user-defined data types in C can be created by using:

1. Structure
2. Union➲
3. Enumerat

Forward

1.8 Type Q

The declaration s

1.8.1 Type Qu

A **type qualifier**
clared object. Th
type qualifiers av

1. **const**[††] **qual**
   during the
2. **volatile qual**
   relevant to

1.8.2 Type Mo

A **type modifier**
arithmetic proper
available in C are:

1. Signed (sign
2. Unsigned (u
3. Short (short)
4. Long (long)

1.9 Differenc

It is very importa
Declaration only i
fore it is used. Dur
of an identifier me
memory. The amou
of the identifier. Id
The memory space
ing environment u
environment.

[††] Refer Section 1.11.2
[‡‡] Refer Section 1.9 for

1. Structure
2. Union
3. Enumeration

Forward Reference: Structure, union, enumeration (Chapter 9).

1.8 Type Qualifiers and Type Modifiers

The declaration statement can optionally have type qualifiers or type modifiers or both.

1.8.1 Type Qualifiers

A **type qualifier** neither affects the range of values nor the arithmetic properties of the declared object. They are used to indicate the special properties of data within an object. Two type qualifiers available in C are:

1. **const[††] qualifier:** Declaring an object const announces that its value will not be changed during the execution of a program.
2. **volatile qualifier:** volatile qualifier announces that the object has some special properties relevant to optimization.

1.8.2 Type Modifiers

A **type modifier** modifies the base type to yield a new type. It modifies the range[‡‡] and the arithmetic properties of the base type. The type modifiers and the corresponding keywords available in C are:

1. Signed (signed)
2. Unsigned (unsigned)
3. Short (short)
4. Long (long)

1.9 Difference Between Declaration and Definition

It is very important to know the difference between the terms **declaration** and **definition**. **Declaration** only introduces the name of an identifier along with its type to the compiler before it is used. During declaration, no memory space is allocated to an identifier. **Definition** of an identifier means the declaration of an identifier plus reservation of space for it in the memory. The amount of memory space reserved for an identifier depends upon the data type of the identifier. Identifiers of different data types take different amounts of memory space. The memory space required by an identifier also depends upon the compiler and the working environment used. Table 1.2 lists the length of various data types in DOS and Windows environment.

[††] Refer Section 1.11.2.2 for a description on const qualifier.

[‡‡] Refer Section 1.9 for a description on range modification by type modifiers.

Table 1.2 | Data types and their memory requirements

S.No	Data type	Base/Modified	TURBO C 3.0/DOS	MS VC++ 6.0/WINDOWS
1.	char	Base	1 Byte	1 Byte
2.	int	Base	2 Bytes	4 Bytes
3.	float	Base	4 Bytes	4 Bytes
4.	double	Base	8 Bytes	8 Bytes
5.	signed ⟨data type 1, 2⟩	Modified	⟨same as data type 1, 2⟩	⟨same as data type 1, 2⟩
6.	unsigned ⟨data type 1, 2⟩	Modified	⟨same as data type 1, 2⟩	⟨same as data type 1, 2⟩
7.	short int	Modified	2 Bytes	2 Bytes
8.	long int	Modified	4 Bytes	4 Bytes
9.	long float	Modified	8 Bytes	8 Bytes
10.	long double	Modified	10 Bytes	8 Bytes
11.	void	Base	Object of void type cannot be created	

The data type determines the possible values that an identifier can have. The range of a data type depends upon the length of the data type. Table 1.3 lists the range of various data types in DOS and Windows environment.

Table 1.3 | Range of various data types

S.No	Data type	TURBO C 3.0/DOS	MS VC++ 6.0/WINDOWS
1.	char	–128 to 127	–128 to 127
2.	int	–32768 to 32767	–2,147,483,648 to 2,147,483,647
3.	float	$3.4 * 10^{-38}$ to $3.4 * 10^{38}$	$3.4 * 10^{-38}$ to $3.4 * 10^{38}$
4.	double	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$
5.	signed ⟨data type 1, 2⟩	Same as 1, 2 as by default data types are signed	Same as 1, 2 as by default data types are signed
6.	unsigned char	0 to 255	0 to 255
7.	unsigned int	0 to 65535	0 to 4,294,967,295
8.	unsigned long int	0 to 4,294,967,295	0 to 4,294,967,295
9.	short int	–32768 to 32767	–32768 to 32767
10.	long double	$3.4 * 10^{-4932}$ to $1.1 * 10^{4932}$	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$

Despite the big difference between the terms declaration and definition, the word declaration is commonly used in place of definition. All the statements written in Section 1.6 are actually definition statements, but I have referred to them as declarations because at that point I just wanted to focus on the name and the type of an identifier.

The statement int variable=20; mentioned in Section 1.6 is actually a definition statement because it allocates 2 bytes (or 4 bytes) to variable somewhere in the memory (say, at memory location with address 2000) and initializes it with the value 20. The memory allocation is purely random (i.e. any free memory location will be randomly allocated). This is illustrated in Figure 1.1.

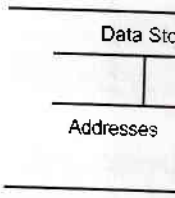


Figure 1.1 | Alloca

If int variable; is a d

If you want t
specifier. ➲ The ke
The extern declarat

Forward R

1.10 Data Ob

You must have kn
in memory depen
location gives rise
concept. These cor

1.10.1 Data Obj

Data object is a ter
ues. Once an identi

1.10.2 L-value

L-value is a data ob
is a sort of name gi
locator. The term l-

1. **Modifiable**
can be access
2. **Non-modifia**
but cannot be

I in l-value
an assignme

1.10.3 R-value

R-value refers to 're

[11] Refer Section 1.11.2.

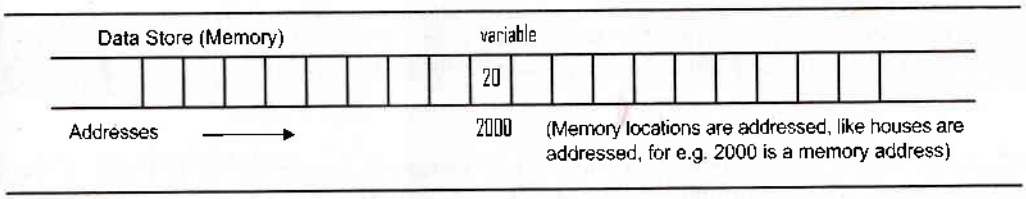


Figure 1.1 | Allocation of memory to variable

If int variable; is a definition statement, then how can I declare variable?

If you want to actually declare variable, write extern int variable;. extern is a storage class specifier.➲ The keyword extern provides a method for declaring a variable without defining it. The extern declaration➲ does not allocate the memory.

Forward Reference: Storage class specifier (Chapter 7), extern declaration (Chapter 7).

1.10 Data Object, L-value and R-value

You must have known by this time that upon definition, an identifier is allocated some space in memory depending upon its data type and the working environment. This memory allocation gives rise to two important concepts known as the **l-value concept** and the **r-value concept**. These concepts are described below.

1.10.1 Data Object

Data object is a term that is used to specify the region of data storage that is used to hold values. Once an identifier is allocated memory space, it will be known as a data object.

1.10.2 L-value

L-value is a data object locator. It is an expression➲ that locates an object. In Figure 1.1, variable is a sort of name given to the memory location 2000. variable here refers to l-value,✍ an object locator. The term l-value can be further categorized as:

1. **Modifiable l-value:** A modifiable l-value is an expression that refers to an object that can be accessed and legally changed in the memory.
2. **Non-modifiable l-value:** A non-modifiable l-value refers to an object that can be accessed but cannot be changed in the memory. [11]

✍ l in **l-value** stands for '**left**'; this means that the l-value could legally stand on the left side of an assignment operator.

1.10.3 R-value

R-value refers to 'read value'. In Figure 1.1, variable has an r-value✍ 20.

[11] Refer Section 1.11.2.2 to learn how to make an l-value non-modifiable.

r in **r-value** stands for '**right**' or '**read**'; this means that if an identifier name appears on the right side of an assignment operator it refers to the r-value.

Consider Figure 1.1 and the expression variable=**variable**+20. variable on the left side of the assignment operator refers to the l-value. **variable** on the right side of the assignment operator (in bold) refers to the r-value. **variable** appearing on the right side refers to 20. The number 20 is added to 20 and the value of expression is 40 (r-value). This outcome (40) is assigned to variable on the left side of the assignment operator, which signifies l-value. The l-value variable locates the memory location where this value is to be placed, i.e. at 2000. After the evaluation of the expression variable=**variable**+20, the contents of the memory are shown in Figure 1.2.

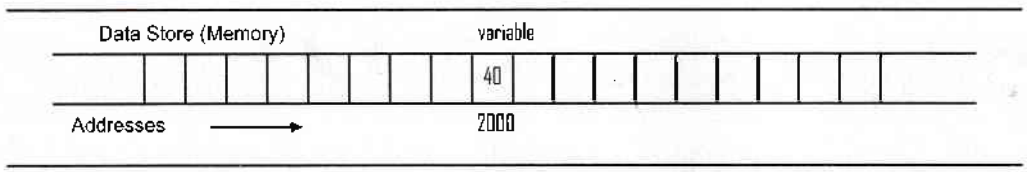


Figure 1.2 | Contents of memory location 2000 after the evaluation of expression variable=**variable**+20

Remember it as:
The **l-value** refers to the location value, i.e. the location of the object, and the **r-value** refers to the read value, i.e. the value of the object.

Forward Reference: Expressions and operators (Chapter 2).

1.11 Variables and Constants

Variables and **constants** are two most commonly used terms in a programming language.

1.11.1 Variables

A **variable** is an entity whose value can vary (i.e. change) during the execution of a program. The value of a variable can be changed because it has a modifiable l-value. Since it has a modifiable l-value, it can be placed on the left side of the assignment operator. Note that only the entities that have modifiable l-values can be placed on the left side of the assignment operator. The variable can also be placed on the right side of the assignment operator. Hence, it has an r-value too. Thus, a variable has both an l-value and an r-value.

1.11.2 Constants

A **constant** is an entity whose value remains the same throughout the execution of a program. It cannot be placed on the left side of the assignment operator because it does not have a modifiable l-value. It can only be placed on the right side of the assignment operator. Thus, a constant has an r-value only. Constants are classified as:

1. Literal constants
2. Qualified constants
3. Symbolic constants

1.11.2.1 Literal Constant

Literal constant or just **literal** denotes a fixed value, which may be an integer, floating point number, character or a string. The type of literal constant is determined by its value. Literal constants are of the following types:

1. Integer literal constant
2. Floating point literal constant
3. Character literal constant
4. String literal constant

1.11.2.1.1 Integer Literal Constant

Integer literal constants are integer values like -1, 2, 8, etc. The rules for writing integer literal constants are as follows:

1. An integer literal constant must have at least one digit.
2. It should not have any decimal point.
3. It can be either positive or negative. If no sign precedes an integer literal constant, then it is assumed to be positive.
4. No special characters (even underscore) and blank spaces are allowed within an integer literal constant.
5. If an integer literal constant starts with 0, then it is assumed to be in an octal number system, e.g. 023 is a valid integer literal constant, which means 23 is in an octal number system and is equivalent to 19 in the decimal number system.
6. If an integer literal constant starts with 0x or 0X, then it is assumed to be in a hexadecimal number system, e.g. 0x23 or 0X23 is a valid integer literal constant, which means 23 is in a hexadecimal number system and is equivalent to 35 in the decimal number system.
7. The size of the integer literal constant can be modified by using a length modifier. The length modifier can be a suffix character l, L, u, U, f or F. If the integer literal constant is terminated with l or L then it is assumed to be long. If it is terminated with u or U, then it is assumed to be an unsigned integer, e.g. 23l is a long integer and 23u is an unsigned integer. The length modifier f or F can only be used with a floating point literal constant and not with an integer literal constant.

1.11.2.1.2 Floating Point Literal Constant

Floating point literal constants are values like -23.1, 12.8, -1.8e12, etc. Floating point literal constants can be written in a **fractional form** or in an **exponential form**. The rules for writing floating point literal constants in a fractional form are as follows:

1. A fractional floating point literal constant must have at least one digit.
2. It should have a decimal point.
3. It can be either positive or negative. If no sign precedes a floating point literal constant, then it is assumed to be positive.

4. No special characters (even underscore) and blank spaces are allowed within a floating point literal constant.
5. A floating point literal constant by default is assumed to be of type double, e.g. the type of 23.45 is double.
6. The size of the floating point literal constant can be modified by using the length modifier f or F, i.e. if 23.45 is written as 23.45f or 23.45F, then it is considered to be of type float instead of double.

The following are valid floating point literal constants in a fractional form:

-2.5, 12.523, 2.5f, 12.5F

The rules for writing floating point literal constants in an exponential form are as follows:

1. A floating point literal constant in an exponential form has two parts: the mantissa part and the exponent part. Both parts are separated by e or E.
2. The mantissa can be either positive or negative. The default sign is positive.
3. The mantissa part should have at least one digit.
4. The mantissa part can have a decimal point but it is not mandatory.
5. The exponent part must have at least one digit. It can be either positive or negative. The default sign is positive.
6. The exponent part cannot have a decimal point.
7. No special characters (even underscore) and blank spaces are allowed within the mantissa part and the exponent part.

The following are valid floating point literal constants in the exponential form:

-2.5E12, -2.5e-12, 2e10 (i.e. equivalent to 2×10^{10})

1.11.2.1.3 Character Literal Constant

A **character literal constant** can have one or at most two characters enclosed within single quotes e.g. 'A', 'a', '\n', etc. Character literal constants are classified as:

1. Printable character literal constants
2. Non-printable character literal constants

1.11.2.1.3.1 Printable Character Literal Constant

All characters of source character set except quotation mark, backslash and new line character when enclosed within single quotes form a **printable character literal constant**. The following are examples of printable character literal constants: 'A', '#', '6'.

1.11.2.1.3.2 Non-printable Character Literal Constant

Non-printable character literal constants are represented with the help of **escape sequences**. An escape sequence consists of a backward slash (i.e. \) followed by a character and both enclosed within single quotes. An escape sequence is treated as a single character. It can be used[§§] in a string like any other printable character. A list of the escape sequences available in C is given in Table 1.4.

[§§]Refer Programs 1-7 and 1-9 for learning the usage of the escape sequences '\t' and '\n'.

Table 1.4 | List

S.No	Escape seq
1.	\'
2.	\"
3.	\?
4.	\\
5.	\a
6.	\b
7.	\f
8.	\n
9.	\r
10.	\t
11.	\v
12.	\0

Forward R
usage of es

1.11.2.1.4 String

A string literal co
enclosed within do
character (i.e. '\0').
more than the num
the terminating nul
due to the presence
is not counted whil
3 although it occup

Forward Re

1.11.2.2 Qualifie

Qualified constants
qualified character c

Consider a defini
of Windows environ
10. The memory loca
It is possible to mo

Table 1.4 | List of escape sequences

S.No	Escape sequence	Character value	Action on output device
1.	\'	Single quotation mark	Prints '
2.	\"	Double quotation mark (")	Prints "
3.	\?	Question mark (?)	Prints ?
4.	\\	Backslash character (\)	Prints \
5.	\a	Alert	Alerts by generating a beep
6.	\b	Backspace	Moves the cursor one position to the left of its current position
7.	\f	Form feed	Moves the cursor to the beginning of next page
8.	\n	New line	Moves the cursor to the beginning of the next line
9.	\r	Carriage return	Moves the cursor to the beginning of the current line
10.	\t	Horizontal tab	Moves the cursor to the next horizontal tab stop
11.	\v	Vertical tab	Vertical tab
12.	\0	Null character	Prints nothing

Forward Reference: Refer Question numbers 35–37 and their answers for examples on the usage of escape sequences.

1.11.2.1.4 String Literal Constant

A string literal constant consists of a sequence of characters (possibly an escape sequence) enclosed within double quotes. Each string literal constant is implicitly terminated by a null character (i.e. '\0'). Hence, the number of bytes occupied by a string literal constant is one more than the number of characters present in the string. The additional byte is occupied by the terminating null character. Thus, the empty string (i.e. "") occupies one byte in the memory due to the presence of the terminating null character. However, the terminating null character is not counted while determining the length of a string. Therefore, the length of string "ABC" is 3 although it occupies 4 bytes in the memory.

Forward Reference: Strings and character arrays (Chapter 6).

1.11.2.2 Qualified Constants

Qualified constants are created by using `const` qualifier. The following statement creates a qualified character constant named `a`:

```
const char a='A';
```

Consider a definition statement `int a=10;`. This statement allocates 2 bytes (or 4 bytes, in case of Windows environment) to `a` somewhere in the memory and initializes it with the value 10. The memory location can be thought of as a transparent box in which 10 has been placed. It is possible to modify the value of `a`. This means that it is possible to open the box and

change the value placed in it. Now, consider the statement const int a=10;. The usage of the const qualifier places a lock on the box after placing the value 10 in it. Since the box is transparent, it is possible to see (i.e. read) the value placed within the box, but it is not possible to modify the value within the box as it is locked. This is depicted in Figure 1.3.

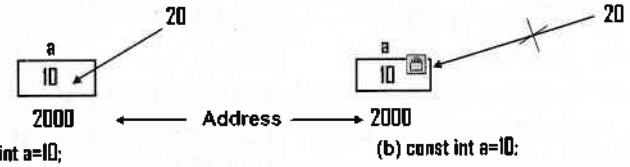


Figure 1.3 | Use of const qualifier

Since qualified constants are placed in the memory, they have l-value. However, as it is not possible to modify them, this means that they do not have a modifiable l-value, i.e. **they have a non-modifiable l-value.**

1.11.2.3 Symbolic Constants

Symbolic constants are created with the help of the define preprocessor directive.[➲] For example: #define PI 3.14124 defines PI as a symbolic constant with value 3.14124. Each symbolic constant is replaced by its actual value during the preprocessing stage.[➲]

Forward Reference: Preprocessor directives, preprocessing stage (Chapter 8).

1.12 Structure of a C Program

In general, a C program is composed of the following sections:

1. Section 1: **Preprocessor directives**
2. Section 2: **Global declarations**
3. Section 3: **Functions**

Sections 1 and 2 are optional, i.e. they may or may not be present in a C program but Section 3 is mandatory. Section 3 should always be present in a C program. Thus, it can be said that '**A C program is made up of functions**'. Look at the simple program in Program 1-1.

Line	Prog 1-1.c	Output window
1	//Comment: First C program	Hello Readers!!
2	#include<stdio.h>	
3	main()	
4	{	
5	printf("Hello Readers!!");	
6	}	

Program 1-1 | A simple program that prints "Hello Readers!!"

Program 1-1 on execution[+++] outputs Hello Readers!!. The contents of Program 1-1 are described below.

1.12.1 Comments

Line 1: is a comment. **Comments** are used to convey a message and to increase the readability of a program. They are not processed by the compiler. There are two types of comments:

1. Single-line comment
2. Multi-line comment

1.12.1.1 Single-line Comment

A **single-line comment** starts with two forward slashes (i.e. //) and is automatically terminated with the end of line. Line 1 of Program 1-1 is a single-line comment.

1.12.1.2 Multi-line Comment

A **multi-line comment** starts with /* and terminates with */. A multi-line comment is used when multiple lines of text are to be commented.

1.12.2 Section 1: Preprocessor Directive Section

Line 2: #include<stdio.h> is a preprocessor directive statement.➲ The **preprocessor directive section** is optional but you will find it in most of the C programs. In the initial phase of learning, just remember that #include<stdio.h> is a preprocessor directive statement, which includes **standard input/output** (i.e. **stdio**) **header** (.h) file. This file is to be included if standard input/output functions like printf or scanf are to be used in a program.
The following points must be remembered while writing preprocessor directives:

1. The preprocessor directive always starts with a pound symbol (i.e. #).
2. The pound symbol # should be the first non-white space character in a line.
3. The preprocessor directive is terminated with a new line character and not with a semicolon.
4. Preprocessor directives are executed before the compiler compiles the source code. These will change the source code, usually to suit the operating environment (pragma directive➲) or to add the code (include directive) that will be required by the calls to library functions.➲

Forward Reference: Preprocessor directives, pragma directive (Chapter 8), library functions (Chapter 5).

1.12.3 Section 2: Global Declaration Section

The **global declaration➲ section** is optional. This section is not present in Program 1-1. In the initial phase of learning, I am not going to use global declarations.

Forward Reference: Global declarations (Chapter 7).

[+++] Refer Section 1.13 to learn how to execute a C program.

1.12.4 Section 3: Functions Section

This section is mandatory and must be present in a C program. This section can have one or more functions. A function named main is always required. The functions section (Lines 3–6) in Program 1-1 consists of only one function, i.e. main function. Every function consists of two parts:

1. Header of the function
2. Body of the function

1.12.4.1 Header of a Function

The general form of the header of a function is

[return_type] function_name([argument_list])

The terms enclosed within square brackets are optional and might not be present in the function header. Since the name of a function is an identifier name, all the rules discussed in Section 1.5.1 for writing an identifier name are applicable for writing the function name. Line 3 in Program 1-1 specifies the header of the function main, in which the return_type➲ and the argument_list➲ are not present. The name of the function is main and it is a valid identifier name. In the initial phase of learning, I will write functions without specifying a return type and an argument list.

Writing a function without specifying a return type may lead to the generation of a warning message during the compilation but we can ignore it for the time being.

Forward Reference: return_type, argument_list (Chapter 5).

1.12.4.2 Body of a Function

The body of a function consists of a set of statements➲ enclosed within curly brackets commonly known as **braces**. Lines 4–6 in Program 1-1 form the body of main function. The body of a function consists of a set of statements. Statements are of two types:

1. Non-executable statements➲: For example: declaration statement
2. Executable statements➲: For example: printf function call statement

It is possible that no statement is present within the braces. In such a case, the program produces no output on execution. However, if there are statements written within the braces, remember that non-executable statements can only come prior to an executable statement, i.e. first non-executable statements are written and then executable statements are written. The body of main function in Program 1-1 has only one executable statement, i.e. printf function call statement.

Forward Reference: Statements, executable statement and non-executable statements (Chapter 3).

1.13 Executing a C Program

If you have finished writing the code listed in Program 1-1, follow these steps to execute your program:

1. **Save prog**
   program c
2. **Compile p**
   TC 3.0 and
   step is the
   invoke the
   to the Buil
   key. After t
   from execu
   If there are
   typing mist
   there is no
3. **Execute/ru**
   the run opt
   C 4.5, the p
   option. It ha
   and invoke
4. **See the out**
   screen. This
   option. The
   6.0, the outp

1.14 More Pr

If you have success
more programs (Pr
are errors, find out
execute them to ge

Line	Prog 1-2.c
1	//Comment: Case
2	#include<stdio.h>
3	Main()
4	{
5	int valid_name=
6	printf("%d", val
7	}

Program 1-2 | A pro

Line	Prog 1-3.c
1	//Comment: Identifi
2	#include<stdio.h>
3	main()
4	{
5	int lst_student=2
6	printf("%d", lst_
7	}

Program 1-3 | A pro

1. **Save program:** with .c extension. This will help you in retrieving the code in case the program crashes upon execution.
2. **Compile program:** Compilation can be done by going to the Compile Menu of Borland TC 3.0 and invoking the compile option available in that menu. The shortcut for this step is the Alt+F9 key. If working with Borland Turbo C 4.5, go to the Project Menu and invoke the compile option. It has the same shortcut key. In Microsoft Visual C++ 6.0, go to the Build Menu and invoke the compile option. The shortcut for this is the Ctrl+F7 key. After the compilation, look for errors and warnings. Warnings will not prevent you from executing the program and if there are any, just ignore them for the time being. If there are errors, check that you have written the code properly. There should be no typing mistake and all the characters listed in Program 1-1 should be present as such. If there is no error, Congrats!! you can now execute your program.
3. **Execute/run program:** Execution can be done by going to the Run Menu and invoking the run option in Borland Turbo C 3.0. The shortcut key is Ctrl+F9. In Borland Turbo C 4.5, the program can be executed by going to the Debug Menu and invoking the run option. It has the same shortcut key. In Microsoft Visual C++ 6.0, go to the Build Menu and invoke the run option. The shortcut key for this is Ctrl+F5.
4. **See the output:** If working with Borland Turbo C 3.0, to see the output go to the user screen. This can be done by going to the Window Menu and invoking the user screen option. The shortcut for this step is Alt+F5. In Borland TC 4.5 and Microsoft Visual C++ 6.0, the output screen will automatically pop-up.

1.14 More Programs for Startup

If you have successfully executed Program 1-1 and have gained some confidence, look at some more programs (Programs 1-2 to 1-11). Type the programs as such and compile them. If there are errors, find out the errors and rectify them. After rectification, recompile the programs and execute them to get a practical feel of all the concepts that we have discussed till now.

Line	Prog 1-2.c	Output window
1	//Comment: Case Sensitivity	Linker error
2	#include<stdio.h>	**Reasons:**
3	Main()	• C Language is case sensitive
4	{	• Main is not same as main
5	int valid_name=20;	**What to do?**
6	printf("%d", valid_name);	• Replace Main by main in line 3 and then recheck
7	}	

Program 1-2 | A program that emphasizes the case sensitivity of C language

Line	Prog 1-3.c	Output window
1	//Comment: Identifier	Compilation error
2	#include<stdio.h>	**Reason:**
3	main()	• 1st_student is not a valid identifier name
4	{	**What to do?**
5	int 1st_student=20;	• Replace it everywhere by student1 and then recheck
6	printf("%d", 1st_student);	
7	}	

Program 1-3 | A program that emphasizes the rules to write an identifier name

Line	Prog 1-4.c	Output window
1	//Comment: Keyword	Compilation error **Reason:** • if is a keyword. It cannot be used as an identifier name **What to do?** • Replace it everywhere by a valid identifier name and then recheck
2	#include<stdio.h>	
3	main()	
4	{	
5	int if=20;	
6	printf("%d", if);	
7	}	

Program 1-4 | A program that emphasizes the fact that keyword is not a valid identifier name

Line	Prog 1-5.c	Output window
1	//Comment: Semicolon is Terminator	Compilation error **Reasons:** • A statement in C is terminated with a semicolon • In line 5, declaration (actually definition) statement is not terminated with a semicolon. This leads to the compilation error **What to do?** • Place semicolon at end of line 5 and then recheck
2	#include<stdio.h>	
3	main()	
4	{	
5	int valid_name=20	
6	printf("%d", valid_name);	
7	}	

Program 1-5 | A program that emphasizes the fact that statements in C are terminated with a semicolon

Line	Prog 1-6.c	Output window
1	//Comment: printf function use	The value is 20
2	#include<stdio.h>	
3	main()	
4	{	
5	int valid_name=20;	
6	printf("The value is %d", valid_name);	
7	}	

Program 1-6 | A program that illustrates the use of printf function to print the value of an identifier

Program 1-6 upon execution outputs The value is 20. The definition statement in line 5 defines an identifier valid_name and initializes it with the value 20. This value is printed with the help of printf function in line 6. The rules for using printf function are as follows:

1. The name of printf function should be in lowercase.
2. The inputs (or arguments) to printf function are given within round or circular brackets, popularly called **parentheses**.
3. At least one input is required, and the first input to printf function should always be a string literal or an identifier of type char*.➲

Forward Reference: Pointers (Chapter 4), Character pointer, i.e. char* (Chapter 6).

4. The inputs are separated by commas.
5. If values of identifiers are to be printed with the help of printf function, the first input to printf function should be a **format string**. For example, in Program 1-6, in line 6, "The value is %d" is a format string. A **format string** consists of **format specifiers**. For example, line 6

in Program
according t
each data t
Table 1.5.

Table 1.5 | Format

S.No	Data type
1.	char
2.	int
3.	int
4.	unsigned int
5.	unsigned int
6.	unsigned int
7.	unsigned int
8.	long int
9.	short int
10.	unsigned long
11.	unsigned short
12.	float
13.	float
14.	float
15.	float
16.	float
17.	double
18.	String type
19.	Pointer type

Line	Prog 1-7.c
1	//Comment: scanf
2	#include<stdio.h>
3	main()
4	{
5	int number;
6	printf("Enter nu
7	scanf("%d",&nu
8	printf("The numb
9	}

Program 1-7 | A pr

in Program 1-6 consists of a format specifier %d. A **format specifier** specifies the format according to which the printing will be done. There is a different format specifier for each data type. Format specifier is written as %x, where x is a character code listed in Table 1.5.

Table 1.5 | Format specifiers in C language

S.No	Data type	x	Format specifier	Remark
1.	char	c	%c	Single character
2.	int	i	%i	Signed integer
3.	int	d	%d	Signed integer in decimal number system
4.	unsigned int	o	%o	Unsigned integer in octal number system
5.	unsigned int	u	%u	Unsigned integer in decimal number system
6.	unsigned int	x	%x	Unsigned integer in hexadecimal number system
7.	unsigned int	X	%X	Unsigned integer in hexadecimal number system
8.	long int	ld	%ld	Signed long
9.	short int	hd	%hd	Signed short
10.	unsigned long	lu	%lu	Unsigned long
11.	unsigned short	hu	%hu	Unsigned short
12.	float	f	%f	Signed single precision float in form of [-]dddd.dddd e.g. 22.25, -12.34
13.	float	e	%e	Singed single precision float in form of [-]d.ddddde[+/-]ddd e.g. -2.3e4, 2.25e-2
14.	float	E	%E	Same as %e, with E for exponent
15.	float	g	%g	Singed value in either e or f form, based on given value and precision
16.	float	G	%G	Same as %g, with E for exponent if e format is used
17.	double	lf	%lf	Signed double-precision float
18.	String type	s	%s	String
19.	Pointer type	p	%p	Pointer

Line	Prog 1-7.c	Output window
1	//Comment: scanf function use	Enter number 12 The number entered is 12 **Remarks:** • '\t' present in line 6 is an escape sequence and is used to create tab-spacing • Observe the tab-space between the string "Enter number" and the value 12 in the output window
2	#include<stdio.h>	
3	main()	
4	{	
5	int number;	
6	printf("Enter number\t");	
7	scanf("%d",&number);	
8	printf("The number entered is %d",number);	
9	}	

Program 1-7 | A program that illustrates the use of scanf function

Program 1-7 upon execution prompts the user to enter a value of number. In response, the user enters the value 12. The entered value is then printed by the printf function. The scanf function is used to take the input just like the printf function is used to print the output. The rules for using scanf function are as follows:

1. The name of scanf function should be in lowercase.
2. The inputs (or arguments) to scanf function are given within parentheses.
3. The first input to scanf function should always be a format string or an identifier of type char*. Ideally, the format string of a scanf function should only consist of blank separated format specifiers.

Forward Reference: Refer Question number 14 and its answer to know why the format string of a scanf function should only consist of blank separated format specifiers.

4. The inputs are separated by commas.
5. The inputs following the first input should denote l-values. For example, in line 7 of Program 1-7, the second input is &number. The symbol & is address-of operator and is used to find the l-value of its operand. Thus, &number refers to the l-value.

The scanf function takes inputs from the user according to the available format specifiers in the specified format string and stores the entered values at the specified l-values. Thus, the scanf function specified in line 7 of Program 1-7 takes an integer value (due to %d format specifier) and stores it at the l-value (i.e. &number).

Forward Reference: Address-of operator, operand (Chapter 2).

Line	Prog 1-8.c	Output window
1	//Comment: Add two numbers	Enter numbers 12 13
2	#include<stdio.h>	The sum is 25
3	main()	
4	{	
5	int number1, number2, number3;	
6	printf("Enter numbers\t");	
7	scanf("%d %d",&number1, &number2);	
8	number3 = number1+number2;	
9	printf("The sum is %d",number3);	
10	}	

Program 1-8 | A program to add two numbers entered by the user

Line	Prog 1-9.c	Output window
1	//Comment: Swap two numbers	Enter numbers 12 13
2	#include<stdio.h>	Numbers before swap 12 13
3	main()	Numbers after swap 13 12
4	{	
5	int number1, number2, number3;	

(Contd...)

6	printf("Ente
7	scanf("%d
8	printf("Num
9	number3=n
10	number1=nu
11	number2=n
12	printf("Numl
13	}

Program 1-9 | A

Line	Prog 1-10.c
1	//Comment: Sw
2	#include<stdio.h
3	main()
4	{
5	int number1,
6	printf("Enter
7	scanf("%d %
8	printf("Numb
9	number2=nu
10	number1=nun
11	number2=nu
12	printf("Numb
13	}

Program 1-10 | A

Line	Prog 1-11.c
1	//Comment: Usa
2	#include<stdio.h>
3	main()
4	{
5	printf("Charac
6	printf("Integer
7	printf("Float t
8	printf("Long ta
9	printf("Double
10	}

Program 1-11 | A

Program 1-11 mak
output is the result
6.0, the size of integ

Forward Re

```
 6    printf("Enter numbers\t");
 7    scanf("%d %d",&number1, &number2);
 8    printf("Numbers before swap %d %d\n",number1, number2);
 9    number3=number1;
10    number1=number2;
11    number2=number3;
12    printf("Numbers after swap %d %d\n",number1, number2);
13 }
```

Remark:
- '\n' present in line 8 is an escape sequence and is used to place a new line character in the output

Program 1-9 | A program to swap two numbers

Line	Prog 1-10.c	Output window
1	//Comment: Swap two numbers without using a third number	Enter numbers 12 13
2	#include<stdio.h>	Numbers before swap 12 13
3	main()	Numbers after swap 13 12
4	{	
5	int number1, number2;	
6	printf("Enter numbers\t");	
7	scanf("%d %d",&number1, &number2);	
8	printf("Numbers before swap %d %d\n",number1, number2);	
9	number2=number1+number2;	
10	number1=number2-number1;	
11	number2=number2-number1;	
12	printf("Numbers after swap %d %d\n",number1, number2);	
13	}	

Program 1-10 | A program to swap two numbers without using a third number

Line	Prog 1-11.c	Output window
1	//Comment: Usage of sizeof operator	Character takes 1 byte in memory
2	#include<stdio.h>	Integer takes 2 bytes in memory
3	main()	Float takes 4 bytes in memory
4	{	Long takes 4 bytes in memory
5	printf("Character takes %d byte in memory\n", sizeof(char));	Double takes 8 bytes in memory
6	printf("Integer takes %d bytes in memory\n", sizeof(int));	**Remark:**
7	printf("Float takes %d bytes in memory\n", sizeof(float));	• The output of the program may vary with the compiler and the working environment
8	printf("Long takes %d bytes in memory\n", sizeof(long));	
9	printf("Double takes %d bytes in memory\n", sizeof(double));	
10	}	

Program 1-11 | A program to find the size of various data types

Program 1-11 makes the use of sizeof➲ operator to find the size of data types. The specified output is the result of execution using Borland Turbo C 3.0/4.5. If it is executed using MS VC++ 6.0, the size of integer would be 4 bytes.

Forward Reference: sizeof operator (Chapter 2).

1.15 Summary

1. C is a general-purpose, block-structured, procedural, case-sensitive, free-flow, portable, high-level language.
2. There are various C standards: Kernighan & Ritchie (K&R) C standard; ANSI C/Standard C/C89 standard; ISO C/C90 standard; C99 standard.
3. ANSI C and ISO C are the most popular C standards. Most popular compilers nowadays are ANSI compliant.
4. C character set consists of letters, digits, special characters and white space characters.
5. Identifier refers to the name of an object. It can be a variable name, a label name, a typedef name, a macro name, name of a structure, a union or an enumeration.
6. Keyword cannot form a valid identifier name. The meaning of keyword is predefined and cannot be changed.
7. Every identifier (except label name) needs to be declared before its use. They can be declared by using a declaration statement.
8. The declaration statement introduces the name of an identifier along with its data type to the compiler before its use.
9. Data types are categorized as: basic data types, derived data types and user-defined data types.
10. The declaration statement can optionally have type qualifiers or type modifiers or both.
11. A type qualifier does not modify the type.
12. A type modifier modifies the base type to yield a new type.
13. Declaration is different from definition in the sense that definition in addition to declaration allocates the memory to an identifier.
14. Variables have both l-value and r-value.
15. Constants do not have a modifiable l-value. They have an r-value only.
16. C program is made up of functions.
17. C program should have at least one function. A function named `main` is always required.

Exercise Questions

Conceptual Questions and Answers

1. *What method is adopted for locating includable source files in ANSI specifications?*

   For including source files, `include` directive is used. The `include` directive can be used in two forms:

   ```
   #include<name-of-file>
   ```
   or
   ```
   #include"name-of-file"
   ```

   `#include<name-of-file>` searches the prespecified list of directories (names of include directories can be specified in IDE settings) for the source file (whose name is given within angular brackets),

and text embeds the entire content of the source file in place of itself. If the file is not found there, it will show an error 'Unable to include 'name-of-file''.
#include"name-of-file" searches the file first in the current working directory. If this search is not supported or if the search fails, this directive is reprocessed as if it reads #include<name-of-file>, i.e. search will be carried out in the prespecified list of directories. If the search still fails, it will show the error 'Unable to include 'name-of-file''.

IDE stands for Integrated Development Environment. All the tools (like text editor, preprocessor, compiler and linker) required for developing programs are integrated into one package, known as IDE.

2. *Is there any difference that arises if double quotes, instead of angular brackets are used for including standard header files?*

   If double quotes instead of angular brackets are used for the inclusion of standard header files, the search space unnecessarily increases (because in addition to the prespecified list of directories, the search will unnecessarily be carried out first in the current working directory) and thus, the time required for the inclusion will be more.

3. *Under what circumstances should the use of quotes be preferred over the use of angular brackets for the inclusion of header files, and under what circumstances is the use of angular brackets beneficial?*

   Self-created or user-defined header files should be included with double quotes because inclusion with double quotes makes files to be searched first in the current working directory (where the user has kept self-created header files) and then in the prespecified list of directories. If standard header files are to be included, angular brackets should be used because the standard header files are present in the prespecified list of directories and there is no use of searching them in the current working directory. Usage of double quotes for including standard header files will also work, but will take more time.

4. *'C is a case-sensitive language'. Therefore, does it create any difference if instead of* #include<stdio.h>, #include<STDIO.H> *is used? If no, why?*

   'C is a case-sensitive language' means that the C constructs are case sensitive (i.e. depends upon whether uppercase (like A) or lowercase (like a) is used). The name of the source file specified for inclusion is not a C construct. Whether it will be case sensitive or not depends upon the working environment. In case of DOS and Windows environment, file names are case insensitive. In Unix and Linux environment, file names are case sensitive. So, if working in DOS or Windows environment, <STDIO.H> can be used instead of <stdio.h>, it does not create any difference. But, in case of Unix or Linux environment, it does create a difference.

5. *A program file contains the following five lines of the source code:*

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

   When the program is compiled, the compiler shows the number of lines compiled to be greater than 5, why it is happening so?

   During the preprocessing stage, include preprocessor directive (the first line of source code) searches the file stdio.h in the prespecified list of directories and if the header file is found, it (the include directive) is replaced by the entire content of the header file. If the included header file contains another include directive, it will also be processed. This processing is carried out recursively

till either no include directive remains or till maximum translation limit is achieved (ISO specifies the nesting level of include files to be at most 15). Hence, one line of source code gets replaced by multiple lines of the header file. During the compilation stage, these added lines will also be compiled; hence, the compiler shows the number of lines compiled to be greater than five.

Forward Reference: Preprocessing stage (Chapter 8).

6. *Is* `int a;` *actually a declaration or a definition?*
The role of the declaration statement is to introduce the name of an identifier along with its data type (or just type) to the compiler before its use. During the declaration, no memory space is allocated to an identifier. Since `int a;` statement in addition to introducing the name and the type of identifier `a`, allocates memory to `a`, it actually becomes a definition.

7. *How are negative integral numbers stored in C?*
Internally, numbers are stored in the form of bits (i.e. **bi**nary dig**its**) and are represented in the binary number system. In the binary number system, negative numbers are not stored directly. To store both the sign and magnitude of a number, some convention for storage has to be used. In C language, the convention used for storing an integral number is **sign-two's complement representation.**
What is sign-two's complement representation?
   1. For every integral number, the **M**ost **S**ignificant **B**it (MSB) contains the sign, and the rest of the bits contain the magnitude.
   2. If the sign is positive, the MSB is 0 and if the sign is negative, the MSB is 1.
   3. If the MSB contains bit 0 (i.e. a positive number), the magnitude is in the direct binary representation.
   4. If the MSB contains bit 1 (i.e. a negative number), the magnitude is not in the direct binary representation. The magnitude is stored in two's complement form. To get the value of the magnitude, take two's complement of the stored magnitude.
   How to find two's complement of a binary number?
   Two's complement of a binary number is its one's complement plus one.
   One's complement of a binary number can be determined by negating every bit (i.e. by converting 0's to 1's and 1's to 0's). For e.g. One's complement of 100101 is 011010 (i.e. every bit is negated). Two's complement of 100101 is its one's complement plus one (i.e. 011010 + 1 = 011011). The following tables show how 200 and –200 are stored in memory:

Storage representation of 200:

Sign Bit 16 MSB	Magnitude (MSB is 0, so direct binary representation of 200)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0

Storage representation of –200:

Sign Bit 16 MSB	Magnitude (MSB is 1, so magnitude is two's complement representation of 200)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	0

Integral type consists of integer type and character type.

Forward Reference: Binary number system (Appendix A).

8. *How does the maximum value that an integral data type supports depends upon its size?*

Consider integer data type, taking 2 bytes, i.e. 16 bits in memory. The maximum value it can have is as follows:

Sign Bit 16 MSB	Magnitude (MSB is 0, so direct binary representation)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Sign Bit = 0 (means number is positive), magnitude is maximum (as all the magnitude bits have maximum value, i.e. 1). The stored number is 32767 (i.e. $2^{15}-1$).

Now, consider character data type (taking 1 byte, i.e. 8 bits in memory). The maximum value it can have is $2^7-1 = 127$. This can be shown as follows:

Sign Bit 8 MSB	Magnitude (MSB is 0, so direct binary representation)						
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1

This shows that the maximum value that an integral type can take is directly in relation to its size. That is why, if an integer variable is not able to store a value (e.g. 70000), we switch to `long` integer because `long` integer takes 32 bits in memory. Thirty-one bits will be used for storage of magnitude. Hence, the maximum value (2147483647, i.e. $2^{31}-1$) of `long` integer is far greater than the maximum value of integer (32767, i.e. $2^{15}-1$), which has only 15 bits for the storage of magnitude.

Data type as such does not take any space in memory. Objects associated with the defined identifiers take memory space according to their data types. Wherever it is referred in the text that data type takes some space in memory, it implies that the object of the specified data type takes that much memory space.

9. *What will the output of the following program segment be? (Assume that integer data type takes 2 bytes of memory.)*

```
#include<stdio.h>
main()
{
   int a=32768;
   printf("%d",a);
}
```

The output that this program snippet prints is -32768. This can be well understood if one knows how integers are stored in the memory.

If integer type takes 2 bytes in the memory, 32767 is stored as follows:

Sign Bit 16 MSB	Magnitude (MSB is 0, so direct binary representation)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Now, 32768 is 32767+1. If 1 is added in the above representation:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
															1
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

The value that comes in the memory is given in bold. The carry generated from Bit 15 has moved into Bit 16 (i.e. sign bit). Now, the sign bit becomes 1 (i.e. number becomes negative). If sign bit is 1, the magnitude of number is stored in two's complement form. The magnitude of number, i.e.

Magnitude (MSB is 1, so magnitude is in two's complement representation)														
Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

is in two's complement form. To get the value of magnitude, take two's complement of two's complemented representation of the magnitude. The magnitude can be found as follows:

		Magnitude													
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
Magnitude in two's complement form (Row 1)		**0**	0	0	**0**	**0**	0	**0**	**0**	**0**	0	**0**	**0**	0	0	0
One's complement		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Two's complement of value in row 1	1	**0**	0	0	**0**	**0**	0	0	**0**	**0**	0	0	**0**	0	0	0

Decimal equivalent of the value obtained is 2^{15} = 32768. The sign was negative, so the number becomes –32768. Hence, whenever the value of an integral data type exceeds the range, the value **wraps around** to the other side of the range.

10. *If a value assigned to an integral variable exceeds the range, the assigned value wraps around to the other side of range. Why?*

A value greater than the maximum value that the magnitude field can hold makes the sign bit 1, i.e. makes the number negative and it seems like that value has wrapped around to the other side of range; e.g. for character data type, 127 (the maximum value) can be stored as follows:

Sign Bit 8 MSB	Magnitude						
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1

If the value is further increased by 1, it becomes as follows:

Sign Bit 8 MSB	Magnitude						
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	0	0	0	0	0	0	0

The sign bit turns out to be 1. Hence, the number is negative and the magnitude is in two's complement form. To get the value of magnitude, take two's complement of 0000000. It comes out to be 10000000. This is equivalent to 128 and because the sign bit was 1, the value becomes −128 (seems as the value has wrapped around to the other side of the range).

11. *What are l-value and r-value?*

Backward Reference: Refer Section 1.10 for a description on l-value and r-value.

12. *Are nested multi-line comments by default allowed in C? If no, how can nested comments be allowed?*
No, by default nested multi-line comments are not allowed in C. Multi-line comments do not nest, i.e. we cannot have a multi-line comment within another multi-line comment. This happens because after finding /*, which marks the beginning of the multi-line comment, the contents of comments are examined only to find the characters */, which terminates the comment.
In the following example:

```
/* comment starts here
...../*nested comment starts here
.....this terminator gets associated with marker of the first line*/
.....this line will not become comment*/
```

In the first line /* is encountered and the multi-line comment starts. Now only */ will be searched. It appears in line 3. This occurrence of */ gets associated with /* of the first line, and the comment is assumed to be finished but some part of the outer comment still persists and this leads to an error.
So in the above example, the portion that gets commented out is given in bold:

```
/* comment starts here
...../*nested comment starts here
.....this terminator gets associated with marker of the first line*/
.....this line will not become comment*/
```

Nested comments can be allowed by making changes in IDE settings or by using pragma directive. Use #pragma option -C to allow nested multi-line comments.

Comment is a feature provided by almost all the programming languages. It is used to increase the readability of the program.

Forward Reference: Pragma directive (Chapter 8).

13. *How are real floating-type numbers treated in C?*

Real floating-type numbers in C, by default, are treated as that of type **double** (i.e. using double precision), so that there should be lesser loss in precision. The following piece of code on execution (using Turbo C 3.0):

```
#include<stdio.h>
main()
{
    printf("%d",sizeof(7.0));
}
```

prints 8 instead of 4. This is because 7.0 is treated as **double** (double precision) and not as **float** (single precision). To make it **float**, write it as 7.0f.

14. *The following piece of code is written to get a value from the user:*

```
main()
{
   int number;
   scanf("Enter a number %d",&number);
   printf("The number entered is %d",number);
}
```

Irrespective of the number that I enter, I get a garbage value. Why?

This problem is because of the string present inside scanf function. The scanf function cannot print a string on to the screen. Therefore, **Enter a number** will not be printed. In addition, the entered input should exactly match the format string present inside the scanf function. Therefore, if a number say 10 is entered, it does not match with the format string and the output will be garbage. However, if **Enter a number 10** is given in the input, the string in the input exactly matches the format string. The number will take the value 10, and the output will be **The number entered is 10.**

The **format specifiers** in a format string are generic terms and get matched with any value of the corresponding type. For example, %d gets matched with 10, 20, -23 or any other integer value.

15. *I have written the following piece of code keeping in mind the fact that the format string of* scanf *function should only consist of format specifiers. Still, I get a garbage value. Why?*

```
main()
{
    int number;
    printf("Enter a number\t");
    scanf("%d",number);
    printf("The number entered is %d",number);
}
```

The given piece of code gives a garbage value due to the erroneous use of scanf function. Since, the second argument to the scanf function is not an l-value of the variable number, it will not be able to place the entered value at the designated memory position. The rectified statement can be written as scanf("%d",&number);.

16.
```
main()
{
    int a,b;
    printf("Enter two numbers");
```

```
    scanf("%d %d",&a,&b);
    printf("%d + %d = %d\n",a,b,a+b);
    printf("%d / %d= %d\n",a,b,a/b);
    printf("%d % %d=%d\n",a,b,a%b);
}
```

The above piece of code on giving inputs 3 and 4 prints

3 + 4 = 7
3 / 4 = 0
3 % %d= 4

The last line is not printed correctly. How can I rectify the problem?

This problem can be rectified by using character stuffing. Instead of writing printf("%d % %d=%d\n",a,b,a%b); use printf("%d %% %d=%d\n",a,b,a%b);.

17. *What will the output of the following code snippet be and why?*

```
main()
{
    char *p="Hello\n";
    printf(p);
    printf("Hello ""Readers!..");
}
```

The output of the code snippet is as follows:

Hello
Hello Readers!..

The printf function requires the first argument to be of char* type (i.e. a string); hence, printf(p) is perfectly valid and on execution prints Hello.
Adjacent string literals get concatenated; hence, "Hello ""Readers!.." will get concatenated to form "Hello Readers!..". It will be printed by the next printf statement.

Forward Reference: Phase of translation during which the adjacent string literals are concatenated (Chapter 8).

18. *What will the output of the following code snippet be and why?*

```
main()
{
    char *p="Hello\n";
    printf(p"Readers!..");
}
```

There is a compilation error in this code snippet. This error is due to the fact that only adjacent string literals are concatenated. p is a variable and is not a string literal. It will not concatenate with the string literal "Readers!..". Hence the error.

19. *What will the output of the following piece of code be?*

```
main()
{
    int a=10,b=5,c;
    c=a/**//b;
    printf("%d",c);
}
```

The output of the code snippet will be 2. /**/ is a comment and will be neglected. Hence, the expression becomes c=a/b. Its output is 2.

20. *How are floating point numbers stored in C?*

Institute of Electrical & Electronics Engineers (IEEE) has produced a standard **(IEEE 754)** for floating point numbers. The standard specifies how single precision (4 bytes, i.e. 32 bits) and double precision (8 bytes, i.e. 64 bit) floating point numbers are represented.
An IEEE single-precision floating point number is stored in 4 bytes (32 bits). The MSB is Sign-bit, the next 8 bits are the Exponent bits 'E' and the final 23 bits are the fraction bits 'F'.

How is a floating point number stored?
Look at the following example to understand the concept. To store 5.75:

1. **Convert 5.75 from the decimal number system (DNS) to the binary number system (BNS).** The integer part 5 in DNS is equivalent to 101 in BNS.
   The fractional part 0.75 in DNS is equivalent to 0.110 in BNS.
   Therefore, 5.75 in DNS is equivalent to 101.110 in BNS.
2. **The straight binary representation of a floating point number is normalized to make it IEEE 754 compliant.** Normalized numbers are represented in the form of 1.ffffff.....ffff (f is binary digit) * 2^p, where p is the exponent. In a normalized number, the integer part is always 1. The decimal point is adjusted by selecting a suitable value of exponent, i.e. p. 101.110 in the normalized form is expressed as 1.01110 * 2^2.

   The value after the decimal point is stored in 23 fraction bits and the integer value is not stored (as it is always 1 in all normalized numbers, so there is no need to store it). So, in 1.01110 * 2^2, only 01110 is stored in 23 bits as fraction.
3. **The exponent is biased with a magic number 127_{10},** i.e. 127 is added to the exponent to make it 129. The binary equivalent of 129 (i.e. 10000001) is stored in 8 bits reserved for the storage of the exponent.

Thus, 5.75 is stored as follows:

Why are exponents biased with magic number 127_{10}?
Exponents are biased with magic number 127_{10}, so that floating point numbers can be compared for equality, greater than or less than.
Suppose exponents are not biased with magic number 127_{10}. Instead, sign-two's complement representation is used to store the value of the exponent. If such a representation is used:
2.0, i.e. 1.0 * 2^1 will be stored as follows:

0.5, i.e. 1.0 * 2^{-1} will be stored as follows:

Now, if it is checked that whether 2.0 > 0.5, it turns out to be false as 0.5 is stored as a greater value than the value of 2.0.

Now, consider that exponents are biased with the magic number 127_{10}.
2.0 is stored as: Sign Bit = 0, Exponent = 1000 0000 (128 = 1+127), Fraction = 00.....0000
0.5 is stored as: Sign Bit = 0, Exponent = 0111 1110 (126 = −1+127), Fraction = 00.....0000

It can be shown as follows:

2.0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0.5	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Value	S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
		8-bits for exponent								23-bits for mantissa																						

2.0 is stored as a greater value than 0.5. Hence, greater than operator will give the correct result.

Conclusion with another example: To find storage representation of 0.4:

1. Convert 0.4 to binary.
   0.4 * 2 = 0.8
   0.8 * 2 = 1.6
   0.6 2 = 1.2
   0.2 * 2 = 0.4
   0.4 * 2 = 0.8; this sequence repeats.
   Therefore, 0.4= **0.01100110011001100**...
2. Normalize **0.0110011001100**... After normalization it can be written as $1.10011001100...*2^{-2}$.
3. Exponent is biased with the magic number 127. Therefore, the exponent becomes −2+127=125.
   Its binary equivalent is 0111 1101.
   Hence, 0.4 will be stored as follows:

0	0	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								23-bits for mantissa																							

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

21.
```
main()
{
   printf("%d %d %d %d",72,072,0x72,0X72);
}
```

22.
```
main()
{
   printf("%d %o %x",72,72,72);
}
```

23.
```
main()
{
   printf("%i %i %i %i",72,072,0x72,0X72);
}
```

```
24. main()
    {
       printf("%05d,%5d,%-5d",32,32,32);
    }
25. main()
    {
       printf("%6.3f,%06.3f,%09.3f,%-09.3f,%6.0f,%6.0f",45.6,45.6,45.6,45.6,45.6,4.45.6);
    }
26. main()
    {
       int a=32768;
       unsigned int b=65536;
       printf("%d %d",a,b);
    }
27. main()
    {
       char a=128;
       unsigned char b=256;
       printf("%d %d\n",a,b);
    }
28. main()
    {
       float a=3.5e38;
       double b=3.5e309;
       printf("%f %lf",a,b);
    }
29. main()
    {
       printf("%d %c",'A','A');
    }
30. main()
    {
       printf("char occupies %d byte\n", sizeof(char));
       printf("int occupies %d bytes\n", sizeof(int));
       printf("float occupies %d bytes", sizeof(float));
    }
31. main()
    {
       printf("bytes occupied by '7'=%d\n",sizeof('7'));
       printf("bytes occupied by 7=%d\n",sizeof(7));
       printf("bytes occupied by 7.0=%d",sizeof(7.0));
    }
32. main()
    {
       printf("%d",sizeof('\n'));
    }
```

```
33. main()
    {
       printf("%d %c"
    }
34. main()
    {
       printf("%d %d
       printf("%d %d
    }
35. main()
    {
       printf("\nab");
       printf("\bsi");
       printf("\rha");
    }
36. main()
    {
       printf("c:\tc\bi
    }
37. main()
    {
       printf("c:\\tc\\
    }
38. main()
    {
       printf("hello.wor
       ");
    }
39. main()
    {
       printf("hello.wor
       ");
    }
40. main()
    {
       char *p="Welcom
       printf(p);
    }
```

Multiple-choice Qu

41. The primary use
    a. System prog
    b. General-pur

33.
```
main()
{
   printf("%d %c");
}
```

34.
```
main()
{
   printf("%d %d %d %d %d\n",sizeof(032),sizeof(0x32),sizeof(32),sizeof(32U),sizeof(32L));
   printf("%d %d %d",sizeof(32.4),sizeof(32.4f),sizeof(32.4F));
}
```

35.
```
main()
{
   printf("\nab");
   printf("\bsi");
   printf("\rha");
}
```

36.
```
main()
{
   printf("c:\tc\bin");
}
```

37.
```
main()
{
   printf("c:\\tc\\bin");
}
```

38.
```
main()
{
   printf("hello,world
   ");
}
```

39.
```
main()
{
   printf("hello,world\
   ");
}
```

40.
```
main()
{
   char *p="Welcome!.."" to C programming";
   printf(p);
}
```

Multiple-choice Questions

41. The primary use of C language was intended for

    a. System programming
    b. General-purpose use
    c. Data processing
    d. None of these

42. C is a/an
    a. Assembly-level language
    b. Machine-level language
    c. High-level language
    d. None of these

43. C is a
    a. General-purpose language
    b. Case-sensitive language
    c. Procedural language
    d. All of these

44. Which of the following cannot be the first character of the C identifier?
    a. A digit
    b. A letter
    c. An underscore
    d. None of these

45. Which of the following cannot be used as an identifier?
    a. Variable name
    b. Constant name
    c. Function name
    d. Keyword

46. Which of the following is not a basic data type?
    a. char
    b. float
    c. long
    d. double

47. Which of the following is not a type modifier?
    a. long
    b. unsigned
    c. signed
    d. double

48. Which of the following is a type qualifier?
    a. const
    b. signed
    c. long
    d. short

49. Which of the following is used to make an identifier a constant?
    a. const
    b. signed
    c. volatile
    d. None of these

50. Which of the following have both l-value and r-value?
    a. Variables
    b. Constants
    c. Both variables and constants
    d. None of these

51. Which of the following is not a C keyword?
    a. typedef
    b. enum
    c. volatile
    d. type

52. Qualified constant can be
    a. Initialized with a value
    b. Assigned a value
    c. Both initialized and assigned
    d. Neither initialized nor assigned

53. Which of the following is not a valid literal constant?
    a. 'A'
    b. 1.234
    c. "ABC"
    d. None of these

54. Which of the following is not a valid floating point literal constant?
    a. +3.2e-5
    b. 4.1e8
    c. -2.8e2.3
    d. +325.34

55. By default, an
    a. A float
    b. A double

56. Which of the
    a. \r
    b. \a

57. Escape seque
    a. /
    b. \

58. Single-line co
    a. //
    b. End of lin

59. The maximur
    a. 0
    b. 1

60. Which of the
    a. New line
    b. Backslash

61. Attributes tha
    a. Its name a
    b. Its value a

62. In the assignm
    assignment sy
    a. Location (
    b. Value (r-va

63. Which one is a
    a. Array typ
    b. Pointer ty

64. In C language
    a. Structural
    b. Name equ

65. In the assignm
    the assignmen
    a. Location (
    b. Value (r-va

66. If specific impl
    the maximum
    a. 32767
    b. 32768

55. By default, any real constant in C is treated as
    a. A float
    b. A double
    c. A long double
    d. Depends upon the memory model

56. Which of the following is not a valid escape sequence?
    a. \r
    b. \a
    c. \v
    d. \m

57. Escape sequence begins with
    a. /
    b. \
    c. %
    d. -

58. Single-line comment is terminated by
    a. //
    b. End of line
    c. */
    d. None of these

59. The maximum number of characters in a character literal constant can be
    a. 0
    b. 1
    c. 2
    d. Any number

60. Which of the following character is not a printable character?
    a. New line character
    b. Backslash character
    c. Quotation mark
    d. All of these

61. Attributes that characterize variables in C language are
    a. Its name and location in the memory
    b. Its value and its type
    c. Its storage class
    d. All of these

62. In the assignment statement x=x+1; the meaning of the occurrence of the variable x to the left of the assignment symbol is its
    a. Location (l-value)
    b. Value (r-value)
    c. Type
    d. None of these

63. Which one is an example of derived data type?
    a. Array type
    b. Pointer type
    c. Function type
    d. All of these.

64. In C language, which method is used for determining the type equivalence?
    a. Structural equivalence
    b. Name equivalence
    c. Both of these
    d. None of these

65. In the assignment statement x=x+1; the meaning of the occurrence of the variable x to the right of the assignment symbol is its:
    a. Location (l-value)
    b. Value (r-value)
    c. Type
    d. None of these

66. If specific implementation of C language uses 2 bytes for the storage of integer data type, what is the maximum value that an integer variable can take?
    a. 32767
    b. 32768
    c. -32768
    d. 65535

67. If specific implementation of C language uses 2 bytes for the storage of integer data type, then what is the minimum value that an integer variable can take?
    a. –32767
    b. –32768
    c. 0
    d. None of these

68. Which of the following format specifier is used for printing an integer value in octal format?
    a. %x
    b. %X
    c. %o
    d. %i

69. How many bytes are occupied by the string literal constant "xyz" in the memory?
    a. 1
    b. 2
    c. 3
    d. 4

70. The variables and constants of which of the following type cannot be declared?
    a. int**
    b. int(*)[]
    c. void
    d. float

Outputs and Explanations to Code Snippets

21. 72 58 114 114

Explanation:

All the outputs are desired in the decimal number system because of %d specifier. Now, 72 is a decimal number, 072 is an octal number equivalent to 58 in the decimal number system, 0x72 and 0X72 are hexadecimal numbers equivalent to 114 in the decimal number system. Hence, the output is 72 58 114 114.

22. 72 110 48

Explanation:

72 is to be printed in the decimal (%d specifier), the octal (%o specifier) and the hexadecimal number system (%x specifier). The octal equivalent of 72 is 110 and the hexadecimal equivalent of 72 is 48. Hence, the output is 72 110 48.

23. 72 58 114 114

Explanation:

%i specifier is used for integers. By default, it will output integer in the decimal number system as it is the most commonly used number system.

24. 00032, 32,32

Explanation:

In the given format string, width specifiers are used along with the format specifiers. Width specifier sets the minimum width for an output value.
%5d means output will be minimum 5 columns wide and will be right justified.
%-5d means output will be minimum 5 columns wide and will be left justified.
%05d means output will be minimum 5 columns wide, right justified, and the blank columns will be padded by zeros.

0	0	0	3	2	,				3	2	,	3	2			
%05d (*cw = 5, rj, padding of 0's)						%5d (*cw = 5, rj)						%–5d (*cw = 5, - is used for lj)				

*cw is **column width, rj is right justified and lj is left justified.**
Hence, the output is 00032. 32.32.

25. 45.600,45.600,00045.600,45.600 . 45. 46

Explanation:

In the given format string, width specifiers and precision specifiers are used along with the format specifiers. Precision specification always begins with a period to separate it from the preceding width specifier.
%6.3f means output is 6 columns wide. 3 is the number of digits after decimal. It is shown as

4	5	.	6	0	0

%06.3f means output is 6 columns wide. 3 is the number of digits after decimal. 0 means blank spaces are to be padded by zeros. It is shown as

4	5	.	6	0	0

%09.3f means output is 9 columns wide. 3 is the number of digits after decimal. 0 means blanks spaces are to be padded by zeros. By default, the output is right justified. It is shown as

0	0	0	4	5	.	6	0	0

%-09.3f means output is 9 columns wide. 3 is the number of digits after decimal. Since – is used, the output will be left justified. Here the output shows blank spaces, and padding by zeros has not been done because only 3 digits can be printed after the decimal. It is shown as

4	5	.	6	0	0			

%6.0f means output is 6 columns wide. 0 is the number of digits after the decimal. Rounding off will take place and 45.4 will be rounded to 45. The output will be

				4	5

%6.0f means output is 6 columns wide. 0 is the number of digits after the decimal. Rounding off will take place and 45.6 will be rounded to 46. It is shown as

				4	6

26. -32768 0

Explanation:

Since the assigned values exceed the maximum value of integer type and unsigned integer type, the values wrap around to the other side of the range. Hence, the outputs are -32768 (minimum value of signed integer) and 0 (minimum value of unsigned integer).

27. -128 0

Explanation:

Since the assigned values exceed the maximum value of character type and unsigned character type, the values wrap around to the other side of the range. Hence, the outputs are -128 (minimum value of signed character) and 0 (minimum value of unsigned character).

28. +INF +INF

Explanation:

Range wraps around only in case of integral data type. Wrap around does not occur in case of float and double data types. In case of float and double data types, if the value falls outside the range +INF or -INF is the output.

+INF refers to +Infinity and –INF refers to –Infinity.

29. 65 A

Explanation:

Integers and characters together form integral data type and are not separated internally. If characters are printed using %d specifier, it gives the ASCII equivalent of the character. Hence, the output is 65, ASCII code of 'A'. If %c specifier is used, it prints the character, i.e. 'A'.

30. char occupies 1 byte
int occupies 2 bytes
float occupies 4 bytes

Explanation:

sizeof operator outputs the size of the given data type.

31. bytes occupied by '7'=1
bytes occupied by 7=2
bytes occupied by 7.0=8

Explanation:

sizeof operator can also take constant as input and returns the number of bytes required by the data type of that constant as output. 7.0 is a real floating number and will be treated as double type. Hence, sizeof(7.0) gives 8.

32. 1

Explanation:

'\n' is a character, more specifically a new line character. Hence, sizeof operator returns 1, i.e. the size of a character.

33. Garbage

Explanation:

Since format specifiers %d and %c are not linked to any value, they will output garbage. This is only applicable for %d and %c specifiers. If %f specifier is not linked, it leads to abnormal program termination.

34. 22224
844

Explanation:

032, 0x32, 32 all are integers in different number systems. 32U is an unsigned integer. Hence, their size is 2. 32L is a long integer of size 4. 32.4 is a real floating-type number and is treated as a double of size 8. 32.4f and 32.4F are float and their size is 4. Hence, the output.

35.
hai

Explanation

'\n' is a new
backspace ch
the output b
starting of th
"as". Hence,

36. c: in

Explanation:

'\t' is a tab ch
from "c:". Th
Hence, the ou

37. c:\tc\bin

Explanation:

The usage of
tab character
acter.

38. Compilation erro

Explanation:

String cannot

39. hello,world

Explanation:

Each instance
leted. This pr
source lines.
such a splice.
Physical sour

```
main()
{
    printf("hello,wo
    ");
}
```

after splicing

```
main()
{
    printf("hello,wo
}
```

Logical source

Forward Re

35. ←Blank line
hai

Explanation:

'\n' is a new line character. Due to '\n', cursor appears in a new line and "ab" gets printed. '\b' is a backspace character. It places the cursor below the character 'b' and "si" gets printed. Therefore, the output becomes "asi". '\r' is a carriage return character. It will make the cursor return to the starting of the same line. The cursor will be placed below 'a'. "ha" gets printed and overwrites "as". Hence, the output becomes "hai".

36. c: in

Explanation:

'\t' is a tab character and '\b' is a backspace character. Due to '\t' character 'c' gets tab separated from "c:". The output becomes "c: c". '\b' makes character 'c' to erase and "in" gets printed. Hence, the output becomes "c: in".

37. c:\tc\bin

Explanation:

The usage of an extra backslash is known as character stuffing. Now '\t' will not be treated as a tab character and will actually get printed. Similarly '\b' will not be treated as a backspace character.

38. Compilation error

Explanation:

String cannot span multiple lines in this way. Hence, the error.

39. hello,world

Explanation:

Each instance of the backslash character (\) immediately followed by a new line character is deleted. This process is known as **line splicing.** Physical source lines are spliced to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice.

Physical source lines

```
main()
{
   printf("hello,world\
   ");
}
```

after splicing will form the following logical source lines:

```
main()
{
   printf("hello,world");
}
```

Logical source lines are processed by the compiler. Hence, on execution, hello,world is the output.

Forward Reference: Phase of translation during which line splicing is carried out (Chapter 8).

40. Welcome!..to C programming

Explanation:

Adjacent string literals get concatenated. Hence, "Welcome!.." "to C programming" gets concatenated and becomes "Welcome!..to C programming". printf needs first argument to be of char* type. In printf(p) this constraint is satisfied as p is the only argument and is of char* type. Hence, the value of p, i.e. Welcome!..to C programming gets printed.

Answers to Multiple-choice Questions

41. a 42. c 43. d 44. a 45. d 46. c 47. d 48. a 49. a 50. a 51. d 52. a 53. d 54. c
55. b 56. d 57. b 58. b 59. c 60. d 61. d 62. a 63. d 64. a 65. b 66. a 67. b 68. c
69. d 70. c

Programming Exercises

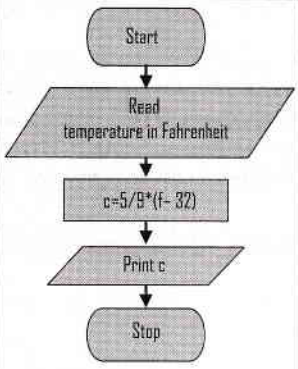
Program 1 | Convert the temperature given in Fahrenheit to Celsius

Algorithm:
Step 1: Start
Step 2: Read the temperature given in Fahrenheit (f)
Step 3: Temperature in Celsius (c) = 5/9*(f-32)
Step 4: Print temperature in Celsius
Step 5: Stop

PE 1-1.c

```
1  //Convert temperature in Fahrenheit to
2   //Celsius
3  #include<stdio.h>
4  main()
5  {
6     float f,c;
7     printf("Enter temperature in Fahrenheit\t");
8     scanf("%f",&f);
9     c=5.0/9.0*(f-32);
10    printf("Temperature in Celsius is %6.2f",c);
11 }
```

Flowchart depicting the flow of control in program



Output window

```
Enter temperature in Fahrenheit  106
Temperature in Celsius is 41.11
```

Flowchart is a graphical representation that depicts the flow of program control.

Forward Reference: Algorithms and Flowcharts (Appendix B).

Program 2 | Find t

Algorithm:
Step 1: Start
Step 2: Read the radi
Step 3: Circumferenc
Step 4: Area area = 22
Step 5: Print circumfe
Step 6: Stop

PE 1-2.c

```
1  //Circumference and
2  #include<stdio.h>
3  main()
4  {
5     float r, cir, area;
6     printf("Enter the ra
7     scanf("%f",&r);
8     cir=2*22.0/7*r;
9     area=22.0/7*r*r;
10    printf("Circumferen
11    printf("Area of circ
12 }
```

Program 3 | Find th

Algorithm:
Step 1: Start
Step 2: Read numbers
Step 3: Average avg =
Step 4: Print avg
Step 5: Stop

PE 1-3.c

```
1  //Average of three nur
2  #include<stdio.h>
3  main()
4  {
5     float no1, no2, no3,av
6     printf("Enter three n
7     scanf("%f %f %f",&r
8     avg=(no1+no2+no3)/
9     printf("Average of nu
10 }
```

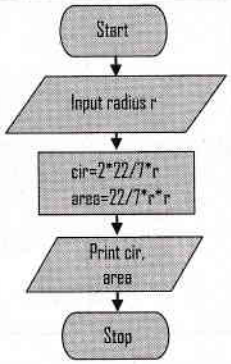
Program 2 | Find the area and circumference of a circle with radius r

Algorithm:
Step 1: Start
Step 2: Read the radius of circle (r)
Step 3: Circumference cir = 2*22/7*r
Step 4: Area area = 22/7*r*r
Step 5: Print circumference and area
Step 6: Stop

PE 1-2.c

```
1  //Circumference and area of circle
2  #include<stdio.h>
3  main()
4  {
5     float r, cir, area;
6     printf("Enter the radius of circle\t");
7     scanf("%f",&r);
8     cir=2*22.0/7*r;
9     area=22.0/7*r*r;
10    printf("Circumference of circle is %6.2f\n",cir);
11    printf("Area of circle is %6.2f\n",area);
12 }
```

Flow chart depicting the flow of control in program



Output window

```
Enter the radius of circle   5
Circumference of circle is 31.43
Area of circle is 78.57
```

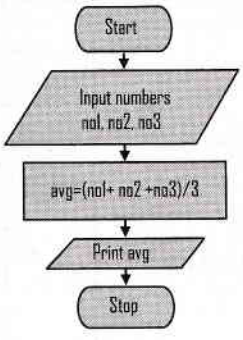
Program 3 | Find the average of three numbers

Algorithm:
Step 1: Start
Step 2: Read numbers no1, no2, no3
Step 3: Average avg = (no1+no2+no3)/3
Step 4: Print avg
Step 5: Stop

PE 1-3.c

```
1  //Average of three numbers
2  #include<stdio.h>
3  main()
4  {
5     float no1, no2, no3,avg;
6     printf("Enter three numbers\t");
7     scanf("%f %f %f",&no1, &no2, &no3);
8     avg=(no1+no2+no3)/3;
9     printf("Average of numbers is %6.2f\n",avg);
10 }
```

Flow chart depicting the flow of control in program



Output window

```
Enter three numbers   12 11 14
Average of numbers is 12.33
```

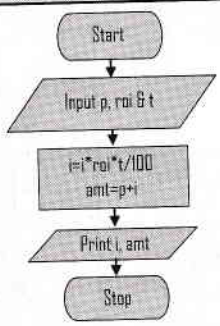
Program 4 | Simple Interest and the Maturity Amount

Algorithm:
Step 1: Start
Step 2: Read principle (p), rate of interest (roi), time period (t)
Step 3: Interest i = p*roi*t/100
Step 4: Amount amt = p+i
Step 5: Print i, amt
Step 6: Stop

PE 1-4.c

```
1  //Simple Interest
2  #include<stdio.h>
3  main()
4  {
5     float p, roi, t, i, amt;
6     printf("Enter principle, rate and time\t");
7     scanf("%f %f %f",&p, &roi, &t);
8     i=p*roi*t/100;
9     amt=p+i;
10    printf("Interest is %6.2f\n",i);
11    printf("Amount is %6.2f\n",amt);
12 }
```

Flow chart depicting the flow of control in program



Output window

```
Enter principle, rate and time  1000 7 2
Interest is 140.00
Amount is 1140.00
```

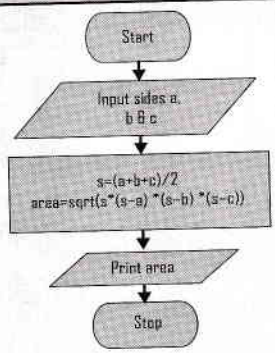
Program 5 | Find area of a triangle whose sides are a, b and c

Algorithm:
Step 1: Start
Step 2: Read sides a, b and c of triangle
Step 3: s = (a+b+c)/2
Step 4: area = sqrt(s*(s–a)*(s–b)*(s–c))
Step 5: Print area
Step 6: Stop

PE 1-5.c

```
1  //Area of a triangle
2  #include<stdio.h>
3  #include<math.h>
4  main()
5  {
6     float a, b, c, s, area;
7     printf("Enter the sides of a triangle\t");
8     scanf("%f %f %f",&a, &b, &c);
9     s=(a+b+c)/2;
10    area=sqrt(s*(s-a) *(s-b) *(s-c));
11    printf("Area of triangle is %6.2f sq. units",area);
12 }
```

Flow chart depicting the flow of control in program



Output window

```
Enter the sides of a triangle  12 5 14
Area of triangle is 29.23 sq. units
```

Program 6 | The ve
from km/hr to m/sec

Algorithm:
Step 1: Start
Step 2: Input the veloc
Step 3: velocity in m/s
Step 4: Print velocity i
Step 5: Stop

PE 1-6.c

```
1  //Convert units of vel
2  #include<stdio.h>
3  main()
4  {
5     float velk, velm;
6     printf("Enter velocit
7     scanf("%f",&velk);
8     velm=velk*5/18;
9     printf("Equivalent v
10 }
```

Program 7 | An obj
and the acceleration

Algorithm:
Step 1: Start
Step 2: Input the initia
Step 3: Input the time
Step 4: Velocity v = u+
Step 5: Print value of
Step 6: Stop

PE 1-7.c

```
1  //Compute velocity af
2  #include<stdio.h>
3  main()
4  {
5     float u, v,a, t;
6     printf("Enter the va
7     scanf("%f",&u);
8     printf("Enter the ac
9     scanf("%f",&a);
10    printf("Enter the ti
11    scanf("%f",&t);
12    v=u+a*t;
13    printf("Velocity afte
14 }
```

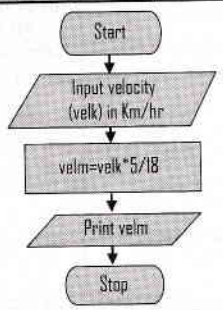
Program 6 | The velocity of an object is given in km/hr. Write a C program to convert the given velocity from km/hr to m/sec

Algorithm:
Step 1: Start
Step 2: Input the velocity (velk) given in km/hr
Step 3: velocity in m/sec (velm) = velk*5/18
Step 4: Print velocity in m/sec (velm)
Step 5: Stop

PE 1-6.c

```
1  //Convert units of velocity
2  #include<stdio.h>
3  main()
4  {
5     float velk, velm;
6     printf("Enter velocity in Km/hr\t");
7     scanf("%f",&velk);
8     velm=velk*5/18;
9     printf("Equivalent velocity is %f m/sec",velm);
10 }
```

Flow chart depicting the flow of control in program



Output window

```
Enter velocity in km/hr   12
Equivalent velocity is 3.333333 m/sec
```

Program 7 | An object undergoes uniformly accelerated motion. The initial velocity (u) of the object and the acceleration (a) are known. Write a C program to find the velocity (v) of the object after time t

Algorithm:
Step 1: Start
Step 2: Input the initial velocity (u) and acceleration (a) of the object in SI units
Step 3: Input the time (t) after which velocity is to be computed
Step 4: Velocity v = u+a*t
Step 5: Print value of velocity (v)
Step 6: Stop

PE 1-7.c

```
1  //Compute velocity after time t
2  #include<stdio.h>
3  main()
4  {
5     float u, v,a, t;
6     printf("Enter the value of initial velocity in m/s\t");
7     scanf("%f",&u);
8     printf("Enter the amount of acceleration\t");
9     scanf("%f",&a);
10    printf("Enter the time in sec.\t");
11    scanf("%f",&t);
12    v=u+a*t;
13    printf("Velocity after %4.2f sec is %4.2f m/s",t,v);
14 }
```

Output window

```
Enter the value of initial velocity in m/s   2.4
Enter the amount of acceleration   4
Enter the time in sec.   2
Velocity after 2.00 sec is 10.40 m/s
```

Program 8 | A year approximately consists of 3.156×10^7 seconds. Write a C program that accepts your age in years and convert it into equivalent number of seconds

Algorithm:
Step 1: Start
Step 2: Enter age (age) in years
Step 3: Age in seconds (age_in_sec) = 3.156 × 10^7*age
Step 4: Print equivalent age in seconds (age_in_sec)
Step 5: Stop

PE 1-8.c

```
1  //Equivalent age in seconds
2  #include<stdio.h>
3  main()
4  {
5     int age;
6     float age_in_sec;
7     printf("How old are you (years)?\t");
8     scanf("%d",&age);
9     age_in_sec=3.156E7*age;
10    printf("Your age in seconds is %5.2E",age_in_sec);
11 }
```

Output window

```
How old are you (years)?   18
Your age in seconds is 5.68E+08
```

Remark:

- %E specifier is used to print floating point value in exponent form

Test Yourself

1. Fill in the bla
   a. The C lan
   b. An identi
   c. One of th
   d. int var; is a
   e. _____
   f. Constant
   g. _____
   h. Non-prin
   i. The first
   j. Floating
   k. A C prog
   l. Every sta
   m. The printf
      _____
   n. The amo
      _____
   o. The argu
2. State whether
   a. C is a cas
      ters and
   b. An identi
   c. All the va
   d. Commen
      produce
   e. Keyword
   f. int a=20, b=
   g. A type qu
   h. Constant
   i. A charact
      quotes.
   j. The scanf
3. Determine wh
   a. main
   b. MAIN
   c. NewStud
   d. New_Stu
   e. a+b
   f. for_while
   g. 123abc
   h. abc123
   i. name&nu
   j. _classnur
   k. _number

ogram that accepts your

floating point value in

Test Yourself

1. Fill in the blanks in each of the following:
   a. The C language was developed by _____.
   b. An identifier name in C starts with a _____.
   c. One of the most important attributes of an identifier is its _____.
   d. `int var;` is a _____ statement.
   e. _____ is a data object locator.
   f. Constants do not have _____ value.
   g. _____ qualifier is used to create a qualified constant.
   h. Non-printable character constants are represented with the help of _____.
   i. The first argument of `printf` function should always be a _____.
   j. Floating point literal constant by default is assumed to be of type _____.
   k. A C program is made up of _____.
   l. Every statement in C is terminated with a _____.
   m. The `printf` function prints the value according to the _____ specified in the _____.
   n. The amount of memory that an object of a data type would take can be found by using _____ operator.
   o. The arguments following the first argument in a `scanf` function should denote _____.
2. State whether each of the following is true or false. If false, explain why.
   a. C is a case-sensitive language, which means that it distinguishes between uppercase characters and lowercase characters.
   b. An identifier name in C cannot start with a digit.
   c. All the variable names must be declared before they are used in a C program.
   d. Comments play an important role in a C program and are processed by the C compiler to produce an executable code.
   e. Keyword or a reserved word cannot be used as a valid identifier name.
   f. `int a=20, b=30, c;` is an example of a longhand declaration statement.
   g. A type qualifier modifies the base type to yield a new type.
   h. Constants have both l-value and r-value.
   i. A character literal constant can have one or at most two characters enclosed within single quotes.
   j. The `scanf` function can be used to read only one value at a time.
3. Determine which of the following are valid identifier names in C:
   a. main
   b. MAIN
   c. NewStudent
   d. New_Student
   e. a+b
   f. for_while
   g. 123abc
   h. abc123
   i. name&number
   j. _classnumber
   k. _number_

4. Determine which of the following are valid constants:
   a. "ABC"
   b. '#'
   c. Abc
   d. 1,234
   e. –22.124
   f. 1.23E-2.0
   g. 0x2AG
   h. '\r'
   i. 0x23
   j. 23L
   k. –7.0f
5. Identify and correct the errors in each of the following statements:
   a. int a=10, int b=20;
   b. int a=10, float b=2.5;
   c. int a=23u, b=2f;
   d. const int number=100;
      number=500;
   e. printf(1,2,3);
   f. Printf("To err is human");
   g. printf("%d %d" no1, no2);
   h. printf("Humans learn by making mistakes")
   i. scanf("%d %d", no1, no2);
   j. first_value+second_value=sum_of_values

2

OPERATORS AND EXPRESSIONS

Learning Objectives

In this chapter, you will learn about:

- Operands and operators
- Expressions
- Simple expressions and compound expressions
- How compound expressions are evaluated
- Precedence and associativity of operators
- How operators are classified
- Classification based on number of operands
- Unary, binary and ternary operators
- Classification based on role of operator
- Arithmetic, relational, logical, bitwise, assignment and miscellaneous operators
- Rules for evaluation of arithmetic expressions
- Implicit and explicit-type conversions
- Promotions and demotions
- Conditional, comma, sizeof and address-of operator
- Combined precedence of all operators

2.1 Introduction

In Chapter 1, you have learnt about identifiers (i.e. variables and functions specifically printf and scanf functions), constants and data types. In this chapter, I will take you a step forward and tell you how to create expressions from identifiers, constants and operators. Finally, we will look at how expressions are evaluated and the intricacies involved in this evaluation process.

2.2 Expressions

An **expression** in C is made up of one or more operands. The simplest form of an expression consists of a single operand. For example, 3 is an expression that consists of a single operand, i.e. 3. Such an expression does not specify any operation to be performed and is not meaningful. In general, a meaningful expression consists of one or more operands and operators that specify the operations to be performed on operands. For example, a=2+3 is a meaningful expression, which involves three operands, namely a, 2 and 3 and two operators, i.e. = (assignment operator) and + (arithmetic addition operator). Thus, an expression is a sequence of operands and operators that specifies the computation of a value. Let us look at the fundamental constituents of an expression, i.e. operands and operators.

2.2.1 Operands

An **operand** specifies an entity on which an operation is to be performed. An operand can be a variable name, a constant, a function call or a macro name. For example, a=printf("Hello")+2 is a valid expression involving three operands, namely a variable name, i.e. a, a function call, i.e. printf("Hello") and a constant, i.e. 2.

Forward Reference: Macros (Chapter 8).

2.2.2 Operators

An **operator** specifies the operation to be applied to its operands. For example, the expression a=printf("Hello")+2 involves three operators, namely function call operator, i.e. (), arithmetic addition operator, i.e. + and assignment operator, i.e. =.

Based on the number of operators present in an expression, expressions are classified as simple expressions and compound expressions.

2.3 Simple Expressions and Compound Expressions

An expression that has only one operator is known as a **simple expression** while an expression that involves more than one operator is called a **compound expression**. For example, a+2 is a simple expression and b=2+3*5 is a compound expression. The evaluation of a simple expression is easier as compared to the evaluation of a compound expression. Since, there is more than one operator in a compound expression, while evaluating compound expressions one must determine the order in which operators will operate. For example, to determine the result of evaluation of the expression b=2+3*5, one must determine the order in which =, + and * will operate. This order determination becomes trivial in the case of evaluation of simple expressions like a+2, as there is only one operator and it has to operate in any case. The order

in which the operators will operate depends upon the **precedence** and the **associativity** of operators.

2.3.1 Precedence of Operators

Each operator in C has a **precedence** associated with it. In a compound expression, if the operators involved are of different precedence, the operator of higher precedence operates first. For example, in an expression b=2+3*5, the sub-expression 3*5 involving multiplication operator (i.e. *) is evaluated first as the multiplication operator has the highest precedence among =, + and *. The result of evaluation of an expression is an r-value. The sub-expression 3*5 evaluates to an r-value 15. This r-value will act as a second operand for an addition operator and the expression becomes b=2+15. In the resultant expression, the sub-expression 2+15 will be evaluated next as the addition operator (i.e. +) has a higher precedence than the assignment operator (i.e. =). The expression after the evaluation of the addition operator reduces to b=17. Now, there is only one operator in the expression. The assignment operator will operate and the value 17 is assigned to b.

The knowledge of precedence of operators alone is not sufficient to evaluate a compound expression in case two or more operators involved are of the same precedence. For example, in the expression b=2*3/5, the multiplication operator (i.e. *) and the division operator (i.e. /) have the same precedence. The sub-expression 2*3/5 will evaluate to 1 if the multiplication operator operates before the division operator and to 0 if the division operator operates prior to the multiplication operator. To determine which of these operators will operate first, the associativity of these operators is to be considered.

2.3.2 Associativity of Operators

In a compound expression, when several operators of the same precedence appear together, the operators are evaluated according to their **associativity**. An operator can be either left-to-right associative or right-to-left associative. The operators with the same precedence always have the same associativity. If operators are left-to-right associative, they are applied in a left-to-right order, i.e. the operator that appears towards the left will be evaluated first. If they are right-to-left associative, they will be applied in the right-to-left order. The multiplication and the division operators are left-to-right associative. Hence, in expression 2*3/5, the multiplication operator is evaluated prior to the division operator as it appears before the division operator in the left-to-right order.

Now, let us look at various operators, their classification, precedence and associativity.

2.4 Classification of Operators

The operators in C are classified on the basis of the following criteria:

1. The number of operands on which an operator operates.
2. The role of an operator.

2.4.1 Classification Based on Number of Operands

Based upon the number of operands on which an operator operates, the operators are classified as:

1. Unary operators — A **unary** operator operates on only one operand. For example, in the expression -3, - is a unary minus operator as it operates on only one operand, i.e. 3. The operand can be present towards the right of the unary operator, as in -3 or towards the left of the unary operator, as in the expression a++. Examples of unary operators are: & (address-of operator), sizeof operator, ! (logical negation), ~ (bitwise negation), ++ (increment operator), -- (decrement operator), etc.
2. Binary operators — A **binary** operator operates on two operands. It requires an operand towards its left and right. For example, in expression 2-3, - acts as a binary minus operator as it operates on two operands, i.e. 2 and 3. Examples of binary operators are: * (multiplication operator), / (division operator), << (left shift operator), == (equality operator), && (logical AND), & (bitwise AND), etc.
3. Ternary operator — A **ternary** operator operates on three operands. Conditional operator (i.e. ?:) is the only ternary operator available in C.

2.4.2 Classification Based on Role of Operator

Based upon their role, operators are classified as:

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

2.4.2.1 Arithmetic Operators

Arithmetic operations like addition, subtraction, multiplication, division, etc. can be performed by using arithmetic operators. The arithmetic operators available in C are given in Table 2.1.

Table 2.1 | Arithmetic operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among arithmetic class	Associativity
1.	+ - ++ --	Unary plus Unary minus Increment Decrement	Unary operators	Unary	Level-I (Highest)	R→L (Right-to-left)
2.	* / %	Multiplication Division Modulus	Multiplicative operators	Binary	Level-II (Intermediate)	L→R (Left-to-right)
3.	+ -	Addition Subtraction	Additive operators	Binary	Level-III (Lowest)	L→R

nly one operand. For example,
minus operator as it operates
perand can be present towards
as in -3 or towards the left of
ression a++. Examples of unary
rator), sizeof operator, ! (logical
, ++ (increment operator), --

two operands. It requires an
nt. For example, in expression
perator as it operates on two
es of binary operators are: *
vision operator), << (left shift
, && (logical AND), & (bitwise

three operands. Conditional
y operator available in C.

vision, etc. can be performed
in C are given in Table 2.1.

lence among etic class	Associativity
I est)	R→L (Right-to-left)
II nediate)	L→R (Left-to-right)
III st)	L→R

The operators within a row have the same precedence, and the order in which they are written does not matter.

The following rules are observed while evaluating arithmetic expressions:

1. The parenthesized sub-expressions are evaluated first.
2. If the parentheses are nested, the innermost sub-expression is evaluated first.
3. The precedence rules are applied to determine the order of application of operators while evaluating sub-expressions.
4. The associativity rule is applied when two or more operators of the same precedence appear in the sub-expression.
5. If the operands of a binary arithmetic operator are of different but compatible types, C automatically applies **arithmetic-type conversion** to bring the operands to a common type. This automatic-type conversion is known as **implicit-type conversion**.The result of the evaluation of an operator will be of the **common type**. The basic principle behind the implicit arithmetic-type conversion is that if operands are of different types, the lower type (i.e. smaller in size) should be converted to a higher type (i.e. bigger in size) so that there is no loss in value or precision. Since a lower type is converted to a higher type, it is said that the lower type is **promoted** to a higher type and the conversion is known as **promotion**. The following are common **arithmetic-type conversions**:
   a. If one operand is long double, the other will be converted to long double, and the result will be long double.
   b. If one operand is double, the other will be converted to double, and the result will be double.
   c. If one operand is float, the other will be converted to float, and the result will be float.
   d. If one of the operands is unsigned long int, the other will be converted to unsigned long int, and the result will be unsigned long int.
   e. If one operand is long int and the other is unsigned int, then
      i. If unsigned int can be converted to long int, then unsigned int operand will be converted as such, and the result will be long int.
      ii. Else, both operands will be converted to unsigned long int, and the result will be unsigned long int.
   f. If one of the operands is long int, the other will be converted to long int, and the result will be long int.
   g. If one operand is unsigned int, the other will be converted to unsigned int, and the result will be unsigned int.
   h. If none of the above is carried out, both the operands are converted to int.

The above-mentioned rules can be **summarized** as:
Binary arithmetic operators can be used in one of the following three different modes:

1. **Integer mode:** If both the operands of a binary arithmetic operator are of integer type, the mode of operation is said to be **integer mode** and the result will be of integer type. For example: the result of 4/3 will be 1 instead of 1.3333, as integer mode operation results in the value of integer type.

2. **Floating point mode:** If both the operands of a binary arithmetic operator are of floating point type, the mode of operation is said to be **floating point mode** and the result will be of floating point type. For example: the result of 4.0/3.0 will be 1.333333, as the result of floating point mode operation is of floating point type.

3. **Mixed mode:** If one of the operands of a binary arithmetic operator is of integer type and another operand is of floating point type, the mode of operation is said to be **mixed mode**. The operand of integer type is promoted to floating point type and the result will be of floating point type. For example: the result of 4/3.0 will be 1.333333.

Consider Program 2-1.

Line	Prog 2-1.c	Output window
1	//Arithmetic expression	The result of evaluation is 10
2	#include<stdio.h>	
3	main()	
4	{	
5	int a;	
6	a=2*3.25+((3+6)/2);	
7	printf("The result of evaluation is %d",a);	
8	}	

Program 2-1 | A program that illustrates the evaluation of an arithmetic expression

Let us look at how the expression a=2*3.25+((3+6)/2) as specified in Program 2-1 gets evaluated. The innermost parenthesized sub-expression (3+6) gets evaluated first. This sub-expression evaluates to an r-value, i.e. 9. This r-value acts as an operand for the division operator. Now, the expression reduces to a=2*3.25+(9/2). The sub-expression (9/2) gets evaluated next. Since both the operands of the division operator are of integer type, the arithmetic involved is integer arithmetic and thus, the result is an r-value of integer type, i.e. 4 instead of 4.5. The expression becomes a=2*3.25+4. Since the multiplication operator (i.e. *) has a higher precedence than the addition operator (i.e. +) and the assignment operator (i.e. =), the sub-expression 2*3.25 gets evaluated next. In this sub-expression, the arithmetic involved is mixed mode arithmetic as one of the operands is of integer type and the other is of floating point type. The operand 2 is promoted to 2.0. The result of sub-expression 2.0*3.25 turns out to be 6.50. After the evaluation of this sub-expression, the expression gets reduced to a=6.50+4. The sub-expression 6.50+4 involves mixed mode operation and is evaluated to 10.50. Finally, the expression becomes a=10.50. In this expression, the value of floating point type is assigned to a variable of integer type. The operand of floating point type (i.e. 10.50) is automatically converted to an integer type so that it can be assigned to the integer variable a. Since a higher type (i.e. float, bigger in size) is converted to a lower type (i.e. int, smaller in size), it is said that the higher type is **demoted** to the lower type and this conversion is called **demotion**. The method followed during demotion is **truncation**. Thus, 10.50 is demoted (i.e. truncated) to 10 and is assigned to a. This value of a is printed by the printf function.

The important points about the arithmetic operators are as follows:

1. The **unary plus operator** can appear only towards the left side of its operand.
2. The **unary minus operator** can appear only towards the left side of its operand.

3. **Incremen**
   a. The inc
      its oper
      the **pre**
      a++), it i
   b. The inc
      l-value.
      will be

Line	Prog 2-2.c
1	//Increment/ D
2	#include<stdio.h
3	main()
4	{
5	int a;
6	a=++2;
7	printf("The re
8	}

Program 2-2 | A
m

   c. ++a or a++
   d. The **diff**
      which th
      i. In case
         and th
      ii. In case
         uation

The differenc
in Program 2

Line	Prog 2-3.c
1	//Difference betwe
2	#include<stdio.h>
3	main()
4	{
5	int a=2, b=2,c,d;
6	c=++a;
7	d=b++;
8	printf("a=%d, b=
9	}

Program 2-3 | A pro

3. **Increment operator**
   a. The increment operator can appear towards the left side or towards the right side of its operand. If it appears towards the left side of its operand (e.g. ++a), it is known as the **pre-increment operator**. If it appears towards the right side of its operand (e.g. a++), it is known as the **post-increment operator**.
   b. The increment operator can only be applied to an operand that has a modifiable l-value. If it is applied to an operand that does not have a modifiable l-value, there will be 'L-value required' error. Try executing the code listed in Program 2-2.

Line	Prog 2-2.c	Output window
1	//Increment/ Decrement operator's operand	Compilation error "L-value required"
2	#include<stdio.h>	**Reasons:**
3	main()	• Operand of increment/decrement operator should have a modifiable l-value
4	{	• 2 is a constant and does not have modifiable l-value
5	int a;	**What to do?**
6	a=++2;	• Create a variable b, place value 2 in it and instead of ++2 write ++b
7	printf("The result of application of pre-increment operator is %d",a);	
8	}	

Program 2-2 | A program to illustrate that operand of increment/decrement operator should have a modifiable l-value

   c. ++a or a++ is equivalent to a=a+1.
   d. The **difference between pre-increment and post-increment** lies in the point at which the value of their operand is incremented.
      i. In case of the pre-increment operator, first the value of its operand is incremented and then it is used for the evaluation of expression.
      ii. In case of the post-increment operator, the value of operand is used first for the evaluation of the expression and after its use, the value of the operand is incremented.

The difference between two versions of increment operator is shown in the code listed in Program 2-3.

Line	Prog 2-3.c	Output window
1	//Difference between Pre-increment and Post-increment	a=3, b=3, c=3, d=2
2	#include<stdio.h>	**Reasons:**
3	main()	• The value of a is incremented and then it is assigned to c as a is pre-incremented
4	{	• The value of b is assigned to d before it is incremented as b is post-incremented
5	int a=2, b=2,c,d;	
6	c=++a;	
7	d=b++;	
8	printf("a=%d, b=%d, c=%d, d=%d",a,b,c,d);	
9	}	

Program 2-3 | A program that illustrates the difference between pre-increment and post-increment

e. Increment operator is a **token**, i.e. one unit. There should be no white-space character between two '+' symbols. If white space is placed between two '+' symbols, they become two unary plus (+) operators. Execute the code listed in Program 2-4 to understand the significance of white-space character.

Line	Prog 2-4.c	Output window
1	//++ is a token. Don't place white space in between + symbols	The result of evaluation is 2
2	#include<stdio.h>	**Remark:**
3	main()	• There will be no compilation error as in Program 2-2 because the expression a=+ +2 does not have an increment operator. Instead it has two unary plus operators, which can be applied on an operand that does not have a modifiable l-value
4	{	
5	int a;	
6	a=+ +2;	
7	printf("The result of evaluation is %d",a);	
8	}	

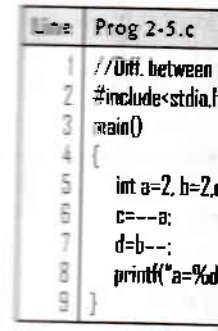
Program 2-4 | A program that illustrates the significance of white-space character in increment operator

Tokens are the basic building blocks of a source code. Characters are combined into tokens according to the rules of the programming language. There are five classes of tokens: identifiers, reserved words, operators, separators and constants.

4. **Decrement operator**
   a. The decrement operator can appear towards the left side or towards the right side of its operand. If it appears towards the left side of its operand (e.g. --a), it is known as the **pre-decrement operator**. If it appears towards the right side of its operand (e.g. a--), it is known as the **post-decrement operator**.
   b. The decrement operator can only be applied to an operand that has a modifiable l-value. If it is applied on an operand that does not have a modifiable l-value, there will be a compilation error 'L-value required'.
   c. --a or a-- is equivalent to a=a-1.
   d. The **difference between pre-decrement and post-decrement** lies in the point at which the value of their operand is decremented.
      i. In case of the pre-decrement operator, first the value of its operand is decremented and then used for the evaluation of the expression in which it appears.
      ii. In case of the post-decrement operator, first the value of the operand is used for the evaluation of the expression in which it appears and then its value is decremented.

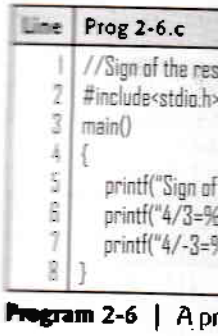
      The difference between two versions of the decrement operator is shown in the code listed in Program 2-5.
   e. Decrement operator is a token, i.e. one unit. There should be no white-space character between two '-' symbols. If white space is placed between two '-' symbols, they become two unary minus (-) operators.

5. **Division operator**
   a. The division operator is used to find the quotient.



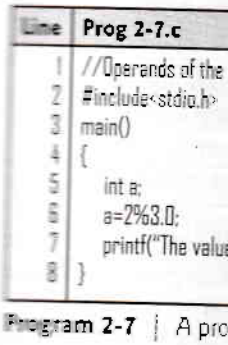
Program 2-5 | A p

b. The sig
of both
will be
negativ
This ca



Program 2-6 | A pr

6. **Modulus op**
   a. The mod
   b. The oper
   tor cann
   Program



Program 2-7 | A pro

Line	Prog 2-5.c	Output window
1	//Diff. between Pre-decrement & Post-decrement operator	a=1, b=1, c=1, d=2
2	#include<stdio.h>	**Reasons:**
3	main()	• The value of a is decremented and then it is assigned to c as a is pre-decremented
4	{	• The value of b is assigned to d before it is decremented as b is post-decremented
5	int a=2, b=2,c,d;	
6	c=--a;	
7	d=b--;	
8	printf("a=%d, b=%d, c=%d, d=%d",a,b,c,d);	
9	}	

Program 2-5 | A program that illustrates the difference between pre-decrement and post-decrement

b. The sign of the result of evaluation of the division operator depends upon the sign of both the numerator as well as the denominator. If both are positive, the result will be positive. If both are negative, the result will be positive. If either of the two is negative, the result will be negative. For example: 4/3=1, -4/3=-1, 4/-3=-1 and -4/-3=1. This can be observed by executing the code listed in Program 2-6.

Line	Prog 2-6.c	Output window
1	//Sign of the result of division operator	Sign of the result of division operator:
2	#include<stdio.h>	4/3=1, -4/3=-1
3	main()	4/-3=-1, -4/-3=1
4	{	**Remark:**
5	printf("Sign of the result of division operator:\n");	• The sign of the result of evaluation of the division operator depends upon the sign of the numerator as well as the denominator
6	printf("4/3=%d, -4/3=%d\n",4/3,-4/3);	
7	printf("4/-3=%d, -4/-3=%d",4/-3,-4/-3);	
8	}	

Program 2-6 | A program that illustrates the sign of result of division operator

6. **Modulus operator**
   a. The modulus operator is used to find the remainder.
   b. The operands of modulus operator (i.e. %) must be of integer type. Modulus operator cannot have operands of floating point type. Try executing the code listed in Program 2-7.

Line	Prog 2-7.c	Output window
1	//Operands of the modulus operator must be of integer type	Compilation error "Illegal use of floating point in the function main"
2	#include<stdio.h>	**Reason:**
3	main()	• Operands of modulus operator should be of integer type
4	{	**What to do?**
5	int a;	• Write 3 instead of 3.0 or type cast 3.0 to int by writing (int)3.0
6	a=2%3.0;	
7	printf("The value of a is %d",a);	
8	}	

Program 2-7 | A program to illustrate that the operands of modulus operator must be of integer type

c. The sign of the result of evaluation of modulus operator depends only upon the sign of the numerator. If the sign of the numerator is positive, the sign of the result will be positive else negative. For example: 4%3=1, -4%3=-1, 4%-3=1 and -4%-3=-1. This can be observed by executing the code listed in Program 2-8.

Line	Prog 2-8.c	Output window
1	//Sign of the result of modulus operator	Sign of the result of modulus operator: 4%3=1, -4%3=-1 4%-3=1, -4%-3=-1 **Remarks:** • The sign of the result of evaluation of the modulus operator depends only upon the sign of the numerator • The % sign marks the beginning of format specifier. If it is to be actually printed, use it twice. Refer Question number 16, Chapter 1
2	#include<stdio.h>	
3	main()	
4	{	
5	printf("Sign of the result of modulus operator:\n");	
6	printf("4%%3=%d, -4%%3=%d\n",4%3,-4%3);	
7	printf("4%%-3=%d, -4%%-3=%d",4%-3,-4%-3);	
8	}	

Program 2-8 | A program that illustrates the sign of result of modulus operator

2.4.2.2 Relational Operators

Relational operators are used to compare two quantities (i.e. their operands). There are six relational operators in C, which are given in Table 2.2.

Table 2.2 | Relational operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among relational class	Associativity
1.	< > <= >=	Less than Greater than Less than or equal to Greater than or equal to	Relational operators	Binary	Level-I	L→R
2.	== !=	Equal to Not equal to	Equality operators	Binary	Level-II	L→R

The important points about the relational operators are as follows:

1. There should be no white-space character between two symbols of a relational operator.
2. The result of evaluation of a relational expression (i.e. involving relational operator) is a boolean constant, i.e. 0 or 1.
3. Each of the relational operators yields 1 if the specified relation is true and 0 if it is false. The result has type int.
4. The expression a<b<c is valid and is not interpreted as in ordinary mathematics. Since the less than operator (i.e. <) is left-to-right associative, the expression is interpreted as (a<b)<c. This means that 'if a is less than b, compare 1 with c, otherwise, compare 0 with c'.
5. An expression that involves a relational operator forms a **condition**. For example, a<b is a condition.

Consider Program 2-9 that illustrates the evaluation of a relational expression.

Line	Prog 2-9.c	Output window
1	//Relational operators	The value of a is 1
2	#include<stdio.h>	**Remark:**
3	main()	• The expression a=2<3!=2 is interpreted as a=(2<3)!=2. The sub-expression 2<3 is true (i.e. 1). 1!=2 is true (i.e. 1). So, 1 is assigned to a
4	{	
5	int a;	
6	a=2<3!=2;	
7	printf("The value of a is %d",a);	
8	}	

Program 2-9 | A program that illustrates the use of relational operators

2.4.2.3 Logical Operators

Logical operators are used to logically relate the sub-expressions. The logical operators available in C are given in Table 2.3.

Table 2.3 | Logical operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among logical class	Associativity
1.	!	Logical NOT	Unary	Unary	Level-I	R→L
2.	&&	Logical AND	Logical operator	Binary	Level-II	L→R
3.	||	Logical OR	Logical operator	Binary	Level-III	L→R

In C language, there is no operator available for logical eXclusive-OR (XOR) operation.

The important points about the logical operators are as follows:

1. Logical operators consider operand as an entity, a unit.
2. Logical operators operate according to the truth tables given in Table 2.4.

Table 2.4 | Truth tables of logical operations

AND Operation		
Operand1	Operand2	Result
False	False	False
False	True	False
True	False	False
True	True	True

(a)

OR Operation		
Operand1	Operand2	Result
False	False	False
False	True	True
True	False	True
True	True	True

(b)

NOT Operation	
Operand	Result
False	True
True	False

(c)

3. If an operand of a logical operator is a non-zero value, the operand is considered as true. If the operand is zero, it is considered as false.
4. Each of the logical operators yields 1 if the specified relation evaluates to true and 0 if it evaluates to false. The evaluation is done according to the truth tables mentioned in Table 2.4. The result has type int.
5. The logical AND (i.e. &&) operator and the logical OR (i.e. ||) operator guarantee left-to-right evaluation.
6. Expressions connected by the logical AND (&&) or the logical OR (||) operator are evaluated left to right and the evaluation stops as soon as truthfulness or falsehood of the expression is determined. Thus, in an expression:
   a. E1&&E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to 0 (i.e. false), E2 will not be evaluated and the result of the overall expression will be 0 (i.e. false). If E1 evaluates to a non-zero value (i.e. true) then E2 will be evaluated to determine the truth value of the overall expression. The fragment of code in Program 2-10 illustrates the mentioned fact.

Line	Prog 2-10.c	Output window
1	//Logical AND operator	Resultant values after evaluation are:
2	#include<stdio.h>	0 1 2 0
3	main()	**Remark:**
4	{	• The expression l=i&&j++&&k++ is interpreted as l=(i&&j++)&&k++. Since i is false, j++ will not be evaluated and (i&&j++) evaluates to 0 (i.e. false). Since (i&&j++) is false, k++ will not be evaluated and the expression i&&j++&&k++ evaluates to 0, i.e. false. So, 0 is assigned to l
5	int i=0,j=1,k=2,l;	
6	l=i&&j++&&k++;	
7	printf("Resultant values after evaluation are:\n");	
8	printf("%d %d %d %d",i ,j,k,l);	
9	}	

Program 2-10 | A program that illustrates logical AND operation

   b. E1||E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to a non-zero value (i.e. true), E2 will not be evaluated and the result of the overall expression will be 1 (i.e. true). If E1 evaluates to 0 (i.e. false) then E2 will be evaluated to determine the truth value of the overall expression. The fragment of code in Program 2-11 illustrates the mentioned fact.

Line	Prog 2-11.c	Output window
1	//Logical OR operator	Resultant values after evaluation are:
2	#include<stdio.h>	0 1 3 1
3	main()	**Remark:**
4	{	• The expression l=i&&j++||k++ is interpreted as l=(i&&j++)||k++. Since i is false, j++ will not be evaluated and (i&&j++) evaluates to 0 (i.e. false). Since (i&&j++) is false, k++ needs to be evaluated. k++ evaluates to 2 (i.e. true) and k becomes 3. The overall expression l=i&&j++||k++ evaluates to 1 (i.e. true). So, l=1
5	int i=0,j=1,k=2,l;	
6	l=i&&j++||k++;	
7	printf("Resultant values after evaluation are:\n");	
8	printf("%d %d %d %d",i ,j,k,l);	
9	}	

Program 2-11 | A program that illustrates logical OR operation

2.4.2.4 Bitwis

The C language p
the operand as o
erators available

Table 2.5 | Bitwi

S.No	Operator
1.	~
2.	<< >>
3.	&
4.	^
5.	|

The important p

1. Bitwise oper
   manipulatio
2. They can o
   unsigned.
3. The bitwise-
   operands acc
4. The expressi
   bits of operar

Value, operator and result	Sign bit	
	Bit 16	Bit 15
2	0	0
3	0	0
2&3=2	0	0
2|3=3	0	0

Figure 2.1 | Bitwise-

5. X-OR operator

2.4.2.4 Bitwise Operators

The C language provides six operators for bit manipulation. These operators do not consider the operand as one entity and operate on the individual bits of the operands. The bitwise operators available in C are given in Table 2.5.

Table 2.5 | Bitwise operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among bitwise class	Associativity
1.	~	Bitwise NOT	Unary	Unary	Level-I	R→L
2.	<< >>	Left Shift Right Shift	Shift operators	Binary	Level-II	L→R
3.	&	Bitwise AND	Bitwise operator	Binary	Level-III	L→R
4.	^	Bitwise X-OR	Bitwise operator	Binary	Level-IV	L→R
5.	|	Bitwise OR	Bitwise operator	Binary	Level-V	L→R

The important points about the bitwise operators are as follows:

1. Bitwise operators operate on the individual bits of the operands and are used for bit manipulation.
2. They can only be applied on operands of type char, short, int, long, whether signed or unsigned.
3. The bitwise-AND and the bitwise-OR operators operate on the individual bits of the operands according to the truth tables specified in Table 2.4.
4. The expression 2&3 evaluates to 2 and 2|3 evaluates to 3. The operations on individual bits of operands (i.e. 2 and 3) are shown in Figure 2.1.

Value, operator and result	Sign bit	Magnitude														
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
X=2	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
Y=3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**

Figure 2.1 | Bitwise-AND and bitwise-OR operator operating on the individual bits of the operands

5. X-OR operator operates according to the truth table given in Table 2.6.

Table 2.6 | Truth table of X-OR operation

X-OR OPERATION		
Operand1	**Operand2**	**Result**
False	False	False
False	True	True
True	False	True
True	True	False

6. The bitwise NOT operator results in 1's complement of its operand.
7. Left shift by 1 bit is equivalent to multiplication by 2. Left shift by n bits is equivalent to multiplication by 2^n, provided the magnitude does not overflow.
8. Right shift by 1 bit is equivalent to an integer division by 2. Right shift by n bits is equivalent to integer division by 2^n.
9. The expression 4<<1 evaluates to 8 and 4>>1 evaluates to 2. This is shown in Figure 2.2.

Value, operator and result	Sign bit	Magnitude													
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
4<<1=8	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
4>>1=2	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Figure 2.2 | Left-shift and right-shift operations

2.4.2.5 Assignment Operators

A variable can be assigned a value by using an assignment operator. The assignment operators available in C language are given in Table 2.6.

Table 2.6 | Assignment operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	=	Simple assignment	Assignment & Shorthand assignment operators	Binary	Level-I	R→L
	*=	Assign product				
	/=	Assign quotient				
	%=	Assign modulus				
	+=	Assign sum				
	-=	Assign difference				
	&=	Assign bitwise AND				
	|=	Assign bitwise OR				
	^=	Assign bitwise XOR				
	<<=	Assign left shift				
	>>=	Assign right shift				

The important points about the assignment operators are as follows:

1. The operand that appears towards the left side of an assignment operator should have a modifiable l-value. If the operand appearing towards the left side of the assignment operator does not have a modifiable l-value, there will be a compilation error 'L-value required'.
2. The shorthand assignment is of the form `op1 op=op2`, where `op1` and `op2` are operands and `op=` is a shorthand assignment operator. It is a shorter way of writing `op1 = op1 op op2`. For example, `a/=2` is equivalent to `a=a/2`.

Forward Reference: Refer Question numbers 53 and 65 and their answers for examples on the usage of a shorthand assignment operator.

3. There should be no white-space character between two symbols of shorthand assignment operators.
4. If two operands of an assignment operator are of different types, the type of operand on the right side of the assignment operator is automatically converted to the type of operand present on its left side. To carry out this conversion, either promotion or demotion is applied.
5. The result of evaluation of an assignment expression is the value that is assigned. For example, in the expression `a=10;`, the value `10` is assigned to `a` and the overall expression evaluates to `10` (i.e. the value that is assigned).
6. The terms assignment and initialization are related but it is important to note the differences between them. They are listed in Table 2.7.

Table 2.7 | Differences between initialization and assignment

S.No	Initialization	Assignment
1.	First time assignment at the time of definition is called initialization. For example: `int a=10;` is initialization of `a`	Value of a data object after initialization can be changed by the means of assignment. For example: Consider the following statements `int a=10; a=20;`. The value of `a` is changed to 20 by the assignment statement
2.	Initialization can be done only once	Assignment can be done any number of times
3.	Qualified constant can be initialized with a value. For example, `const int a=10;` is valid	Qualified constant cannot be assigned a value. It is erroneous to write `a=10;` if `a` is a qualified constant

2.4.2.6 Miscellaneous Operators

Other operators available in C are:

1. Function call operator (i.e. ())
2. Array subscript operator (i.e. [])
3. Member select operator
   a. Direct member access operator (i.e. . (dot operator or period))
   b. Indirect member access operator (i.e. -> (arrow operator))
4. Indirection operator (i.e. *)
5. Conditional operator

6. Comma operator
7. sizeof operator
8. Address-of operator (i.e. &)

Forward Reference: Function call operator (Chapter 5), array subscript operator (Chapter 4), member select operator (Chapter 9), indirection operator (Chapter 4).

2.4.2.6.1 Conditional Operator

Conditional operator is the only ternary operator available in C (Table 2.8).

Table 2.8 | Conditional operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	?:	Conditional operator	Conditional	Ternary	Level-I	R→L

The important points about the conditional operator are as follows:

1. The general form of conditional operator is E1?E2:E3, where E1, E2 and E3 are sub-expressions.
2. The sub-expression E1 must be of scalar type.
3. The sub-expression E1 is evaluated first. If it evaluates to a non-zero value (i.e. true), then E2 is evaluated and E3 is ignored. If E1 evaluates to zero (i.e. false), then E3 is evaluated and E2 is ignored.

Integer and floating types are collectively called **arithmetic types**. Arithmetic types and pointer types are collectively called **scalar types**.

Forward Reference: Refer Question number 33 and its answer for an example on the usage of a conditional operator.

2.4.2.6.2 Comma Operator

The comma operator is used to join multiple expressions together (Table 2.9).

Table 2.9 | Comma operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	,	Comma operator	Comma	Binary	Level-I	L→R

The important points about the comma operator are as follows:

1. Every instance of a comma symbol is not a comma operator. The commas separating arguments in a function call are not comma operators. If commas separating arguments in a function call are considered as comma operators, then no function could have more than one argument. The commas used in the declaration/definition statement are not considered as comma operators. The commas appearing between the arguments in a function call or commas appearing in a declaration/definition statement are separators.

2. The comma operator guarantees left-to-right evaluation.
3. In expression E1, E2, E3...En, the sub-expressions E1, E2, E3...En are evaluated in left-to-right order. The result and type of evaluation of the overall expression is the value and type of the evaluation of the rightmost sub-expression, i.e. En.
4. The comma operator has least precedence.

The piece of code in Program 2-12 illustrates the use of a comma operator.

Line	Prog 2-12.c	Output window
1	//Use of comma operator	Resultant values of a and b are:
2	#include<stdio.h>	1 5
3	main()	**Remarks:**
4	{	• The precedence of assignment operator is greater than comma operator
5	int a,b;	• Thus, in the expression a=1,2,3,4,5, the sub-expression a=1 gets evaluated first. Hence, the value assigned to a is 1
6	a=1, 2, 3, 4, 5;	• In the expression b=(1,2,3,4,5), the sub-expression 1,2,3,4,5 is parenthesized and will be evaluated first. The result of evaluation of comma operator is the result of evaluation of the rightmost sub-expression, i.e. 5. Thus, (1,2,3,4,5) evaluates to 5 and is assigned to b
7	b=(1, 2, 3, 4, 5);	
8	printf("Resultant values of a and b are:\n");	
9	printf("%d %d",a ,b);	
10	}	

Program 2-12 | A program to illustrate the use of comma operator

Forward Reference: Arguments, function (Chapter 5).

2.4.2.6.3 sizeof Operator

The sizeof operator is used to determine the size in bytes, which a value or a data object will take in memory (Table 2.10).

Table 2.10 | sizeof operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	sizeof	Size-of operator	Unary	Unary	Level-I	R→L

The important points about the sizeof operator are as follows:

1. The general form of a sizeof operator is:
   a. **sizeof expression** or **sizeof (expression)** (For example: sizeof 2, sizeof(a), sizeof(2+3))
   b. **sizeof (type-name)** (For example: sizeof(int), sizeof(int*), sizeof(char))
2. Parentheses should be used if the sizeof operator is applied on a type-name, as indicated in point 1 b) above.
3. The type of result of evaluation of the sizeof operator is int.
4. The operand of the sizeof operator is not evaluated. This fact can be seen by executing the code listed in Program 2-13.

Line	Prog 2-13.c	Output window
1	//sizeof operator	Resultant values of a and b are:
2	#include<stdio.h>	1 2
3	main()	**Remark:**
4	{	• The operand of sizeof operator is not evaluated. Hence, ++a is not evaluated and thus, the value of a remains unchanged, i.e. 1. The value of a takes 2 bytes in memory (in case of MS-VC++ 6.0, it takes 4 bytes). Thus, the value of b is 2
5	int a=1,b;	
6	b=sizeof(++a);	
7	printf("Resultant values of a and b are:\n");	
8	printf("%d %d",a ,b);	
9	}	

Program 2-13 | A program to illustrate that operand of sizeof operator is not evaluated

5. The sizeof operator cannot be applied on operands of incomplete type➲ or function type.➲

Forward Reference: Incomplete type (Chapter 9), function type (Chapter 5).

2.4.2.6.4 Address-of Operator

The address-of operator is used to find the address, i.e. l-value of a data object (Table 2.11).

Table 2.11 | Address-of Operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	&	Address-of operator	Unary	Unary	Level-I	R→L

The important points about the address-of operator are as follows:

1. The address-of operator must appear towards the left side of its operand.
2. The syntax of using the address-of operator is &operand.
3. The operand of the address-of operator should be a variable or a function designator.➲ The address-of operator cannot be applied to constants, expressions, bit-fields➲ and to the variables declared with register➲ storage class.

Forward Reference: Function designator (Chapter 5), bit-field (Chapter 9), storage class specifier (Chapter 7).

2.5 Combined Precedence of All Operators

Till now, I have described different operators according to their role and have categorized them into various classes like arithmetic operators, relational operators, etc. I have described the precedence of operators within a class (i.e. intra-class precedence). Now, it is the time to consider the precedence of an operator with respect to the operators in other classes (i.e. inter-class precedence). Table 2.11 provides a combined table of precedence.

Table 2.11 | Co

S.No	Operator
1.	() [] -> .
2.	! ~ + - ++ -- & * sizeof
3.	* / %
4.	+ -
5.	<< >>
6.	< > <= >=
7.	== !=
8.	&
9.	^
10.	|
11.	&&
12.	||
13.	?:

Table 2.11 | Combined precedence chart

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	() [] -> .	Function call Array subscript Indirect member access Direct member access			Level-I (Highest)	
2.	! ~ + - ++ -- & * sizeof	Logical NOT Bitwise NOT Unary plus Unary minus Increment Decrement Address-of Deference Sizeof	Unary operators	Unary	Level-II	R→L
3.	* / %	Multiplication Division Modulus	Multiplicative operators	Binary	Level-III	L→R
4.	+ -	Addition Subtraction	Additive operators	Binary	Level-IV	L→R
5.	<< >>	Left Shift Right Shift	Shift operators	Binary	Level-V	L→R
6.	< > <= >=	Less than Greater than Less than or equal to Greater than or equal to	Relational operators	Binary	Level-VI	L→R
7.	== !=	Equal to Not equal to	Equality operators	Binary	Level-VII	L→R
8.	&	Bitwise AND	Bitwise operator	Binary	Level-VIII	L→R
9.	^	Bitwise X-OR	Bitwise operator	Binary	Level-IX	L→R
10.	|	Bitwise OR	Bitwise operator	Binary	Level-X	L→R
11.	&&	Logical AND	Logical operator	Binary	Level-XI	L→R
12.	||	Logical OR	Logical operator	Binary	Level-XII	L→R
13.	?:	Conditional operator	Conditional	Ternary	Level-XIII	R→L

(Contd...)

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
14.	= *= /= %= += -= &= |= ^= <<= >>=	Simple assignment Assign product Assign quotient Assign modulus Assign sum Assign difference Assign bitwise AND Assign bitwise OR Assign bitwise XOR Assign left shift Assign right shift	Assignment & Shorthand assignment operators	Binary	Level-XIV	R→L
15.	,	Comma operator	Comma	Binary	Level-XV (Least)	L→R

2.6 Summary

1. Operand is an entity on which an operation is performed.
2. Operator specifies the operation to be performed on an operand.
3. Expression is made up of operands and operators.
4. Operands constituting an expression can be identifiers, constants or expressions themselves. The identifiers allowed to constitute an expression are variables, functions and macros. However, label names, typedef name, tags of structure, union or enumeration cannot be a part of an expression. The expressions forming an expression are called **sub-expressions**. An expression that is not a part of another expression is called **full expression**.
5. Based upon the number of operators in an expression, the expressions are classified as simple expressions and compound expressions.
6. Simple expressions have only one operator.
7. There is more than one operator present in a compound expression. To evaluate a compound expression, the order in which the operators will operate is to be determined.
8. The order in which operators operate depends upon the precedence and the associativity of the operators.
9. In a compound expression, if operators of different precedence appear together, the operator of the higher precedence operates first.
10. In a compound expression, if operators of the same precedence appear together, then precedence is not sufficient to determine the order in which operators will operate. The order of evaluation can be determined by looking at the associativity of the operators.
11. If operators are left-to-right associative, the operator that appears first in the left-to-right traversal will operate first.
12. If operators are right-to-left associative, the operator that appears first in the right-to-left traversal will operate first.
13. The operators with the same precedence have the same associativity but vice versa is not true.

14. In an arithmetic expression, if the operands of a binary operator are of a different type, C automatically applies arithmetic-type conversion to bring the operands to a common type. The type of result of the binary operator will also be the common type.
15. Automatic-type conversion is called implicit-type conversion.
16. Type can also be changed by applying explicit-type conversion.
17. Explicit-type conversion is done with the help of a type cast operator, i.e. (). The syntax of using the type cast operator is **(target-type-name) expression**.

Exercise Questions

Conceptual Questions and Answers

1. *I have heard that white-space characters are ignored in C. If I write the statement* `a+ =2;` *in a C program, there is a compilation error. However, if I write it as* `a+=2;` *it works. Why is the blank space (i.e. a white-space character) between + and = not getting ignored?*

   Every white-space character is not ignored in C. White-space characters separating tokens are not significant and are ignored in C. Here, '+=' is a token (i.e. a single unit, one operator). We cannot have white space in between + and =. The occurrence of a white-space character between them makes '+' and '=' two different tokens (i.e. two different operators and both are binary operators). Two binary operators cannot come next to each other without having any operand in between. This leads to an error.
   The following are allowed:

   ```
   1. a     +=     2;
   2. a+=          2;
   3. a            +=2;
   ```

   because white-space characters come in between tokens and not within a token. Similarly, `printf ("Hello");` can be written and will work but `pri ntf("Hello");` will not work because the white-space character does not separate different tokens but comes within a token (i.e. `printf`). Thus, the statement 'White-space characters are ignored in C' can be corrected and refined as 'Non-significant white-space characters are ignored in C'.

2. *I want to check whether a number* `b` *lies in between numbers* `a` *and* `c`. *I have written the following segment of code:*

   ```
   if a<b<c)
     printf("b lies between a and c");
   else
     printf("b is an outlier");
   ```

   The above segment of code does not work for all test cases. Why? Correct the code so that it starts working as intended.

   The answer to why this code does not work for all test cases lies in understanding how expression `a<b<c` gets evaluated. In the expression `a<b<c`, two less than operators (<) are involved. The less than operator is left-to-right associative, thus the expression `a<b<c` is interpreted as `(a<b)<c`. `a<b` is evaluated first. Less than is a relational operator and the outcome of a relational operator is a boolean constant, i.e. `1` (true) or `0` (false). Therefore, `a<b` can be `1` or `0`, depending upon whether `a` is less than `b` or not. Then, the result of comparison of `a` and `b` gets compared with `c`. Therefore, instead of `b` getting compared with `c`, `0` or `1` gets compared with `c`. Here lies the flaw.

edence	Associativity
el-XIV	R→L
vel-XV (Least)	L→R

rand.

nstants or expressions them-
variables, functions and mac-
union or enumeration cannot
ession are called **sub-expres-**
is called **full expression**.
expressions are classified as

xpression. To evaluate a com-
perate is to be determined.
recedence and the associativ-

cedence appear together, the

edence appear together, then
ch operators will operate. The
ssociativity of the operators.
at appears first in the left-to-

t appears first in the right-to-

associativity but vice versa is

Suppose a=2, b=1 and c=5, in a<b<c (i.e. 2<1<5), 2<1 is false, i.e. 0. Therefore, the expression becomes 0<5. 0<5 is true, i.e. 1; hence, the output will be 'b lies between a and c', which is wrong.
Instead of writing a<b<c, the expression should be written as a<b&&b<c. The correct code is:

```
if(a<b&&b<c)
    printf("b lies between a and c");
else
    printf("b is an outlier");
```

In the expression a<b&&b<c (i.e. 2<1&&1<5), 2<1 is false. Therefore, the entire expression evaluates to false and the output is 'b is an outlier'.

3. *A programmer wants to find the average of three numbers. He has written the following piece of code in C:*

```
main()
{
    int a=10,b=12,c=13, average;
    average=a+b+c/3;
    printf("Average is %d",average);
}
```

Does the mentioned piece of code produce the correct result as intended? If no, why?
No, the code does not produce the intended result due to the following reasons:

1. The division operator has a higher precedence than the addition operator. Hence, the expression `average=a+b+c/3` is interpreted as `average=a+b+(c/3)` instead of being interpreted as `average=(a+b+c)/3`.
2. The type of the variable `average` is taken as `int` instead of `float`.

4. *If the code in the previous question is rectified and rewritten as*

```
main()
{
    int a=10,b=12,c=13;
    float average;
    average=(a+b+c)/3;
    printf("Average is %f",average);
}
```

does this code produce the correct result? If no, why? Rewrite the code, so that it produces the correct result.

Still the code will not produce the correct result. This is due to the fact that in the expression `average=(a+b+c)/3`, the sub-expression `a+b+c` will be evaluated first and then it is divided by 3. 10+12+13 turns out to be 35. 35/3 gives 11. (As both 35 and 3 are integers, integer mode arithmetic is applicable. In this mode, the result of evaluation of binary arithmetic operator is an integer.) Now, 11 is assigned to a float variable. Before assigning an integer value to a `float` variable, the integer value gets promoted (i.e. converted into `float`). Thus, 11 get promoted to 11.0. Therefore, the average value that gets printed is 11.000000 instead of 11.666667.

The reason behind this problem is the application of integer arithmetic instead of floating point arithmetic. We must do something so that floating point arithmetic or mixed mode arithmetic is applied. To make this happen, any one of the below-mentioned ways can be adopted:

```
1. average=(a+b+c)/3.0;          //←Implicit-type conversion
2. average=(float)(a+b+c)/3;     //←Explicit-type conversion
3. average=(a+b+c)/(float)3;     //←Explicit-type conversion
```

In all the three cases, division is carried out between an int value and a float value. Thus, mixed mode arithmetic is applicable instead of integer arithmetic and the result of computation turns out to be a float value. By using any one of the above three ways, 'Average is 11.666667' gets printed.

5. *The output of the following piece of code turns out to be 81 instead of the expected output 300. Why does this happen? Suggest possible ways to rectify this problem.*

```
main()
{
    int a=100,b=900,c;
    c=a*b/300;
    printf("The value that c gets is %d",c);
}
```

The expression c=a*b/300 contains three operators, namely assignment operator, multiplication operator and division operator. Multiplication and division operators have the same precedence. The assignment operator has a lesser precedence than these operators. Therefore, multiplication and division operators will be evaluated prior to the assignment operator. Being left-to-right associative, multiplication will be carried out first as the multiplication operator appears towards the left. When 100 and 900 (i.e. both integers) get multiplied, the result turns out to be 90000, which exceeds the range of integer data type. Since the value exceeds the range, wrap around will occur and 90000 will be mapped to 90000–65536 = 24464. Now, this number is divided by 300 to give 81 as the result. Therefore, this problem occurs due to overflow and wrap-around effect.

In order to avoid this problem, we should prevent this overflow and wrap around. This can be done by using range of long integer type instead of integer type. The following alternatives will solve the problem:

1. c=(long)a*b/300;
2. c=a*(long)b/300;

Now, long integer and integer gets multiplied and the result turns out to be a long integer. 90000 is well within the range of long integer type; hence no overflow occurs.

It is very important to note that the following ways do not solve the problem:

1. c=(long)(a*b)/300;
2. c=a*b/(long)300;
3. c=a*b/300L;

This happens because type casting does not prevent overflow in the above-mentioned statements. In 1, first a and b are multiplied. At this stage, overflow occurs and the value becomes 24464. Therefore, there is no benefit now in type-casting it to a long integer. A similar reason applies for 2 and 3.

6. *Why does an assignment operator fail on constants, i.e. why cannot constants be placed on the left side of an assignment operator?*

Assignment operator fails on constants because the assignment operator on its left side expects an operand that has a modifiable l-value. Constants do not have a modifiable l-value and thus cannot be placed on the left side of the assignment operator. If a constant is placed on the left side of the assignment operator, the compiler shows 'L-value required' error.

7. *A programmer wants to find the exponent of a number. He has written the following piece of code:*

```
main()
{
    int x=10,y=2,result;
    result=x^y;
```

```
    printf("The result of exponent operation is %d",result);
}
```

Does the above-mentioned piece of code produce the intended result?

No, the mentioned piece of code does not produce the correct result. There is no operator in C to find the exponent of a number. The ^ operator is a bitwise XOR operator. Hence, the mentioned piece of code finds 'x bitwise-XOR y' instead of 'x exponent y'.

8. *I have read that 'Every statement in C is terminated with a semicolon'. The line number 1 in the given piece of code is terminated with a comma instead of a semicolon. Will this piece of code work? If yes, what would its output be?*

```
main()
{
    printf("Hello"),                //←line 1
    printf("Readers!!..") ;         //←line 2
}
```

As new line characters and comments are ignored during the translation phase by the compiler, the given piece of code:

```
main()
{
    printf("Hello"),                //←line 1
    printf("Readers!!..");          //←line 2
}
```

will be interpreted as

```
main()
{
    printf("Hello"), printf("Readers!!..");
}
```

The interpreted code has only one statement that consists of two comma-separated expressions, i.e. printf("Hello") and printf("Readers!!.."). As the operands of the comma operator are evaluated in left-to-right order, printf("Hello") is evaluated first followed by printf("Readers!.."). Hence, the output of the code would be HelloReaders!!..

9. *From the previous question, I have inferred that semicolons separating two* printf *functions can be replaced by commas. Is my inference correct?*

No. Consider the following piece of code:

```
main()
{
    printf("Hello"); ;printf("Readers!!..");
}
```

The given piece of code on execution prints HelloReaders!!... If the semicolons appearing between the printf functions are replaced by commas, the given code becomes

```
main()
{
    printf("Hello"), ,printf("Readers!!..");
}
```

The resultant code on compilation gives 'Expression syntax error'. This error is due to the fact that two comma operators cannot appear consecutively. There must be an operand in between them. Hence, the drawn inference is not correct.

10. *What will the output of the following code segment be?*

```
main()
{
    int a=10,b=20;
    printf("%d %d\n",a,b);
    a=a*b;
    b=a/b;
    a=a/b;
    printf("%d %d\n",a,b);
    a=a^b;
    b=a^b;
    a=a^b;
    printf("%d %d\n",a,b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf("%d %d\n",a,b);
}
```

The code provides three different ways to swap the contents of two variables without using a temporary variable.
Initially, a=10, b=20. On execution of statements:
a=a*b; //←a will become 200
b=a/b; //←b will become 10
a=a/b; //←a will become 20
Values are swapped.
a=a^b; //←a will become 30
b=a^b; //←b will become 20
a=a^b; //←a will become 10
Values are swapped again.
a=a+b; //←a will become 30
b=a-b; //←b will become 10
a=a-b; //←a will become 20
Values are swapped again.
Hence, the output of the code would be:

```
10 20
20 10
10 20
20 10
```

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

11.
```
main()
{
    int a;
    a=2*3+4%5-3/2+6;
    printf("%d",a);
}
```

12.
```
main()
{
    printf("%d %d %d %d",6/5,-6/5,6/-5,-6/-5);
}
```
13.
```
main()
{
    printf("%d %d %d %d",6%5,-6%5,6%-5,-6%-5);
}
```
14.
```
main()
{
    int a=12,b;
    printf("%d %d",b,b=a);
}
```
15.
```
main()
{
    int a=23,b=12,c=10,d;
    d=c=b=a;
    printf("%d %d %d %d",a,b,c,d);
}
```
16.
```
main()
{
    int a=23,b=12,c=10,d;
    d=c+2=b+1=a;
    printf("%d %d %d %d",a,b,c,d);
}
```
17.
```
main()
{
    int a=2,b=3,c=1,d;
    d=a<b>c;
    printf("%d",d);
}
```
18.
```
main()
{
    int a=3,b=2,c=1,d;
    d=a<b<c-1;
    printf("%d",d);
}
```
19.
```
main()
{
    int a=10,b=20,c=30;
    c==a=b;
    printf("%d %d %d",a,b,c);
}
```
20.
```
main()
{
    int a=10,b=20,c=30;
    c=a==b;
```
```
    printf("%d %
}
```
21.
```
main()
{
    int a=10,b=20,
    c==a==b;
    printf("%d %d
}
```
22.
```
main ()
{
    int a=012,b=03
    int x=0x12,y=0x
    int c,d,u,v;
    c=a&b;
    d=a|b;
    u=x&y;
    v=x|y;
    printf("%d %d
}
```
23.
```
main ()
{
    int a=012,b=034
    int x=0x12,y=0x3
    int c,d,u,v;
    c=a&&b;
    d=a||b;
    u=x&&y;
    v=x||y;
    printf("%d %d %
}
```
24.
```
main()
{
    int c=10,d,e;
    d=!c;
    e=~c;
    printf("%d %d",d
}
```
25.
```
main()
{
    int c=-4,d=4;
    printf("%d %d %
}
```
26.
```
main()
{
    int i=10;
    printf("%d",i++*i+
}
```

```
    printf("%d %d %d",a,b,c);
}
21. main()
{
    int a=10,b=20,c=30;
    c==a==b;
    printf("%d %d %d",a,b,c);
}
22. main ()
{
    int a=012,b=034;
    int x=0x12,y=0x34;
    int c,d,u,v;
    c=a&b;
    d=a|b;
    u=x&y;
    v=x|y;
    printf("%d %d %d %d",c,d,u,v);
}
23. main ()
{
    int a=012,b=034;
    int x=0x12,y=0x34;
    int c,d,u,v;
    c=a&&b;
    d=a||b;
    u=x&&y;
    v=x||y;
    printf("%d %d %d %d",c,d,u,v);
}
24. main()
{
    int c=10,d,e;
    d=!c;
    e=~c;
    printf("%d %d",d,e);
}
25. main()
{
    int c=-4,d=4;
    printf("%d %d %d %d",~c,~d,c^d,~c^~d);
}
26. main()
{
    int i=10;
    printf("%d",i++*i++);
}
```

27.
```
main()
{
    int i=10,j;
    j=++i++;
    printf("%d %d",i,j);
}
```

28.
```
main()
{
    int i=10,j=11,k,l;
    k=i+++j;
    l=i+++++j;
    printf("%d %d",l,k);
}
```

29.
```
main()
{
    int i=10,j=11,k,l;
    k=i+++j;
    l=i+++ ++j;
    printf("%d %d",l,k);
}
```

30.
```
main()
{
    int i=10,j=11,k,l;
    k=i+++j;
    l=i++ +++j;
    printf("%d %d",l,k);
}
```

31.
```
main()
{
    int x=20,y=35;
    x=y++ + x++;
    y=++y + ++x;
    printf("%d %d",x,y);
}
```

32.
```
main()
{
    int i=100,j=20;
    i++=j;
    printf("%d %d",i,j);
}
```

33.
```
main()
{
    int a=10,b;
    a>=5?b=100:b=200;
    printf("%d",b);
}
```

```
34. main()
    {
        int i=0,j=1,k=2,l;
        l=i||j++&&++k;
        printf("%d %d %d %d",i,j,k,l);
    }
35. main()
    {
        int i=0,j=1,k=2,l;
        l=i&&j++&&++k;
        printf("%d %d %d %d",i,j,k,l);
    }
36. main()
    {
        int i=0,j=1,k=2,l;
        l=++i&&j++&&++k;
        printf("%d %d %d %d",i,j,k,l);
    }
37. main()
    {
        int i=0,j=1,k=2,l;
        l=++i||j++&&++k;
        printf("%d %d %d %d",i,j,k,l);
    }
38. main()
    {
        int i=0,j=1,k=2,l;
        l=++i&&j++||++k;
        printf("%d %d %d %d",i,j,k,l);
    }
39. main()
    {
        int i=0,j=1,k=2,l;
        l=++i&&--j||++k;
        printf("%d %d %d %d",i,j,k,l);
    }
40. main()
    {
        int i=0,j=1,k=2,l;
        l=++i&&j--||++k;
        printf("%d %d %d %d",i,j,k,l);
    }
41. main()
    {
        int x=4;
        printf("%d %d %d",x,x<<2,x>>2);
    }
```

42.
```
main()
{
    int x=32767;
    printf("%d",x<<1);
}
```

43.
```
main()
{
    int num=3;
    printf("%d",num<<2<<2);
}
```

44.
```
main()
{
    int num=3;
    printf("%d",num<<(2<<2));
}
```

45.
```
main()
{
    int num=5,i=1;
    printf("%d",(num<<i&1<<15)?1:0);
}
```

46.
```
main()
{
    int num=5,i=1;
    printf("%d",(num<<i&&1<<15)?1:0);
}
```

47.
```
main()
{
    float a=0.9;
    int c;
    c=a<0.9;
    printf("%d",c);
}
```

48.
```
main()
{
    float a=0.5;
    int c;
    c=a<0.5;
    printf("%d",c);
}
```

49.
```
main()
{
    float a=0.9;
    int c;
    c=a<0.9f;
    printf("%d",c);
}
```

```
50. main()
    {
        int a=0,b=0;
        ++a==0||++b==1);
        printf("%d %d",a,b);
    }
51. main()
    {
        int x=4+2%-8;
        printf("%d",x);
    }
52. main()
    {
        int i=5;
        i=!i>3;
        printf("%d",i);
    }
53. main()
    {
        int a=10,b=70,c;
        c=b=a*=2;
        printf("%d %d %d",a,b,c);
    }
54. main()
    {
        printf("%x",-1<<4);
    }
55. main()
    {
        int c=- -2;
        printf("%d",c);
    }
56. main()
    {
        int c=--2;
        printf("%d",c);
    }
57. main()
    {
        int i=5;
        printf("%d %d %d %d %d",i++,i--,++i,--i,i);
    }
58. main()
    {
        200;
        printf("%d",200);
    }
```

59.
```
main()
{
    int i=-l;
    +i;
    printf("%d %d",i,+i);
}
```

60.
```
main()
{
    char not;
    not=!2;
    printf("%d",not);
}
```

61.
```
main()
{
    int k=l;
    printf("%d==l is ""%s",k,k==l?"True":"False");
}
```

62.
```
main()
{
    const int i=4;
    float j;
    j=++i;
    printf("%d %d",i,++j);
}
```

63.
```
main()
{
    int i=5;
    printf("%d",i=++i==6);
}
```

64.
```
main()
{
    int i=5,j=10;
    j=i8=j&&10;
    printf("%d %d",i,j);
}
```

65.
```
main()
{
    float x,y;
    x=7; y=10;
    x*=y*=y+28.5;
    printf("%f %f",x,y);
}
```

66.
```
main()
{
    unsigned int a=0xffff;
    ~a;
    printf("%x",a);
}
```

67.
```
main()
{
    unsigned char i
    printf("%d",i<<
}
```

68.
```
main()
{
    unsigned a=-l;
    int b;
    printf("%u ",a);
    printf("%u ",++a
}
```

69.
```
main()
{
    float u=3.5;
    int v,w,x,y;
    v=(int)(u+0.5);
    w=(int)u+0.5;
    x=(int)((int)u+0.5
    y=(u+(int)0.5);
    printf("%d %d %
}
```

70.
```
main()
{
    int u=3.5,v,w,x,y;
    v=(int)(u+0.5);
    w=(int)u+0.5;
    x=(int)((int)u+0.5);
    y=(u+(int)0.5);
    printf("%d %d %d
}
```

Multiple-choice Que

71. The location of a g
    a. Load time
    b. Procedure ent

72. Which of the follo
    a. *
    b. +

73. Which of the follo
    a. &&
    b. |

74. Which of the follo
    a. %
    b. /

```
67. main()
    {
        unsigned char i=0x80;
        printf("%d",i<<1);
    }
68. main()
    {
        unsigned a=-1;
        int b;
        printf("%u ",a);
        printf("%u ",++a);
    }
69. main()
    {
        float u=3.5;
        int v,w,x,y;
        v=(int)(u+0.5);
        w=(int)u+0.5;
        x=(int)((int)u+0.5);
        y=(u+(int)0.5);
        printf("%d %d %d %d",v,w,x,y);
    }
70. main()
    {
        int u=3.5,v,w,x,y;
        v=(int)(u+0.5);
        w=(int)u+0.5;
        x=(int)((int)u+0.5);
        y=(u+(int)0.5);
        printf("%d %d %d %d",v,w,x,y);
    }
```

Multiple-choice Questions

71. The location of a global variable is bound at
    a. Load time
    b. Procedure entry time
    c. Run time
    d. None of these

72. Which of the following is not an arithmetic operator?
    a. *
    b. +
    c. &
    d. %

73. Which of the following is not a bitwise operator?
    a. &&
    b. |
    c. ^
    d. >>

74. Which of the following operators in arithmetic class has the lowest precedence?
    a. %
    b. /
    c. *
    d. +

75. What is the correct way to round off a float variable z into an integer?
    a. x=(int)(z+0.5)
    b. x=(z+(int)z+0.5
    c. x=(int)z+0.5
    d. x=(int)((int)z+0.5)

76. Comma operator is a/an
    a. Unary operator
    b. Binary operator
    c. Ternary operator
    d. None of these

77. The location of local variables and reference parameter is typically bound at
    a. Load time
    b. Procedure entry time
    c. Run time
    d. None of these

78. Evaluation of the expression involving || operator
    I. Takes place from left to right
    II. Takes place from right to left
    III. Stops when one of the operand evaluates to true
    IV. Stops when one of the operand evaluates to false
    a. I and III
    b. III only
    c. II only
    d. IV

79. Evaluation of the expression involving && operator
    I. Takes place from left to right
    II. Takes place from right to left
    III. Stops when one of the operand evaluates to true
    IV. Stops when one of the operand evaluates to false
    a. I and III
    b. I and IV
    c. II only
    d. IV

80. Expressions in C can be made from
    I. Operands alone
    II. Operators alone
    III. Operators and operands
    IV. None of these
    a. I and III
    b. III only
    c. II only
    d. IV

81. What is the fundamental unit of execution in C?
    a. Expression
    b. Sub-expression
    c. Statement
    d. Function

82. What is the minimum number of temporary variables required to swap the content of two variables?
    a. 1
    b. 2
    c. 0
    d. None of these

83. int a; is actually a
    a. Declaration
    b. Definition
    c. Neither a definition nor a declaration
    d. None of these

84. The output of the following C code will be as follows:

```
main()
{
```

```
int a=10,b=20;
printf("%d %d",a,b);
a ^=b ^=a ^=b;
printf("%d %d",a,b);
}
```

a. 10 20 10 20
b. 10 20 20 10
c. 10 10 10 10
d. None of these

```
main()
{
if(-0 == -1)
printf("Perfect");
}
```

a. Perfect
b. No output
c. Compilation error
d. None of these

Outputs and Explanations to Code Snippets

11. 15

Explanation:

The expression a=2*3+4%5-3/2+6 gets evaluated as
a=6+4%5-3/2+6
a=6+4-3/2+6
a=6+4-1+6
a=10-1+6
a=9+6
a=15

12. 1 -1 -11

Explanation:

The sign of the result of evaluation of division operator depends upon the sign of both the numerator as well as the denominator. If both are positive, the result will be positive. If either is negative, the result will be negative and if both are negative, the result will be positive.

13. 1 -11 -1

Explanation:

The sign of the result of evaluation of modulus operator depends upon the sign of numerator only. If the numerator is positive, the result will be positive. If the numerator is negative, the result will be negative.

14. 12 12

Explanation:

The comma operator guarantees left-to-right evaluation, but the commas separating the arguments in a function call are not comma operators. They are considered as separators. If commas separating arguments in a function call are considered as comma operators, then no function could have more than one argument. Hence, these arguments are not guaranteed to be evaluated from left to right. The order of evaluation of arguments in a function call is compiler dependent. In Borland TC 3.0 & Borland TC 4.5, evaluation takes place from right to left. Thus, if the code in the given question is executed using the specified compilers, b=a gets evaluated first and b gets the value 12. The result of the evaluation of the expression b=a turns out to be 12, i.e. the value that is assigned. Therefore, 12 12 gets printed.

Separators are used to separate two tokens. Unlike other programming languages:
- Semicolon in C language is a terminator and not a separator. **Terminator** terminates a statement. Statements in C are terminated with semicolon.
- In C language, the white-space character acts as separator.

Forward Reference: Arguments, function call (Chapter 5).

15. 23 23 23 23

Explanation:

d=c=b=a is a valid expression with no compilation error. The assignment operator is right-to-left associative. Thus, d=c=b=a is interpreted as (d=(c=(b=a))). Thus, first the value of a will be placed in b, then the value of b will be placed in c and then the value of c will be placed in d. Hence, all b, c and d will have a value of a, i.e. 23.

The result of evaluation of an expression is an r-value. Assignment expression is no exception to this rule. The result of evaluation of an assignment expression is the value that is assigned. For example, a=10; assigns 10 to a and the overall expression evaluates to 10 (i.e. the assigned value). As described in the explanation above, the value of b is not assigned to c. Actually, the result of evaluation of expression b=a is assigned to c. However, since the result of evaluation of expression b=a is the same as the value of variable b after assignment, the above explanation is also correct.

16. Compilation error (l-value required error)

Explanation:

c+2 and b+1 are expressions. The result of the evaluation of an expression is an r-value, and the assignment operator cannot have an r-value on its left side. Hence, the placement of c+2 and b+1 on the left side of the assignment operator is erroneous and leads to 'L-value required' error.

17. 0

Explanation:

In expression d=a<b>c, three operators namely, assignment operator (=), less than operator (<) and greater than operator (>) are involved. The precedence of the assignment operator is least and less than operator and greater than operator have the same precedence. < and > operators are left-to-right associative. Thus, less than operator (<) will be evaluated first. a<b, i.e. 2<3 turns out to be true, i.e. 1. Now 1>c, i.e. 1>1 is checked and it turns out to be false, i.e. 0. This is assigned to d. Hence, d got the value 0.

18. 0

Explanation:

Out of =, < and - operators, - operator has the highest precedence. Thus, c-1 will be evaluated first and turns out to be 0. a<b is evaluated then and turns out to be 0 (as 3<2 is false). Then 0<0 is evaluated and turns out to be 0. This outcome is assigned to d. Therefore, d will have the value 0.

19. Compilation error (l-value required)

Explanation:

Out of == and = operator, the equality (==) operator has a higher precedence than the assignment operator. Remember that the assignment operator has a lower precedence than every

other operator except the comma operator. The equality operator will be evaluated first. c==a, i.e. 10==30 evaluates to false, i.e. 0. Now, the expression becomes 0=b (i.e. trying to assign a value of b to a constant). It is not allowed, as the assignment operator cannot have a constant (i.e. r-value) on its left side and if it happens (as in this case), there will be 'L-value required' error.

20. 10 20 0

Explanation:

a==b is evaluated first and turns out to be 0. 0 is assigned to c. The values of a and b are not manipulated and remain the same. Hence, the result is 10 20 0.

21. 10 20 30

Explanation:

The equality operator is left-to-right associative. Hence, the expression c==a==b is interpreted as (c==a)==b. The sub-expression c==a is evaluated first and turn out to be 0. Then, 0==b, i.e. 0==20 is evaluated and results in 0. This outcome is not assigned to any variable and will be ignored. Values of a, b and c are not modified anywhere in the function. Hence, the output is 10 20 30.

22. 8 20 16 54

Explanation:

a will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

b will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0

Result of & (Bitwise AND) and |(Bitwise OR) operators is shown in the figure below:

Operator and result	Sign	Magnitude														
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**
b	0	0	0	**0**	**0**	0	0	0	**0**	**0**	**0**	**1**	1	1	0	0
c=a&b=8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
d=a|b=30	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0

c will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0

y will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0

Result of & (Bitwise AND) and |(Bitwise OR) operators is shown in the figure below:

Operator and result	Sign	Magnitude (Magnitude is in two's complement representation)														
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
x	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
y	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0
u=x&y=16	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
v=x|y=54	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0

23. 1111

Explanation:

In C language, a non-zero value is treated as true and zero value is treated as false. Therefore, 012,034, 0x12 and 0x34 are treated as true as all are non-zero values. True && True evaluates to true, i.e. 1. True || True evaluates to true, i.e. 1. Hence, 1 is assigned to c, d, u and v.

24. 0 -11

Explanation:

! is logical NOT operator and ~ is bitwise NOT operator. Logical NOT operator, i.e. ! operates on its operand considering it as a single entity while bitwise NOT operator, i.e. ~ operates on the individual bits of its operand. In d=!c, c is 10, i.e. true. The result of logical negation of true turns out to be false, i.e. 0. Hence, d will have a value of 0.

The value of c (i.e. 10) will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

Bitwise operator (~) negates every bit of c. Hence, the result of bitwise negation will be as follows:

Operator and result	Sign Bit 16 MSB	Magnitude of: c is in normal binary representation e is in two's complement representation														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
c	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
e=~c	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1

Sign bit of e is
two's complem
figure below:

e in two's complement form
Its two's complement

Since sign bit wa

25. 3 -5 -8 -8

Explanation:

Operator and result
c=-4
d=4
~c=3
~d=-5
c^d=-8 (XOR)
~c^~d=-8

26. 10

Explanation:

Actually the resu
operator, the valu
the value of the o
left undefined. Tw

1. According to
   evaluation of
2. According to
   the evaluation

If the value is inc
expression i++*i++
full expression (i.
ent compilers use
TC 3.0 and Borla
expression. Hence

An expression t
is not part of an

Sign bit of e is 1. Therefore, the number e will be negative. Its value can be determined by taking two's complement of the two's complemented representation of its magnitude as shown in the figure below:

	Magnitude (MSB is 1, so magnitude is in two's complement representation)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
e in two's complement form	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1
Its two's complement	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

Since sign bit was 1, the value of e will be -11.

25. 3 -5 -8 -8

Explanation:

Operator and result	Sign Bit 16 MSB	Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
c=-4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
d=4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
~c=3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
~d=-5	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
c^d=-8 (XOR)	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
~c^~d=-8	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0

26. 110

Explanation:

Actually the result of this program snippet is compiler dependent. In the case of a post-increment operator, the value of the operand is used first for the evaluation of expression and **after** its use the value of the operand is incremented. The precise meaning of words '**after**' and '**expression**' is left undefined. Two possible interpretations are as follows:

1. According to the first interpretation, the value of the operand (i.e. i) is incremented after the evaluation of the sub-expression (i.e. i++).
2. According to the second interpretation, the value of the operand (i.e. i) is incremented after the evaluation of full expression (i.e. i++*i++).

If the value is incremented after the evaluation of the sub-expression (i.e. interpretation 1), the expression i++*i++ will be evaluated to 10*11=110. If the value is incremented after the evaluation of full expression (i.e. interpretation 2), the expression i++*i++ will be evaluated to 10*10=100. Different compilers use different interpretations; hence, the result is compiler dependent. In Borland TC 3.0 and Borland TC 4.5 compilers, increment takes place after the evaluation of the sub-expression. Hence, the result is 110.

An expression that is part of another expression is called **sub-expression**. An expression that is not part of another expression is called **full expression**.

27. Compilation error (l-value required)

Explanation:

As the increment operator is right-to-left associative, the expression j=++i++ will be interpreted as j=++(i++). The result of evaluation of sub-expression i++ will be an r-value. This r-value will act as an operand for other increment operator, i.e. pre-increment operator. Thus, the expression reduces to ++(r-value). This reduced expression is erroneous, as increment and decrement operators can only work on operands that have a modifiable l-value. Hence, there will be 'L-value required' error.

28. Compilation error (l-value required)

Explanation:

The tokenizer of C language is greedy in nature. It always tries to create the biggest possible token. Thus, the expression k=i+++j will be treated as k=i++ +j and it is a well-formed expression. The operator ++ will operate first and then operator + will operate. The outcome is assigned to k. In expression l=i+++++j, the tokenizer will divide the operator sequence +++++ into ++ ++ +. Thus, the expression l=i+++++j will be treated as l=i++ ++ +j. The sub-expression i++ will evaluate to an r-value. The second ++ operator cannot operate on an r-value and hence, will lead to 'L-value required' compilation error.

The first phase of a compiler that divides the sequence of input characters into tokens is known as a **tokenizer** or a **lexical analyzer**. C language has **'Greedy Tokenizer'**. It always tries to create the biggest possible token. For example, the sequence of input characters i++-+j will be divided into token sequences i, ++, -, + and j. Consider another example, the sequence of input characters i+++ ++j will be divided into token sequences i, ++, +, ++ and j. Note that the white-space character between two characters is not ignored while tokenizing.

29. 23 21

Explanation:

The expression k=i+++j will be treated as k=i++ +j. First, the value of i is used for the evaluation of sub-expression i++ and then it is incremented by 1. The value of j (i.e. 11) is added to the result of evaluation of i++ (i.e. 10) and the outcome is assigned to k. Therefore, k will be 10+11=21. i will become 11 and j remains 11.

The expression l=i+++ ++j will be treated as l=i++ + ++j. First, the value of i is used for the evaluation of sub-expression i++ and then it is incremented by 1. The value of j will be incremented first and then its value is used for the evaluation of full expression. The result of evaluation of two sub-expressions (i.e. i++ and ++j) is added and is assigned to l. Thus, the value of l becomes 11+12=23. Both i and j become 12 after the evaluation of full expression l=i+++ ++j.

30. Compilation error (l-value required error)

Explanation:

The expression l=i++ +++j will be treated as i++ ++ +j and will give an error due to the reason mentioned in Answer 28.

31. 57 94

Explanation:

In the expression x=y++ + x++, the values of x (i.e. 20) and y (i.e. 35) are used for the evaluation of sub-expressions: y++ and x++. The outcomes of evaluation of these sub-expressions are added, and the result is assigned to variable x (i.e. 20+35=55 and 55 is assigned to x). Then the values of y and x are incremented (i.e. y becomes 36 and x becomes 56). In the next expression, the values of y and x get incremented first (i.e. x becomes 57 and y becomes 37) and then they are used for the evaluation of full expression y=++y + ++x (i.e. y=37+57=94). Hence, x and y become 57 and 94, respectively.

32. Compilation error (l-value required)

Explanation:

In the expression i++=j, the increment operator and the assignment operator are involved. The increment operator ++ has a higher precedence than the assignment operator and will get evaluated first. The result of evaluation of increment operator is an r-value. This r-value lies on the left side of the assignment operator and thus, leads to 'L-value required' error.

33. 100

Explanation:

The expression a>=5 evaluates to true. Hence, the expression b=100 gets evaluated. Value 100 is assigned to variable b and is printed by the next printf statement.

In conditional expression E1?E2:E3, the sub-expression E1 is evaluated first. If it evaluates to a non-zero value (i.e. true), then E2 is evaluated and E3 is ignored. If E1 evaluates to zero (i.e. false), then E3 is evaluated and E2 is ignored.

34. 0231

Explanation:

Rules to be followed:

1. The precedence of the logical AND operator (&&) is higher than the precedence of the logical OR (||) operator. The precedence of the logical AND operator and the logical OR operator is only used to parenthesize the expression involving them.
2. The logical AND operator (&&) and the logical OR operator (||) always guarantee left-to-right evaluation irrespective of their precedence.
3. If the first operand of the logical OR operator (||) evaluates to true, the second operand will not be evaluated, as TRUE || anything (true or false) is TRUE.
4. If the first operand of the logical AND operator (&&) evaluates to false, the second operand will not be evaluated, as FALSE && anything (true or false) is FALSE.

Expression i=i||j++&&++k will be treated as i=i||(j++&&++k), as the logical AND operator has a higher precedence than the logical OR operator. The logical AND and logical OR operator guarantee left-to-right execution. Hence, the expression i=i||(j++&&++k) is executed from left to right. The first operand of the logical OR operator (||), i.e. i is 0, i.e. false; hence, the second operand needs to be evaluated to determine the truth value of full expression. The sub-expression j++&&++k starts evaluation. In sub-expression j++, j is post-incremented. The sub-expression j++ evaluates to 1 and the value of j is incremented to 2. Since the first operand of the logical AND operator, i.e. j++ evaluates to 1 (i.e. true), the second operand (i.e. ++k) needs to be evaluated. In sub-expression ++k, k is pre-incremented. The value of k is incremented first and then its value is used for the evaluation of expression. Thus, the value of k used for the evaluation of expression is 3. Therefore, 1&&3 turns out to be 1. Thus, the second operand of the logical OR operator evaluates to 1. Hence, 0||1 will be evaluated and turns out to be 1. The outcome is assigned to i.
Therefore, the values are i=0, j=2, k=3, l=1.

35. 0120

Explanation:

Since the logical AND operator is left-to-right associative, the expression i=i&&j++&&++k will be interpreted as i=(i&&j++)&&++k. Recall Rule 4 mentioned in the previous answer. In the sub-expression i&&j++, as the first operand of && operator, i.e. i is 0 (i.e. false), j++ will not be evaluated and the sub-

expression i&&j++ evaluates to 0. Due to the same reason, the sub-expression ++k will not be evaluated and the full expression evaluates to 0. 0 is assigned to i. Hence, i=0, j=1, k=2 and i=0.

36. 1 2 3 1

Explanation:

The expression i=++i&&j++&&++k will be interpreted as i=(++i&&j++)&&++k. In the sub-expression ++i&&j++, i is pre-incremented. i becomes 1 and the sub-expression ++i, evaluates to 1. Since the first operand of the && operator evaluates to 1, i.e. true, the sub-expression j++ needs to be evaluated. The sub-expression j++ evaluates to 1 and the value of j becomes 2. As 1&&1 evaluates to 1, the sub-expression ++k will be evaluated. k will become 3. 1&&3 evaluates to 1. Hence, i will get value 1. Therefore, the values are i=1, j=2, k=3, i=1.

37. 1 1 2 1

Explanation:

Since the precedence of the logical AND operator is higher than the logical OR operator, the expression i=++i||j++&&++k will be interpreted as i=++i||(j++&&++k). In the sub-expression ++i, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. 1|| anything (0 or 1) is 1. Hence, the sub-expression (j++&&++k) will not be evaluated. The values of j and k remain 1 and 2, respectively. Therefore, the values are i=1, j=1, k=2, i=1.

38. 1 2 2 1

Explanation:

The expression i=++i&&j++||++k will be interpreted as i=(++i&&j++)||++k. In the sub-expression ++i&&j++, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. Since the first operand of the logical AND operator evaluates to true, the second operand needs to be evaluated. The sub-expression j++ evaluates to 1 and j is incremented to 2. 1&&1 evaluates to 1. 1 || anything (0 or 1) is 1. Therefore, the sub-expression ++k will not be evaluated and the value of k remains 2. Thus, the expression ++i&&j++||++k evaluates to 1 and is assigned to i. Hence, i=1, j=2, k=2 and i=1.

39. 1 0 3 1

Explanation:

The expression i=++i&&--j||++k will be interpreted as i=(++i&&--j)||++k. In the sub-expression ++i&&--j, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. Since the first operand of the logical AND operator evaluates to true, the second operand (i.e. --j) needs to be evaluated. j is decremented to 0 and the sub-expression --j evaluates to 0. Thus, 1&&0 evaluates to 0. As the first operand of the logical OR operator is 0, the sub-expression ++k needs to be evaluated. k becomes 3. 0||3 evaluates to 1 and is assigned to i. Hence, the values are i=1, j=0, k=3 and i=1.

40. 1 0 2 1

Explanation:

The expression i=++i&&j--||++k will be interpreted as i=(++i&&j--)||++k. In the sub-expression ++i&&j--, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. Since the first operand of the logical AND operator evaluates to true, the second operand (i.e. j--) needs to be evaluated. The sub-expression j-- evaluates to 1 and j is decremented to 0. 1&&1 evaluates to 1. The first operand of the logical OR operator evaluates to 1, so the second operand need not be evaluated. Hence, ++k will not be evaluated and the value of k remains 2. The expression ++i&&j--||++k evaluates to 1 and is assigned to i. Hence, i=1, j=0, k=2, i=1.

41. 4 16 1

Explanatio

<< is the left
2. A shift by
flow.
>> is the righ
by 2. A shift
4<<2 is equiva
4>>2 is equiva

The stater
true till th
is stored i

42. -2

Explanation:

x=32767 will be

Sign Bit 16 MSB	Bit 15
0	1

Shift by 1 bit

Sign Bit 16 MSB	Bit 15
1	1

Sign bit becom
two's complem
of 1111 1111 1111 1110 is

A good meth
1. Look f
2. Till 1 is
3. After 1
For example,
two's comple

43. 48

Explanation:

Since, the shift
as (num<<2)<<2. Th
operand for the

41. 4 16 1

Explanation:

<< is the left shift operator. A shift by 1 bit in the left direction is equivalent to multiplication by 2. A shift by n bits is equivalent to multiplication by 2^n, provided the magnitude does not overflow.

>> is the right shift operator. A shift by 1 bit in the right direction is equivalent to integer division by 2. A shift by n bits is equivalent to integer division by 2^n.

4<<2 is equivalent to $4*2^2 = 4*4 = 16$

4>>2 is equivalent to $4/2^2 = 4/4 = 1$.

The statement 'A shift by 1 bit in the left direction is equivalent to multiplication by 2' holds true till there is no overflow in the magnitude field of the number. For example, if an integer is stored in 2 bytes, 32767<<2 will not be 65534 because the magnitude field has overflowed.

42. -2

Explanation:

i=32767 will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Shift by 1 bit in left direction will lead to

Sign Bit 16 MSB	Magnitude is in two's complement representation														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Sign bit becomes 1. Hence, the number will be negative and its value can be determined by taking two's complement of two's complemented representation of its magnitude. Two's complement of 1111 1111 1111 1110 is 0000 0000 0000 0010, i.e. 2. Sign bit was 1, i.e. negative. Hence, the result will be -2.

A good method to **find two's complement** of a number is:

1. Look from the right side in the bits sequence.
2. Till 1 is encountered keep the bits sequence same.
3. After 1 has been encountered, negate every bit, i.e. 0 to 1 and 1 to 0.

For example, consider number 111 1111 1111 1110 ←

two's complement will be 000 0000 0000 0010

43. 48

Explanation:

Since, the shift operator is left-to-right associative, the expression num<<2<<2 will be interpreted as (num<<2)<<2. The sub-expression num<<2, i.e. 3<<2 evaluates to an r-value 12. This r-value acts as an operand for the second shift operator and the sub-expression 12<<2 evaluates to 48.

44. 768

Explanation:

In expression num<<(2<<2), the sub-expression 2<<2 will be evaluated first. The result of its evaluation will be an r-value, i.e. 8. Then, num<<8 (i.e. 3<<8) will be evaluated and results in 768.

Parenthesized sub-expressions are evaluated first.

45. 0

Explanation:

Since the shift operator has a higher precedence than the bitwise AND (&) operator, the expression (num<<i&1<<15)?1:0) will be interpreted as ((num<<i)&(1<<15))?1:0). First, the operand1 of the conditional operator (i.e. sub-expression num<<i&1<<15) will be evaluated as follows:

Operator and result	Sign bit 16 MSB	Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
num=5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
num<<i=num<<1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1<<15	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
num<<1&1<<15=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Since the sub-expression num<<i&1<<15 evaluates to 0, i.e. false, the outcome of the conditional operator will be the result of evaluation of operand3, i.e. 0.

46. 1

Explanation:

The expression (num<<i&1<<15)?1:0) will be interpreted as ((num<<i)&&(1<<15))?1:0). The sub-expression num<<i&1<<15 will be evaluated first. The sub-expression num<<i (i.e. 5<<1) evaluates to 10 and the sub-expression 1<<15 evaluates to -32768. Both are non-zero values and non-zero values are considered as true. Also, true&&true is true. Hence, num<<i&1<<15 evaluates to true, i.e. 1. Since, operand1 of the conditional operator evaluates to true, operand2 (i.e. 1) will be evaluated and results in 1.

47. 1

Explanation:

This question can only be answered after looking at some of the technicalities and intricacies involved in storing floating point numbers. The following facts must be remembered:

1. **Each real floating-type number cannot be represented exactly in memory** (i.e. with infinite precision).
   During their storage, some round-off errors occur. Some real floating-type numbers are stored as a greater value and some are stored as a lesser value.
   Execute the given code and have a look at the output:

```
main()
{
  float a=0.4, b=0.9;
  printf("0.4 is stored as %.20f\n",a);
  printf("0.9 is stored as %.20f",b);
}
```

The output of this code turns to be
0.4 is stored as 0.400000005960464477800 (i.e greater value)
0.9 is stored as 0.899999976158142090000 (i.e smaller value)

2. **Floats are stored in 32 bits (1 bit for Sign, 8 bits for Exponents and 23 bits for Fraction).**
0.9 as a float will be stored in memory as follows:

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
3				F					6			6			6				6				6				6					
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								23-bits for mantissa																						

Backward Reference: Refer Answer number 20 in Chapter 1 to review how float is stored in memory.

3. **Doubles are stored in 64 bits (1 bit for Sign, 11 bits for Exponents and 52 bits for Fraction).**
To store 0.9 (i.e. 0.1110011001100110011001100110011001100...) as double:
   I. Normalize it. Value becomes $1.1100110011001100...^{*} 2^{-1}$
   II. Bias the double exponent with value 1023 like float exponent is biased with 127. Therefore, exponent after biasing becomes -1+1023=1022 i.e. 01111111110 (in binary)
   III. Fractional part is 1100110011001100110011001...

0.9 as double will be stored in memory as follows:

64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
0	0	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
3				F				E				C				C				C				C				C			
S	E	E	E	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	11-bits for exponent											52-bits for mantissa (Continued in the next table)																			

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
C				C				C				C				C				C				C				D			
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
(Continued from the previous table)																															

4. **Last nibble gets rounded off**
Why is the last nibble (i.e. 4 bits) in double represented by D instead of C?
This is because of rounding. C gets rounded to D. This can be confirmed by running the following piece of code:

```
main()
{
  float a=0.9;
  double b=0.9;
  char *p;
  int i;
  p=(char*)&a;
```

```
    printf("Float is stored in memory as:\t");
    for(i=0;i<=3;i++)
    printf("%02X ",(unsigned char)p[i]);
    p=(char*)&b;
    printf("\n Double is stored in memory as:\t");
    for(i=0;i<=7;i++)
    printf("%02X ",(unsigned char)p[i]);
}
```

The above code gives as output

Float is stored in memory as: 66 66 66 3F
Double is stored in memory as: CD CC CC CC CC CC EC 3F

Why is the output like CD CC CC CC CC CC EC 3F instead of 3F EC CC CC CC CC CC CD?
The output is like this because the Intel family of micro-processors stores numbers in **little-endian format.** Therefore, the least significant byte, i.e. CD gets stored in the lowest memory location and hence gets printed first. The most significant byte, i.e. 3F is stored in the highest memory location and will get printed last.

In **little-endian format** of storing numbers, the least significant byte is always stored in the lowest numbered memory location, and the most significant byte is stored in the highest.

5. **When float and double are compared, float gets converted into double first. This type of conversion is called promotion. We say that float gets promoted to double.**
The float value 3F 66 66 66 is promoted to double and becomes:

64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
0	0	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
3				F				E				C				C				C				C				C			
S	E	E	E	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	11-bits for exponent											52-bits for mantissa (Continued in the next table)																			

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				0				0				0				0				0				0				0			
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
(Continued from the previous table)																															

Only 23 fraction bits are available in float. Therefore, when float is promoted to double, the rest of the fraction bits (shown in gray) will be taken as zero. Hence, when 0.9 as float is promoted to double, it becomes 3F EC CC CC C0 00 00 00.

This can be confirmed by running the below-mentioned piece of code:

```
main()
{
  float a=0.9;
  double c;
  int i;
  char *p;
  p=(char*)&a;
  printf("Float value is stored as:\t");
  for(i=0;i<=3;i++)
```

```
    printf("%02X ",(unsigned char)p[i]);
  printf("\n");
  printf("Now float is converted to double\n");
  c=a;
  p=(char*)&c;
  printf("Promoted value is getting stored as:\t");
  for(i=0;i<=7;i++)
    printf("%02X ",(unsigned char)p[i]);
}
```

6. **Comparison of double value and promoted float value**
   Therefore, when this promoted float value (Step No. 5) is compared with the actual double value (Step No. 3) with a less than operator, it results in 1 (i.e. true) because 3F EC CC CC C0 00 00 00 is lesser than 3F EC CC CC CC CC CD.

58. 0

Explanation:

0.5 as float will be stored as:

0	0	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
3				F				0				0				0				0				0			
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								23-bits for mantissa																		

0.5 as double will be stored as follows:

64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3				F				E				0				0				0				0				0			
S	E	E	E	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	11-bits for exponent											52-bits for mantissa (Continued in the next table)																			

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				0				0				0				0				0				0				0			
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
(Continued from the previous table)																															

The fractional part in both the cases is zero. Therefore, when 0.5 as float is promoted to double, it becomes 3F E0 00 00 00 00 00 00. This promoted value is equal to double value (3F E0 00 00 00 00 00 00). Hence, the less than operator on comparison gives zero.

59. 0

Explanation:

0.9, a double value, is demoted to float and is assigned to a float variable a. 0.9f is also stored as a float. In the expression c=a<0.9f, float is compared with float. Loss of precision is the same in both the demotions. Hence, a<0.9f evaluates to 0 and is assigned to c.

Floating-point literal constant by default is of type double.

Backward Reference: Refer Section 1.11.2.1.2 (Floating Point Literal Constant) in Chapter 1 to review length modifiers.

50. 11

Explanation:

The logical OR operator || guarantees left-to-right evaluation. Thus, in the expression ++a==0||++b==11, the sub-expression ++a==0 will be evaluated first. a will be incremented by 1 and the sub-expression ++a evaluates to 1. The sub-expression ++a==0 (i.e. 1==0) evaluates to 0 (i.e. false). As the first operand of the logical OR operator is false, the second operand needs to be evaluated to determine the truth value of the full expression. Thus, the sub-expression ++b==11 will be evaluated. b is incremented by 1 and the expression ++b evaluates to 1. The sub-expression ++b==11 (i.e. 1==11) evaluates to 0, i.e. false. Both the operands of the logical OR operator have evaluated to 0. Thus, the full expression evaluates to zero. This outcome is not assigned to any variable and will be ignored. Hence, the values of a and b that get printed are 1 and 1.

51. 6

Explanation:

In expression 4+2%-8, the modulus operator has the highest precedence. The result of the modulus operator depends only upon the sign of the numerator. Thus, the sub-expression 2%-8 evaluates to 2. This outcome is added to 4 and is assigned to x. Therefore, x will have value 6.

52. 0

Explanation:

As the logical NOT operator (i.e. !) has a higher precedence than the greater than operator (>), it gets evaluated first. The sub-expression !i (i.e. !5) evaluates to 0. This outcome is compared with 3 and the sub-expression 0>3 evaluates to 0 (i.e. false). This outcome is assigned to i.

53. 20 20 20

Explanation:

The assignment operator is right-to-left associative. The sub-expression a*=2 (i.e. a=a*2) is evaluated first. It evaluates to 20 and is assigned to a. The value of a is then assigned to b and b will become 20. The value of b is assigned to c and c will also become 20. Hence, all a, b and c are 20.

54. fff0

Explanation:

-1 will be stored in memory as follows:
MSB will be 1 as the sign is negative. Magnitude will be two's complemented representation of 1, i.e. 111 1111 1111 1111. This value is shifted in the left direction by 4 bits and the outcome of the shift operation is as follows:

Operator and result	Sign bit 16 MSB	Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-1<<4	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**
In Hexadecimal	f				f				f				0			

55. 2

Explanation:

In the expres
to-left associ
minus makes

56. Compilation error

Explanation:

-- is not the s
decrement op
l-value. In the

57. 4 5 5 4 5

Explanation:

In Borland TC
i is 5. so the su
to 4 and the s
and evaluates
to 5 and then
value of i (i.e.
printf function
Therefore, the

58. 200

Explanation:

200; is a valid s

Forward R

59. -1 -1

Explanation:

In expression
as ++i. In the n

Unary plus

60. 0

Explanation:

2 is considered
assigned to th

61. 1==1 is True

Explanation:

In Borland TC
expression k==
result of the c
enated. Hence
matched with
the result that

55. 2

Explanation:

In the expression - -2, both the occurrences of - are instances of unary minus operator. It is right-to-left associative. The rightmost unary minus will first make 2 as -2. Then the second unary minus makes this -2 as 2. Therefore, the result will be 2.

56. Compilation error (l-value required)

Explanation:

-- is not the same as - -. It is one token (i.e. one operator, namely pre-decrement operator). The pre-decrement operator cannot operate on constants and requires an operand that has a modifiable l-value. In the given question, since -- is applied on constant, it shows 'L-value required' error.

57. 4 5 5 4 5

Explanation:

In Borland TC 3.0 and 4.5, arguments of printf function are evaluated from the right. The value of i is 5, so the sub-expression i evaluates to 5. In the sub-expression --i, the value of i is decremented to 4 and the sub-expression evaluates to 4. The sub-expression ++i increments the value of i to 5 and evaluates to 5. In the sub-expression i--, i is post-decremented. The sub-expression evaluates to 5 and then i is decremented to 4. In the sub-expression i++, i is post-incremented, so first the value of i (i.e. 4) will be used and then it is incremented to 5. After the evaluation of values, the printf function prints the values in a left-to-right order according to the given format specifiers. Therefore, the values that get printed are 4 5 5 4 5.

58. 200

Explanation:

200; is a valid statement but does nothing. In the next statement 200 is printed by the printf function.

Forward Reference: Statements (Chapter 3).

59. -1 -1

Explanation:

In expression +i, + is unary plus and will not have any effect on the value of i. It is not the same as ++i. In the next statement, the unmodified value of i gets printed. Hence, -1 -1 is the result.

Unary plus does nothing and is known as the **Dummy operator.**

60. 0

Explanation:

2 is considered as true as it is a non-zero value. !TRUE evaluates to false, i.e. 0. The outcome is assigned to the identifier not. This value of the identifier not is printed in the next statement.

61. 1==1 is True

Explanation:

In Borland TC 3.0 and 4.5, arguments of the printf function are evaluated from the right. Thus, expression k==1?"True":"False" is evaluated first. The sub-expression k==1 evaluates to true, hence the result of the conditional operator turns out to be "True". Also, adjacent string literals get concatenated. Hence, "%d==1 is""%s" will get concatenated to form "%d==1 is %s". The integer specifier is matched with k, which has value 1, and the string specifier %s is matched with string "True". Hence, the result that gets printed is "1==1 is True".

62. Compilation error (Cannot modify a constant object)

Explanation:

The expression ++i is erroneous as i is defined as a qualified constant.

Qualified constants do not have a modifiable l-value. Hence, it cannot be used as the operand of an increment/decrement operator.

63. 1

Explanation:

First, the value of i is incremented by 1 and it becomes 6. 6 is compared for equality with 6 and evaluates to true, i.e. 1. This outcome is then assigned to i and gets printed.

64. 11

Explanation:

The logical AND operator && has a higher precedence than the assign-bitwise AND operator &=. The sub-expression j&&10 is evaluated first (i.e. 10&&10) and turns out to be true, i.e. 1. The sub-expression i&=1 is equivalent to i=i&1 (i.e. i=5&1). On evaluation it gives 1. Therefore, i will take value 1. This value of i is assigned to j. Hence, both i and j will have value 1.

65. 2695.000000 385.000000

Explanation:

y+28.5 is computed first and turns out to be 38.5. y*=38.5 is computed then and the value of y becomes 385.0. Then, x*=385.0 is computed and the value of x becomes 2695.0. Therefore, the values that get printed are 2695.000000 and 385.000000.

66. ffff

Explanation:

~a does not change the value of a. The value of a remains the same and gets printed as ffff.

67. 256

Explanation:

0x80 is 1000 0000, shifting by 1 bit in the left direction gives 1 0000 0000 and this is equivalent to 256 in decimal.

68. 65535 0

Explanation:

-1 will be stored in memory as follows:

Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

i.e. all sixteen 1's. -1 is assigned to a. Therefore, a becomes

	Sign bit 16	Magnitude													
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
a=-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
++a 1 (carry gets overflowed)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

a is declared as unsigned. Therefore, the 16th bit is not considered as a sign bit but it is considered as a magnitude bit. Therefore, the value of a that gets printed is 65535. If one is added to a, carry overflows and the result turns out to be 0.

69. 4 3 3 3

Explanation:

In v=(int)(u+0.5), first 3.5+0.5 is evaluated and turns out to be 4.0. This is then type casted to the integer and becomes 4. 4 is then assigned to v.
In w=(int)u+0.5, first u is type casted to the integer, i.e. it becomes 3. Then 0.5 is added to make it 3.5. This value is then assigned to an integer variable. Before assignment, demotion will be carried out. 3.5 will be demoted to 3 and then assigned to w.
In x=(int)((int)u+0.5), x will get a value 3. Instead of implicitly demoting 3.5 to 3 as in the previous case, it is now explicitly type casted to 3.
In y=(u+(int)0.5), first 0.5 is type casted to 0. 0 is added to 3.5 and it comes out to be 3.5. 3.5 after implicit demotion is assigned to an integer variable y. Hence, the value assigned to y will be 3.

Type casting can be done explicitly by using a type cast operator. The syntax of using a type cast operator is (target-type-name) expression.

70. 3 3 3 3

Explanation:

The identifier u is declared as int. Therefore, 3.5 will be demoted to 3 and will then be assigned to u. Hence, u will have the value 3 instead of 3.5. All the remaining computations are carried out in the same way as in the previous answer.

Answers to Multiple-choice Questions

71. a 72. c 73. a 74. d 75. a 76. b 77. b 78. a 79. b 80. a 81. c 82. c 83. b 84. b 85. a

Programming Exercises

Program 1 | Find one's and two's complement of a number

Algorithm:
Step 1: Start
Step 2: Read the number (num)
Step 3: One's complement (oc) = ~num i.e. negate every bit using bitwise NOT operator
Step 4: Two's complement (tc) = oc+1 i.e. two's complement is one's complement plus 1
Step 5: Print values of oc and tc
Step 6: Stop

Line | PE 2-1.c

```
1  //One's and Two's complement
2  #include<stdio.h>
3  main()
4  {
5      int num, oc, tc;
6      printf("Enter number\t");
7      scanf("%d",&num);
8      oc=~num;
9      tc=oc+1;
10     printf("One's complement is %d\n",oc);
```

Memory content

num=2

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

oc = -3 (Two's complement representation)

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1

Output window

```
Enter number  2
One's complement is -3
Two's complement is -2
```

Remarks:

- ~ is bitwise NOT operator
- The sign bits of oc and tc are 1. Hence, they are negative and are stored in two's-complement representation

(Contd...)

Line	PE 2-1.c	Memory content	Output window
11 12	printf("Two's complement is %d\n",tc); }	tc = -2 (Two's complement representation)	

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Program 2 | Assuming that bit numbering starts from 1. Write a C program to set a particular bit in a given number

Algorithm:
Step 1: Start
Step 2: Read the number (num)
Step 3: Read the bit number (bit) that is to be set (i.e. to be made 1) in the given number
Step 4: Construct a temporary number such that it has 1 at the bit position that is to be set in the given number and zero elsewhere. Temporary number can be constructed by using left-shift operator as temp=1<<(bit-1)
Step 5: To set the bit in the given number, perform bitwise OR of the number with the constructed temporary number and save result in the number i.e. num=num|temp
Step 6: Print number (num)
Step 7: Stop

Line	PE 2-2.c	Memory content	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
	//Set particular bit in a given number
	#include<stdio.h>
	main()
	{
	int num, bit, temp;
	printf("Enter number\t");
	scanf("%d",&num);
	printf("Enter the bit number to be set\t");
	scanf("%d",&bit);
	temp=1<<(bit-1);
	num=num|temp;
	printf("Value after setting bit is %d", num);
	}
		num=5
		After setting bit 2, the value of num becomes
		i.e.7
			Enter number 5
			Enter the bit number to be set 2
			Value after setting bit is 7

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Program 3 | Assuming that bit numbering starts from 1. Write a C program to negate a particular bit in a given number

Algorithm:
Step 1: Start
Step 2: Read the number (num)
Step 3: Read the bit number (bit) that is to be negated (i.e. to be made 1 if it is 0 and vice-versa) in the given number
Step 4: Construct a temporary number such that it has 1 at the bit position that is to be negated in the given number and zero elsewhere. Temporary number can be constructed by using left-shiftoperator as temp=1<<(bit-1)
Step 5: To negate the bit in the given number, perform bitwise XOR of the number with the constructed temporary number and save result in the number i.e. num=num^temp
Step 6: Print number (num)
Step 7: Stop

(Contd...)

Line	PE 2-3.c	Memory content	Output window
			Enter number 5 Enter the bit number to be negated 2 Value after negating bit is 7
			Output window (second execution)
			Enter number 5 Enter the bit number to be negated 3 Value after negating bit is 1

```
1  //Negate a particular bit in a given number
2  #include<stdio.h>
3  main()
4  {
5  int num, bit, temp;
6  printf("Enter number\t");
7  scanf("%d",&num);
8  printf("Enter the bit number to be negated\t");
9  scanf("%d",&bit);
10 temp=1<<(bit-1);
11 num=num^temp;
12 printf("Value after negating bit is %d", num);
13 }
```

num=5

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

After negating bit 2, the value of num becomes

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

i.e.7

num=5

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

After negating bit 3, the value of num becomes

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

i.e.1

Program 4 | Given two numbers, say val and key. Wherever the bits of number key are 1, set the corresponding bits of number val. Leave all other bits of number val unchanged

Algorithm:
Step 1: Start
Step 2: Read the numbers, val and key
Step 3: val=val|key
Step 4: Print number (val)
Step 5: Stop

Line	PE 2-4.c	Memory content	Output window
			Enter two numbers 4 10 After setting bits, result is 14
			Output window (second execution)
			Enter two numbers 4 5 After setting bits, result is 5

```
1  //Set the corresponding bits
2  #include<stdio.h>
3  main()
4  {
5  int val, key;
6  printf("Enter two numbers\t");
7  scanf("%d %d",&val, &key);
8  val=val|key;
9  printf("After setting bits, result is %d",val);
10 }
```

val=4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

key = 10

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

After setting the corresponding bits, val becomes 14

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

Program 5 | Given two numbers, say val and key. Wherever the bits of number key are 1, negate the corresponding bits of number val. Leave all other bits of number val unchanged

Algorithm:
Step 1: Start
Step 2: Read the numbers, val and key
Step 3: val=val^key
Step 4: Print number (val)
Step 5: Stop

Line	PE 2-5.c
1	//Negate the corresponding bits
2	#include<stdio.h>
3	main()
4	{
5	int val, key;
6	printf("Enter two numbers\t");
7	scanf("%d %d",&val, &key);
8	val=val^key;
9	printf("After negating bits, result is %d",val);
10	}

Memory content

val = 4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

key = 5

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

After negating the corresponding bits, val becomes 1

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Output window

```
Enter two numbers 2 5
After negating bits, result is 7
```

Output window (second execution)

```
Enter two numbers 4 5
After negating bits, result is 1
```

Program 6 | Given two numbers, say val and key. Wherever the bits of number key are 1, reset (i.e. make 0) the corresponding bits of number val. Leave all other bits of number val unchanged

Algorithm:
Step 1: Start
Step 2: Read the numbers, val and key
Step 3: Construct a temporary which is one's complement of the key i.e. temp=~key.
Step 4: val=val&temp
Step 5: Print number (val)
Step 6: Stop

Line	PE 2-6.c
1	//Reset the corresponding bits
2	#include<stdio.h>
3	main()
4	{
5	int val, key, temp;
6	printf("Enter two numbers\t");
7	scanf("%d %d",&val, &key);
8	temp=~key;
9	val=val&temp;
10	printf("After resetting bits, result is %d",val);
11	}

Memory content

val = 4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

key = 5

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

After resetting the corresponding bits, val becomes 0

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Output window

```
Enter two numbers 4 5
After resetting bits, result is 0
```

Output window (second execution)

```
Enter two numbers 2 5
After resetting bits, result is 2
```

Program 7 | Write a C program to multiply a given number with 2^n, without using a multiplication operator. The value of n will be entered by the user

Algorithm:
Step 1: Start
Step 2: Read a number (num).
Step 3: Input the value of n.
Step 4: To multiply number with 2^n, shift the bits of number in left direction n times i.e. res=num<<n
Step 5: Print number (res)
Step 6: Stop

Line	PE 2-7.c
1	//Multiply by 2 raise to the power n
2	#include<stdio.h>
3	main()
4	{
5	int num, n, res;
6	printf("Enter number to be multiplied\t");
7	scanf("%d",&num);
8	printf("Enter value of n\t");
9	scanf("%d",&n);
10	res=num<<n;
11	printf("Result of multiplication is %d",res);
12	}

Memory content

val = 4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

res = 16

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Output window

Enter number to be multiplied 2
Enter value of n 3
Result of multiplication is 16

Remark:

- Left shift by n bits is equivalent to multiplication by 2^n, provided the magnitude does not overflow

Test Yourself

1. Fill in the blanks in each of the following:
   a. An _____ specifies an entity on which operation is to be performed.
   b. An expression that has only one operator is known as _____.
   c. Assignment operator is _____ associative.
   d. The operands of a modulus operator must be of _____ type.
   e. The result of evaluation of a relational expression is a _____.
   f. In a compound expression, if operators of different precedence appear together, the operator of _____ precedence operates first.
   g. The order in which operators operate depends upon the _____ and the _____ of the operators.
   h. If the operands of an operator are of different types, C automatically applies _____ to bring the operands to a common type.
   i. The _____ operator returns the number of bytes the operand occupies.
   j. _____ operator has least precedence.

2. State whether each of the following is true or false. If false, explain why.
   a. The operators with the same precedence always have the same associativity.
   b. The multiplication and division operators are left-to-right associative.
   c. The sign of the result of evaluation of the modulus operator depends upon the sign of both the numerator as well as the denominator.
   d. The knowledge of precedence alone is sufficient to evaluate a compound expression.
   e. Conditional operator is a binary operator.
   f. The increment operator can only be applied to an operand that has a modifiable l-value.
   g. The expression ++a is equivalent to a+=1.
   h. In C language, there is no operator available for logical eXclusive-OR (XOR) operation.
   i. Qualified constant cannot be initialized with a value.
   j. The expression !(x>=y) is equivalent to the expression x<y.

3. Find the result of evaluation for the following expressions:
   a. 5*3/4-2
   b. ~5+3&2
   c. 4-5&&2
   d. 2<<2>>2
   e. 2<<2>2
   f. 2<3,4,5*3+2
   g. 5 != 10 && 2 | 3&5
   h. 2?2^2:2|5
   i. 3?~2?~5:4:3
   j. +2.25+-3.85

4. Which of the following expressions are valid? If valid, find the result of evaluation of expressions, assuming identifiers a and b are defined and their values are a=10 and b=15. If invalid, identify the errors.
   a. a+++b=20
   b. a=b==12==b
   c. ++a=23*5-4
   d. b=7.5%2.5

e. 2==3+5+=6
f. 2*3/2.0&3
g. a+++++b
h. ~2~3^4
i. a^=b^=10
j. a&&=10

3

STATEMENTS

Learning Objectives

In this chapter, you will learn about:

- Statements
- How statements are classified
- Non-executable statements and executable statements
- Simple statements and compound statements
- Declaration statement and definition statement
- Null statement and expression statements
- Labeled statements
- Flow control statements
- How to implement decision making
- Selection statements and jump statements
- How to perform iteration
- Iteration statements
- Role of **break** and **continue** statements
- Graphical representation of flow of control

3.1 Introduction

In the last chapter, you have learnt how to form and evaluate expressions. In C language, the expressions do not have any independent existence. To make them exist, they must be converted into statements. A **statement** is the smallest logical entity that can independently exist in a C program. In this chapter, I will tell you how to convert expressions into statements. I will also describe how statements are executed, how to make decisions with the help of branching statements and how to make a set of statements execute a number of times by using iteration statements. Finally, we will look at how various statements can be used in conjunction to perform meaningful tasks.

3.2 Statements

A **statement** is the smallest logical entity that can independently exist in a C program. No entity smaller than a statement, i.e. expressions, variables, constants, etc. can independently exist in a C program unless and until they are converted into statements. The code snippet in Program 3-1 proves the above-mentioned fact.

Line	Prog 3-1.c	Output window
1	//Expressions cannot exist independently	Compilation error "Statement missing ; in function main"
2	#include<stdio.h>	**Reason:**
3	main()	• Expression in line 6 cannot exist independently. It should be a part of some statement
4	{	**What to do?**
5	int a;	• Convert expression a=2+3 into a statement by terminating it with a semicolon and re-execute the code
6	a=2+3	
7	printf("Value of a is %d",a);	
8	}	

Program 3-1 | A program to illustrate that an expression cannot exist independently

A statement in a programming language is analogous to a sentence in a natural language. Just as sentences are terminated with a period (i.e. full stop) in the English language, statements in C language are terminated with a semicolon. When an expression is terminated with a semicolon, it forms an **expression statement**. For example, a=2+3 is an expression. When it is terminated with a semicolon, it forms an expression statement, i.e. a=2+3;. Expression statements are classified according to the type of operator involved in the expression. Since an assignment operator is involved in the expression statement a=2+3;, it can be called an **assignment statement**. Moreover, as an arithmetic operator (+) is also involved in the expression statement a=2+3;, it can also be called an **arithmetic statement**.

3.3 Classification of Statements

Statements in C are classified according to the following criteria:

1. Based upon the type of action they perform
2. Based upon the number of constituent statements
3. Based upon their role

3.3.1 Based Upon the Type of Action they Perform

A statement specifies an action to be performed. Based upon the type of action it performs, the statements in C are classified into the following:

1. Non-executable statements
2. Executable statements

3.3.1.1 Non-executable Statements

Non-executable statements tell the compiler how to build a program. The important points about non-executable statements are listed as follows:

1. These statements help the compiler to determine how to allocate memory, interpret and compile other statements in a program.
2. These statements are intended mainly for the compiler, and no machine code is generated for them. Only executable† statements play a role during the execution of a program.
3. The order in which non-executable statements appear in a program is important. When a compiler compiles a program, it scans all the statements from top to bottom. A non-executable statement can only affect the statements that appear below it. Thus, a non-executable statement should appear only before executable statements within a block.‡
4. Only non-executable statements can appear outside the body of a function.➲
5. Examples of non-executable statements are function prototypes,➲ global variable➲ declarations, constant declarations and preprocessor directive➲ statements.

Although the separation between executable and non-executable statements is simple and effective, it was rather rigid earlier. This rigidity was relaxed in the C99 standard, and flexibility in terms of freely mixing executable and non-executable statements was provided.

Forward Reference: Function and function prototype (Chapter 5), global variables (Chapter 7), preprocessor directives (Chapter 8).

3.3.1.2 Executable Statements

Executable statements represent the instructions that are to be performed when the program is executed. The important points about executable statements are listed as follows:

1. For an executable statement, some machine code is generated by the compiler.
2. An executable statement can appear only inside the body of a function.
3. The examples of executable statements are assignment statements, branching statements, looping statements, function call statements, etc.
4. A global definition like `const int obj=10;` appears to be an executable statement, but it is a non-executable statement.

The code segment in Program 3-2, if compiled with compilers conforming to pre-C99 standards, illustrates the fact that within a block, non-executable statements can appear only before an executable statement.

---

†Refer Section 3.3.1.2 for a description on executable statements.
‡Refer Section 3.3.2.2 for a description on blocks.

Line	Prog 3-2.c	Output window
1	//Executable and Non-executable statements	Compilation error "Declaration is not allowed here"
2	#include<stdio.h>	**Remarks:**
3	#include<conio.h>	• Borland TC 3.0 generates this error but some compilers (like Borland TC 4.5 and other latest compilers) do not enforce this constraint and does not produce an error
4	main()	• File must be saved with .C extension and not with .CPP extension
5	{	**Reason:**
6	clrscr();	• Line 6 is an executable statement but line 7 is a non-executable statement. If a compiler conforming to pre-C99 standards is used, non-executable statements can appear only before executable statements
7	int a=10;	**What to do?**
8	printf("Value of a is %d",a);	• Interchange lines 6 and 7 and re-execute the code
9	}	

Program 3-2 | A program that emphasizes on the order of occurrence of executable and non-executable statements

The code snippet in Program 3-3 illustrates the fact that executable statements can appear only inside the body of a function while non-executable statements can even appear outside the body of a function, i.e. in global scope.

Line	Prog 3-3.c	Output window
1	//Executable and Non-executable statements	Compilation error "Type name expected"
2	#include<stdio.h>	**Reason:**
3	#include<conio.h>	• Line 5 is an executable statement. Executable statements can appear only inside the body of a function, i.e. in local scope. Hence, line 5 leads to the compilation error
4	int a=10;	**What to do?**
5	a=a*2;	• Place content of lines 5 after line 7 and re-execute the code
6	main()	
7	{	
8	printf("Value of a is %d",a);	
9	}	

Program 3-3 | A program to show that executable statements can appear only inside the body of a function

3.3.2 Based Upon the Number of Constituent Statements

Based upon the number of constituent statements, statements in C language are classified as follows:

1. Simple statements
2. Compound statements

3.3.2.1 Simple Statements

A **simple statement** consists of a single statement. It is terminated with a semicolon. Examples of simple statements are as follows:

```
1. int variable=10;          //←definition statement
2. variable+5;               //←expression statement
3. variable=variable+10;     //←assignment statement
```

3.3.2.2 Compound Statements

A **compound statement** consists of a sequence of simple statements enclosed within a pair of braces.[‡] An example of a compound statement is as follows:

```
{                            //← a compound statement consisting of three simple statements
int variable=10;
variable=variable*2;
variable+=5;
}
```

The important points about compound statements are listed below:

1. A compound statement is also known as a **block**.
2. A compound statement need not be terminated with a semicolon. However, if it is terminated with a semicolon, there will be no compilation error but it will be interpreted in a different way.[§]
3. A compound statement can be empty, i.e. there is no simple statement present inside the pair of braces, like {}. An empty compound statement is equivalent to a null[¶] statement, but it cannot act as a terminator for a statement. Figure 3.1 illustrates the interpretation of this fact.

`if(a==b)`
`{`
`}`
	Equivalent to	`if(a==b)`
		`:  //←null statement`
			Valid as {} is equivalent to null statement (i.e..)

`printf("Hello"){}`	Not equivalent to	`printf("Hello");`	Invalid as {} cannot act as a terminator

Figure 3.1 | Empty compound statement acts as a null statement but not as a terminator

4. A compound statement is treated as a single unit. If there is no jump[††] statement present inside the block, all the constituent simple statements will be executed in a sequence if the program control enters the block.
5. A compound statement can appear at any point in a program wherever a simple statement can appear.
6. In a block, non-executable statements (e.g. declaration statements) should come before executable statements.

[‡] Refer Section 3.3.3.2 for a description on how a compound statement terminated with a semicolon is interpreted.

[§] Refer Section 3.3.3.2 for a description on null statement.

[¶] Refer Section 3.3.3.4.1.2 for a description on jump statements.

Curly brackets, i.e. {}, are known as **braces**.

3.3.3 Based Upon their Role

Based upon their role, statements are classified as follows:

1. Declaration statement and definition statement
2. Null statement and expression statement
3. Labeled statements
4. Flow control statements
   a. Branching statements
      i. Selection statements
      ii. Jump statements
   b. Iteration statements

3.3.3.1 Declaration Statement and Definition Statement

The role of a **declaration statement** is to introduce the name of an identifier along with its data type to the compiler before its use. All identifier names (except label names) need to be declared before they are used. During declaration, no memory is allocated to an identifier. The memory space for an identifier can be reserved by using a **definition statement**. The definition statement declares an identifier and also reserves the memory space for it depending upon its data type. For example, `int a;` is a definition statement, which reserves 2 bytes (or 4 bytes) for `a` in the memory. To declare `a`, write `extern int a;`.

Forward Reference: extern, storage class specifiers (Chapter 7).

3.3.3.2 Null Statement and Expression Statements

A **null statement** just consists of a semicolon. For example:

```
;        //← is a null statement
```

A null statement is the simplest form of program statement and **performs no operation**. It is just used as a place-holder, i.e. it is used when the syntax of a language construct requires a statement to be present, but the logic of a program does not require it. A null statement is equivalent to an empty compound statement, i.e. {}. A compound statement need not be terminated with a semicolon. However, if it is terminated with a semicolon, it is interpreted as a compound statement followed by a null statement. Figure 3.2 illustrates the interpretation of a compound statement, which is terminated with a semicolon.

Computations in C language are performed with the help of expression statements. An expression terminated with a semicolon forms an **expression statement.** For example:

```
a=2+3;     //← is an expression statement
```

Expression statements like `printf("Hello Readers");` in which the function call operator (i.e. ( )) is involved are called **function call statements** or **function invocations**.

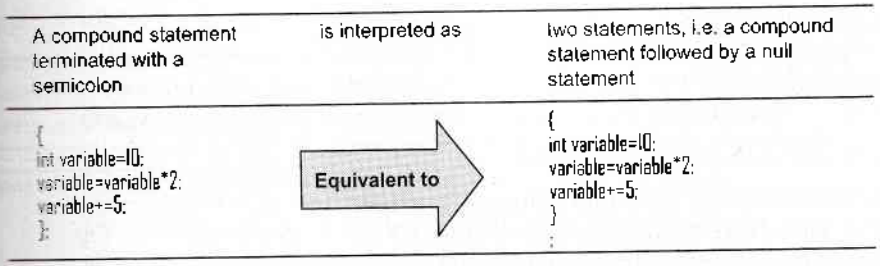


Figure 3.2 | Interpretation of a compound statement terminated with a semicolon

3.3.3.3 Labeled Statements

Labeled statements are rarely used in isolation. They have practical significance only when they are used in conjunction with branching statements. In the following sub-sections, the syntax of labeled statements is described. Their practical application will be discussed along with the branching statements.[++] Labeled statements are of three types:

1. Identifier-labeled statements
2. Case-labeled statements
3. Default-labeled statements

Practically, an identifier-labeled statement is used in conjunction with a goto[§§] statement. Case-labeled and default-labeled statements are useful only when they are used in conjunction with a switch[¶¶] statement.

3.3.3.3.1 Identifier-labeled Statements

The general form of an **identifier-labeled statement** is:

```
identifier: statement
```

The important points about identifier-labeled statements are listed below:

1. The identifier used in an identifier-labeled statement is called a **label name**. For example, in the following identifier-labeled statement, lab is the label name:

```
lab: printf ("Labeled statement");
```

2. Unlike other identifiers, i.e. variable names, function names, etc., label names are not explicitly declared by using declaration statements. They are not explicitly declared because:
   a. There is no type associated with them.
   b. No operation is allowed on them. Unlike other identifiers, they cannot be used as an operand in an expression.
3. Label names are implicitly (i.e. automatically) declared by their syntactic appearance, i.e. an identifier followed by a colon and a statement is implicitly treated as a label name.
4. The statement after the label name in an identifier-labeled statement can be any statement, even some another labeled statement. For example, the following statement is an identifier-labeled statement whose constituent statement is another identifier-labeled statement.

[++] Refer Section 3.3.3.4.1 for a description on branching statements.
[§§] Refer Section 3.3.3.4.1.2.1 for a description on goto statement.
[¶¶] Refer Section 3.3.3.4.1.1.5 for a description on switch statement.

```
label1:          //←An identifier-labeled statement whose
    label2:      //←Constituent statement is another identifier-labeled statement
        printf("Identifier labeled statement's statement is another identifier labeled statement");
```

5. Label name should be unique within a function.
6. Label names do not alter the flow of control.[†††]
7. Identifier-labeled statements have practical significance only when they are used in conjunction with a goto statement.

The piece of code in Program 3-4 illustrates that label names do not impede the flow of control.

Line	Prog 3-4.c	Output window
1	//Identifier labeled statements	Identifier labeled statement
2	#include<stdio.h>	**Remarks:**
3	main()	• Label names do not alter the flow of control
4	{	• label1 followed by label2, followed by the printf statement is one statement. Thus, the mentioned code has only one simple statement
5	label1:	
6	label2:	
7	printf("Identifier labeled statement");	
8	}	

Program 3-4 | A program to illustrate that label names do not alter the flow of control

3.3.3.3.2 Case-labeled Statements

The general form of a **case labeled statement** is:

```
case constant-expression: statement
```

The important points about case labeled statements are as follows:

1. A case-labeled statement consists of the keyword case followed by a constant expression (i.e. **case label**), followed by a colon and then a statement. An example of a valid case-labeled statement is as follows:

```
case 2: printf("case labeled statement");
```

2. The case label should be a compile time constant expression of integral type. For example, the following case-labeled statements are valid:

```
a. case 2+3: printf("Valid");   //←2+3 is compile time constant expression of int type
b. case a: printf("Valid");     //←where a is qualified constant of integral type
c. case 'A': printf("Valid");   //←'A' is a character constant
```

The following case-labeled statements are not valid:

```
a. case j: printf("Invalid");   //←j is variable and not a constant
b. case 2.5: printf("Invalid"); //←2.5 is an expression of float type and not of integral type
```

3. Case-labeled statements can appear only inside the body of a switch[‡‡‡] statement.

[†††] Refer Section 3.3.3.4 for a description on flow of control and flow control statements.
[‡‡‡] Refer Section 3.3.3.4.1.1.5 for a description on switch statement.

4. The constituting statement of a case-labeled statement can be any statement, even some other case-labeled statement with a different case label. For example, a case-labeled statement whose constituent statement is another case-labeled statement having a different case label is as follows:

```
case 1:          //←Case-labeled statement whose
    case 2:      //←Constituent statement is another case-labeled statement
        printf("Case labeled statement's statement is another case labeled statement");
```

2.3.3.3 Default-labeled Statements

The general form of a **default labeled statement** is:

```
default: statement
```

The important points about default labeled statements are as follows:

1. A default-labeled statement consists of the keyword default followed by a colon and a statement.
2. A default-labeled statement can appear only inside the body of a switch statement.
3. The constituting statement of a default-labeled statement can be any statement except the default-labeled statement. If a default-labeled statement is the constituting statement of another default-labeled statement, it leads to 'Too many default cases' compilation error. For example, the following default-labeled statement is not valid:

```
default:         //←Default-labeled statement cannot have another default-labeled statement
    default:
        printf("This is not valid");
```

2.3.3.4 Flow Control Statements

By default, statements in a C program are executed in a sequential order. The order in which the program statements are executed is known as **'flow of program control'** or just **'flow of control'**. By default, the program control flows sequentially from top to bottom. All the programs that we have developed till now have default flow of control. Many practical situations like decision making, repetitive execution of a certain task, etc. require deviation or alteration from the default flow of program control. The default flow of control can be altered by using **flow control statements**. Flow control statements are of two types:

1. Branching statements
   a. Selection statements
   b. Jump statements
2. Iteration statements

2.3.3.4.1 Branching Statements

Branching statements are used to transfer the program control from one point to another. They are categorized as:

1. Conditional branching: In **conditional branching**, also known as **selection**, program control is transferred from one point to another based upon the outcome of a certain condition. The following **selection statements** are available in C: if statement, if-else statement and switch statement.
2. Unconditional branching: In **unconditional branching**, also known as **jumping**, program control is transferred from one point to another without checking any condition. The following **jump statements** are available in C: goto statement, break statement, continue statement and return statement.

3.3.3.4.1.1 Selection Statements

Based upon the outcome of a particular condition, **selection statements** transfer control from one point to another. Selection statements select a statement to be executed among a set of various statements. The selection statements available in C are as follows:

1. if statement
2. if-else statement
3. switch statement

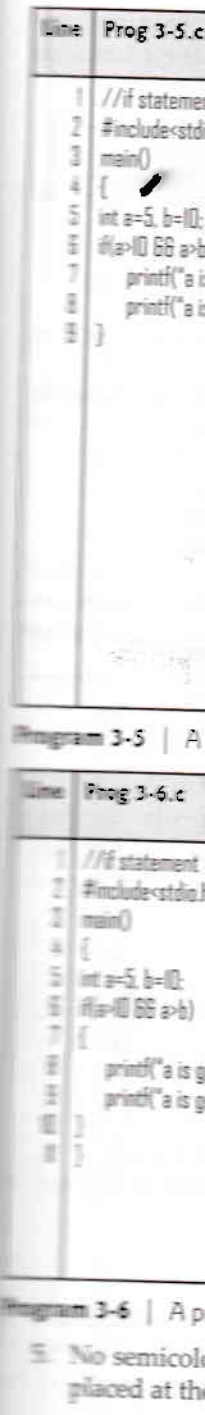
3.3.3.4.1.1.1 if Statement

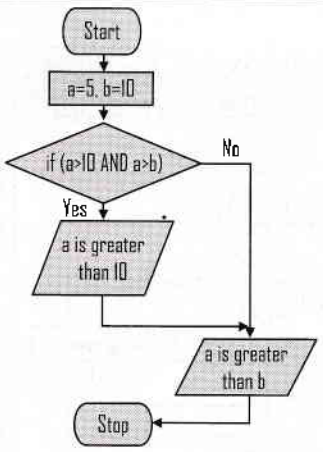
The general form of **if statement** is:

```
if(expression)      //←if header
statement           //←if body
```

The important points about an if statement are as follows:

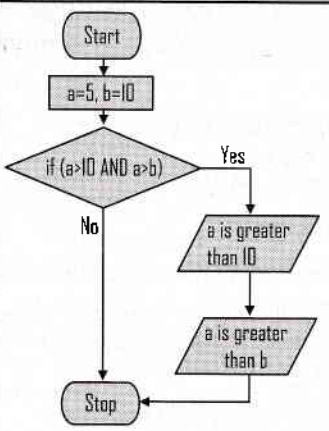
1. An if statement consists of an if header and an if body.
2. An if header consists of an if clause followed by an **if controlling expression** enclosed within parentheses.
3. An if statement is executed as follows:
   a. The if controlling expression is evaluated.
   b. If the if controlling expression evaluates to true, the statement constituting if body is executed.
   c. If the if controlling expression evaluates to false, if body is skipped and the execution continues from the statement following the if statement.
4. The syntax of an if statement permits only a single statement to be associated with if header. Practical applications often require that the execution of two or more statements should depend upon the outcome of a particular condition. In such cases, the dependent statements should be clubbed together to form a compound statement. This concept is clarified with the help of the code snippet listed in Program 3-5 and its corresponding flow chart.



Prog 3-5.c	Flow chart depicting the flow of control in program	Output window
1 //if statement 2 #include<stdio.h> 3 main() 4 { 5 int a=5, b=10; 6 if(a>10 && a>b) 7 printf("a is greater than 10"); 8 printf("a is greater than b"); 9 }		a is greater than b **Reasons:** • Only one statement can be associated with if header • Irrespective of the indentation made in the program, printf statement in line 8 is not associated with the if header and is not dependent upon the result of evaluation of if controlling expression • Statement in line 8 is statement next to if statement and will always be executed irrespective of the outcome of if controlling expression **What to do?** • Club statements in lines 7 and 8 into one compound statement as shown in Program 3-6

Program 3-5 | A program to illustrate the execution of if statement

Prog 3-6.c	Flow chart depicting the flow of control in program	Output window

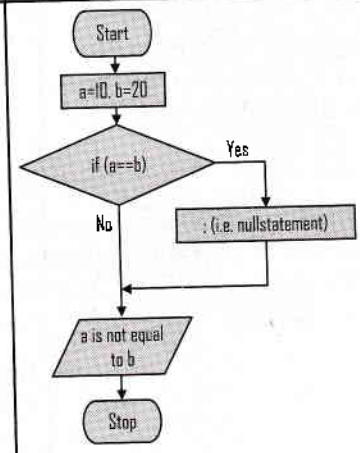
1 //if statement
2 #include<stdio.h>
3 main()
4 {
5 int a=5, b=10;
6 if(a>10 && a>b)
7 {
8 printf("a is greater than 10");
9 printf("a is greater than b");
10 }
11 }
		No output
		Reasons:
		• Lines 7–10 constitute a compound statement
		• The execution of both the statements in lines 8 and 9 is dependent upon the result of evaluation of if controlling expression
		• Since the if controlling expression evaluates to false, statements in lines 8 and 9 do not get executed

Program 3-6 | A program to illustrate the execution of if statement

5. No semicolon should be placed at the end of the if header. However, if a semicolon is placed at the end of the if header, there will be no compilation error (although this may

lead to logical error). This is one of the potential logical errors most amateur programmers do. The logical error due to the semicolon placed at the end of the if header is depicted in the code listed in Program 3-7.

Line	Prog 3-7.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
	//if statement
	#include<stdio.h>
	main()
	{
	int a=10, b=20;
	if(a==b);
	printf("a is not equal to b");
	}
			a is not equal to b
			Expected output:
			No output
			Reason for deviation:
			Presence of semicolon at the end of the if header
			How is the listed code interpreted?
			• It is interpreted as:
			if(a==b)
			;
			printf("a is not equal to b");
			• if body is a null statement
			• printf statement is next to the if statement and its execution does not depend upon the outcome of if controlling expression

Program 3-7 | A program to illustrate the effect of the semicolon placed at the end of the if header

3.3.3.4.1.1.2 if-else Statement

Most of the problems require one set of actions to be performed if a particular condition is true, and another set of actions to be performed if the condition is false. To implement such a decision, C language provides an **if-else statement**. The general form of the if-else statement is:

```
if(expression)          //←if-else header
statement1              //←if body
else                    //←else clause
statement2              //←else body
```

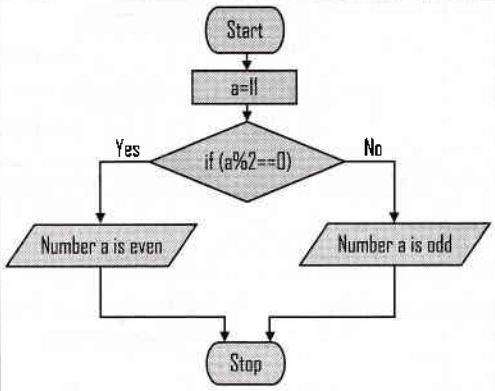
The important points about an if-else statement are as follows:

1. An if-else statement consists of an if-else header, if body, else clause and else body.
2. An if-else header consists of an if clause followed by an **if-else controlling expression** enclosed within parentheses.
3. An if-else statement is executed as follows:
   a. The if-else controlling expression is evaluated.
   b. If the if-else controlling expression evaluates to true, the statement constituting the if body is executed and the else body is skipped.
   c. If the if-else controlling expression evaluates to false, the if body is skipped and the else body is executed.

d. After the execution of the if body or the else body, the execution continues from the statement following the if-else statement.

The code snippet in Program 3-8 illustrates the use of the if-else statement.

Line	Prog 3-8.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
10
11
	//if-else statement
	//Find whether no. is even or odd
	#include<stdio.h>
	main()
	{
	int a=11;
	if(a%2==0)
	printf("Number a is even");
	else
	printf("Number a is odd");
	}
			Number a is odd
			Remarks:
			• The if-else controlling expression a%2==0 evaluates to false
			• The if body is skipped and the else body gets executed

Program 3-8 | A program to illustrate the use of the if-else statement

c. The syntax of if-else statement permits only a single statement to be associated with if clause and else clause. However, this single statement can be a compound statement constituting a number of simple statements. Consider the piece of code in Program 3-9.

Line	Prog 3-9.c	Output window

1
2
3
4
5
6
7
8
9
10
11
	//if-else statement
	#include<stdio.h>
	main()
	{
	int a=11;
	if(a>10)
	printf("The value of a is %d",a);
	printf("Value a is greater than 10");
	else
	printf("Value a is less than 10");
	}
		Compilation error "Misplaced else in function main"
		Reasons:
		• Only a single statement can be associated with if clause and else clause
		• The mentioned code is interpreted as:
		if(a>10) //←**If statement**
		printf("The value of a is %d",a);
		printf("Value a is greater than 10"); //←**statement next to if statement**
		else //←**else clause without any matching if clause**
		printf("Value a is less than 10");
		• else clause cannot exist without a matching if clause
		What to do?
		• Club statements in lines 7 and 8 into one compound statement and re-execute the code

Program 3-9 | A program to illustrate the use of the if-else statement

d. Care must be taken that no semicolon is placed at the end of the if-else header or after the else clause.

3.3.3.4.1.1.3 Nested if Statement

If the body of the if statement is another if statement or contains another if statement (as shown below), then we say that if's are nested and the statement is known as a **nested if statement**. The general form of a nested if statement is:

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

(a)
Body of an **if** statement is another **if** statement

or

```
if(expression)          //←nested if statement
{       statement
        ----------
        if statement
        ----------
        statement
}
```

(b)
Body of an **if** statement contains another **if** statement

This nesting can be done up to any level as shown below:

```
if(expression1)
        if(expression2)
                if(expression-n)
                        statement
```

The above structure seems to form a ladder and is known as the **if ladder**.

The number of levels up to which nesting can be done depends upon the translation limits of the compiler. The translation limits constrain the implementation of language translators and libraries.

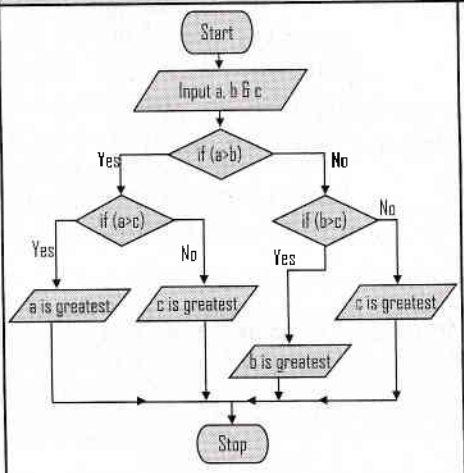
Forward Reference: Translation limits mentioned in ANSI specifications (Appendix C).

3.3.3.4.1.1.4 Nested if-else Statement

In a **nested if-else statement**, the if body or else body of an if-else statement is, or contains, another if statement or if-else statement. Program 3-10 illustrates the use of a nested if-else statement to find the greatest of three numbers.

The careless use of a nested if-else statement introduces a source of potential ambiguity referred to as the **dangling else ambiguity**. When **a statement** contains more number of if clauses than else clauses, then there exists a potential ambiguity regarding with which if clause does the else clause properly matches up. This ambiguity is known as **dangling else problem**. The code listed in the column 1 of Table 3.1 suffers from a dangling else problem. The other columns in the table show the two possible interpretations of the code listed in column 1.

Prog 3-10.c	Flow chart depicting the flow of control in program	Output window

//Nested if-else statement
#include<stdio.h>
main()
{
int a, b, c;
printf("Enter three numbers\t");
scanf("%d %d %d",&a,&b,&c);
if(a>b)
{//←if body starts
if(a>c)
printf("%d is greatest", a);
else
printf("%d is greatest",c);
}//←if body ends
else
{//←else body starts
if(b>c)
printf("%d is greatest",b);
else
printf("%d is greatest",c);
}//←else body ends
}
		Enter three numbers 1 4 2
		4 is greatest
		Remarks:
		• The program illustrates the use of nested if-else statement
		• Both if body and else body of if-else statement consists of if-else statement

Program 3-10 | A program that uses a nested **if-else** statement to find the greatest of three numbers

Table 3.1 | The code in column 1 suffers from dangling else ambiguity. Columns 2 and 3 depict the two possible interpretations of the code listed in column 1

Code suffering from dangling else problem (Column 1)	Interpretation-I (Column 2)	Interpretation-II (Column 3)

//Dangling else problem
#include<stdio.h>
main()
{
int a=10, b=20;
if(a==100)
if(b==20)
printf("Match-I");
else
printf("Match-II");
}
	//Interpretation-I
	#include<stdio.h>
	main()
	{
	int a=10, b=20;
	if(a==100)
	if(b==20)
	printf("Match-I");
	else
	printf("Match-II");
	}
		//Interpretation-II
		#include<stdio.h>
		main()
		{
		int a=10, b=20;
		if(a==100)
		if(b==20)
		printf("Match-I");
		else
		printf("Match-II");
		}

Output	**If interpreted in this way, the output would be:**	**If interpreted in this way, the output would be:**
No output	Match-II	No output

The dangling else problem is solved in two ways:

1. **Implicitly by compiler:** The dangling else ambiguity is implicitly resolved by the compiler by matching the else clause with the last occurring unmatched if, i.e. interpreted in a way as shown in column 3 of Table 3.1. The outputs in columns 1 and 3 are the same. This indicates that the code in column 1 is interpreted in the same way as shown in column 3 of Table 3.1.
2. **Explicitly by user:** The dangling else ambiguity can be explicitly removed by the user by using braces. This is shown in Table 3.2.

Table 3.2 | Dangling else ambiguity removed explicitly by the user

Line	Code suffering from dangling else problem (Column 1)	Dangling else ambiguity removed from the code listed in column 1 by using braces (Column 2)	Dangling else ambiguity removed from the code listed in column 1 by using braces (Column 3)
1	//Dangling else problem	//Dangling else problem	//Dangling else problem
2	#include<stdio.h>	#include<stdio.h>	#include<stdio.h>
3	main()	main()	main()
4	{	{	{
5	int a=10, b=20;	int a=10, b=20;	int a=10, b=20;
6	if(a==100)	if(a==100)	if(a==100)
7	if(b==20)	{	{
8	printf("Match-I");	if(b==20)	if(b==20)
9	else	printf("Match-I");	printf("Match-I");
10	printf("Match-II");	}	else
11	}	else	printf("Match-II");
		printf("Match-II");	}
		}	}
	Output	**Output**	**Output**
	No output	Match-II	No output

3.3.3.4.1.1.5 switch Statement

A **switch statement** is used to control complex branching operations. When there are many conditions, it becomes too difficult and complicated to use if and if-else constructs. In such cases, the switch statement provides an easy and organized way to select among multiple options, depending upon the outcome of a particular condition. The general form of a switch statement is:

```
switch(expression)      //←switch header
statement               //←switch body
```

The important points about a switch statement are as follows:

1. A switch statement consists of a switch header and a switch body.
2. A switch header consists of the keyword switch followed by a switch selection expression enclosed within parentheses.

3. The switch selection expression must be of integral type (i.e. integer type or character type).
4. The switch body consists of a statement. The statement constituting a switch body can be a null statement, an expression statement, a labeled statement, a flow control statement, a compound statement, etc.
5. Generally, a switch body consists of a compound statement, whose constituent statements are case-labeled statements, expression statements, flow control statements and an optional default-labeled statement.
6. Case labels of case-labeled statements constituting the body of a switch statement should be unique, i.e. no two case labels should have or evaluate to the same value.
7. There can be at most one default labeled statement within the switch body.
8. A switch statement is executed as follows:
   a. The switch selection expression is evaluated.
   b. The result of evaluation of switch selection expression is compared with the case labels of the case-labeled statements until there is a match or until all the case-labeled statements have been examined.
      i. If the result of evaluation of switch selection expression is matched with the case label of a case-labeled statement, the execution starts from the matched case-labeled statement, and all the statements after the matched case-labeled statement within the switch body gets executed.
      ii. If no case label of case-labeled statements within the switch body matches the result of evaluation of switch selection expression, the execution starts with the default-labeled statement, if it is present, and all the statements after the default-labeled statement within the switch body gets executed.
      iii. If none of the case labels match the result of evaluation of switch selection expression and there is no default-labeled statement present within the switch body, no statement within the switch body will be executed and the execution continues from the statement following the switch statement.

It is a common misunderstanding that only the matched case-labeled statement or the default-labeled statement (if none of the case labels match) gets executed. In fact, the execution begins with the matched case labeled statement or the default labeled statement, and all the statements after the matched case labeled statement or the default labeled statement within the switch body get executed.

The code snippets in Programs 3-11 to 3-13 clarify the points discussed above.

	Prog 3-11.c	Output window
1	//switch statement	This is case option 1
2	#include<stdio.h>	Value of a is 1
3	main()	This is case option 2
4	{	This is default option

(Contd...)

is implicitly resolved by the
clause with the last occurring
n a way as shown in column
olumns 1 and 3 are the same.
olumn 1 is interpreted in the
3 of Table 3.1.
be explicitly removed by the
own in Table 3.2.

Dangling else ambiguity removed from the code listed in column 1 by using braces (Column 3)

```
//Dangling else problem
#include<stdio.h>
main()
{
int a=10, b=20;
if(a==100)
{
   if(b==20)
      printf("Match-I");
   else
      printf("Match-II");
}
}
```

Output

No output

tions. When there are many
and **if-else** constructs. In such
to select among multiple op-
The general form of a switch

dy.
y a switch selection expression

Line	Prog 3-11.c
5	int a=1;
6	switch(a)
7	{
8	case 1:
9	printf("This is case option 1\n");
10	printf("Value of a is %d\n",a);
11	case 2:
12	printf("This is case option 2\n");
13	default:
14	printf("This is default option\n");
15	}
16	}

Output window

Remarks:

- A switch body consists of four statements and is interpreted as:

```
{
case 1:                    //←Statement 1: case-labeled statement
   printf("This is case option 1\n");
printf("Value of a is %d\n",a);  //←Statement 2: function call statement
case 2:                    //←Statement 3: case-labeled statement
   printf("This is case option 2\n");
default:                   //←Statement 4: default-labeled statement
   printf("This is default option\n");
}
```

- Since case label 1 matches the result of evaluation of switch selection expression, the execution starts from the statement with the case label 1, and all the statements after it within the switch body gets executed

Program 3-11 | A program to illustrate the working of a switch statement

Line	Prog 3-12.c
1	//switch statement
2	#include<stdio.h>
3	main()
4	{
5	int a=3;
6	switch(a)
7	{
8	case 1:
9	printf("This is case option 1\n");
10	printf("Value of a is %d\n",a);
11	default:
12	printf("This is default option\n");
13	case 2:
14	printf("This is case option 2\n");
15	}
16	}

Output window

```
This is default option
This is case option 2
```

Remarks:

- There is no constraint about the position of a default-labeled statement within the switch body
- Since none of the case labels match the result of evaluation of a switch selection expression, the execution begins with the default-labeled statement
- All the statements after the default-labeled statement within the switch body gets executed

Program 3-12 | A program to illustrate that there is no constraint about the position of a default-labeled statement within the switch body

Line	Prog 3-13.c
1	// switch statement & ranges
2	#include<stdio.h>
3	main()
4	{
5	char exp='E';
6	switch(exp)

Output window

```
Upper case vowel
```

Remarks:

- A switch body consists of two **case-labeled statements**. The code is interpreted as:

```
case 'a':     //←Statement 1: case-labeled statement
   case 'e':  //←Constituent statement of case-labeled statement is
```

(Contd...)

7	{
8	case 'a':
9	case 'e':
10	case 'i':
11	case 'o':
12	case 'u':
13	printf("Lower cas
14	case 'A':
15	case 'E':
16	case 'I':
17	case 'O':
18	case 'U':
19	printf("Upper cas
20	}
21	}

Program 3-13 | A prog

3.3.3.4.1.2 Jump S

A jump **statement** tra
tion, i.e. unconditiona

1. goto statement
2. break statement
3. continue statemen
4. return statement

3.3.3.4.1.2.1 goto S

The **goto statement** is u
tion. It provides a high
to another within a fu
modify. Thus, the use
languages like C. The

The important poin

1. The goto stateme
   Within the body
   statement with a
   be present.
2. The goto statemen
   statement having

```
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    printf("Lower case vowel\n");
case 'A':
case 'E':
case 'I':
case 'O':
case 'U':
    printf("Upper case vowel\n");
}
}
```

```
case 'i':   //←another case-labeled statement
    case 'o':
        case 'u':
            printf("Lower case vowel\n");
case 'A':   //←Statement 2: case-labeled statement
    case 'E':
        case 'I':
            case 'O':
                case 'U':
                    printf("Upper case vowel\n");
```

- In this way, the switch statement can be used to switch on ranges
- This is only beneficial when the ranges are small
- C language does not support the following ways for switching on ranges:
  - case 'A'-'Z' //←if used, it will be interpreted as case -25 (i.e. ASCII code of 'A'- ASCII code of 'Z')
  - case 'A' to 'Z' //←allowed in Visual Basic but not in C language

Program 3-13 | A program to illustrate the use of a switch statement to switch on ranges

2.3.3.4.1.2 Jump Statements

A **jump statement** transfers the control from one point to another without checking any condition, i.e. unconditionally. The following jump statements are present in C language:

1. goto statement
2. break statement
3. continue statement
4. return statement

2.3.3.4.1.2.1 goto Statement

The **goto statement** is used to branch unconditionally from one point to another within a function. It provides a highly unstructured way of transferring the program control from one point to another within a function. It often makes the program control difficult to understand and modify. Thus, the use of a goto statement is discouraged in powerful structured programming languages like C. The syntactic form of a goto statement is:

goto label;

The important points about a goto statement are as follows:

1. The goto statement is always used in conjunction with an identifier-labeled statement. Within the body of a function in which the goto statement is present, an identifier-labeled statement with a label name, same as the label name used in the goto statement should be present.
2. The goto statement on execution transfers the program control to an identifier-labeled statement having a label name same as the label name used in the goto statement.

3. The goto statement can be used to make a forward jump as well as a backward jump. If the goto statement is present before its corresponding identifier-labeled statement, the jump made will be known as a **forward jump.** If the goto statement is present after its corresponding identifier-labeled statement, the jump made will be known as a **backward jump.** The forward and backward jumps are shown in Figure 3.3.

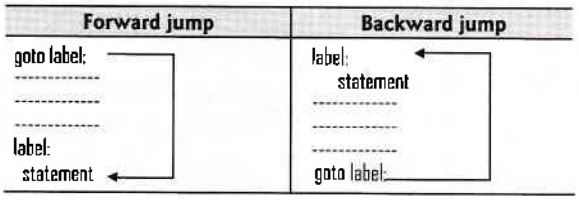


Figure 3.3 | Forward and backward jump

4. There can be two or more goto statements corresponding to an identifier-labeled statement but there cannot be two or more identifier-labeled statements corresponding to a goto statement. The interpretation of this rule is illustrated in Figure 3.4.

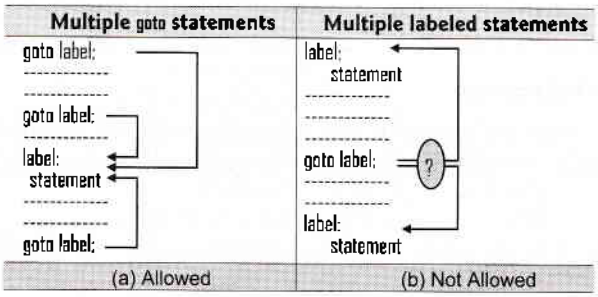


Figure 3.4 | (a) Multiple goto statements corresponding to one identifier-labeled statement are allowed; (b) multiple identifier-labeled statements corresponding to one goto statement are not allowed

5. The goto statement can transfer control anywhere within a function, i.e. it can take control in or out of a nested if statement, nested if-else statement or nested loops. However, a goto statement in no way can take control out of the function in which it is used.

3.3.3.4.1.2.2 break Statement

The syntactic form of a **break statement** is:

```
break;
```

The important points about a break statement are as follows:

1. A break statement can appear only inside, or as a body of, a switch statement or a loop.[§§§] The code snippet listed in Program 3-14 verifies this fact.

[§§§] Refer Section 3.3.3.4.2 for a description on loops.

Prog 3-14.c	Output window

//break statement
#include<stdio.h>
main()
{
int a=10;
if(a==10)
{
printf("if controlling expression evaluates to true");
break;
printf("The value of a is %d",a);
}
}
	Compilation error "Misplaced break in function main"
	Reasons:
	• The break statement can appear only inside or as a switch/loop body
	• The break statement present in line 9 is neither a part of a switch body nor a loop body
	What to do?
	• Either remove the break statement from if body or place the if statement inside a loop body or a switch body

Program 3-14 | A program to illustrate that the break statement cannot appear outside the switch body or a loop

2. A break statement terminates the execution of the nearest enclosing switch or the nearest enclosing loop. The execution resumes with the statement present next to the terminated switch statement or the terminated loop. The interpretation of this point is illustrated in the code snippet listed in Program 3-15.

Prog 3-15.c	Output window

//break statement
#include<stdio.h>
main()
{
int a=1;
switch(a)
{
case 1:
printf("One");
break;
case 2:
printf("Two");
break;
default:
printf("Default");
}
printf("\nThis statement is next to switch");
}
	One
	This statement is next to switch
	Remarks:
	• The switch body consists of five statements and is interpreted as:
	{
	case 1: //←**Statement 1: case-labeled statement**
	printf("One");
	break; //←**Statement 2: break statement**
	case 2: //←**Statement 3: case-labeled statement**
	printf("Two");
	break; //←**Statement 4: break statement**
	default: //←**Statement 5: default-labeled statement**
	printf("Default");
	}
	• Execution starts from the statement with the case label 1
	• Execution of break statement terminates the switch statement and the control immediately transfers to the statement present next to the switch statement, i.e. printf("\nThis statement is next to switch");

Program 3-15 | A program to illustrate the execution of a switch statement

3.4.1.2.3 continue Statement

The syntactic form of a **continue statement** is:

```
continue;
```

The important points about a continue statement are as follows:

1. A continue statement can appear only inside, or as the body of, a loop.
2. A continue[111] statement terminates the current iteration of the nearest enclosing loop. The semantics of the continue statement will be discussed after iteration statements.

3.3.3.4.1.2.4 return Statement

The general forms of a **return statement** are:

```
return;   or          //←Form 1
return expression;    //←Form 2
```

The important points about a return statement are as follows:

1. A return statement without an expression (i.e. Form 1) can appear only in a function whose return type➲ is void.
2. A return statement with an expression (i.e. Form 2) should not appear in a function➲ whose return type is void.
3. A return statement terminates the execution of a function and returns the control to the calling➲ function.

The syntax and semantics of a return statement will be discussed in Chapter 5.

Forward Reference: Functions and their return types, calling function and called function (Chapter 5).

3.3.3.4.2 Iteration Statements

Iteration is a process of repeating the same set of statements again and again until the specified condition holds true. Humans find iterative tasks boring but computers are very good at performing iterative tasks. Computers execute the same set of statements again and again by putting them in a loop. The C language provides the following three **iteration statements**:

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

In general, loops are classified as:

1. Counter-controlled loops
2. Sentinel-controlled loops

3.3.3.4.2.1 Counter-Controlled Loops

Counter-controlled looping is a form of looping in which the number of iterations to be performed is known in advance. Counter-controlled loops are so named because they use a control variable, known as the **loop counter**, to keep a track of loop iterations. The counter-controlled loop starts with the initial value of the loop counter and terminates when the final value of the loop counter is reached. Since the counter-controlled loops iterate a fixed number of times, which is known in advance, they are also known as **definite repetition loops**. There are three main ingredients of counter-controlled looping:

[111] Refer Section 3.3.3.4.2.4.2 for a description on the semantics of a continue statement.

1. Initialization of the loop counter.
2. An expression (specifically a condition) determining whether the loop body should be executed or not.
3. An expression that manipulates the value of the loop counter so that the condition in step 2 eventually becomes false and the loop terminates.

Firstly, I will describe the syntax of looping statements available in C language and how they can be used for counter-controlled looping. In Section 3.3.3.4.2.2, I will describe the use of available iteration statements for sentinel-controlled looping.

3.3.3.4.2.1.1 for Statement

Out of all the looping constructs available in C, **for statement** is the most popular one. The general form of a for statement is:

```
for(expression1; expression2; expression3)      //←for header
statement                                       //←for body
```

The important points about a for statement are as follows:

1. The for statement consists of for header and for body.

Points about for header:

2. The for header consists of the keyword for followed by three expressions separated by semicolons and enclosed within parentheses.
3. All the expressions in the for header are optional and can be skipped. Even if all the expressions are missing, it is mandatory to create three sections by placing two semicolons.
4. Three sections are named as: initialization section, condition section and manipulation section.
   a. **Initialization section:** $expression_1$ constitutes the initialization section. It is used to initialize (i.e. assign a starting value to) the loop counter. If the loop counter has already been initialized, the initialization expression, i.e. $expression_1$ can be skipped. However, a semicolon is necessary and must be placed.
   b. **Condition section:** $expression_2$ forms the condition section. $expression_2$ tests the value of the loop counter. This section determines whether the body of the loop is to be executed or not. In case of infinite loops, the condition section can be skipped.
   c. **Manipulation section:** $expression_3$ is part of the manipulation section. The manipulation expression manipulates the value of the loop counter so that the $expression_2$ present in the condition section eventually evaluates to false and the loop terminates.
5. Care must be taken that the for header is not terminated with a semicolon. If it is terminated with a semicolon, the semicolon is interpreted as a null statement following the for header (i.e. it is treated as for body).

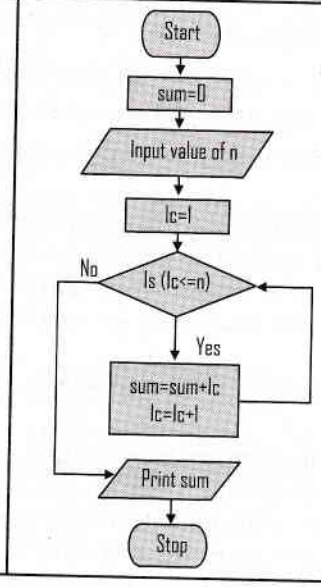
A point about for body:

6. The syntax of for statement permits only a single statement to be associated with for header. If a number of statements are to be executed repeatedly, the statements should be clubbed together to form a compound statement.

Execution of for statement:

7. The for statement is executed as follows:
   a. Initialization section is executed only once at the start of the loop.
   b. The expression present in the condition section is evaluated.
      i. If it evaluates to true, the body of the loop is executed.
      ii. If it evaluates to false, the loop terminates and the program control is transferred to the statement present next to the for statement.
   c. After the execution of the body of the loop, the manipulation expression is evaluated.
   d. These three steps represent the first iteration of the for loop. For the next iterations, Steps b and c are repeated until the expression in Step b evaluates to false.

The facts mentioned above are illustrated in Program 3-16.

Line	Prog 3-16.c	Flow chart depicting the flow of control in program	Output window
1	//Use of for statement to find the		Enter the value of n 10
2	//sum of first n natural numbers		Sum is 55
3	#include<stdio.h>		
4	main()		
5	{		
6	int n, lc, sum=0;		
7	printf("Enter the value of n\t");		
8	scanf("%d",&n);		
9	for(lc=1;lc<=n;lc++)		
10	sum=sum+lc;		
11	printf("Sum is %d",sum);		
12	}		

Program 3-16 | A program to illustrate the use of for statement for finding the sum of first n natural numbers

The codes in Table 3.3 are equivalent to the code specified in Program 3-16.

Table 3.3 | Codes equivalent to the code listed in Program 3-16

Line	Equivalent Code-I	Equivalent Code-II	Equivalent Code-III
1	//Use of for statement to find the sum	//Use of for statement to find the sum	//Use of for statement to find the
2	// of first n natural numbers	//of first n natural numbers	//sum of first n natural numbers
3	#include<stdio.h>	#include<stdio.h>	#include<stdio.h>
4	main()	main()	main()
5	{	{	{
6	int n, lc=1, sum=0; **//Initialization of lc**	int n, lc, sum=0;	int n, lc=1, sum=0;
7	printf("Enter the value of n\t");	printf("Enter the value of n\t");	printf("Enter the value of n\t");

(Contd...)

```
scanf("%d",&n);
for( ;lc<=n;lc++)//Initialization missing
    sum=sum+lc;
printf("Sum is %d",sum);
}
```

```
scanf("%d",&n);
for(lc=1;lc<=n; )//Manipulation missing
    sum=sum+lc++;//Manipulation of lc
printf("Sum is %d",sum);
}
```

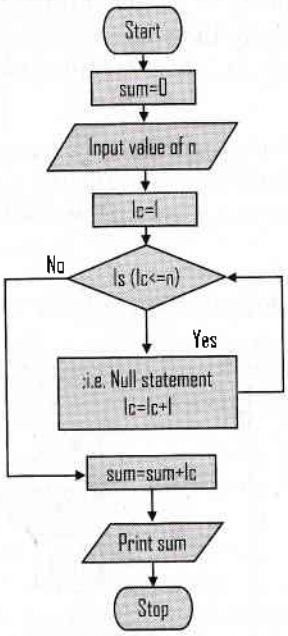
```
scanf("%d",&n);
for( ;lc<=n; )//Both missing
    sum=sum+lc++;
printf("Sum is %d",sum);
}
```

The code snippet in Program 3-17 illustrates the effect of presence of a semicolon at the end of for header.

Prog 3-17.c

```
//Effect of ; at end of for header
#include<stdio.h>
main()
{
int n, lc, sum=0;
printf("Enter the value of n\t");
scanf("%d",&n);
for(lc=1;lc<=n;lc++);
    sum=sum+lc;
printf("Sum is %d",sum);
}
```

Flow chart depicting the flow of control in program



Output window

```
Enter the value of n    10
Sum is 11
```

Remarks:

- for header is terminated with a semicolon
- Semicolon is interpreted as null statement and forms the for body
- The statement sum=sum+lc; is a statement present next to the for statement and thus gets executed only once
- The value of lc on the termination of loop will be 11
- This value of lc is added to sum to produce the mentioned output

Program 3-17 | A program to illustrate the effect of a semicolon at the end of a for header

2.2.3.4.2.1.2 while Statement

The general form of a **while statement** is:

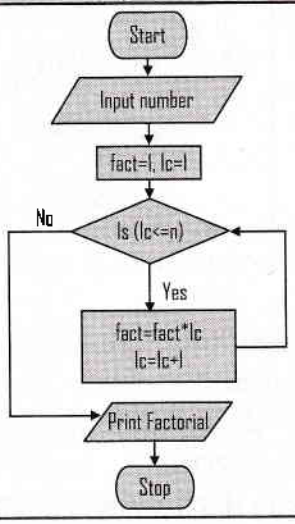
```
while(expression)        //←while header
statement                //←while body
```

The important points about a while statement are as follows:

1. The while statement consists of while header and while body.
2. The while header consists of keyword while followed by while controlling expression enclosed within the parentheses.
3. The controlling expression in while header is mandatory and cannot be skipped.

4. The while header should not be terminated with a semicolon. If it is terminated with a semicolon, the semicolon is interpreted as a null statement following the while header (i.e. it is treated as a while body).
5. The syntax of a while statement permits only a single statement to be associated with while header. If a number of statements are to be executed repeatedly, the statements should be clubbed together to form a compound statement.
6. The while statement is executed as follows:
   a. The while controlling expression is evaluated.
      i. If it evaluates to true, the body of the loop is executed.
      ii. If it evaluates to false, the program control is transferred to the statement present next to the while statement.
   b. After executing the while body, the program control returns back to the while header.
   c. Steps a and b are repeated until the while controlling expression in Step a evaluates to false.
7. While making the use of while statement, always remember to initialize the loop counter before the while controlling expression is evaluated and to manipulate the loop counter inside the body of while statement, i.e. before the while controlling expression is evaluated again.

The facts mentioned above are illustrated in Program 3-18.

Line	Prog 3-18.c	Flow chart depicting the flow of control in program	Output window
1	//Use of while statement to find		Enter number 5 Factorial is 120 **Remarks:** • Line 9 initializes the value of fact to 1. It is important to initialize fact to 1 else garbage will be the result • Line 10 provides the initialization of loop counter • Line 14 manipulates the loop counter
2	//the factorial of a number		
3	#include<stdio.h>		
4	main()		
5	{		
6	int num, lc, fact;		
7	printf("Enter number\t");		
8	scanf("%d",&num);		
9	fact=1;		
10	lc=1; **//Initialization of loop counter**		
11	while(lc<=num)		
12	{		
13	fact=fact*lc;		
14	lc=lc+1; **//Manipulation of loop counter**		
15	}		
16	printf("Factorial is %d",fact);		
17	}		

Program 3-18 | A program to find the factorial of a number using while loop

The codes in Table 3.4 are equivalent to the code specified in Program 3-18.

Table 3.4 | Codes equivalent to the code listed in Program 3-18

Line	Equivalent Code-I	Equivalent Code-II	Equivalent Code-III
1	//Use of while statement to find	//Use of while statemnt to find	//Use of while statement to find
2	//the factorial of a number	//the factorial of a number	//the factorial of a number
3	#include<stdio.h>	#include<stdio.h>	#include<stdio.h>
4	main()	main()	main()
5	{	{	{
6	int num, lc=1, fact=1;	int num, lc=1, fact=1;	int num, lc=0, fact=1;
7	printf("Enter number\t");	printf("Enter number\t");	printf("Enter number\t");
8	scanf("%d",&num);	scanf("%d",&num);	scanf("%d",&num);
9	while(lc<=num)	while(lc<=num)	while(lc++<num)
10	{	fact=fact*lc++;	fact=fact*lc;
11	fact=fact*lc;	printf("Factorial is %d",fact);	printf("Factorial is %d",fact);
12	lc=lc+1;	}	}
13	}		
14	printf("Factorial is %d",fact);		
15	}		

3.3.3.4.2.1.3 do-while Statement

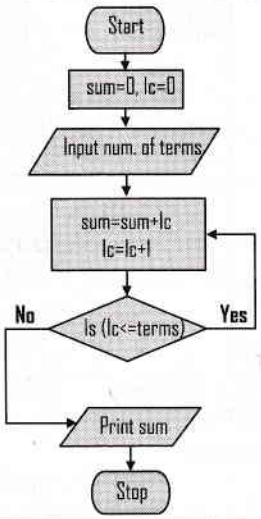
The general form of **do-while statement** is:

```
do                          //←do-while header
statement                   //←do-while body
while(expression);          //←while clause
```

The important points about a do-while statement are as follows:

1. The do-while statement consists of a do clause, followed by a statement that constitutes do-while body, followed by the while clause consisting of while keyword followed by do-while controlling expression enclosed within parentheses. The while clause is terminated with a semicolon.
2. The controlling expression in a do-while statement is mandatory and cannot be skipped.
3. The syntax of a do-while statement permits only a single statement to be present. If a number of statements are to be executed repeatedly, the statements should be clubbed together to form a compound statement.
4. The do-while statement is executed as follows:
   a. The statement, i.e. body of do-while statement, is executed.
   b. After the execution of a do-while body, the do-while controlling expression is evaluated.
      i. If it evaluates to true, the statement, i.e. do-while body is executed again and Step b is repeated.
      ii. If it evaluates to false, the program control is transferred to the statement present next to the do-while statement.
5. While making the use of a do-while statement, always remember to initialize the loop counter before the do-while statement and to manipulate it inside the body of the do-while statement so that the do-while controlling expression eventually becomes false.
6. The statement, i.e. body of the do-while loop is executed once, even when the do-while controlling expression is initially false.

The code snippet in Program 3-19 illustrates the use of a do-while statement to find the sum of the series 1 + 2 + 3... *n* terms.

Line	Prog 3-19.c	Flow chart depicting the flow of control in program	Output window
1	//Use of do-while statement to find		Enter number of terms 5 Sum of series is 15
2	//the sum of the series 1+2+3.....n terms		**Remarks:**
3	#include<stdio.h>		• Initialize the value of variable sum to 0 else the result will be garbage
4	main()		• Look at the position of controlling expression
5	{		• It is present at the end (i.e. exit point) of the loop
6	int terms, sum=0, lc=0;		• That is why, do-while is known as **exit-controlled loop**
7	printf("Enter number of terms\t");		• for and while are known as **entry-controlled loops**
8	scanf("%d",&terms);		
9	do		
10	{		
11	sum=sum+lc;		
12	lc=lc+1;		
13	}		
14	while(lc<=terms);		
15	printf("Sum of series is %d",sum);		
16	}		

Program 3-19 | A program to illustrate the use of a do-while statement

3.3.3.4.2.2 Sentinel-Controlled Loops

In **sentinel-controlled looping**, the number of times the iteration is to be performed is not known beforehand. The execution or termination of the loop depends upon a special value called the **sentinel value**. If the sentinel value is true, the loop body will be executed, otherwise it will not. Since the number of times a loop will iterate is not known in advance, this type of loop is also known as **indefinite repetition loop**.

Consider the problem of finding the maximum and the minimum from a set of numbers. However, the set (i.e. numbers) and the cardinality of set (i.e. how many numbers are there in the set) are not known beforehand; therefore, the user will enter them at the run time. The mentioned problem can be solved by using sentinel-controlled looping as given in Programs 3-20 and 3-21.

Line	Prog 3-20a.c	Prog 3-20b.c	Output window
1	//while statement for sentinel control	//for statement for sentinel control	Enter number 5
2	#include<stdio.h>	#include<stdio.h>	Want to enter more y
3	#include<conio.h>	#include<conio.h>	Enter number 3
4	main()	main()	Want to enter more y
5	{	{	Enter number 8
6	char choice;	char choice;	Want to enter more y
7	int num, max, min;	int num, max, min;	Enter number -2
8	printf("Enter number\t");	printf("Enter number\t");	Want to enter more n
9	scanf("%d",&num);	scanf("%d",&num);	Maximum is 8

(Contd...)

```
max=min=num;
printf("Want to enter more\t");
choice=getche();
while(choice=='y'||choice=='Y')
{
printf("\nEnter number\t");
scanf("%d",&num);
if(num>max)
    max=num;
else
    if(num<min)
        min=num;
printf("Want to enter more\t");
choice=getche();
}
printf("\nMaximum is %d",max);
printf("\nMinimum is %d",min);
}
```

```
max=min=num;
printf("Want to enter more\t");
choice=getche();
for( ;choice=='y'||choice=='Y'; )
{
printf("\nEnter number\t");
scanf("%d",&num);
if(num>max)
    max=num;
else
    if(num<min)
        min=num;
printf("Want to enter more\t");
choice=getche();
}
printf("\nMaximum is %d",max);
printf("\nMinimum is %d",min);
}
```

```
Minimum is -2
```

Remarks:

- The loop terminates when the user does not enter the choice 'Y' or 'y'
- choice is the **sentinel value**
- The number of iterations after which the user will say 'no' is not known in advance
- The header file conio.h is to be included for using the function getche
- The function getche is used to get a character from the user. It also outputs, i.e. echoes the entered character onto the screen. The character e in getche stands for echo
- The variant of getche function that is used to get a character from the user without echoing it on the screen is getch function

Program 3-20 | A program to illustrate the use of while statement and for statement for sentinel-controlled looping

Line	Prog 3-21.c
1	//do-while statement for sentinel controlled looping
2	#include<stdio.h>
3	#include<conio.h>
4	main()
5	{
6	char choice;
7	int num, iteration=1, max, min;
8	do
9	{
10	printf("Enter number\t");
11	scanf("%d",&num);
12	if(iteration++==1)
13	max=min=num;
14	else
15	if(num>max)
16	max=num;
17	else
18	if(num<min)
19	min=num;

Output window

```
Enter number    5
Want to enter more    y
Enter number    3
Want to enter more    y
Enter number    8
Want to enter more    y
Enter number    -2
Want to enter more    n

Maximum is 8
Minimum is -2
```

Remarks:

- The loop terminates when the user does not enter the choice 'Y' or 'y'
- choice is the sentinel value
- The number of iterations after which the user will say 'no' is not known in advance

(Contd...)

(Contd...)

Line	Prog 3-21.c	Output window
20	printf("Want to enter more\t");	
21	choice=getche();	
22	printf("\n");	
23	}	
24	while(choice=='y'||choice=='Y');	
25	printf("\nMaximum is %d",max);	
26	printf("\nMinimum is %d",min);	
27	}	

Program 3-21 | A program to illustrate the use of a do-while statement for sentinel-controlled looping

3.3.3.4.2.3 Nested Loops

If the body of a loop is, or contains another iteration statement, then we say that the **loops are nested**. An example of a nested for loop is given in Program 3-22.

Line	Prog 3-22.c	Output window
1	//Nested for loop	****
2	#include<stdio.h>	****
3	main()	****
4	{	****
5	int olc,ilc;	**Remarks:**
6	for(olc=1;olc<=4;olc++)	• olc is the outer loop counter and ilc is the inner loop counter
7	{	• The inner loop is responsible for printing 4 stars in a row
8	for(ilc=1;ilc<=4;ilc++)	• The outer loop is responsible for printing 4 such rows
9	printf("*");	
10	printf("\n");	
11	}	
12	}	

Program 3-22 | A program to illustrate the use of a nested for loop

3.3.3.4.2.4 Semantics of break and continue Statements

After the discussion of iteration statements, it is time to discuss the use of break and continue statements. The break statement helps in terminating a loop, while the continue statement helps in terminating the current iteration of a loop.

3.3.3.4.2.4.1 Semantics of break Statement

The important points about the usage of a break statement along with loops are as follows:

1. When the break statement present inside a loop is executed, it terminates the loop and the program control is transferred to the statement present next to the loop.
2. When the break statement present inside a nested loop is executed, it only terminates the execution of the nearest enclosing loop. The execution resumes with the statement present next to the terminated loop.
3. There is no constraint about the number of break statements that can be present inside a loop.

The meaning of

Line	Prog 3-23.c
1	//Use of break st
2	#include<stdio.h>
3	main()
4	{
5	int i;
6	for(i=1;i<=10;i++)
7	{
8	if(i==5)
9	break;
10	printf("%d ",i);
11	}
12	if(i<11)
13	printf("\nPrem
14	}

Program 3-23 | A pr

Program 3-24 illu

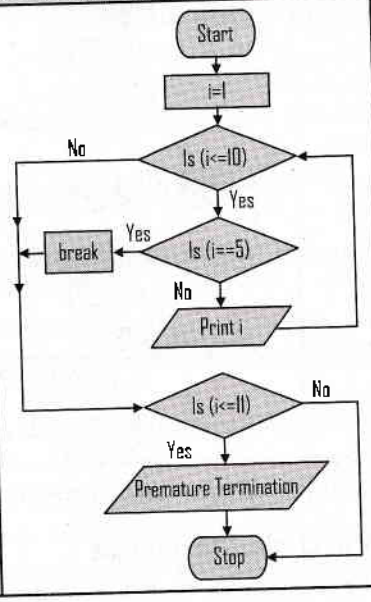
Line	Prog 3-24.c
1	//Use of break stat
2	#include<stdio.h>
3	main()
4	{
5	int olc, ilc;
6	for(olc=1;olc<=3;olc<
7	for(ilc=1;ilc<=4;il
8	{
9	if(ilc==3)
10	break;
11	printf("%d %d\n
12	}
13	}

Program 3-24 | A pro

The use of a break
Program 3-25.

The meaning of the above-mentioned points is illustrated in Program 3-23.

Line	Prog 3-23.c	Flow chart depicting the flow of control in program	Output window

	//Use of break statement
	#include<stdio.h>
	main()
	{
	int i;
	for(i=1;i<=10;i++)
	{
	if(i==5)
	break;
	printf("%d ",i);
	}
	if(i<11)
	printf("\nPremature Termination");
	}
			1 2 3 4
			Premature Termination
			Remark:
			• break statement is used to prematurely terminate the loop

Program 3-23 | A program to illustrate the use of break statement

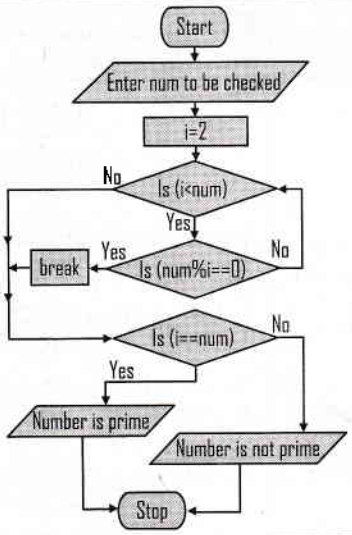
Program 3-24 illustrates a break statement, which terminates the nearest enclosing loop.

Line	Prog 3-24.c	Values of olc and ilc	Output window

	//Use of break statement in nested loops
	#include<stdio.h>
	main()
	{
	int olc, ilc;
	for(olc=1;olc<=3;olc++)
	for(ilc=1;ilc<=4;ilc++)
	{
	if(ilc==3)
	break;
	printf("%d %d\n",olc,ilc);
	}
	}
		olc=1
		ilc=1 //←prints 1 1
		ilc=2 //←prints 1 2
		ilc=3 //←break executes & terminates the inner loop
		olc=2
		ilc=1 //←prints 2 1
		ilc=2 //←prints 2 2
		ilc=3 //←break executes
		olc=3
		ilc=1 //←prints 3 1
		ilc=2 //←prints 3 2
		ilc=3 //←break executes
			1 1
			1 2
			2 1
			2 2
			3 1
			3 2
			Remarks:
			• break statement terminates only the inner loop
			• The control still remains inside the outer loop

Program 3-24 | A program to illustrate that break statement terminates the nearest enclosing loop

The use of a break statement in checking whether a number is prime or not is illustrated in Program 3-25.

Line	Prog 3-25.c	Flow chart depicting the flow of control	Output window
1	//Use of break statement to check		Enter the number 9 Number is not prime **Remarks:** • Whether a number is prime or not can be determined by checking whether the number is divisible by any value from 2 to num-1 • When it is found that the number num is divisible by some value of i, there is no need to check the divisibility of num for rest of the values of i
2	//whether a number is prime or not		
3	#include<stdio.h>		
4	main()		
5	{		
6	int num, i;		
7	printf("Enter the number\t");		
8	scanf("%d",&num);		
9	for(i=2;i<num;i++)		
10	if(num%i==0)		
11	break;		
12	if(i==num)		
13	printf("Number is prime");		
14	else		
15	printf("Number is not prime");		
16	}		

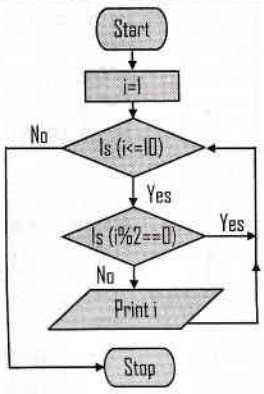
Program 3-25 | A program to check whether a number is prime or not

3.3.3.4.2.4.2 Semantics of continue Statement

The important points about a continue statement are as follows:

1. A continue statement terminates the current iteration of the loop.
2. When a continue statement present inside a nested loop is executed, it only terminates the current iteration of the nearest enclosing loop.
3. On the execution of a continue statement, the program control is immediately transferred to the header of the loop.
4. There is no constraint about the number of continue statements that can be present inside a loop.

The semantics of a continue statement is illustrated in Program 3-26.

Line	Prog 3-26.c	Flow chart depicting the flow of control	Output window
1	//Use of continue statement		1 3 5 7 9 **Remark:** • For even values of i, the printf statement will not be executed as the continue statement transfers the control to the header of the loop
2	#include<stdio.h>		
3	main()		
4	{		
5	int i;		
6	for(i=1;i<=10;i++)		
7	{		
8	if(i%2==0)		
9	continue;		
10	printf("%d ",i);		
11	}		
12	}		

Program 3-26 | A program to illustrate the use of a continue statement

Output window

```
Enter the number  9
Number is not prime
```

Remarks:

- Whether a number is prime or not can be determined by checking whether the number is divisible by any value from 2 to num-1
- When it is found that the number num is divisible by some value of i, there is no need to check the divisibility of num for rest of the values of i

op.
uted, it only terminates the

is immediately transferred

can be present inside a loop.
-26.

Output window

```
13579
```

Remark:

- For even values of i, the printf statement will not be executed as the continue statement transfers the control to the header of the loop

3.4 Summary

1. Statement is the smallest logical entity that can independently exist in a C program.
2. No entity smaller than a statement, i.e. expressions, variables, constants, etc. can exist in a C program unless and until they are converted into statements.
3. A single statement is known as a simple statement.
4. A group of single statements can be clubbed together into one statement by enclosing them within braces. A clubbed statement is known as a block or a compound statement.
5. Non-executable statements are meant for the compiler. No machine code is generated for non-executable statements.
6. Only executable statements play a role during the execution of a program. Only for these statements, the machine code is generated.
7. A null statement performs no operation and consists of just a semicolon.
8. An expression statement performs the computation and is formed by terminating an expression with a semicolon.
9. By default, the flow of program control is sequential and it flows from top to bottom.
10. Flow of control needs to be altered to implement decision making and iteration.
11. To alter the default flow of control, flow control statements are used.
12. To implement decision making, selection statements are used.
13. Selection statements are: if statement, if-else statement, and switch statement.
14. Selection statements can be nested.
15. Careless use of nested if-else statement may lead to dangling else problem.
16. Dangling else problem can be implicitly as well as explicitly solved.
17. A switch statement is a better alternative to a nested if-else statement and is used in the complex decision making.
18. Looping can be performed by using iteration statements.
19. Three iteration statements available in C are: for statement, while statement and do-while statement.
20. A break statement is used to terminate the nearest enclosing loop.
21. A continue statement is used to terminate the current iteration of the nearest enclosing loop.

Exercise Questions

Conceptual Questions and Answers

1. *What is the smallest logical unit that can independently exist in a C program?*

   Statement is the smallest logical unit that has an independent existence in a C program. No entity smaller than a statement (i.e. expressions, variables and constants, etc.) can exist unless and until they are converted into statements. Consider the following program segment:

```
main()
{
    int a=10,b=20,c;
```

```
    c=a+b                          //← Error: Statement missing ; in function main()
    printf("The value of c is %d",c);
}
```

On compilation, the above-mentioned piece of code gives 'Statement missing ; error'. This error is due to the fact that c=a+b is an expression and not a statement. Expressions cannot independently exist in a C program. To make them exist, they must be converted into statements by terminating them with a semicolon. The following is the rectified code:

```
main()
{
    int a=10,b=20,c;
    c=a+b;                         //←Expression terminated with a semicolon forming a statement
    printf("The value of c is %d",c);
}
```

2. *What is meant by a simple statement and a compound statement?*

   A simple statement consists of a single statement. For example, c=a+b; is a simple statement. A compound statement consists of a sequence of simple statements enclosed within braces. The following is an example of a compound statement:

```
{
    c=a+b;
    a*=2;
    b+3;
}
```

3. *What are executable statements and non-executable statements?*

   Executable statements are the statements that call for a processing action by the computer, such as performing arithmetic, reading data, making decision and so on. Non-executable statements are the statements that provide the information about the nature of data (e.g. declaration statement). Non-executable statements can be placed outside the bodies of functions (i.e. in global scope➲), but executable statements can only be placed within the body of some function (i.e. local scope➲).

Forward Reference: Global and local scope (Chapter 7).

4. *Write a simple C statement to accomplish the following tasks:*
   a. *Assign sum of* x *and* y *to* z *and increment the value of* x *by 1 after the calculation.*
   b. *Decrement the variable* x *by 1 then subtract it from the variable* total.

   a. z=x++ + y;
      Note: Writing z=x++ + y is not valid as it is not a statement. It is an expression.
   b. total-=--x;
      or
      total=total- --x;
      Carefully note the position of white-space character. Writing total=total---x; or total=total-- -x is not the same as writing total=total- --x;. The difference between them is shown in the table given below:

Column I	Initial values		After execution of statement in Column I	
	total	x	total	x
total-=--x;	15	5	11	4
total=total- --x;	15	5	11	4
total=total-- -x;	15	5	9	5
total=total---x;	15	5	9	5

5. *What is the difference between initialization and assignment?*

First time assignment at the time of definition is called initialization. Assigning a value to an identifier after initialization will be treated as an assignment. The clear understanding of difference between terms initialization and assignment becomes important when we talk about qualified constants. Consider the following piece of code:

```
main()
{
   const int a=20;   //← Initialization of a qualified constant is valid.
   a=30;             //← Compilation error: Value cannot be assigned to a qualified constant.
}
```

The above-mentioned code highlights the fact that:
'We cannot assign a value to a qualified constant but we can initialize it'.

6. *What is null statement and where is it used?*

Backward Reference: Refer Section 3.3.3.2 for a description on null statement.

A null statement is used when the syntax of a language construct requires a statement to be present but the logic of the program does not require it. Its use is illustrated in the next answer.

7. *How can you print "Hello World" without using a semicolon in a C program?*

The following code segment prints "Hello World" without using a semicolon.

```
main()
{
   if(printf("Hello World"))
   {}     //←Null statement. Syntax of if statement requires a statement to be present but
}         //    the logic of the program does not require it. Hence, null statement is placed.
```

8. *What is dangling else problem? How is it solved by a compiler and how can it be avoided?*

Backward Reference: Refer Section 3.3.3.4.1.1.4 for the answer to this question.

9. *Why does the following piece of code on compilation gives an error?*

```
main()
{
int a=1;
if(a==1)
   printf("This is if body\n");
```

```
    printf("This statement does not belong to if body");
else
    printf("This is else body");
}
```

The given piece of code on compilation gives 'Misplaced else error.' The source of error can be found by looking at the syntax of an if-else statement. The general form of an if-else statement is:

```
if(expression)          //← if header
statement1              //← if body
else                    //← else clause
statement2              //← else body
```

It should be noted that only one statement can be associated with if clause and else clause. If more than one statement needs to be associated with if clause or else clause, then a block comprising those simple statements must be created. This block of statements, although comprising more than one simple statement, will be treated as a unit, as one statement and can be associated with if clause or else clause. The given piece of code is interpreted as:

```
main()
{
    int a=1;
    if(a==1)
        printf("This is if body\n");                    //←Only this statement is associated with if clause
    printf("This statement does not belong to if body");          //←This statement is not in if body
    else            //←else clause is left without any matching if clause and this leads to error
        printf("This is else body");
}
```

To remove this error, club both the simple statements into a compound statement. The rectified code is as follows:

```
main()
{
    int a=1;
    if(a==1)
    {                           //←Compound statement: It will be treated as a unit
        printf("This is if body\n");
        printf("This statement does not belong to if body");
    }
    else                        //←Now the else clause is properly matched with if clause
        printf("This is else body");
}
```

10. *Can the selection expression of a* switch *statement be a string?*

No, the selection expression of a switch statement cannot be a string. The switch selection expression and case labels must be of integral type. Hence, the switch statement can be used to switch only on integral data types (i.e. character and integer). Consider the following program segment:

```
main()
{
    switch("Hello")
    {
    case "Hello":
        printf("Hello");
```

```
    case "Hi":
        printf("Hi");
    }
}
```

In the above-mentioned piece of code, switch selection expression and case labels (shown in bold) are strings. This is not allowed and thus, the code on compilation gives an error.

11. *Can a* switch *statement have more than one* default *label?*
No, a switch statement cannot have more than one default label. In a switch statement, all the case labels must be unique and at most one default label can be present. The presence of more than one default label or duplicate case labels leads to ambiguity, which results in a compilation error.

12. *Why does the following piece of code on compilation gives an error?*

```
main()
{
int i=65;
switch(i)
{
case 65:
    printf("This statement should get executed\n");
    break;
case 'A':
    printf("A has ASCII code of 65, this statement should get executed\n");
    break;
default:
    printf("Duplicate case labels lead to error\n");
}
}
```

The mentioned piece of code on compilation gives 'Duplicate case in function main' error. This is due to the fact that integers and characters are not treated separately in C language. Characters are stored internally in terms of their ASCII values. Character 'A' has ASCII value 65. So, writing case 'A': is equivalent to writing case 65:. However, case label 65 is already present. Duplicate case labels are not allowed. Hence, this leads to 'Duplicate case in function main' error.

13. *Can we use a* continue *statement within the body of a* switch *statement like we can use a* break *statement within it?*
No, a continue statement can appear only in or as a loop body. A switch statement is a branching statement and not a looping statement. Hence, the continue statement cannot appear inside the body of a switch statement.

14. *Is it mandatory to have* case *labeled or* default *labeled statements within a* switch *body? If the* switch *body does not contain any* case *or* default *labeled statements, will there be a compilation error?*
The general form of a switch statement is:

```
switch(expression)        //←switch header
statement                 //←switch body
```

A switch body consists of a statement. This statement can be a null statement, an expression statement, a labeled statement, a flow control statement, a compound statement, etc. There is no constraint that only labeled statements can form the switch body. Hence, it is not mandatory to have case labeled or default labeled statements within the switch body. The following usages of switch statement (without any case labeled or default labeled statements) are valid:

```
a. switch(expr);                    //←switch body is a null statement
b. switch(expr)                     //←switch body consisting of two function call statements
   {
      printf("Two expression statements");
      printf("This is valid");
   }
c. switch(expr)                     //←switch body has a labeled statement, but the labeled
   {                                //  statement is an identifier labeled statement and not a case labeled
   lab:                             //  or a default labeled statement
   printf("This is also valid");
   goto lab;
   }
```

15. *Can* case *labeled or* default *labeled statement exist outside the* switch *body?*

No, case labeled statements and default labeled statements can appear only inside the switch body. Placing case labeled statements or default labeled statements outside the switch body leads to 'Case/Default outside of switch' compilation error.

16. *Why does the following piece of code gives an error on compilation?*

```
main()
{
int exp=2;
switch(exp)
{
case 1:
   int j=2;
   printf("The value of j in case 1 is %d\n",j);
case 2:
   printf("The value of j in case 2 is %d\n",j);
}
}
```

This compilation error is due to the fact that the placement of the definition statement associated with a case or default label is illegal unless it is placed within a statement block. The placement of the definition statement within a statement block is mandatory because if the definition is not enclosed within a statement block, the defined identifier would be visible➲ (i.e. can be used) across the case labels, but is initialized only if the case label within which it is defined is executed. The presence of the statement block ensures that the name is initialized whenever it is visible. In the given piece of code, int j=2; is not placed within a statement block. Hence, there is a compilation error. The compilation error can be removed by placing int j=2; within a statement block.

Forward Reference: Visibility and scope (Chapter 8).

17. *I have tried to rectify the problem in the code mentioned in the previous question. Does the following piece of code compile successfully?*

```
main()
{
int exp=2;
switch(exp)
```

```
{
case 1:
{
    int j=2;
    printf("The value of j in case 1 is %d\n",j);
}
case 2:
    printf("The value of j in case 2 is %d\n",j);
}
}
```

No, the given piece of code does not yet compile successfully. The given piece of code on compilation gives 'Undefined symbol 'j' in function main' error. This error is due to the fact that the identifier j defined within the statement block of case label 1 is visible (i.e. can be used) only inside it. The identifier j is not visible (i.e. does not exist) outside the statement block in which it is defined. Hence, reference to j in the printf statement of case label 2 is not valid and leads to the compilation error.

18. *Why does the following piece of code show just a sequence of zeros in its output?*

```
main()
{
    int number=2;
    while(1)
    {
        printf("%d ",number);
        number*=2;
    }
}
```

The code actually outputs
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 -32768 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...infinite times.
Initially number is two. It is represented in memory as:

Sign Bit 16 MSB	Magnitude													
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Multiplying by two makes it four (i.e. equivalent to shifting in left direction by 1 bit).

Sign Bit 16 MSB	Magnitude													
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

This shifting is continued and after 14 iterations, the number becomes:

Sign Bit 16 MSB	Magnitude													
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

i.e. -32768. If the shifting is further carried out, the number becomes zero.

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

From this point onwards irrespective of how many times shifting is carried out (i.e. number is multiplied by two), the number remains zero. Hence, from this point onwards, the output will only have zeros.
Now, the speed of processing is so fast that first few outputs will be skipped (cannot be seen in the output as the screen scrolls) and only a sequence of zeros can be seen. If you want to see all the outputs, put some delay mechanism inside the loop. This can be done by using either the function getch(), delay(int) or sleep(int).

The function **delay(int)** suspends the execution of the program for a given time interval. The time interval is an integer value and specifies the time in milliseconds. **sleep(int)** is a function equivalent to the delay function. The delay function is provided in the DOS environment and the sleep function is usually available with the WINDOWS environment.

19. *I want to test whether a character entered by the user lies in the range 'A' to 'C' or 'X' to 'Z'. Can I use a* switch *statement to do this?*

Yes, a switch statement can be used to accomplish it. Use the following piece of code to check whether the character entered lies in the range 'A' to 'C' or 'X' to 'Z'.

```
main()
{
    char ch;
    printf("Enter a character\t");
    scanf("%c",&ch);
    switch(ch)
    {
        case 'A':
        case 'B':
        case 'C':
            printf("The entered character is in range A-C");
            break;
        case 'X':
        case 'Y':
        case 'Z':
            printf("The entered character is in range X-Z");
            break;
        default:
            printf("The entered character is neither in range A-C nor in range X-Z");
    }
}
```

However, this method would not be practical if the ranges are bigger. In case of bigger ranges, usage of if-else statements with the involvement of logical operators in the controlling expressions is preferred.

Do labels have scope like variables?

Yes, labels do have scope like variables. A label name is a type of identifier that has only function scope. It can be used anywhere in the function in which it appears.

Forward Reference: Function scope (Chapter 8).

Can we use a goto *statement to take control from one function to some other function?*

A goto statement in no way can transfer control from one function to another function. Consider the following piece of code:

```
main()
{
   goto here;
}
other_function()
{
   here:
      printf("The label is in other function");
}
```

The above-mentioned code on compilation gives 'Undefined label here in function main' error. To remove this error, use label here somewhere inside the body of main function.

Can a label be followed by another label or should it be followed by a statement?

The general forms of labeled statements are:

1. identifier: statement
2. case constant-expression: statement
3. default: statement

After identifier label, case label or default label, there should be a statement. This statement can itself be another labeled statement. Hence, label can follow another label. For e.g.

```
lab:            //←label followed by another labeled statement
   try:
      printf("This is valid");
```

Due to this definition of a labeled statement, the following form of a switch statement is valid:

```
switch(expr)
{
   case 1:
   case 2:
      printf("Case 1 and Case 2");
   case 3: case 4: case 5:
      printf("Case 3,4 and 5");
}
```

Can a label name be the same as a function name or a variable name?

Yes, label name can be the same as a function name or a variable name. Consider the following piece of code:

```
main()
{
   int i=1;
```

```
    main:
        printf("Function name is used as label name\n");
    i++;
    if(i==2)
        goto i;
    goto main;
    i:
        printf("Variable name is used as label name\n");
}
```

In the given code, the function name, i.e. main and the variable name, i.e. i are used as label names. The given code on execution gives an output as:

```
Function name is used as label name
Variable name is used as label name
```

24. *Can a reserved word or a keyword like* while, if, *etc. be used as a label name?*

Reserved words or keywords cannot form valid identifier names. Since label names are identifiers, reserved words or keywords cannot be used as valid label names.

25. *All the identifiers need to be declared before their use. Label names are also identifiers. So, do we need to declare label names?*

No, label names need not be declared. Label names are identifiers but no type is associated with them. Hence, there is no need to explicitly declare them. Label names are implicitly declared by their syntactic appearance. An identifier followed by a colon and a statement is implicitly treated as a label name.

26. *Is a* goto *statement capable of taking the control in or out of a nested loop?*

Yes, a goto statement is capable of taking the control in or out of a nested loop. The goto statement is capable of taking control anywhere within a function in which it is used. Consider the following piece of code:

```
main()
{
    int i,j;
    for(i=1;i<5;i++)
        for(j=1;j<5;j++)
        {
            printf("This statement will be executed only once\n");
            goto label;
        }
    label:
        printf("goto statement has taken the control out of nested loop");
}
```

Upon execution, it gives the output as:

```
This statement will be executed only once
goto statement has taken the control out of nested loop
```

27. *Can a single* break *statement be used to terminate a nested loop?*

No, a single break statement cannot be used to terminate a nested loop. A break statement can only terminate the execution of the nearest enclosing switch or the nearest enclosing loop. Consider the following piece of code:

```
main()
{
    int i,j;
    for(j=1;j<3;j++)
    {
        for(j=1;j<3;j++)
        {
            break;
            printf("This will not get printed");
        }
        printf("This will be executed twice as it is inside outer loop\n");
    }
}
```

The break statement only terminates the inner for loop. The printf statement in the outer for loop executes normally. The above piece of code on execution outputs:

This will be executed twice as it is inside outer loop
This will be executed twice as it is inside outer loop

25. *What are entry-controlled and exit-controlled loops?*

In **entry-controlled loops**, condition is checked before the execution of body of the loop. The for loop and while loop are examples of entry-controlled loops. In **exit-controlled loops**, the condition is checked after the execution of body of the loop. do-while is an example of an exit-controlled loop. In entry-controlled loops, if the condition is initially false, the body of the loop will not be executed. However, in exit-controlled loops, even if the condition is initially false, the body of the loop will be executed once. Consider the following piece of code:

```
main()
{
    int i=2;
    do
        printf("Condition is false, but this will be printed");
    while(i<1);
}
```

The condition of a do-while loop is initially false; even then "Condition is false, but this will be printed" is the output. This indicates that the body of the exit-controlled loop gets executed once, even if the condition of the loop is initially false.

26. *What are counter-controlled and sentinel-controlled loops?*

Counter-controlled looping is a form of looping in which the number of times the loop will execute is known in advance. The counter-controlled loop starts with the initial value of the loop counter and terminates when the final value of the loop counter is reached. Since a counter-controlled loop iterates a fixed number of times, it is also known as a definite repetition loop. In sentinel-controlled looping, the number of times the loop will execute is not known beforehand. The execution or termination of the loop depends upon a special value called the sentinel value. If the sentinel value is true, the loop body gets executed else not. Since the number of times the loop will iterate is not known in advance, this type of loop is also known as an indefinite repetition loop.

27. *What are the three main ingredients of counter-controlled looping?*

Three main ingredients of counter-controlled looping are:

1. Initialization of the loop counter.
2. A condition determining whether the loop body should be executed or not.
3. An expression that manipulates the value of the loop counter so that the condition in Step 2 eventually becomes false and the loop terminates.

31. *For every usage of a* for *loop, we can write an equivalent* while *loop. So, when should one prefer to use a* for *loop and when should a* while *loop be preferred?*

A while loop should be preferred over a for loop when the number of iterations to be performed is not known in advance. The termination of the while loop is based on the occurrence of some particular condition, i.e. a specific sentinel value. The usage of a for loop should be preferred when the number of iterations to be performed is known beforehand. In short, a while loop is preferred for sentinel-controlled looping and a for loop is preferred for counter-controlled looping.

32. *Why does the following piece of code on compilation give a compilation error?*

```
main()
{
int i=2;
while(i<10);
{
    printf("The value of i is %d",i);
    if(i==5)
        break;
}
}
```

The given piece of code gives a compilation error due to the fact that the break statement can appear only in or as a switch body or a loop body. Here, the break statement does not appear inside the body of the while loop. The body of the while loop consists of a null statement. To rectify the given code, remove the semicolon present at the end of the while header.

33. *I want to terminate the nearest enclosing loop. Which construct in C provides me this functionality?*

To terminate the nearest enclosing loop, a break statement can be used. This can be seen by executing the following piece of code:

```
main()
{
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<5;j++)
        {
            if(i!=0 || j!=0)
                break;
            printf("This will be printed only once\n");
        }
    printf("This will be printed two times\n");
    }
}
```

34. *I want to terminate the current iteration of the nearest enclosing loop. Which construct in C provides me this functionality?*

To terminate the current iteration of the nearest enclosing loop, a continue statement can be used. This can be seen by executing the following piece of code:

```
main()
{
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<5;j++)
        {
            if(i!=0 || j!=0)
                continue;
            printf("This will be printed only once\n");
        }
    printf("This will be printed two times\n");
    }
}
```

The syntactic form of a for *loop is as follows:*

```
for(expression1;expression2;expression3)
statement
```

What is the order in which $expression_1$, $expression_2$, $expression_3$ and statement get evaluated?

The order in which the expressions are evaluated is:

1. $expression_1$ is evaluated before the first evaluation of the controlling expression $expression_2$. $expression_1$ is evaluated only once.
2. $expression_2$ is the controlling expression and is evaluated every time before the execution of the loop body. If $expression_2$ evaluates to true, the loop (i.e. statement) body will be executed otherwise the control will come out of the loop.
3. $expression_3$ is evaluated after the execution of the loop body.

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

```
int a=10,b=20,c;
c=a+b;
main()
{
    printf("Value of c is %d",c);
}
```

```
main()
{
    int a=10,b=20,c;
    c=a+2*b
    printf("The value of c is %d",c);
}
```

```
main()
{
    int a=10,b=20;
    if(a==b)
        printf("a=10,b=20");
```

```
        printf("a and b are not equal");
    }
```

39.
```
main()
{
    int a=10,b=20;
    if(a==b)
    {
        printf("a=10, b=20");
        printf("a and b are not equal");
    }
}
```

40.
```
main()
{
    int a=10,b=20;
    if(a=b)
        printf("a and b are equal");
    else
        printf("a and b are not equal");
}
```

41.
```
main()
{
    int a=10,b=20;
    if(a==b);
        printf("a and b are equal");
    else
        printf("a and b are not equal");
}
```

42.
```
main()
{
    int a=10,b=10;
    if(a==b)
        printf("a and b are equal\n");
    else;
        printf("a and b are not equal\n");
}
```

43.
```
main()
{
    if(1)
        printf("This will always get executed");
    else
        printf("This will never get executed");
}
```

44.
```
main()
{
    if(printf("Hello"))
        printf("Students");
}
```

45.
```
main()
{
int a=10,b=20;
if(a==10)
if(b==10)
printf("Value of a
else
printf("Value of a
}
```

46.
```
main()
{
    int a=10,b=20
    if(a==10)
    {
    if(b==10)
        printf("Val
    }
    else
        printf("Val
}
```

47.
```
main()
{
    int expr=10;
    switch(expr)
        printf("This
}
```

48.
```
main()
{
    int expr=10;
    switch(expr);
        printf("Tell
}
```

49.
```
main()
{
    float expr=2.0;
    switch(expr)
    {
        case 1: print
        case 2: print
        default: prin
    }
}
```

50.
```
main()
{
    int expr=2,j=1;
    switch(expr)
    {
    case j:
```

```
45. main()
    {
    int a=10,b=20;
    if(a==10)
    if(b==10)
    printf("Value of a and b is 10");
    else
    printf("Value of a is 10 and b is something else");
    }
46. main()
    {
        int a=10,b=20;
        if(a==10)
        {
        if(b==10)
            printf("Value of a and b is 10");
        }
        else
            printf("Value of a is 10 and b is something else");
    }
47. main()
    {
        int expr=10;
        switch(expr)
            printf("This is valid but will not get executed");
    }
48. main()
    {
        int expr=10;
        switch(expr);
            printf("Tell whether this will get executed or not");
    }
49. main()
    {
        float expr=2.0;
        switch(expr)
        {
            case 1: printf("One");
            case 2: printf("Two");
            default: printf("Default");
        }
    }
50. main()
    {
        int expr=2,j=1;
        switch(expr)
        {
        case j:
```

```
            printf("This is case 1");
        case 2:
            printf("This is case 2");
        default:
            printf("This is default case");
        }
    }
```

51.
```
main()
{
    char ch='A';
    switch(ch)
    {
        case 'A':
            printf("Case label is A");
        case "B":
            printf("Case label is B");
    }
}
```

52.
```
main()
{
    int expr=1;
    switch(expr)
    {
        case 1: printf("One\n");
        case 2: printf("Two\n");
        default: printf("Three\n");
    }
}
```

53.
```
main()
{
    int expr=1;
    switch(expr)
    {
        case 1:
            printf("One\n");
            break;
        case 2:
            printf("Two\n");
            break;
        default: printf("Three\n");
    }
}
```

54.
```
main()
{
    int expr=3;
    switch(expr)
    {
        default: printf("Three\n");
```

```
            case 1: pri
            case 2: pri
        }
    }
```

55.
```
main()
{
    int expr=2;
    switch(expr)
    {
        case 1:
            printf("
        case 2-1:
            printf("
    }
}
```

56.
```
main()
{
    int i=1,j=3;
    switch(i)
    {
        case 1:
            printf("
            switch(j
            {
                case
                    p
                    b
                defau
                    p
            }
        case 2:
            printf("Th
    }
}
```

57.
```
main()
{
    int expr=2;
    switch(expr)
    {
        case 1:
            printf("Thi
            break;
        case 2:
            printf("Thi
            continue;
        default:
```

```
        case 1: printf("One\n");
        case 2: printf("Two\n");
    }
}

main()
{
    int expr=2;
    switch(expr)
    {
        case 1:
            printf("This is case 1");
        case 2-1:
            printf("This is case 2");
    }
}

main()
{
    int i=1,j=3;
    switch(i)
    {
        case 1:
            printf("This is outer case 1\n");
            switch(j)
            {
                case 3:
                    printf("This is inner case 1\n");
                    break;
                default:
                    printf("This is inner default case");
            }
        case 2:
            printf("This is outer case 2");
    }
}

main()
{
    int expr=2;
    switch(expr)
    {
        case 1:
            printf("This is case 1");
            break;
        case 2:
            printf("This is case 2");
            continue;
        default:
```

```
            printf("Default");
        }
    }
```

58.
```
main()
{
    default:
        printf("This is default labeled statement");
    goto default;
}
```

59.
```
main()
{
    int i=1;
    while(i<=5)
    {
        printf("%d ",i);
        i=i+1;
    }
    printf("\nThe value of i after the loop is %d",i);
}
```

60.
```
main()
{
    int i=1;
    while(i<=5);
    {
        printf("%d\n ",i);
        i=i+1;
    }
    printf("The value of i after the loop is %d",i);
}
```

61.
```
main()
{
    int i=1;
    while(i<=5)
        printf("%d ",i);
    printf("The value of i after loop is %d",i);
}
```

62.
```
main()
{
    int i=1;
    for(    )
    {
        printf("%d",i);
        if(i=5)
            break;
    }
}
```

```
main()
{
    int i;
    for(i=1;i<=32767;i++)
        printf("%d ",i);
}

main()
{
    int i=1;
    for(;;)
    {
        printf("%d ",i);
        if(i==5)
            break;
    }
}

main()
{
    int i=1;
    for(;;)
    {
        printf("%d",i);
        if(i=5)
            break;
    }
}

main()
{
    int i=1;
    for(i<=5;printf("%d ",i++));
}

main()
{
    int i=1;
    for(;i<=10;i++)
    {
        if(i%2==0)
            continue;
        printf("%d ",i);
    }
}

main()
{
    int i=1;
    loop:
        printf("%d ",i++);
```

```
        if(i==5) break;
        goto loop;
    }
```

69.
```
main()
{
    int i=1;
    loop:
        printf("%d ",i++);
    if(i==5) goto out;
    goto loop;
    out:
    ;
}
```

70.
```
main()
{
    int i,j;
    for(i=1;i<3;i++)
        for(j=1;j<4;j++)
        {
            if(j==2) break;
            printf("%d %d\n",i,j);
        }
}
```

71.
```
main()
{
    int i,j;
    for(i=1;i<3;i++)
        for(j=1;j<4;j++)
        {
            if(j==2) continue;
            printf("%d %d\n",i,j);
        }
}
```

72.
```
main()
{
    int i=3;
    for(;i++=0;)
        printf("%d",i);
}
```

73.
```
main()
{
    int a=0, b=20;
    char x=1, y=10;
    if(y,x,b,a)
        printf("hello");
}
```

```
74. main()
    {
        int i=0;
        for(;i++;)
            printf("%d",i);
    }
75. main()
    {
        int i=0;
        for(;++i;)
            printf("%d",i);
    }
76. main()
    {
        int i=3,j=3;
        for(;i<6,j<4;i++,j++)
            printf("%d %d\n",i,j);
    }
77. main()
    {
        int i=1;
        while (i<=5)
        {
            printf("%d",i);
            if (i>2)
                goto here;
            i++;
        }
    }
    other_function()
    {
        here:
            printf("The label is in other function");
    }
78. main()
    {
        int i=3;
        goto label;
        for(i=0;i<5;i++)
        {
            label:
                printf("%d ",i);
        }
    }
79. main()
    {
        int i=5;
```

```
    do
    {
        printf("%d",i);
        i++;
    }while(i<10)
}
```

80.
```
main()
{
    int i=5;
    do
    {
        printf("%d",i);
        i++;
    }while(i<0);
}
```

Multiple-choice Questions

81. The smallest independent logical unit in a C program is
    a. Expression
    b. Token
    c. Statement
    d. None of these

82. In C language, statements are terminated with
    a. Period
    b. Semicolon
    c. New-line character
    d. None of these

83. By default, statements in a C program are executed
    a. Randomly
    b. Sequentially in top to bottom order
    c. Sequentially in bottom to top order
    d. None of these

84. int ival; is actually a
    a. Declaration statement
    b. Definition statement
    c. Neither a declaration statement nor a definition statement
    d. Declaration as well as a definition statement

85. Sentinel-controlled loop is also known as
    a. Definite repetition loop
    b. Infinite repetition loop
    c. Indefinite repetition loop
    d. None of these

86. Case label inside switch body must be
    a. An expression
    b. An integral expression
    c. A constant integral expression
    d. An integer constant

87. Which of the following forms of for statement is syntactically valid
    a. for(;;);
    b. for(;;)
    c. for(;)
    d. for();

88. The selection expression of switch statement must be of
    a. Integer type
    b. Float type
    c. Integral type
    d. String type

The C construct that is used to terminate the current iteration of a loop is

a. break statement
b. continue statement
c. return statement
d. None of these

Dangling else is an ambiguity that arises when in a statement the number of else clauses are

a. Equal to the number of if clauses
b. Less than the number of if clauses
c. Greater than the number of if clauses
d. None of these

The C construct that is used to terminate a loop is

a. break statement
b. continue statement
c. return statement
d. None of these

Minimum number of times a do-while loop will be executed is

a. 0
b. 1
c. Cannot be predicted
d. None of these

Which of the following statement is true about continue statement?

a. It terminates the loop
b. It terminates the current iteration of the loop
c. It can be used in or as a switch body
d. None of these

The body of a switch statement must consist of

a. Case-labeled statements
b. Default-labeled statements
c. A statement
d. Null statement

A continue statement can only be used in or as

a. switch body
b. Loop body
c. if body
d. None of these

Labels have

a. Block scope
b. Global scope
c. Function scope
d. File scope

A goto statement cannot take control

a. Out of nested if-else
b. Out of a nested loop
c. Out of a function
d. None of these

Consider the following segment of C code:

```
int j;
j=1;
while(j<=n)
  j=j*2;
```

The number of comparison made in the execution of the loop for any n>0 is

a. ceiling($\log_2 n$)+2
b. n
c. ceiling($\log_2 n$)+1
d. floor($\log_2 n$)+2

Consider the following fragment of C code in which i, j and n are integer variables.

```
for(i=n,j=0;i>0;i/=2,j+=1);
```

The value of j after the termination of for loop is

a. floor($\log_2 n$)+1
b. n/2+1
c. n
d. ceiling($\log_2 n$)+1

100. Consider the following fragment of C code. How many times will the following loop be executed?

```
x=500;
while(x<=500)
{
   x=x-600;
   if(x<0) break;
}
```

a. 0
b. 100
c. 500
d. 1

Outputs and Explanations to Code Snippets

36. Compilation error

Explanation:

Non-executable statements can be placed outside the body of a function but executable statements can only be placed within the body of a function. c=a+b; is an executable statement and cannot be placed outside the body of a main function. To remove this error, place the statement c=a+b; inside the body of the main function.

37. Compilation error (Statement missing ; in function main)

Explanation:

c=a+2*b is not a statement. It is an expression. No entity smaller than a statement can independently exist in a C program. Hence, the error. To remove this error, convert the expression c=a+2*b into a statement by terminating it with a semicolon.

38. a and b are not equal

Explanation:

printf("a and b are not equal"); does not belongs to if body. It is a statement next to if statement and will always be executed irrespective of the result of evaluation of if controlling expression.

39. No output

Explanation:

The if body is a compound statement consisting of two printf statements. Being a compound statement, it will be treated as a unit, i.e. a single statement. Either all of its constituent statements will be executed or none will get executed depending upon the outcome of the if controlling expression. Here, the if controlling expression evaluates to false. Hence, if body (i.e. printf statements) will not be executed and thus, there is no output.

40. a and b are equal

Explanation:

The controlling expression of if-else statement is a=b. An assignment operator has been used instead of equality operator. The value of b is assigned to a and the value of expression comes out to be 20 (i.e. the assigned value of b). 20 is a non-zero value, i.e. true. If the if-else controlling expression evaluates to true, if body will get executed. Hence, if body (i.e. printf("a and b are equal");) gets executed and a and b are equal is the result.

41. Compilation error (Misplaced else in function main)

Explanation:

This error is due to the presence of a semicolon after the if-else controlling expression. The mentioned code will be interpreted in the following way:

```
main()
{
    int a=10,b=20;
    if(a==b)
        ;                       //←if body is a null statement
    printf("a and b are equal");    //←This statement is next to if statement
    else                        //←else clause is without any if clause
        printf("a and b are not equal");
}
```

To rectify this code, either remove the semicolon or make the null statement and printf("a and b are equal"); statement a single statement by enclosing them within braces.

42. a and b are equal
a and b are not equal

Explanation:

a and b are not equal is a part of the output due to the presence of a semicolon after the else clause. Null statement forms the else body. printf("a and b are not equal"); statement is a statement next to the if-else statement and will always get executed irrespective of the result of the evaluation of if-else controlling expression.

43. This will always get executed

Explanation:

The controlling expression of if-else statement is 1. 1 is a non-zero value and is considered as true. Every time you run this program, This will always get executed is the output as if-else controlling expression always evaluates to true.

44. HelloStudents

Explanation:

The controlling expression of if statement is evaluated first. Controlling expression of if statement is printf("Hello"). Function calls are valid expressions, so writing if(printf("Hello")) will not lead to any compilation error. The expression gets evaluated and Hello is printed on the screen. The printf function also returns an integer value. The value returned by the printf function is the number of characters it prints. The number of characters in Hello is 5; hence, printf function returns 5. 5 is a non-zero value and is treated as true. As the controlling expression of if statement evaluates to true, if body gets executed and Students is printed on the screen. Hence, the output that gets printed is: HelloStudents.

Forward Reference: Functions and the values returned by them (Chapter 5).

45. Value of a is 10 and b is something else

Explanation:

The code suffers from dangling else ambiguity. The ambiguity is implicitly resolved by the compiler and the code is interpreted in the following way:

```
main()
{
    int a=10,b=20;
    if(a==10)
        if(b==10)
            printf("Value of a and b is 10");
```

```
        else
            printf("Value of a is 10 and b is something else");
}
```

The given code has an if statement whose body consists of an if-else statement. The controlling expression of if statement (i.e. a==10) evaluates to true, so its body (i.e. if-else statement) gets executed. The controlling expression of if-else statement (i.e. b==10) evaluates to false, and hence the else body, i.e. printf("Value of a is 10 and b is something else"); gets executed.

46. No output

Explanation:

This code does not suffer from dangling else ambiguity. There is an if-else statement whose if body consists of another if statement and else body consists of a printf statement. The controlling expression of an if-else statement (i.e. a==10) evaluates to true, hence its if body will be executed and else body will be skipped. The if statement present inside the if body of if-else statement starts execution and its controlling expression (i.e. b==10) evaluates to false. Hence, its body will not be executed and thus, nothing gets printed.

47. No output

Explanation:

The switch statement is executed according to the rule mentioned below:
The switch selection expression is evaluated and the result of evaluation of the switch selection expression is compared against the value associated with each case label until either a match is successful or all labels have been examined. If the result of evaluation of the switch selection expression matches the value of a case label, the execution begins from the statement with that case label. The execution continues across case/default boundaries till the end of the switch statement. If there is no match, the execution begins from the statement with the default label if it is present; otherwise the execution of the program continues with the statement following the switch statement.
According to the above-mentioned rule, execution can start only with the matched case labeled statement or the default labeled statement, if it is present. Since the printf statement is neither a matched case labeled statement nor a default labeled statement, it will not be executed. Hence, there will be no output.

48. Tell whether this will get executed or not

Explanation:

Null statement present after the switch controlling expression forms the switch body. The printf statement does not belong to switch body and is a statement present next to the switch statement. This statement will always be executed irrespective of the value of switch selection expression.

49. Compilation error

Explanation:

switch selection expression and case labels must be of integral type. Since in the given code switch selection expression is of float type, there will be a compilation error.

50. Compilation error

Explanation:

Case label must be a compile time constant integral expression. Since in the given code, variable j is used as case label, there is a violation of syntactic rule and this leads to the compilation error.

52. Compilation error

Explanation:

Case label must be of integral type, i.e. either integer type or character type. Usage of string as case label (i.e. case "B") is a violation of syntactic rule and leads to the compilation error.

53. One
Two
Three

Explanation:

A common misunderstanding is that only the statements associated with the matched case label are executed. Rather, execution begins there and continues across case/default boundaries until the end of switch statement is encountered.

54. One

Explanation:

The case label 1 gets matched with the value of switch selection expression. Execution begins from the statement with the case label 1. printf("One\n"); gets executed and One is printed on the screen. The execution of statements would have been carried out till the end of switch statement but the break statement is encountered after the printf statement. This break statement terminates the switch statement. Hence, the rest of the case labeled, default labeled and other statements do not get executed. Thus, One is the output.

55. Three
One
Two

Explanation:

There is no constraint about the position of default labeled statement within the switch body. It can be placed before the case labeled statements, in-between the case labeled statements or after the case labeled statements. Generally, it is placed after the case labeled statements but it can be placed anywhere within the switch body. In the given piece of code, default labeled statement is placed before the case labeled statements. The result of evaluation of switch selection expression is matched with the case labels. Since none of the case labels (i.e. 1 and 2) get matched with the evaluated value of the switch selection expression (i.e. 3), the execution starts from the statement with the default label and is carried out across the case boundaries till the end of the switch statement. Hence, the printf statements associated with case labels 1 and 2 also gets executed.

56. Compilation error

Explanation:

The case labels should be unique. Although the case labels in the given piece of code seems to be unique but they are actually the same. The constant expression 2-1 gets evaluated to 1. Since case label 1 is already present, there is 'Duplicate case in function main' error.

57. This is outer case 1
This is inner case 1
This is outer case 2

Explanation:

The body of the switch statement consists of three statements:

1. case labeled statement-1
   case 1:

printf("This is outer case 1\n");

2. switch(j) { ...}
3. case labeled statement-2

   case 2:

   printf("This is outer case 2\n");

The execution of the statements starts from the statement with the matched case label. Since case label 1 gets matched with the value of the switch selection expression (i.e. value of i), the execution starts with printf("This is outer case 1\n");. The execution from this point is carried out till the end of the switch statement. After the execution of the printf statement, statement 2, i.e. the inner switch statement starts execution. The body of the inner switch also consists of three statements:

1. Case-labeled statement-1

   case 3:

   printf("This is inner case 1\n");
2. break;
3. Default-labeled statement

   default:

   printf("This is inner default case\n");

Since the value of selection expression of the inner switch (i.e. value of j) matches the case label 3, execution starts with printf("This is inner case 1\n");. Execution from this point would have been carried out till the end of the inner switch statement but after the execution of printf statement break statement is encountered. This break statement terminates the execution of the nearest enclosing switch (i.e. inner switch statement). Hence, the default labeled statement is not executed and the control is immediately transferred to the case labeled statement-2 of the outer switch statement. The statement printf("This is outer case 2\n"); gets executed.

This illustrates that there can be a switch statement within the body of another switch statement. Hence, switch statements can be nested.

57. Compilation error: "Misplaced continue in function main()"

Explanation:

A continue statement shall appear only in or as a loop body. It cannot appear in or as a switch body. In the given piece of code, continue is placed inside the switch body. This is a violation of the syntactic rule and leads to the compilation error 'Misplaced continue in function main.'

58. Compilation error

Explanation:

Remember the following syntactic rules:

1. case labeled and default labeled statements can appear only inside the switch statement.
2. case label and default label cannot be used with a goto statement. Only identifier labels can be used with the goto statement.

Since there is violation of both the above-mentioned rules, there are compilation errors:

1. 'Default outside of switch in function main' (Due to violation of rule 1)
2. 'Goto statement missing label in function main' (Due to violation of rule 2)

59. 1 2 3 4 5

The value of i aft

Explanation

The controlli
is executed.
with the valu
gets printed.
loop termina
statement.

60. No output

Caution:

Infinite loop

Explanation:

The presence
infinite loop.
ment gets exe
no output. Th
of the loop co
absence of a
evaluates to t
the program

61. 1 1 1 1 ...infinite tim

Caution:

Infinite loop

Explanation:

An expressior
sion of the whil

62. Compilation error

Explanation:

The general fo
for(expression$_1$;exp
statement
All the express
missing, it is n
the for header
a compilation

63. 1 2 3 ...32767 -3276

Caution:

Infinite loop

Explanation:

The loop count
the loop body
of i becomes 2.

59. 1 2 3 4 5
The value of i after the loop is 6

Explanation:

The controlling expression (i<=5) is evaluated first and comes out to be true. The body of the loop is executed. 1 gets printed and value of i becomes 2. The controlling expression is evaluated again with the value of i being 2 (i.e. 2<=5). It comes out to be true and 2 gets printed. In this way 3 4 5 gets printed. The value of i becomes 6. The controlling expression (i.e. 6<=5) becomes false and the loop terminates. The value of i when the loop terminates is 6 and gets printed by the next printf statement.

60. No output

Caution:

Infinite loop

Explanation:

The presence of a semicolon at the end of the while header makes this program to stick into an infinite loop. The controlling expression of a while statement is true and the body of the while statement gets executed. The body of the while statement is a null statement. Null statement produces no output. There is no expression in the body of the while statement that manipulates the value of the loop counter so that the controlling expression eventually evaluates to false. Due to the absence of a manipulating expression, the controlling expression of the while statement always evaluates to true and keeps on executing the null statement. Hence, there will be no output and the program will not terminate as it is trapped inside an infinite loop.

61. 1 1 1 1 ...infinite times

Caution:

Infinite loop

Explanation:

An expression that manipulates the value of the loop counter is missing. The controlling expression of the while statement always evaluates to true. Thus, an infinite loop.

62. Compilation error

Explanation:

The general form of for statement is:
for(expression$_1$;expression$_2$;expression$_3$)
statement
All the expressions in for header are optional and can be skipped. Even if all the expressions are missing, it is mandatory to create three sections by placing two semicolons. In the given code, the for header does not have the required sections. Thus, it is syntactically incorrect and leads to a compilation error.

63. 1 2 3 ...32767 -32768 -32767...32767 -32768 -32767...infinite times

Caution:

Infinite loop

Explanation:

The loop counter i is initialized to 1. The condition i<=32767 (i.e. 1<=32767) evaluates to true. Hence, the loop body gets executed and 1 is printed. The expression i++ gets evaluated and the value of i becomes 2. Condition i<=32767 (i.e. 2<=32767) evaluates to true. The loop body gets executed

and 2 is printed. This process is continued till the value 32767 gets printed. Now, when i++ is evaluated, the value of i does not becomes 32768 as 32768 exceeds the range of the integer data type. Instead it becomes -32768 due to the wrap around effect. Thus, the condition i<=32767 (i.e. -32768<=32767) still evaluates to true. Hence, the condition never becomes false and the loop will not terminate.

64. 1 1 1 ...infinite times

Caution:

Infinite loop

Explanation:

for(;;) is syntactically valid and semantically (i.e. logically) it is an infinite loop. Inside the body of for loop, a break statement is present and it seems to be an exit path from the loop. The break statement will only be executed if the value of i becomes 5. Since the body of the for loop contains no expression to manipulate the value of i, the value of i will never become 5 and thus the break statement will never be executed. Hence, 1 will be printed infinite number of times.

65. 1

Explanation:

The initial value of i is 1. No condition is present inside the header of the for loop. Hence, without checking any condition, the body of the loop starts execution. printf("%d",i) gets executed and the value 1 gets printed. The statement present next to the printf statement is an if statement. The controlling expression of if statement is evaluated. The if controlling expression (i.e. i=5) has an assignment operator instead of an equality operator. The value 5 is assigned to i and the if controlling expression evaluates to true. Thus, the body of if statement, i.e. break statement gets executed. The break statement terminates the for loop. Hence, 1 is the output.

66. 1 2 3 4 5

Explanation:

In the given piece of code, condition i<=5 (i.e. 1<=5) evaluates to true. Thus, the body of the for loop, i.e. a null statement gets executed. After the execution of the body, the manipulation section (i.e. printf("%d ",i++)) gets executed. It prints the current value of i (i.e. 1) and then increments the value of i to 2. Again, the condition is checked and the above process is repeated. In this way 2 3 4 5 also gets printed.

67. 1 3 5 7 9

Explanation:

For even values of i, the if controlling expression i%2==0 evaluates to true. The body of the if statement (i.e. continue statement) gets executed. The continue statement on execution, immediately transfers the control to the header of the loop and the rest of the statements in the body of the loop will not be executed for the current iteration. Thus, for the even values of i, printf statement will not be executed.

68. Compilation error

Explanation:

break statement shall appear only in or as a switch body or a loop body. Logically, we have created a loop by using goto statement but since no looping construct (i.e. for, while or do-while) is used, a break statement cannot be placed there. Hence, the compilation error 'Misplaced break in function main' occurs.

69. 1 2 3 4

Explanation:

goto loop; state
out of this lo
The value of
the control ou

70. 1 1
2 1

Explanation:

Value of i	
1	
2	
3	

71. 1 1
1 3
2 1
2 3

Explanation:

Value of i	C
1	
2	
3	

1 2 3 4

Explanation:

goto loop; statement is used to create a logical loop and goto out; statement is used to take the control out of this logical loop. The printf statement present inside the logical loop prints the value of i. The value of i is manipulated by the expression i++. When the value of i becomes 5, goto out; takes the control out of the logical loop and the logical loop terminates.

11
21

Explanation:

Value of i	Condition of outer for loop	Value of j	Condition of inner for loop	Controlling expression of if statement (j==2)	Whether break is executed	Whether printf statement is executed	The values that get printed
1	True	1	True	False	No	Yes	11
		2	True	True	Yes	No	
2	True	1	True	False	No	Yes	21
		2	True	True	Yes	No	
3	False	Outer for loop is terminated					

11
13
21
23

Explanation:

Value of i	Condition of outer for loop	Value of j	Condition of inner for loop	Controlling expression of if statement (j==2)	Whether continue is executed	Whether printf statement is executed	The values that get printed
1	True	1	True	False	No	Yes	11
		2	True	True	Yes	No	
		3	True	False	No	Yes	13
		4	False	Inner for loop is terminated			
2	True	1	True	False	No	Yes	21
		2	True	True	Yes	No	
		3	True	False	No	Yes	23
		4	False	Inner for loop is terminated			
3	False	Outer for loop is terminated					

72. Compilation error

Explanation:

++ operator has higher priority than an assignment operator and will get evaluated first. The expression i++ evaluates to an r-value and cannot be placed on the left side of the assignment operator. Thus, i++=0 leads to the compilation error 'L-value required in function main'.

73. No output

Explanation:

The if controlling expression is y,x,b,a. In if controlling expression, the sub-expressions y, x, b and a are separated by comma operators. The comma-separated expressions (i.e. y, x, b and a) are evaluated from left to right and the result of evaluation of full expression is the result of evaluation of the right-most sub-expression (i.e. a). Since a is 0, the value of the entire expression y,x,b,a turns out to be 0. 0 is considered as false and hence the if body will not be executed.

74. No output

Explanation:

i is initialized with 0. The condition of the for loop (i.e. i++) has post-increment operator. This means, firstly the value of i (i.e. 0) is used for the evaluation of expression and then the value of i will be incremented. Thus, the controlling expression of the loop evaluates to 0, i.e. false. Hence, the body of the loop will not be executed and there will be no output.

75. 1 2 ...32767 -32768 -32767...-1

Explanation:

The condition of the for loop has a pre-increment operator. The value of i is incremented first and then used for the evaluation of expression. The value of i first becomes 1 and then is used for the evaluation of the for controlling expression. Since the controlling expression evaluates to true, the body of the loop will be executed and 1 gets printed. The condition is evaluated again, i becomes 2 and gets printed. This process is repeated till 32767 gets printed. Now, when ++i is evaluated, i becomes -32768 instead of 32768 (due to the range wrapping to the other side). This is a non-zero value and will be treated as true and it gets printed. The condition is evaluated again, i becomes -32767 and gets printed. This process is repeated till -1 is printed. After printing of -1, ++i gets evaluated and i becomes 0. 0 is treated as false; hence, the condition of the loop becomes false and the loop terminates.

76. 3 3

Explanation:

The condition of the for loop is an expression i<6,j<4. This expression has two sub-expressions separated by a comma operator. The sub-expressions will be evaluated from left to right but the outcome of the full expression, i.e. i<6,j<4 depends upon the outcome of the right-most sub-expression, i.e. j<4. Hence, till the sub-expression j<4 evaluates to true, the body of the loop will be executed. The initial value of j is 3. The sub-expression j<4 (i.e. 3<4) evaluates to true and the body of the loop will be executed. The value that gets printed is 3 3. After the execution of the body of the loop, the expression i++,j++ gets evaluated. Both i and j become 4. Now the condition j<4 (i.e. 4<4) evaluates to false and the loop gets terminated.

77. Compilation error

Explanation:

A label name is a type of identifier that has only function scope. In function main, goto here statement is present but there is no label named here. The label here present inside the body of other_function is not visible inside the function main, as label names have only function scope. Hence the compilation error 'Undefined label here in function main' occurs.

Forward Reference: Scopes, function scope (Chapter 8).

78. 3 4

Explanation:

The goto statement is capable of taking the program control in or out of a loop. In the given piece of code, the goto statement is used to transfer the program control inside the for loop. Since the goto statement transfers the control inside the for loop, the initialization expression in the for header will not be executed. Hence, the value of i remains 3 instead of being initialized to 0. After this the for loop works normally and 3 4 gets printed.

79. Compilation error

Explanation:

The general form of the do-while statement is:
do
statement
while(expression);
The semicolon after the while controlling expression is a must, else there will be a compilation error 'Statement ; missing'.

80. 5

Explanation:

do-while is an exit-controlled loop. The body of the loop will be executed once, even if the controlling condition is initially false. In the given piece of code, the controlling expression is initially false, even then the body of the do-while loop is executed once, and 5 gets printed.

Answers to Multiple-choice Questions

81. c 82. b 83. b 84. b 85. c 86. c 87. a 88. c 89. b 90. b 91. a 92. b 93. b 94. c 95. b 96. c 97. c 98. d 99. a 100. d

Programming Exercises

Program 1 | Check whether a given number is even or odd without using modulus operator

Whether a number is even or odd can be determined by checking its Least Significant Bit (LSB). If the first bit of a number is:

LSB

- 0, the number is even, e.g. 6, i.e. 0000 0000 0000 0110
- 1, the number is odd, e.g. 13, i.e. 0000 0000 0000 1101

Name: PE 3-1.c

```
1  //Even or odd without using modulus operator
2  #include<stdio.h>
3  main()
4  {
5  int num;
6  printf("Enter the number\t");
7  scanf("%d",&num);
8  if(num&1)==0)
9      printf("Number %d is even",num);
10 else
11     printf("Number %d is odd",num);
12 }
```

Output window

```
Enter the number    12
Number 12 is even
```

In function main, goto here
present inside the body c
only function scope. Hence

Program 2 | Check whether a given year is leap or not

A year is a **leap year**, if:
- It is divisible by 4 but not by 100, or
- It is divisible by 400.

PE 3-2.c

```
1  //Leap year
2  #include<stdio.h>
3  main()
4  {
5  int year;
6  printf("Enter the year\t");
7  scanf("%d",&year);
8  if(((year%4==0) && (year%100!=0)) || (year%400==0))
9      printf("%d is a leap year", year);
10 else
11     printf("%d is not a leap year", year);
12 }
```

Output window

```
Enter the year    2004
2004 is a leap year
```

Program 3 | Calculate the roots of a quadratic equation

The roots of a quadratic equation $ax^2 + bx + c = 0$ can be obtained by using the expression $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, where b^2-4ac is called discriminant.

If $b^2-4ac > 0$, the roots are real and unequal.

If $b^2-4ac = 0$, the roots are real and equal, i.e. $x = \frac{-b}{2a}$.

If $b^2-4ac < 0$, the roots are imaginary, i.e. $x = \frac{-b}{2a} \pm \sqrt{\frac{b^2 - 4ac}{2a}}\, i$.

PE 3-3.c

```
1  //Roots of a quadratic equation
2  #include<stdio.h>
3  #include<math.h>
4  main()
5  {
6  int a, b, c, d;
7  float r1,r2;
8  int num;
9  printf("Enter the coefficients a, b and c\t");
10 scanf("%d %d %d", &a, &b, &c);
11 d=b*b-4*a*c;
12 if(d>0)
13 {
14     r1=(-b+sqrt(d))/(2*a);
15     r2=(-b-sqrt(d))/(2*a);
16     printf("Roots are real and unequal\n");
```

Output window

```
Enter the coefficients a, b and c   1 4 3
Roots are real and unequal
Roots are: -1.000000 -3.000000
```

(Contd...)

```
    printf("Roots are: %f %f",r1,r2);
}
else if(d==0)
{
    r1= -b/(2*a);
    printf("Roots are real and equal\n");
    printf("Roots are: %f %f",r1,r1);
}
else
    printf("No real roots, roots are imaginary");
}
```

Program 4 | Find the sum of individual digits in a given positive integer number

PE 3-4.c

```
//Find sum of digits of a given number
#include<stdio.h>
main()
{
int num,sum=0,digit;
printf("Enter the number\t");
scanf("%d",&num);
while(num!=0)
{
    digit=num%10;
    sum=sum+digit;
    num=num/10;
}
printf("Sum of digits is %d",sum);
}
```

Output window

```
Enter the number    786
Sum of digits is 21
```

Program 5 | Find the reverse of a given number

PE 3-5.c

```
//Reverse of a given number
#include<stdio.h>
main()
{
int num,reverse=0, digit;
printf("Enter the number\t");
scanf("%d",&num);
while(num!=0)
{
    digit=num%10;
    num=num/10;
    reverse=reverse*10+digit;
}
printf("Reverse is %d", reverse);
}
```

Output window

```
Enter the number    534
Reverse is 435
```

(Contd...)

Program 6 | Check whether a given number is a palindrome or not

A number is a **palindrome** if the reverse of the number is equal to the number itself, e.g. 121, 535, etc.

Line	PE 3-6.c
1	//Palindrome
2	#include<stdio.h>
3	main()
4	{
5	int num, temp, digit, reverse=0;
6	printf("Enter the number\t");
7	scanf("%d",&num);
8	temp=num;
9	while(temp!=0)
10	{
11	digit=temp%10;
12	temp=temp/10;
13	reverse=reverse*10+digit;
14	}
15	if(num==reverse)
16	printf("%d is a palindrome", num);
17	else
18	printf("%d is not a palindrome", num);
19	}

Output window

```
Enter the number    1234
1234 is not a palindrome
```

Output window (second execution)

```
Enter the number    12321
12321 is a palindrome
```

Program 7 | Check whether a given number is perfect or not

An integer is said to be a **perfect number** if its factors (including 1) sum to the number, e.g. 6 is a perfect number as 6=1+2+3, 28 is a perfect number as 28=1+2+4+7+14.

Line	PE 3-7.c
1	//Perfect number
2	#include<stdio.h>
3	main()
4	{
5	int num, sum=0, i;
6	printf("Enter the number\t");
7	scanf("%d",&num);
8	for(i=1;i<num;i++)
9	{
10	if(num%i==0)
11	sum=sum+i;
12	}
13	if(num==sum)
14	printf("%d is a perfect number", num);
15	else
16	printf("%d is not a perfect number", num);
17	}

Output window

```
Enter the number    28
28 is a perfect number
```

Output window (second execution)

```
Enter the number 23
23 is not a perfect number
```

Program 8 | Pri

Line	PE 3-8.c
1	//First n perfe
2	#include<stdio.h
3	main()
4	{
5	int num=1, sum=
6	printf("How ma
7	scanf("%d", &n)
8	printf("Perfect n
9	while(count<=n)
10	{
11	for(i=1;i<num
12	{
13	if(num%i
14	sum=
15	}
16	if(num==sum
17	{
18	printf("%d
19	count++;
20	}
21	num++; sum=0
22	}
23	}

Program 9 | Check

A number is said to b
is an Armstrong num

Line	PE 3-9.c
1	//Armstrong numb
2	#include<stdio.h>
3	main()
4	{
5	int num, temp, digit,
6	printf("Enter the nu
7	scanf("%d",&num);
8	temp=num;
9	while(temp!=0)
10	{
11	digit=temp%10;
12	sum=sum+digit*
13	temp=temp/10;
14	}
15	if(num==sum)
16	printf("%d is an A
17	else
18	printf("%d is not
19	}

Program 8 | Print first n perfect numbers

Line	PE 3-8.c
1	//First n perfect numbers
2	#include<stdio.h>
3	main()
4	{
5	int num=1, sum=0, i, count=1, n;
6	printf("How many numbers you want to print\t");
7	scanf("%d", &n);
8	printf("Perfect numbers are:\n");
9	while(count<=n)
10	{
11	for(i=1;i<num;i++)
12	{
13	if(num%i==0)
14	sum=sum+i;
15	}
16	if(num==sum)
17	{
18	printf("%d\t",num);
19	count++;
20	}
21	num++; sum=0;
22	}
23	}

Output window

```
How many numbers you want to print    3
Perfect numbers are:
6    28    496
```

Program 9 | Check whether a given number is an Armstrong number or not

A number is said to be an **Armstrong number** if the sum of cube of its digits is equal to the number itself, e.g. 153 is an Armstrong number as $153=1^3+5^3+3^3$, i.e. 153 = 1 + 125 + 27.

Line	PE 3-9.c
1	//Armstrong number
2	#include<stdio.h>
3	main()
4	{
5	int num, temp, digit, sum=0;
6	printf("Enter the number\t");
7	scanf("%d",&num);
8	temp=num;
9	while(temp!=0)
10	{
11	digit=temp%10;
12	sum=sum+digit*digit*digit;
13	temp=temp/10;
14	}
15	if(num==sum)
16	printf("%d is an Armstrong number", num);
17	else
18	printf("%d is not an Armstrong number",num);
19	}

Output window

```
Enter the number    153
153 is an Armstrong number
```

Output window (second execution)

```
Enter the number 221
221 is not an Armstrong number
```

Program 10 | Fibonacci series

Fibonacci series is a series in which a term is equal to the sum of the previous two terms. The first term of the series is 0 and the second term is 1.

Line	PE 3-10.c	Output window
1	//Fibonacci series: 0 1 1 2 3 5 8 13 21 ...	How many terms do you want to print 5
2	#include<stdio.h>	Fibonacci series:
3	main()	0 1 1 2 3
4	{	
5	int n, count=2, a=0, b=1, c;	
6	printf("How many terms do you want to print\t");	
7	scanf("%d",&n);	
8	printf("Fibonacci series:\n");	
9	printf("%d\t%d\t",a,b);	
10	while(count<n)	
11	{	
12	c=a+b;	
13	printf("%d\t", c);	
14	a=b;	
15	b=c;	
16	count++;	
17	}	
18	}	

Program 11 | Find sum of all odd numbers that lie between 1 and n

Line	PE 3-11.c	Output window
1	//Sum of odd numbers 1+3+5+7...+n	Enter the value of n 5
2	#include<stdio.h>	Sum of odd numbers from 1 to 5 is 9
3	main()	
4	{	
5	int n, sum=0, i=1;	
6	printf("Enter the value of n\t");	
7	scanf("%d",&n);	
8	while(i<=n)	
9	{	
10	if(i%2==1)	
11	sum=sum+i;	
12	i++;	
13	}	
14	printf("Sum of odd numbers from %d to %d is %d",1,n,sum);	
15	}	

Program 12 | Find the sum of series 1+(1+2)+ (1+2+3) +(1+2+3+4)... n terms

Line	PE 3-12a.c	PE 3-12b.c	Output window PE 3-12a.c
1	//Sum of the given series	//Sum of the given series	Enter the number of terms 3
2	#include<stdio.h>	//Output in a better way	Sum of the series is 10
3	main()	#include<stdio.h>	**Output window PE 3-12b.c**
4	{	main()	
5	int num, i=1, j, sum=0;	{	Enter the number of terms 3
6	printf("Enter the number of terms\t");	int num, i=1, j, sum=0;	(1)+(1+2)+(1+2+3)= 10

(Conta...

```
7   scanf("%d",&num);
8   while(i<=num)
9   {
10      j=1;
11      while(j<=i)
12      {
13          sum=sum+
14          j++;
15      }
16      i++;
17  }
18  printf("Sum of the
19  }
20
21
22
23
24
25
26
27
28
```

Program 13 | Find

Line	PE 3-13.c
1	//Sum of the given
2	#include<stdio.h>
3	main()
4	{
5	int n, i=1, sum=0;
6	printf("Enter the nu
7	scanf("%d",&n);
8	while(i<=n)
9	{
10	sum=sum + i*i;
11	i++;
12	}
13	printf("Sum of serie
14	}

Program 14 | Find t

Line	PE 3-14.c
1	//Sum of the given
2	#include<stdio.h>
3	main()
4	{
5	int n, i=1;
6	float sum=0;
7	printf("Enter the nu
8	scanf("%d",&n);

```
scanf("%d",&num);
while(i<=num)
{
    j=1;
    while(j<=i)
    {
        sum=sum+j;
        j++;
    }
    i++;
}
printf("Sum of the series is %d", sum);
}
```

```
printf("Enter the number of terms\t");
scanf("%d",&num);
while(i<=num)
{
    j=1;
    printf("(");
    while(j<=i)
    {
        printf("%d",j);
        sum=sum+j;
        j++;
        if(j<=i)
            printf("+");
        else
            printf(")");
    }
    if(i<num)
        printf("+");
    i++;
}
printf("= %d", sum);
}
```

Program 13 | Find the sum of series $1^2 + 2^2 + 3^2 + \ldots$ n terms

Line	PE 3-13.c	Output window
	`//Sum of the given series`	Enter the number of terms 5
	`#include<stdio.h>`	Sum of series is 55
	`main()`	
	`{`	
	`int n, i=1, sum=0;`	
	`printf("Enter the number of terms\t");`	
	`scanf("%d",&n);`	
	`while(i<=n)`	
	`{`	
	`sum=sum + i*i;`	
	`i++;`	
	`}`	
	`printf("Sum of series is %d",sum);`	
	`}`	

Program 14 | Find the sum of series 1+1/2+1/3+... n terms

Line	PE 3-14.c	Output window
	`//Sum of the given series`	Enter the number of terms 3
	`#include<stdio.h>`	Sum of series is 1.833333
	`main()`	
	`{`	
	`int n, i=1;`	
	`float sum=0;`	
	`printf("Enter the number of terms\t");`	
	`scanf("%d",&n);`	

(Contd...)

Line	PE 3-14.c	Output window
9	while(i<=n)	
10	{	
11	sum=sum + 1/(float)i;	
12	i++;	
13	}	
14	printf("Sum of series is %f",sum);	
15	}	

Program 15 | Making use of sine series, evaluate the value of sin(x), where x is in radians

According to sine series: $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \frac{x^n}{n!}$

Line	PE 3-15.c	Output window
1	//Evaluate sin(x)	Enter the value of x in radians 3.14
2	#include<stdio.h>	Enter the power of end term 25
3	main()	Sin of 3.14 is 0.001593
4	{	
5	int i=1,n;	
6	float sum, term, x;	
7	printf("Enter the value of x in radians\t");	
8	scanf("%f",&x);	
9	printf("Enter the power of end term\t");	
10	scanf("%d",&n);	
11	sum=0;	
12	term=x;	
13	i=1;	
14	while(i<=n)	
15	{	
16	sum=sum + term;	
17	term=(term*x*x*-1)/((i+1)*(i+2));	
18	i=i+2;	
19	}	
20	printf("Sin of %4.2f is %f",x, sum);	
21	}	

Program 16 | Reverse, add and check for palindrome

Problem statement: Take a number, reverse its digits and add the reverse to the original. If the sum is not a palindrome, repeat the procedure with the sum until the result is a palindrome. Write a program that takes a number and gives the resulting palindrome and the number of additions it took to find it.

Test case:	354	807	1515
	+ 453	+ 708	+ 5151
	807	1515	6666

Result: Palindrome is 6666 and the number of additions to find it is 3.

(Contd...)

Line	PE 3-16.c
1	//Comment: Rever
2	#include<stdio.h>
3	#include<conio.h>
4	main()
5	{
6	int num, temp, rever
7	printf("Enter the nu
8	scanf("%d",&num);
9	while(1)
10	{
11	temp=num;
12	reverse=0;
13	while(temp!=0)
14	{
15	digit=temp%
16	reverse=rev
17	temp=temp/
18	}
19	if(num==reverse
20	{
21	printf("\nPal
22	break;
23	}
24	else
25	{
26	printf(" %d\
27	printf("+ %d"
28	num=num+re
29	printf("-----
30	printf(" %d\
31	printf("-----
32	add++;
33	}
34	}
35	}

...gram 17 | Print ...

...id of digits:

...logic to print the pyra...

PE 3-16.c

```
//Comment: Reverse and Add
#include<stdio.h>
#include<conio.h>
main()
{
int num, temp, reverse=0, add=0, digit;
printf("Enter the number\t");
scanf("%d",&num);
while(1)
{
    temp=num;          //←Save num in temp
    reverse=0;
    while(temp!=0)     //←Find the reverse of temp
    {
        digit=temp%10;
        reverse=reverse*10+digit;
        temp=temp/10;
    }
    if(num==reverse)   //←Is it a palindrome
    {
        printf("\nPalindrome is %d and no. of addition is %d",reverse, add);
        break;
    }
    else               //←If no, repeat the procedure with sum
    {
        printf("  %d\n",num);
        printf("+ %d\n",reverse);
        num=num+reverse;
        printf("----------\n");
        printf("  %d\n",num);
        printf("----------\n");
        add++;         //←Keep track of number of additions performed
    }
}
}
```

Output window

```
Enter the number  354
  354
+ 453
----------
  807
----------
  807
+ 708
----------
 1515
----------
 1515
+ 5151
----------
 6666
----------

Palindrome is 6666 and no. of addition is 3
```

Program 17 | Print pyramid of digits as shown below for n number of lines

Pyramid of digits:

```
          1
        2 3 2
      3 4 5 4 3
    4 5 6 7 6 5 4
  .....................
```

Logic to print the pyramid:

			1			
		2	3	2		
	3	4	5	4	3	
4	5	6	7	6	5	4

(Contd...)

(Contd...)

1. Get the number of rows in the pyramid, let it be n.
2. In each row r (where r is the row number) leave (n-r) spaces blank and then print (2r-1) values. The printing of values starts with the row number. The first $\left\lceil \frac{2r-1}{2} \right\rceil$ values are printed by incrementing the previously printed value. The next $\left\lfloor \frac{2r-1}{2} \right\rfloor$ values are printed by decrementing the previously printed value.

Line	PE 3-17.c
1	//Print pyramid of digits
2	#include<stdio.h>
3	main()
4	{
5	int n, r=1, val, j;
6	printf("Enter the number of rows in the pyramid\t");
7	scanf("%d",&n);
8	while(r<=n) //←Print n rows
9	{
10	val=r; //←Printing starts with row number
11	for(j=1;j<=n-r;j++)
12	printf("\t"); //←Print n-r blank spaces
13	for(j=1;j<=2*r-1;j++)
14	if(j<=(2*r-1)/2) //←Printing left half of the row
15	printf("%d\t",val++);
16	else if(j==(2*r-1)/2+1) //←Printing middle element of row
17	printf("%d\t",val);
18	else //←Printing right half of the row
19	printf("%d\t",--val);
20	printf("\n");
21	r++;
22	}
23	}

Output window

```
Enter the number of rows in the pyramid    4
                1
            2   3   2
        3   4   5   4   3
    4   5   6   7   6   5   4
```

Line	PE 3-18.c
1	//Floyd's triangle
2	#include<stdio.h>
3	main()
4	{i
5	nt n, r=1, val=1, j;
6	printf("Enter the num
7	scanf("%d",&n);
8	while(r<=n) /
9	{
10	for(j=1;j<=r;j++)
11	printf("%d\t"
12	printf("\n");
13	r++;
14	}
15	}

Program 18 | Print Floyd's triangle

Floyd's triangle:

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

Logic to print Floyd's triangle:

1. Get the number of rows in the Floyd's triangle, let it be n.
2. In each row r (where r is the row number), print r values. The printing of values starts with 1. Successive values are printed by incrementing the previously printed values.

(Contd...

Line	PE 3-18.c	Output window
1	//Floyd's triangle	Enter the number of rows in the triangle 4
2	#include<stdio.h>	1
3	main()	2 3
4	{	4 5 6
5	int n,r=1, val=1, j;	7 8 9 10
6	printf("Enter the number of rows in the triangle\t");	
7	scanf("%d",&n);	
8	while(r<=n) //←Print n rows	
9	{	
10	for(j=1;j<=r;j++) //←Printing a row	
11	printf("%d\t",val++); //←Printing values	
12	printf("\n"); //←New-line for next row	
13	r++;	
14	}	
15	}	

Test Yourself

1. Fill in the blanks in each of the following:
   a. The smallest logical entity that can independently exist in a C program is _____.
   b. Statements in C language are terminated with a/an _____.
   c. A compound statement is also known as _____.
   d. The types of labeled statements are _____, _____, _____.
   e. A case label should be a compile time constant expression of _____ type.
   f. The form of looping in which the number of iterations to be performed is known in advance is called _____.
   g. The execution or termination of a sentinel-controlled loop depends upon a special value known as _____.
   h. Sentinel-controlled loop is also known as _____.
   i. The statements for which no machine code is generated are called _____.
   j. To alter the default flow of control, _____ statements are used.
   k. _____ statement is used to terminate the current iteration of the enclosing loop.
   l. An expression terminated with a semicolon is known as _____ statement.
   m. _____ is an exit-controlled loop.
   n. The _____ statement when executed in a switch statement causes immediate exit from it.
   o. Careless use of nested if-else statement may lead to _____ problem.
2. State whether each of the following is true or false. If false, explain why.
   a. Only non-executable statements can appear outside the body of a function.
   b. Null statement performs no operation.
   c. An empty compound statement is equivalent to a null statement.
   d. An entry-controlled loop is executed at least once.
   e. Identifier-labeled statement is a branching statement and alters the flow of control.
   f. A continue statement can appear inside, or as a body of switch statement or a loop.
   g. Case-labeled statements can appear only inside the body of a switch statement.
   h. A break statement is used to terminate the current iteration of the loop.
   i. A switch selection expression can be of any type.
   j. In an entry-controlled loop, if the body of the loop is executed n times the expression in the condition section is evaluated n+1 times.
3. Write a simple C statement to accomplish each of the following:
   a. Test if the value of the variable count is greater than 10. If so, print "Count is greater than 10".
   b. Assign the value 10 to the variables a, b and c.
   c. Increment the value of variable var by 10 and then assign it to variable stud.
   d. Test if the least significant bit of the variable num is 1. If so, assign 10 to variable a else assign 20 to it.
   e. Find factorial of a number n and assign it to variable fact.
4. Programming exercise:
   a. Write a C program that prints the integers between 1 and n which are divisible by 7. Get the value of n from the user.
   b. Write a C program that prints the integers from 1 to n omitting those integers which are divisible by 7. Get the value of n from the user.
   c. Write a C program that prints the integers between 1 and n which are divisible by 3, but not divisible by 4.

d. Write a C program to find the sum of all integers that lie between 1 and n and are divisible by 7.
e. Write a C program to evaluate 1×2×3×4×...n. Get the value of n from the user.
f. Write a C program to print first n Armstrong numbers. Get the value of n from the user.
g. Write a C program to print first n prime numbers. Get the value of n from the user.
h. Write a C program to evaluate the following series (Get the value x and n from the user):

i. $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\infty$

ii. $\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots\infty$

iii. $\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} \cdots\infty$

iv. $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} \ldots + \frac{1}{n!}$

v. $\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} \ldots\infty$

i. Write a C program to generate the following patterns (get the number of rows in the pattern from the user):

1.
```
1 2 3 4 5
  1 2 3
    1
```

2.
```
    1
  1 2 3
1 2 3 4 5
  1 2 3
    1
```

4

ARRAYS AND POINTERS

Learning Objectives

In this chapter, you will learn about:

- The limitation of basic data types
- Derived data types: array type and pointer type
- Arrays
- Single-dimensional and multi-dimensional arrays
- Declaration and usage of arrays
- Memory representation of arrays
- Different ways of storing multi-dimensional arrays
- Pointers
- Operations allowed on pointers
- Pointer arithmetic
- void pointer and null pointer
- Relationship between arrays and pointers
- Arrays of pointers
- Pointer to a pointer
- Pointer to an array
- Advantages and limitations of arrays

4.1 Introduction

So far you have learnt about the basic data types, expressions and statements. In the previous chapter, you have learnt the use of iteration statements to perform repetitive tasks like summing first n natural numbers, etc. Consider a problem to find the average of marks secured by five students in a course. A piece of code written for it is given in Program 4-1.

Line	Prog 4-1.c	Output window
1	//Average of marks secured by students	Average marks secured is 11.800000
2	#include<stdio.h>	
3	main()	
4	{	
5	int marks1=10, marks2=12, marks3=9, marks4=11, marks5=17;	
6	int sum; float average;	
7	sum=marks1+marks2+marks3+marks4+marks5;	
8	average=sum/5.0;	
9	printf("Average marks secured is %f ",average);	
10	}	

Program 4-1 | A program to find average marks secured by students

The powerful iteration statements discussed in Chapter 3 have not been used here to sum up the marks secured by the students because the marks are stored in separate variables and it is not possible to access them in a generalized way. Since there are only five students, it is possible to find the average in the above-mentioned manner. Now suppose there are 200 students in a course. For a problem of this scale, it is not feasible to create separate variables for storing the marks and finding the average in the above-mentioned manner. To solve such problems a method is required that helps in storing and accessing data in a generalized and an efficient manner. The C language provides this method in the form of a derived data type known as **array type** or just **array**.

Consider another real-time problem that requires storing and processing names like "Sam" entered by the user. There is no basic data type available in C that provides this flexibility. A variable of char type can be used to store only one character but cannot be used to store all the three characters of the name "Sam". The derived array type provides a solution to this problem. An array enables the user to store the characters of the entered name in a contiguous set of memory locations, all of which can be accessed by only one name, i.e. the array name.

The array type has a close relationship with another derived data type, known as the **pointer type** or just **pointer**. Their relationship is so intimate that they cannot be studied in isolation. In this chapter, I will describe both arrays and pointers. Finally, we will look at the operations that can be applied on them and how to use them to solve problems.

4.2 Arrays

An **array** is a data structure that is used for the storage of homogeneous data, i.e. data of the same type. Figure 4.1 depicts arrays of four different types.

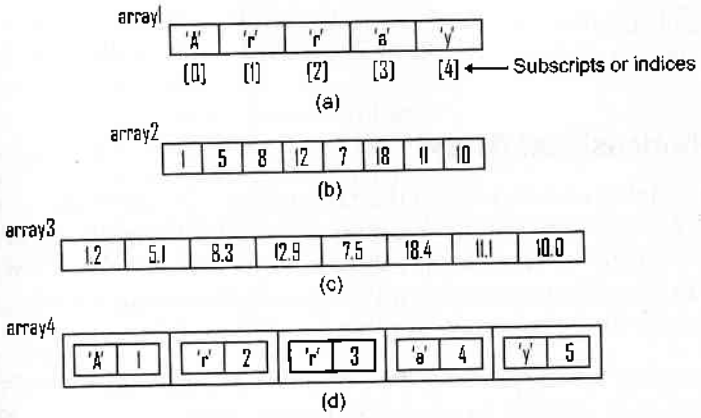


Figure 4.1 | (a) Character array; (b) integer array; (c) float array; (d) array of user-defined type

The important points about arrays are as follows:

1. An **array** is a collection of elements of the same data type. The data type of an element is called **element type**. For example, in Figure 4.1, the element type of array1 is char, array2 is int, array3 is float and array4 is user-defined type.
2. The individual elements of an array are not named. All the elements of an array share a common name, i.e. the **array name**. For example, in Figure 4.1 (a), all the elements of array, i.e. 'A', 'r', 'r', 'a' and 'y' have a common name, i.e. array1.
3. The individual elements of an array are distinguished and are referred to or accessed according to their positions in an array. The position of an element in an array is specified with an integer value known as **index** or **subscript**. Because arrays use indices or subscripts to access their elements, they are also known as **indexed variables** or **subscripted variables**.
4. The array index in C starts with 0, i.e. index of the first element of an array is 0.
5. The memory space required by an array can be computed as (**size of element type**) × (**Number of elements in an array**). For example, in Figure 4.1, array1 takes 1×5, i.e. 5 bytes in the memory, array2 takes 16 bytes (if an integer occupies 2 bytes), array3 takes 32 bytes and array4 takes 15 bytes (if an integer takes 2 bytes) in the memory.
6. Arrays are always stored in contiguous (i.e. continuous) memory locations. For example, in Figure 4.1, if the first element of array1 is stored at memory location 2000, then the successive elements of the array will be stored at the memory locations 2001, 2002, 2003 and 2004. In case of array2, if the first element is stored at memory locations 2000-2001, the next elements will be stored at the memory locations 2002-2003, 2004-2005, and so on.

Data structure is a logical representation of data. It provides systematic mechanisms for storage, retrieval and manipulation of data. Examples of data structures are: **arrays**, stacks, queues, linked lists, trees, etc.

Forward Reference: User-defined data types (Chapter 9).

In general, arrays are classified as:

1. Single-dimensional arrays
2. Multi-dimensional arrays

4.3 Single-dimensional Arrays

A **single-dimensional** or **one-dimensional array** consists of a fixed number of elements of the same data type organized as a simple linear sequence. The elements of a single-dimensional array can be accessed by using a single subscript, thus they are also known as **single-subscripted variables**. The other common names of single-dimensional arrays are **linear arrays** and **vectors**. Single-dimensional arrays are shown in Figure 4.2.

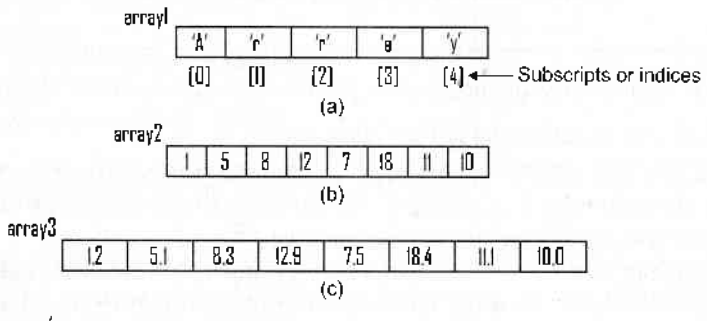


Figure 4.2 | Single-dimensional arrays

There are two aspects of working with arrays:

1. Declaration (i.e. creation) of array
2. Usage (i.e. storing or referring elements) of array

4.3.1 Declaration of a Single-dimensional Array

The general form of a single-dimensional array declaration is:

<storage_class_specifier><type_qualifier><type_mod>**type_specifier identifier[**<size_specifier>**]**<=initialization_list<...>>

Forward Reference: Storage class specifier (Chapter 7).

The important points about a single-dimensional array declaration are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in bold are the mandatory parts of a single-dimensional array declaration.

2. A single-dimensional array declaration consists of a type specifier (i.e. **element type**), an identifier (i.e. **name of array**) and a size specifier (i.e. **number of elements** in the array) enclosed within square brackets (i.e. []). The following declarations of single-dimensional arrays are valid:

```
int array1[8];    //←array1 is an array of 8 integers          (Integer array)
float array2[5];  //←array2 is an array of 5 floating point numbers (Floating point array)
char array3[6];   //←array3 is an array of 6 characters         (Character array)
```

3. The size specifier specifies the number of elements in an array. The syntactic rules about the size specifier are as follows:

   a. It should be a compile time constant expression of integral type.

   Reasons:

   i. The memory space to an array is allocated at the compile time. The memory requirement of an array depends upon its element type and the number of elements (i.e. size) in it. Hence, the size of an array must be known at the compile time so that memory can be allocated to it.

   ii. The size of an array cannot be expanded or squeezed at the run-time. Thus, size must be a constant expression so that it cannot be changed at the run-time.

   The following declarations of single-dimensional arrays are valid:

```
int array1[3+5];      //←3+5 is a compile time constant expression of int type
float array2[size];   //←where size is a qualified constant of integral type
char array3[size];    //←where size is a symbolic constant of integral type
```

   The following declarations of single-dimensional arrays are not valid:

```
int array1[j];        //← j is a variable and not a constant
int array2[3.5];      //←It is not possible to create an array of 3.5 locations
```

   b. It should be greater than or equal to one.

   Reason: It is not possible to create an array of size zero, i.e. having no element.

   It is allowed to create an array of size 1, i.e. having only one element. Array of size 1 is like a simple variable and does not provide any significant advantage.

   The following declarations of single-dimensional arrays are not valid:

```
int array1[-1];       //← It is not possible to create an array of -1 locations
char array2[0];       //← It is not possible to create an array of 0 locations
```

   c. The size specifier is mandatory if an array is not explicitly initialized, i.e. if an initialization list is not present.

   Reason: If an initialization list is present, it is possible to determine the size of array from the number of initializers in the initialization list. In that case, the size specification becomes optional.

   The following declaration of a single-dimensional array is not valid:

```
int array1[];         //←Here, it is not possible to determine the size of array
                      //← Hence, the amount of memory to be allocated cannot
                      //    be determined
```

4. **Initializing elements of a single-dimensional array:** Like variables can be initialized, similarly the elements of an array can also be initialized. The syntactic rules about the initialization of array elements are as follows:
   a. The elements of an array can be initialized by using an **initialization list.** An initialization list is a comma-separated list of initializers enclosed within braces.
   b. An **initializer** is an expression that determines the initial value of an element of the array.
   c. If the type of initializers is not the same as the element type of an array, implicit type casting will be done, if the types are compatible. If types are not compatible, there will be a compilation error. The code segment in Program 4-2 illustrates this fact.

Line	Prog 4-2.c	Output window
1	//Initializers of compatible but different types	Elements of arrays are initialized with
2	#include<stdio.h>	arr1: 2 4 6
3	main()	arr2: 65.000000 66.000000 67.000000
4	{	**Remarks:**
5	int arr1[]={2.3, 4.5, 6.9};	• The element types of the arrays are different from the types of initializers but the types are compatible
6	float arr2[]={'A','B','C'};	• float initializers are demoted and then elements of arr1 are initialized
7	printf("Elements of arrays are initialized with\n");	• char initializers are promoted before initializing the elements of arr2. ASCII values of characters are used
8	printf("arr1: %d %d %d\n",arr1[0],arr1[1],arr1[2]);	
9	printf("arr2: %f %f %f\n",arr2[0],arr2[1],arr2[2]);	
10	}	

Program 4-2 | A program to illustrate that the initializer's type can be different from the element type of an array

   d. The number of initializers in the initialization list should be less than or at most equal to the value of size specifier, if it is present.

      The following declarations of single-dimensional arrays are valid:

```
int array1[]={1,2,3,4,5};  //←Initialization list {1,2,3,4,5} present
int array2[]={2+3,a+5}; //←Initializers are 2+3 and a+5, where a is an int variable
char array3[6]={'A','r','r','a','y'};   //←Number of initializers is less than the value of
                                        //   size specifier
```

      The following declaration of a single-dimensional array is not valid:

```
int array1[2]={1,2,3,4,5}; //← Number of initializers cannot be more than the
                           //   value of size specifier
```

   e. If the number of initializers in the initialization list is less than the value of the size specifier, the leading array locations (i.e. occurring first) equal to the number of initializers get initialized with the values of initializers. The rest of the array locations get initialized to 0 (if it is an integer array), 0.0 (if case of floating point array) and '\0' (i.e. null character, if it is an array of character type). The above-mentioned fact is shown in Figure 4.3.

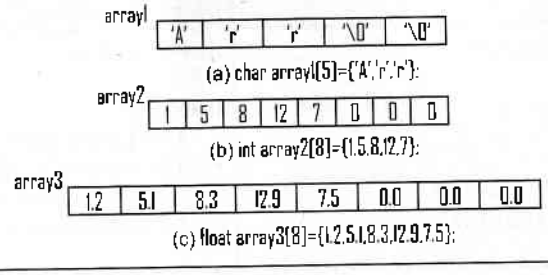


Figure 4.3 | Contents of arrays if the number of initializers is less than their size

4.3.2 Usage of Single-dimensional Array

The elements of a single-dimensional array can be accessed by using a subscript operator (i.e. [] and a subscript. The important points about the usage of single-dimensional arrays are as follows:

1. For accessing the elements of a one-dimensional array, the general form of expression is E1[E2], where E1 and E2 are sub-expressions and [] is the subscript operator. One of the sub-expressions E1 or E2 must be of an array type[‡] or a pointer type[†] and the other sub-expression must be of an integral type.
2. The sub-expression of the integral type (i.e. the subscript) must evaluate to a value greater than or equal to 0.
3. The array subscript in C starts with 0, i.e. the subscript of the first element of an array is 0. Thus, if the size of an array is n, the valid subscripts are from 0 to n-1. However, if the array index greater than n-1 is used while accessing an element of the array, there will be no compilation error. This is due to the fact that C language does not provide compile time or run-time **array index out-of-bound check**. However, using an out-of-bound index[➲] may lead to run-time error or exceptions. Thus, care must be taken to ensure that the array indices are within bounds, i.e. from 0 to n-1.

An **array type** is one of the derived data types. It is said to be derived from an element type and if the element type is T (where T is a generic term and can be int, float, char or any other type), the array type is called '**array of Ts**'. The construction of an array type from an element type is called '**array type derivation**'. Consider the declaration statement **int array[5];** the array type derived from an element type **int** is **int[5]**.

Forward Reference: Refer Question numbers 17 and 42 and their answers.

The code snippet in Program 4-3 illustrates the use of a singe-dimensional array.

[‡] See Section 4.4 for a description on pointer type.

Line	Prog 4-3.c	Output window
1	//Use of single-dimensional array	Element of array are:
2	#include<stdio.h>	10 20 30
3	main()	**Remarks:**
4	{	• a is of array type.
5	int a[3]={10,20,30};	• The expression a[0] refers to the first element, a[1] refers to the second element and a[2] refers to the third element of the array
6	printf("Elements of array are:\n");	
7	printf("%d %d %d",a[0],a[1],a[2]);	
8	}	

Program 4-3 | A program to illustrate the use of subscript operator

4.3.2.1 Reading, Storing and Accessing Elements of a One-dimensional Array

An iteration statement (i.e. loop) is used for storing and reading the elements of a one-dimensional array. The code snippet in Program 4-4 illustrates a method to read, store and access the elements of a single-dimensional array.

Line	Prog 4-4.c	Output window
1	//Use of single-dimensional array	Enter the number of students in class 5
2	#include<stdio.h>	Enter marks of students
3	main()	
4	{	Enter marks of student 1 10
5	int marks[200], lc, studs, sum=0;	Enter marks of student 2 12
6	float average;	Enter marks of student 3 9
7	printf("Enter the number of students in class\t");	Enter marks of student 4 11
8	scanf("%d",&studs);	Enter marks of student 5 17
9	printf("Enter marks of students\n\n");	
10	for(lc=0;lc<studs;lc++)	Average marks of the class is 11.800000
11	{	**Remarks:**
12	printf("Enter marks of student %d\t",lc+1);	• The marks of 200 students can be stored in an array named marks. The elements of the array can be accessed in general way by writing marks[lc], where lc ∈ {0,....199}
13	//Reading and storing elements in a 1-D array	• Although at the runtime marks of only 5 students are entered, the size of array is kept 200 to accommodate the worst case (i.e. 200 students)
14	scanf("%d",&marks[lc]);	• 195 locations are not used. Hence, 195*2=390 bytes of memory got wasted
15	}	• In line number 19, integer variable sum is explicitly type casted to float
16	for(lc=0;lc<studs;lc++)	
17	//Accessing elements stored in the 1-D array	
18	sum=sum+marks[lc];	
19	average=(float)sum/studs;	
20	printf("\nAverage marks of the class is %f",average);	
21	}	

Program 4-4 | A scalable version of Program 4-1

4.3.3 Memory Representation of Single-dimensional Array

The elements of an array are **stored in contiguous** (i.e. continuous) memory locations. This is depicted in Figure 4.4.

The mentioned addresses refer to the starting addresses of the elements. The first element in Figure 4.4(c) occupies the memory locations 2000-2003.

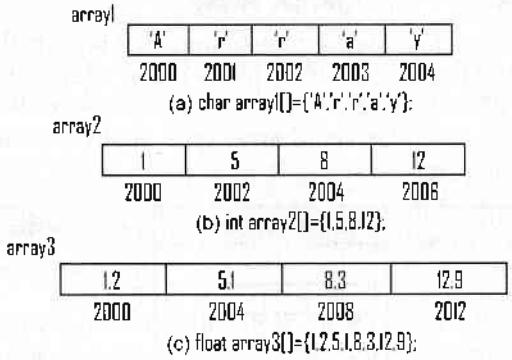


Figure 4.4 | Elements of the array are stored in contiguous memory locations

4.3.4 Operations on a Single-dimensional Array

4.3.4.1 Subscripting a Single-dimensional Array

The only operation allowed on arrays is **subscripting**. Subscripting is an operation that selects an element from an array. To perform subscripting in C language, a subscript operator (i.e. []) is used. The rules for subscripting have already been discussed in Section 4.3.2.

4.3.4.2 Assigning an Array to Another Array

A variable can be assigned to or initialized with another variable but an array cannot be assigned to or initialized with another array. The following statement is not valid and leads to a compilation error:

array1=array2; //←where array1 and array2 are arrays of the same type and size

Reason: In C language, the name of the array refers to the address of the first element of the array and is a constant object. It does not have a modifiable l-value. Since it does not have a modifiable l-value, it cannot be placed on the left side of the assignment operator.

To assign an array to another array, each element must be assigned individually. The code segment in Program 4-5 illustrates the mentioned fact.

Line	Prog 4-5.c	Output window
1	//Assignment of an array to another array	Compilation error "L-value required in function main"
2	#include<stdio.h>	**Reasons:**
3	main()	• The name of the array a refers to the address of the first element of the array and is a constant object
4	{	• It does not refer to a modifiable l-value
5	int a[3], b[3]={10,20,30};	• Hence, it cannot be placed on the left side of the assignment operator
6	printf("Assigning an array to an array:\n");	**What to do?**
7	a=b;	• Making use of a loop, assign individual elements of array b to the elements of array a by writing a[i]=b[i], where i={0,1,2}
8	printf("Elements of array a are:\n");	
9	printf("%d %d %d",a[0],a[1],a[2]);	
10	}	

Program 4-5 | A program to illustrate that an array cannot be assigned to another array in one step

4.3.4.3 Equating an Array with Another Array

When the operands of an equality operator are of the array type, it always evaluates to false. **Reason:** In C language, the name of an array refers to the address of the first element of the array and the addresses of first elements of two arrays can never be the same. Hence, when the operands of an equality operator are of array type, it always evaluates to false. Program 4-6 illustrates the mentioned fact.

Line	Prog 4-6.c
1	//Equality operator & arrays
2	#include<stdio.h>
3	main()
4	{
5	int a[3]={10,20,30}, b[3]={10,20,30};
6	if(a==b)
7	printf("Arrays are equal");
8	else
9	printf("Arrays are not equal");
10	}

Memory contents

a

10	20	30
2000	2002	2004

b

10	20	30
4000	4002	4004

Output window

Arrays are not equal

Reasons:

- The name of arrays a and b refers to the addresses of their first elements i.e. 2000 and 4000, respectively
- Since the addresses are different, the equality operator evaluates to false although the contents of the arrays are the same

What to do?

- For checking equality, check the equality of all individual elements

Program 4-6 | A program to illustrate the behavior of equality operator on arrays

To check whether the contents of two arrays are the same or not, check the equality of each individual element.

Programs 4-5 and 4-6 illustrate that the name of an array refers to the address of the first element of the array. An expression of an array type (e.g. the name of array) is automatically converted to an expression of pointer type. This automatic conversion makes the simultaneous discussion of arrays and pointers essential.

4.4 Pointers

A **pointer** is a variable that holds the address of a variable or a function. A pointer is a powerful feature that adds enormous power and flexibility to C language. A pointer variable can be declared as:

```
[storage_class_specifier][type_qualifier][type_modifier]type_specifier* identifier[=l-value[....]];
```

The important points about pointers are as follows:

1. The terms enclosed within square brackets (i.e. []) are optional and might not be present in a declaration statement. The terms shown in **bold** are the mandatory parts of a pointer variable declaration.
2. A pointer variable declaration consists of a type specifier (i.e. **referenced type**), **punctuator** * and an identifier (i.e. name of pointer variable). The following declarations are valid:

```
int *iptr;          //←iptr is pointer to an integer
float *fptr;        //←fptr is pointer to a float
char *cptr;         //←cptr is pointer to a character
```

```
const int *ptric;        //←ptric is pointer to an integer constant or constant integer
unsigned int *ptrui;     //←ptrui is pointer to an unsigned integer
```

3. Pointer variable declarations are read from the right side. The punctuator * is read as **'pointer to'**. So the declaration statement int *iptr; is read as 'iptr is a pointer to an integer'.

> The concept of pointer declaration is scalable. It is possible to declare a pointer to a variable, which itself is a pointer variable. Such a pointer is known as a **pointer to a pointer.**[1] The declaration statement **int **pptr;** declares a pointer to a pointer and is read as **'pptr is a pointer to a pointer to an integer'**.

4. A pointer variable can hold the address of a variable or a function. In Figure 4.5(a) iptr is an integer pointer and holds the address of an integer variable a. In Figure 4.5(c) pptr is pointer to pointer to an integer and holds the address of an integer pointer iptr, which in turn holds the address of an integer variable val.

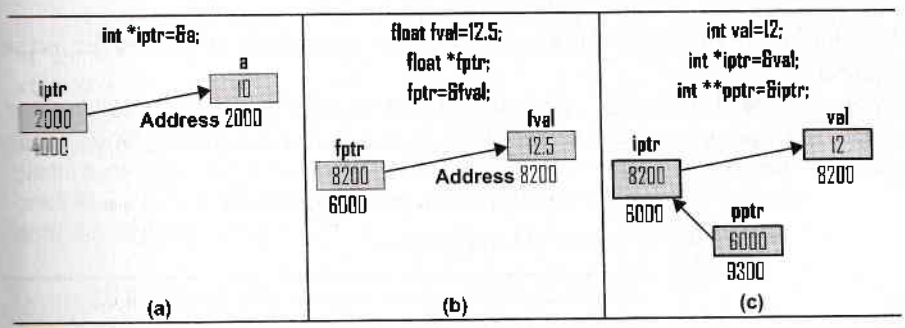


Figure 4.5 | Pointers holding addresses

5. Every pointer variable takes the same amount of memory space irrespective of whether it is a pointer to int, float, char or any other type. This fact is illustrated in the code segment given in Program 4-7.

Prog 4-7.c	Output window

`//Size of pointer variables`
`#include<stdio.h>`
`main()`
`{`
`  char *cptr;`
`  int *iptr;`
`  float *fptr;`
`  printf("Pointer to character takes %d bytes\n",sizeof(cptr));`
`  printf("Pointer to integer takes %d bytes\n",sizeof(iptr));`
`  printf("Pointer to float takes %d bytes\n",sizeof(fptr));`
`}`
	Pointer to character takes 2 bytes
	Pointer to integer takes 2 bytes
	Pointer to float takes 2 bytes
	Remarks:
	• The above output is the result of execution using Borland Turbo C 3.0 IDE
	• In Borland Turbo C 4.5 or MS-VC++ 6.0 each type of pointer variable takes 4 bytes

Program 4-7 | A program to illustrate that a pointer to any type takes the same amount of memory space

[1] See Section 4.6.3 for a description on the pointer to a pointer.

6. The value of a pointer variable is printed with %p format specifier. Since the pointer variables hold addresses, which are unsigned integers, %u format specifier can also be used for printing pointer values. However, the use of %p format specifier is recommended over the use of %u format specifier.

Forward Reference: Pointers to functions (Chapter 5).

4.4.1 Operations on Pointers

The operations allowed on pointers are as follows:

4.4.1.1 Referencing Operation

In **referencing operation**, a pointer variable is made to refer to an object. The reference to an object can be created with the help of a reference operator (i.e. &). The important points about the reference operator are as follows:

1. The reference operator, i.e. & is a unary operator and should appear on the left side of its operand.
2. The operand of the reference operator should be a variable of arithmetic type[✍] or pointer type.[✍] The operand of the reference operator can also be a function designator,[➲] i.e. name of a function.
3. The reference operator is also known as **address-of operator**.

The above-mentioned points are depicted in Figure 4.6.

```
float fval=12.5;        //← fval is a floating point variable initialized with 12.5
float *fptr;            //← fptr is a pointer to float type
fptr=&fval;             //← The address-of fval is assigned to fptr. fval is known as
                        //   referenced object and fptr is known as referencing
                        //   object and references fval
```

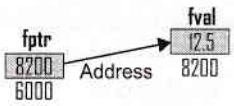


Figure 4.6 | A float pointer referencing a float variable

Integral and floating types are collectively called **arithmetic types**. A **pointer type** describes an object, whose value provides reference to an object of type T. T is a generic term and will be known as **reference type**. It can be int, float, char or any other type. A pointer type derived from the reference type T is called **'pointer to T'**. The construction of a pointer type is called **'pointer-type derivation'**.

Forward Reference: Function designator, pointer to a function (Chapter 5).

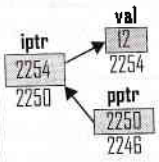
4.4.1.2 Dereferencing a Pointer

The object pointed to or referenced by a pointer can be indirectly accessed by dereferencing the pointer. A dereferencing operation allows a pointer to be followed to the data object

which it points. A pointer can be dereferenced by using a dereference operator (i.e. *). The important points about the dereference operator are as follows:

1. The dereference operator (i.e. *) is a unary operator and should appear on the left side of its operand.
2. The operand of a dereference operator should be of pointer type.
3. The dereference operator is also known as **indirection operator** or **value-at operator**.

The code snippet in Program 4-8 illustrates the use of a dereference operator.

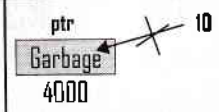
Line	Prog 4-8.c	Memory	Output window
1	//Dereferencing pointers		Value is 12 Value by dereferencing iptr is 12 Value by dereferencing pptr is 12 Value of iptr is 2407:2254 Value of pptr is 2407:2250 **Remarks:** • The printed addresses are in the form of **segment address: offset address** • The segment address and the offset address are in the **hexadecimal number system** • If the memory is assumed to be analogous to a city, the segment address is analogous to a sector number and the offset address is analogous to a house number • The addresses that you get in the output may be different from the mentioned addresses as the memory allocation is purely random • val=12, iptr=2254 and ptr=2250 • *iptr=value-at(iptr)=value-at(2254)=12 • **pptr=value-at(value-at(pptr))=value-at(value-at(2250))=value-at(2254)=12
2	#include<stdio.h>		
3	main()		
4	{		
5	int val=12;		
6	int *iptr=&val;		
7	int **pptr=&iptr;		
8	printf("Value is %d\n",val);		
9	printf("Value by dereferencing iptr is %d\n",*iptr);		
10	printf("Value by dereferencing pptr is %d\n",**pptr);		
11	printf("Value of iptr is %p\n",iptr);		
12	printf("value of pptr is %p\n",pptr);		
13	}		

Program 4-8 | A program to illustrate the dereferencing operation

4.4.1.3 Assigning to a Pointer

1. A pointer can be assigned or initialized with the address of an object. A pointer variable cannot hold a non-address value and thus can only be assigned or initialized with l-values. Program 4-9 illustrates this fact.

Line	Prog 4-9.c	Memory contents	Output window

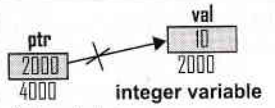
1
2
3
4
5
6
7
8
9
	// Invalid assignment to pointer variable
	#include<stdio.h>
	main()
	{
	int val=10;
	int *ptr=val;
	printf("Value of variable is %d\n",val);
	printf("Pointer holds %p\n", ptr);
	}
			Compilation error "Cannot convert int to int*"
			Reasons:
			• Pointer variables can only hold addresses
			• A pointer variable ptr cannot hold an integer value val
			What to do?
			• Initialize ptr with the address of variable val by writing &val and re-execute the code

Program 4-9 | A program to illustrate that a pointer variable cannot hold a non-address value

There is an exception to this rule. The constant zero can be assigned to a pointer. For example, int *iptr=0; is valid. Assignment or initialization with zero makes the pointer a special pointer known as the **null pointer**.§

2. A pointer to a type cannot be initialized or assigned the address of an object of another type. Program 4-10 illustrates this fact.

Line	Prog 4-10.c	Memory contents	Output window

1
2
3
4
5
6
7
8
9
	// Invalid assignment to pointer variable
	#include<stdio.h>
	main()
	{
	int val=10;
	float *ptr=&val;
	printf("Value of variable is %d\n",val);
	printf("Pointer holds %p\n", ptr);
	}
		
		float pointer
		A float pointer cannot point to an integer variable
			Compilation error "Cannot convert int* to float*"
			Reasons:
			• A pointer variable can only be assigned address of an object of the same type
			• A pointer variable ptr (of type float*) cannot hold the address of an integer variable (i.e. int*)
			What can be done?
			• Explicitly type cast int* to float* by using type cast operator. Write float* ptr=(float*)&val; and then re-execute the code
			Remark:
			• Explicit type casting of pointers may give unexpected results and is not recommended

Program 4-10 | A program to illustrate that a pointer to a type cannot be assigned address of an object of another type

§ Refer Section 4.4.3 for a description on null pointer.

3. A pointer can be assigned or initialized with another pointer of the same type. However, it is not possible to assign a pointer of one type to a pointer of another type without explicit type casting.

There is an exception to Rules 2 and 3. A pointer to any type of object can be assigned to a pointer of type void*[1] but vice-versa is not true. A **void pointer** cannot be assigned to a pointer to a type without explicit type casting.

4.4.1.4 Arithmetic Operations (Pointer Arithmetic)

Arithmetic operations can be applied to pointers in a restricted form. When arithmetic operations are applied on pointers, the outcome of the operation is governed by **pointer arithmetic**. The pointer arithmetic rules are mentioned below.

4.4.1.4.1 Addition Operation

1. An expression of integer type can be added to an expression of pointer type. The result of such operation would have the same type as that of pointer type operand. If ptr is a pointer to an object, then 'adding 1 to pointer' (i.e. ptr+1) points to the next object. Similarly, ptr+i would point to the i^{th} object beyond the one the ptr currently points to. This is shown in Table 4.1.

Table 4.1 | Addition operation on pointers

S. No	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial value	Final value	How to determine?
1.	Addition operator (+)	Pointer to type T	int	Pointer to type T				Result = initial value of pointer + integer operand*sizeof (the reference type T)
	Example1:	**float***	**int**	**float***	**ptr=ptr+1**	**ptr=2000**	**2004**	**2000+1*(4)=2004 as sizeof(float)=4**
	Example2:	**int***	**int**	**int***	**ptr=ptr+5**	**ptr=2000**	**2010**	**2000+5*(2)=2010, if sizeof(int)=2**
2.	Addition operator (+)	Pointer	Pointer	Not allowed				

2. Addition of two pointers is not allowed.
3. The addition of a pointer and an integer is commutative, i.e. ptr+1 is same as 1+ptr.

4.4.1.4.2 Increment Operation

The increment operator can be applied to an operand of pointer type. Table 4.2 depicts the application of an increment operator to an operand of a pointer type.

[1] Refer Section 4.4.2 for a description on void pointer.

Table 4.2 | Increment operation on a pointer

S. No	Operator	Type of operand	Resultant type	Example	Initial values	Final values	How to determine?
1.	Increment operator (++)	Pointer to type T	Pointer to type T				**Post-increment:** Result=initial value of pointer **Pre-increment:** Result = initial value of pointer + sizeof (the reference type T) **In both the cases:** Value of pointer=Value of pointer + sizeof (the reference type T)
Example1:	**Post-increment**	**float***	**float***	**ftr=ptr++**	**ftr=? ptr=2000**	**ftr=2000 ptr=2004**	
Example2:	**Pre-increment**	**float***	**float***	**ftr=++ptr**	**ftr=? ptr=2000**	**ftr=2004 ptr=2004**	

4.4.1.4.3 Subtraction Operation

1. A pointer and an integer can be subtracted. The operation along with examples is shown in Table 4.3.

Table 4.3 | Subtraction operation on pointers

S. No	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial value(s)	Final value	How to determine?
1.	Subtraction operator (-)	Pointer to type T	int	Pointer to type T				Result = initial value of pointer - integer operand *sizeof (the reference type T)
	Example1:	**float***	**int**	**float***	**ptr=ptr-1**	**ptr=2000**	**1996**	**2000-1*(4)=1996 as sizeof(float)=4**
	Example2:	**int***	**int**	**int***	**ptr=ptr-5**	**ptr=2000**	**1990**	**2000-5*(2)=1990, as sizeof(int)=2**
2.	Subtraction operator (-)	Pointer to type T	Pointer to type T	int				Result=(operand1-operand2)/ sizeof (the reference type T)
	Example3:	**float***	**float***	**int**	**a=p2-p1**	**p1=2000 p2=2008**	**2**	**(2008-2000)/ sizeof(float)= (2008-2000)/4=2**

2. Subtraction of integer and pointer is not commutative, i.e. ptr-I is not the same as I-ptr. The operation I-ptr is illegal.
3. Two pointers can also be subtracted. Pointer subtraction is meaningful only if both the pointers point to the elements of the same array. The result of the operation is the difference in subscripts of two array elements. The mentioned rule is described in Table 4.3 and is depicted in Figure 4.7.

Figure 4.7 | Pointer

4.4.1.4.4 Decre

The decrement ope
application of a dec

Table 4.4 | Decreme

S.No	Operator
1.	Decremen operator (--)
Exam-ple1:	**Post-decrement**
Exam-ple2:	**Pre-decrement**

4.4.1.5 Relationa

A pointer can be co
pointers is meaningf
picts the comparison

Table 4.5 | Relational

S.No	Operator
1.	Comparison operators (==, !=, <, <=, >, >=
	Example1:
	Example2:
	Example3:

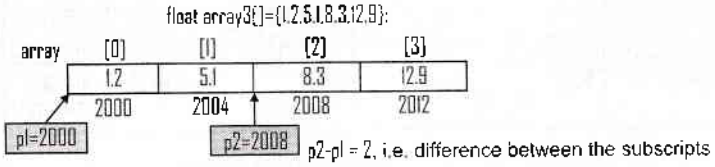


Figure 4.7 | Pointer subtracted from a pointer

4.4.1.4 Decrement Operation

The decrement operator can be applied to an operand of pointer type. Table 4.4 depicts the application of a decrement operator to an operand of pointer type.

Table 4.4 | Decrement operation on a pointer

Example	Operator	Type of operand	Resultant type	Example	Initial values	Final values	How to determine?
	Decrement operator (--)	Pointer to type T	Pointer to type T				**Post- decrement:** Result=initial value of pointer **Pre-decrement:** Result = initial value of pointer - sizeof (the reference type T) **In both the cases:** Value of pointer=Value of pointer - sizeof (the reference type T)
Example1:	Post-decrement	float*	float*	ftr=ptr--	ftr=? ptr=2000	ftr=2000 ptr=1996	
Example2:	Pre-decrement	float*	float*	ftr=--ptr	ftr=? ptr=2000	ftr=1996 ptr=1996	

4.4.1.5 Relational (Comparison) Operations

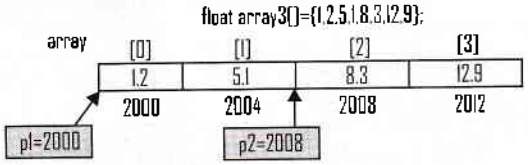
A pointer can be compared with a pointer of the same type or with zero. A comparison of pointers is meaningful only when they point to the elements of the same array. Table 4.5 depicts the comparison of pointers.

Table 4.5 | Relational operations on pointers

Example	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial values	Final value	How to determine?
	Comparison operators (==, !=, <, <=, >, >=)	Pointer to type T	Pointer to type T	int (0 i.e. false or 1 i.e. true)				
	Example1:	float*	float*	int	r=p1!=p2	p1=2000 p2=2008	1	
	Example2:	float*	float*	int	r=p1<p2	p1=2000 p2=2008	1	
	Example3:	float*	float*	int	r=p2>=p1	p1=2000 p2=2008	1	

(Contd...)

S.No	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial values	Final value	How to determine?
	Example4:	float*	float*	int	r=p2==p1	p1=2000 p2=2008	0	



4.4.1.6 Illegal Pointer Operations

The following operations on pointers are not allowed:

1. Addition of two pointers is not allowed.
2. Only integers can be added to pointers. It is not valid to add a float or a double value to a pointer.
3. Multiplication and division operators cannot be applied on pointers.
4. Bitwise operators cannot be applied on pointers.
5. A pointer of one type cannot be assigned to a pointer of another type (except void*) without explicit type casting.
6. A pointer variable cannot be assigned a non-address value (except zero).

The pointer arithmetic discussed above is not applicable to void pointers. However, what actually are void pointers?

4.4.2 void pointer

void is one of the basic data types available in C language. void means nothing or not known. It is not possible to create an object of type void. For example, the following declaration statement is not valid and leads to 'Size of var unknown or zero' compilation error.

```
void var;
```

Although an object of type void cannot be created, it is possible to create a pointer to void. Such a pointer is known as a **void pointer** and has type void*. **void pointer** is a generic pointer and can point to any type of object. Figure 4.8 depicts the mentioned fact.

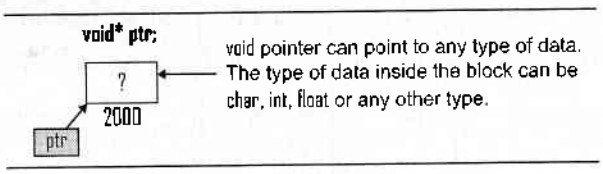


Figure 4.8 | void pointer

4.4.2.1 Operations on void Pointer

The following operations on void pointer are allowed:

1. A pointer to any type of object can be assigned to a void pointer. This is a standard conversion and the compiler will do it implicitly without any explicit type casting. This is shown in Program 4-11.

Line	Prog 4-11.c	Output window
1	//Assigning a pointer to a void pointer	int* is implicitly converted to void*
2	#include<stdio.h>	**Remarks:**
3	main()	• iptr is of int* type
4	{	• vptr is of void* type
5	int a=10;	• int* implicitly gets converted to void* in line number 7
6	int *iptr=&a;	
7	void *vptr=iptr;	
8	printf("int* is implicitly converted to void*");	
9	}	

Program 4-11 | A program to illustrate that pointer to any type implicitly gets converted to void*

2. void pointers can be compared for equality and inequality.

The following operations on void pointers are not allowed:

1. A void pointer cannot be dereferenced.
2. Pointer arithmetic is not allowed on void pointers.

   Reason: A void pointer cannot be dereferenced and pointer arithmetic is not applicable on it because the compiler does not know what kind of object the void pointer is really pointing to. Hence, the precise number of bytes to which the pointer refers to is not known. The compiler must know the number of bytes to which a pointer refers to in order to apply dereference operation and pointer arithmetic.

Before the application of dereference operator or arithmetic operator on a void pointer, it must be explicitly type casted to a pointer to a specific type.

4.4.3 Null Pointer

A **null pointer** is a special pointer that does not point anywhere. It does not hold the address of any object or function. It has numeric value 0. The following declaration statement declares ptr as a null pointer:

```
int *nptr=0;
```

The macro➲ or symbolic constant NULL defined in the header files stdio.h, stddef.h, stdlib.h, alloc.h and mem.h can also be used for the creation of a null pointer. The following declaration statement is equivalent to the declaration statement mentioned above:

```
int *nptr=NULL;
```

The important points about null pointers are as follows:

1. When a null pointer is compared with a pointer to any object or a function, the result of comparison is always false.
2. Two null pointers always compare equal.
3. Dereferencing a null pointer leads to a runtime error.

Forward Reference: Macros and symbolic constants (Chapter 8).

4.5 Relationship Between Arrays and Pointers

In C language, **arrays and pointers** are so closely related that they cannot be studied in isolation. They are often used interchangeably. The following relationships exist between arrays and pointers:

1. The name of an array refers to the address of the first element of the array, i.e. an expression of array type decomposes to pointer type. Program 4-12 illustrates this fact.

Line	Prog 4-12.c	Output window
1	//Arrays and pointers relationship-I	First element of array is at 2407:2242
2	#include<stdio.h>	Second element of array is at 2407:2244
3	main()	Third element of array is at 2407:2246
4	{	**Remarks:**
5	int arr[3]={10,15,20};	• The name of the array (i.e. arr) refers to the address of the first element of the array and is a constant object
6	printf("First element of array is at %p\n",arr);	• The expression arr+I decomposes to pointer type
7	printf("Second element of array is at %p\n",arr+1);	• Thus, in expression arr+I, the arithmetic involved is pointer arithmetic
8	printf("Third element of array is at %p\n",arr+2);	• Note that ++arr cannot be written instead of arr+I as arr is a constant object
9	}	

Program 4-12 | A program to depict the relationship between arrays and pointers

The name of an array refers to the address of the first element of the array but there are two exceptions to this rule:

a. When an array name is operand of sizeof operator it does not decompose to the address of its first element. Program 4-13 illustrates this fact.

Line	Prog 4-13.c	Output window
1	//sizeof operator and arrays	The result of sizeof operator is 10
2	#include<stdio.h>	**Remarks:**
3	main()	• The result of the sizeof operator is the size of the complete array (i.e. 5 elements * 2 bytes each = 10 bytes)
4	{	• This example clearly indicates that the name of the array is not decomposed into pointer type
5	int array[5]={10,15,20,25,30};	• If it would have been decomposed into pointer type, the result would have been 2 as integer pointer takes 2 bytes in the memory (in case of Borland Turbo C 3.0)
6	printf("The result of sizeof operator is %d\n",sizeof(array));	
7	}	

Program 4-13 | A program to illustrate the application of the sizeof operator on arrays

b. When an array name is an operand of reference or address-of operator it does not decompose to the address of its first element.

2. In C language, any operation that involves array subscripting is done by using pointers. The expression of form E1[E2] is automatically converted into an equivalent expression of form *(E1+E2). Program 4-14 illustrates this fact.

Line	Prog 4-14.c	Output window
1	//Arrays and pointers relationship-II	Elements are 10 15 20 Elements are 10 15 20 Elements are 10 15 20 **Remarks:** • E1[E2] is the usual way of subscripting (used in line number 6) • E1[E2] gets converted to *(E1+E2). The transformed way of subscripting is used in line number 7 • 0[array] used in line number 8 is also valid because 0[array] will automatically be converted to *(0+array), which is equivalent to *(array+0), + being a commutative operation • *(array+0) is equivalent to array[0]. Hence, 0[array] is equivalent to array[0]
2	#include<stdio.h>	
3	main()	
4	{	
5	int array[3]={10,15,20};	
6	printf("Elements are %d %d %d\n",array[0], array[1],array[2]);	
7	printf("Elements are %d %d %d\n",*(array+0),*(array+1),*(array+2));	
8	printf("Elements are %d %d %d\n",0[array],1[array],2[array]);	
9	}	

Program 4-14 | A program to depict the relationship between arrays and pointers

4.5 Scaling up the Concept

With all this knowledge at hand, it is the time to scale up the concept and look at array of arrays (i.e. multi-dimensional arrays), array of pointers, pointer to a pointer and pointers to arrays.

4.5.1 Array of Arrays (Multi-dimensional Arrays)

A 2-D array is an array of 1-D (i.e. single dimensional) arrays and can be visualized as a plane that has rows and columns. Each row is a single-dimensional array. A **3-D array** is an array of 2-D arrays and can be visualized as a cube that has planes. Each plane is a 2-D array. This concept can be scaled up to any level and in general, an **n-D array** is an array of (n-1)-D arrays. Arrays having dimensions higher than three are generally not needed unless and until highly data-intensive applications are to be developed. Therefore, I will restrict the discussion only upto two-dimensional arrays.

4.5.1.1 Two-dimensional Arrays

A **two-dimensional array** has its elements arranged in a rectangular grid of rows and columns. The elements of a two-dimensional array can be accessed by using a row subscript (i.e. row number) and a column subscript (i.e. column number). Both the row subscript and the column subscript are required to select an element of a two-dimensional array. A two-dimensional array is popularly known as a **matrix**. Figure 4.9 depicts a two-dimensional array as an array of 1-D arrays.

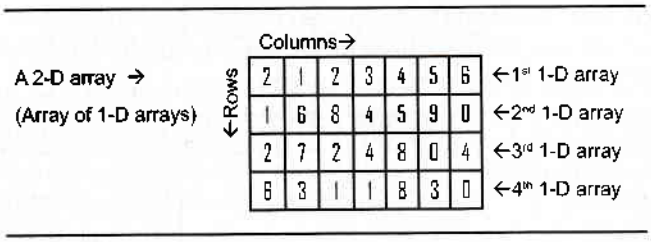


Figure 4.9 | A two-dimensional array

4.6.1.1.1 Declaration of a Two-dimensional Array

The general form of a two-dimensional array declaration is:

<sclass_specifier><type_qualifier><type_modifier>**type identifier[**<row_specifier>**][column_specifier]**<=initialization_list<....>>;

The important points about a two-dimensional array declaration are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in bold are the mandatory parts of a two-dimensional array declaration.
2. A two-dimensional array declaration consists of a type specifier (i.e. element type), an identifier (i.e. name of the array), a row size specifier (i.e. number of rows in an array) and a column size specifier (i.e. number of columns in each row). The size specifiers are enclosed within square brackets. The following declarations of two-dimensional arrays are valid:

```
int array1[2][3];       //←array1 is an integer array of 2 rows and 3 columns
float array2[5][1];     //←array2 is a float array of 5 rows and 1 column
char array3[3][3];      //←array3 is a character array of 3 rows and 3 columns
```

3. The row size specifier and column size specifier should be a compile time constant expression greater than zero.
4. The specification of a row size and column size is mandatory if an initialization list is not present. If the initialization list is present, the row size specifier can be skipped but it is mandatory➲ to mention the column size specifier.
5. **Initializing elements of two-dimensional arrays:** Like one-dimensional arrays, the elements of two-dimensional arrays can also be initialized by providing an initialization list.

   The syntactic rules about the initialization of elements of a two-dimensional array are as follows:

   a. The number of initializers in the initialization list should be less than or at most equal to the number of elements (i.e. row size × column size) in the array.
   b. The array locations are initialized row-wise. If the number of initializers in the initialization list is less than the number of elements in the array, the array locations that do not get initialized will automatically be initialized to 0 (if it is an integer

array), 0.0 (in case of a floating point array) and '\0' (i.e. null character if it is an array of character type). The mentioned fact is shown in Figure 4.10.

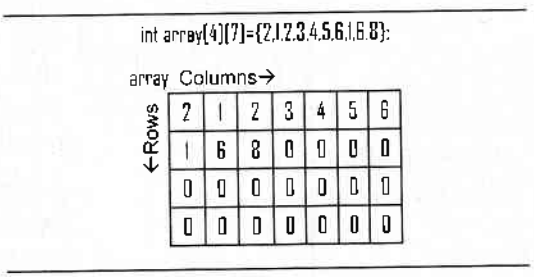


Figure 4.10 | Initialization of a two-dimensional array

c. The initializers in the initialization list can be braced to initialize elements of the individual rows. If the number of initializers within the inner braces is less than the row size, trailing locations of the corresponding row get initialized to 0, 0.0 or '\0', depending upon the element type of the array. The mentioned fact is shown in Figure 4.11.

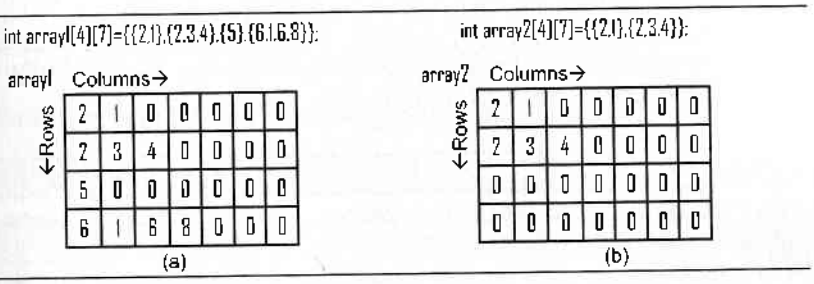


Figure 4.11 | Initialization of individual rows of a two-dimensional array

Forward Reference: Refer Question number 58 and its answer to know why it is mandatory to specify column size specifier even if a 2-D array is explicitly initialized.

4.5.1.1.2 Usage of a Two-dimensional Array

The elements of a two-dimensional array can be accessed by using row and column subscripts. The important points about the usage of a two-dimensional array are as follows:

1. An element of a two-dimensional array can be accessed by writing E1[E2][E3], where E1, E2 and E3 are sub-expressions. One of the sub-expressions E1 or E2 must be of an array type or a pointer type, and the other sub-expressions must be of integral type. Program 4-15 illustrates the use of a subscript operator to access the elements of a two-dimensional array.

Line	Prog 4-15.c
1	//Two-dimensional arrays
2	#include<stdio.h>
3	main()
4	{
5	int a[2][3]={2,1,3,2,3,4};
6	printf("Elements of array are:\n");
7	printf("%d %d %d\n",a[0][0], a[0][1], a[0][2]);
8	printf("%d %d %d\n",1[a][0], 1[a][1], 1[a][2]);
9	}

Memory contents

a	[0]	[1]	[2]
[0]	2	1	3
[1]	2	3	4

Output window

Elements of array are:
2 1 3
2 3 4

Remarks:

- The general form of an expression for accessing an element of a 2-D array is E1[E2][E3]
- In line number 7, E1 is of array type and E2 is of int type
- In line number 8, E1 is of int type and E2 is of array type
- Both types of usage are valid
- The sub-expression E3 must be of integral type and cannot be of array type

Program 4-15 | A program to illustrate the usage of a two-dimensional array

2. The expression E1[E2][E3] is implicitly converted into an equivalent expression of form *(*(E1+E2)+E3). Program 4-16 illustrates this fact.

Line	Prog 4-16.c
1	// Subscript operator and equivalent conversion to pointer form
2	#include<stdio.h>
3	main()
4	{
5	int a[2][3]={2,1,3,2,3,4};
6	printf("Use of subscript operator:\n");
7	printf("%d %d %d\n",a[0][0], a[0][1], a[0][2]);
8	printf("%d %d %d\n",a[1][0], a[1][1], a[1][2]);
9	printf("Use of pointer expressions:\n");
10	printf("%d %d %d\n",*(*(a+0)+0), *(*(a+0)+1), *(*(a+0)+2));
11	printf("%d %d %d\n", *(*(a+1)+0), *(*(a+1)+1), *(*(a+1)+2));
12	printf("Use of mixed form of expressions:\n");
13	printf("%d %d %d\n",*(a[0]+0), *(a[0]+1), *(a[0]+2));
14	printf("%d %d %d\n", *(a[1]+0), *(a[1]+1), *(a[1]+2));
15	}

Output window

Use of subscript operator:
2 1 3
2 3 4
Use of pointer expressions:
2 1 3
2 3 4
Use of mixed form of expressions:
2 1 3
2 3 4

Remark:

- The expression *(a+i) is equivalent to a[i]. Hence, the expression *(*(a+i)+j) is equivalent to *(a[i]+j), which is further equivalent to a[i][j].

Program 4-16 | A program to illustrate the conversion of a subscript operator into an equivalent pointer form

3. In an expression that involves an array, if the number of subscripts used with the array name is less than the dimensions of the array, the expression refers to an address instead of a value. Program 4-17 illustrates this fact.

Line	Prog 4-17.c	Memory contents	Output window
1	//Number of subscripts and values	a	No subscript used:
2	#include<stdio.h>	Indices [0] [1]	234F:2234
3	main()	[0] 2 1	One subscript used:
4	{	2234 2236	234F:2234 234F:2238
5	int a[2][2]={2,1,3,4};	[1] 3 4	Two subscripts used:
6	printf("No subscript used:\n");	2238 2240	2 1
7	printf("%p\n",a);		3 4
8	printf("One subscript used:\n");		**Remarks:**
9	printf("%p %p\n",a[0], a[1]);		• When no subscript is used, the expression a refers to the starting address of the first element (i.e. first row) of the array
10	printf("Two subscripts used:\n");		• When one subscript is used, the expressions a[0] and a[1] refer to the starting address of the first row and the second row, respectively
11	printf("%d %d\n",a[0][0], a[0][1]);		• When two subscripts are used, the expressions in line numbers 11 and 12 refer to the value of the corresponding array element
12	printf("%d %d\n",a[1][0], a[1][1]);		
13	}		

Program 4-17 | A program to illustrate the outcome of an expression that uses lesser subscripts than dimensions

4.6.1.1.2.1 Reading, storing and accessing elements of a 2-D array

The elements can be read and stored in a 2-D array by making use of nested loops. Program 4-18 illustrates the method to read, store and access the elements of a two-dimensional array.

Line	Prog 4-18.c	Output window
1	// Reading, storing and accessing elements of a two-dimensional array	Enter the number of rows(<10): 2
2	#include<stdio.h>	Enter the number of cols(<10): 2
3	main()	Enter the elements:
4	{	2
5	int a[10][10], olc, ilc, rows, cols;	3
6	printf("Enter the number of rows(<10):\t");	3
7	scanf("%d",&rows);	4
8	printf("Enter the number of cols(<10)\t");	The entered elements were:
9	scanf("%d",&cols);	2 3
10	printf("Enter the elements:\n");	3 4
11	for(olc=0;olc<rows;olc++)	**Remarks:**
12	for(ilc=0;ilc<cols;ilc++)	• olc is the outer loop counter
13	scanf("%d",&a[olc][ilc]); //←Reading and storing elements	• ilc is the inner loop counter
14	printf("The entered elements were:\n");	• To read and store elements in a 2-D array, a nested loop consisting of two loops is required
15	for(olc=0;olc<rows;olc++)	• The outer loop is for getting the rows, and the inner loop is for getting the elements of a row (i.e. columns)
16	{	
17	for(ilc=0;ilc<cols;ilc++)	
18	printf("%d ",a[olc][ilc]); //←Accessing elements	
19	printf("\n");	
20	}	
21	}	

Program 4-18 | A program to illustrate the method of reading, storing and accessing elements of a two-dimensional array

4.6.1.1.3 Memory Representation of a Two-dimensional Array

A 2-D array can be visualized as a plane, which has rows and columns. Although multi-dimensional arrays are visualized in this way, they are actually stored in the memory, which is linear (i.e. one dimensional). Hence, a multi-dimensional array is to be stored in one dimension. There are two ways of doing this:

1. Row major order of storage
2. Column major order of storage

4.6.1.1.3.1 Row Major Order of Storage

In **row major order of storage**, the elements of an array are stored row-wise. **In C language, multi-dimensional arrays are stored in the memory by using row major order of storage.** Figure 4.12 shows the row major order of storage.

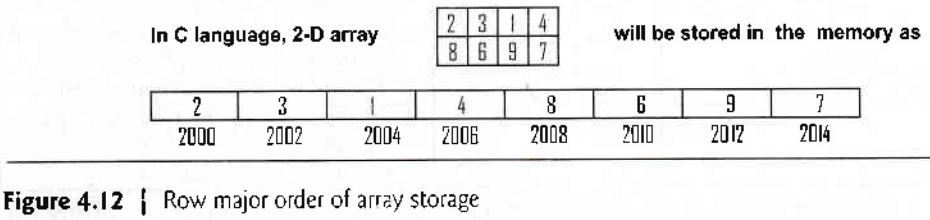


Figure 4.12 | Row major order of array storage

4.6.1.1.3.2 Column Major Order of Storage

In **column major order of storage**, the elements of an array are stored column-wise. Column major order of array storage is used in the languages like FORTRAN, MATLAB, etc. Figure 4.13 shows the column major order of storage.

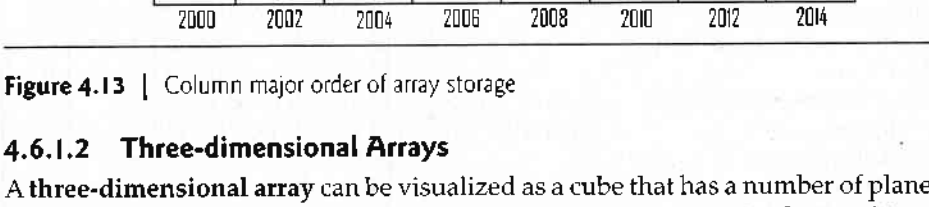


Figure 4.13 | Column major order of array storage

4.6.1.2 Three-dimensional Arrays

A **three-dimensional array** can be visualized as a cube that has a number of planes. Each plane is a two-dimensional array. Thus, a three-dimensional array is made up of two-dimensional arrays. Figure 4.14 depicts a three-dimensional array as an array of 2-D arrays.

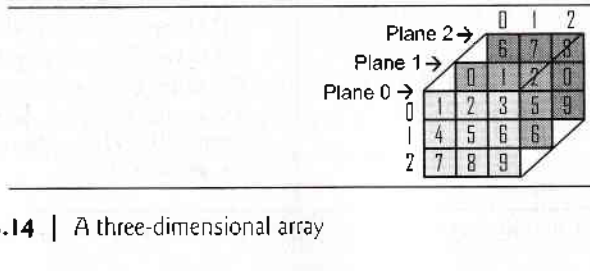


Figure 4.14 | A three-dimensional array

4.6.1.2.1 Declaration of a Three-dimensional Array

The general form of a three-dimensional array declaration is:

<scspec*><type_qual><type_mod>**type identifier**[<plane_specifier>][**row_specifier**][**column_specifier**]<=init_list<...>>;

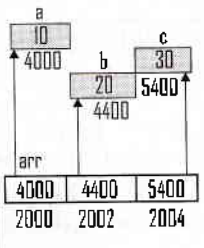
*- scspec means storage class specifier

The important points about a three-dimensional array declaration are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in **bold** are the mandatory parts of a three-dimensional array declaration.
2. A three-dimensional array declaration consists of a type specifier (i.e. element type), an identifier (i.e. name of array), a plane size specifier, a row size specifier and a column size specifier. The size specifiers are enclosed within the square brackets (i.e. []).
3. The plane size specifier, row size specifier and column size specifier should be a compile time constant expression greater than zero.
4. The specification of all size specifiers is mandatory if the elements of an array are not explicitly initialized. If an initialization list is present, the plane size specifier can be skipped but it is mandatory to mention the row size specifier and column size specifier. The general rule is **'While declaring n-D arrays, even if initialization list is present, it is mandatory to specify (n-1) fastest varying specifiers'**. In case of two-dimensional arrays, the column size specifier varies faster as compared to the row size specifier. In case of three-dimensional arrays, column size specifier and row size specifier vary faster than a plane size specifier.
5. **Initializing elements of three-dimensional arrays:** The elements of a three-dimensional array can be initialized in the same way as the elements of a two-dimensional array are initialized, i.e. by providing an initialization list.

4.6.2 Array of Pointers

An **array of pointers** is a collection of addresses. The addresses in an array of pointers could be the addresses of isolated variables or the addresses of array **elements** or any other addresses. The only constraint is that all the pointers in an array must be of the same type. Program 4-19 illustrates the use of array of pointers.

Line	Prog 4-19.c	Memory contents	Output window
1	// Array of pointers		The values of variables are: 10 20 30 10 20 30
2	#include<stdio.h>		**Remarks:**
3	main()		• arr is an array of integer pointers and holds the addresses of variables a, b and c
4	{		• All the variables are of the same type
5	int a=10,b=20, c=30;		
6	int* arr[3]={&a, &b, &c};		
7	printf("The values of variables are:\n");		
8	printf("%d %d %d\n",a,b,c);		
9	printf("%d %d %d\n",*arr[0],*arr[1],*arr[2]);		
10	}		

Program 4-19 | A program to illustrate the use of array of pointers

4.6.3 Pointer to a Pointer

A pointer that holds the address of another pointer variable is known as a **pointer to a pointer**. Such a pointer is said to exhibit multiple levels of indirection. There can be many levels of indirection in a single declaration statement. Consider the code snippet in Program 4-20.

Line	Prog 4-20.c
1	// Pointer to a pointer
2	#include<stdio.h>
3	main()
4	{
5	int i=10;
6	int *p1=&i; //←Pointer to int
7	int **p2=&p1; //←Pointer to pointer to int
8	int ***p3=&p2; //←Pointer to pointer to pointer to int
9	int ****p4=&p3;//←.....concept scales up
10	int *****p5=&p4;
11	int ******p6=&p5;
12	int *******p7=&p6;
13	int ********p8=&p7;
14	int *********p9=&p8;
15	int **********p10=&p9;
16	int ***********p11=&p10;
17	int ************p12=&p11;
18	printf("The values of variables are:\n");
19	printf("%d\t",*p1);
20	printf("%d\t",**p2);
21	printf("%d\t",***p3);
22	printf("%d\t",****p4);
23	printf("%d\t",*****p5);
24	printf("%d\t",******p6);
25	printf("%d\t",*******p7);
26	printf("%d\t",********p8);
27	printf("%d\t",*********p9);
28	printf("%d\t",**********p10);
29	printf("%d\t",***********p11);
30	printf("%d\t",************p12);
31	}

Output window

```
The values of variables are:
10 10 10 10 10 10 10 10 10 10 10 10
```

Remarks:

- The ANSI C standard says that all compilers must handle at least 12 levels of indirection
- Some compilers may support more levels of indirection
- Two levels of indirection are common
- Level of indirection higher than two becomes difficult to understand and visualize
- In an expression, if the number of indirection operators used to dereference a pointer is less than the number of punctuators (*) used to declare the pointer, then the pointer will not be completely dereferenced and the expression refers to an address
- The number of indirection operators required to completely dereference a pointer is equal to the number of punctuators (*) used while declaring it
- For example, in the mentioned code the expression *p2 refers to an address, i.e. address of p1. In the expression **p2, p2 is completely dereferenced and refers to the value of i, i.e. 10

Program 4-20 | A program to illustrate the use of multi-level pointers

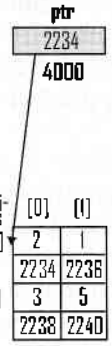
4.6.4 Pointer to an Array

It is possible to create a pointer that points to a complete array instead of pointing to the individual elements of an array or isolated variables. Such a pointer is known as a **pointer to an array**. The following declaration statements declare such pointers:

```
int (*p1)[5];       //←p1 is a pointer to an array of 5 integers
int (*p2)[2][2];    //←p2 is a pointer to an integer array of 2 rows and 2 columns
int (*p3)[2][3][4]; //←p3 is a pointer to an integer array having 2 planes. Each plane
                    //←has 3 rows and 4 columns
```

While declaring pointer to an array, parentheses, i.e. () are used because [] binds more tightly than *. If parentheses are not used, the declaration **int *p[5];** declares p as an array of 5 integer pointers. In the said declaration, p becomes an array instead of becoming a pointer because [] binds p more tightly than *. To make p a pointer to an array of 5 integers, write it as **int(*p)[5]**. In this declaration, parentheses are used to bind p with *.

Program 4-21 illustrates the use of a pointer to an array.

Line	Prog 4-21.c	Memory contents	Output window
1	// Pointer to an array		Address of row 1 is 234F:2234 Address of row 2 is 234F:2238 1st element of row 1 is 2 1st element of row 2 is 3 **Remarks:** • arr refers to the address of the first element of the array • Elements of a 2-D array are 1-D arrays • Thus, arr refers to the address of first 1-D array of two integers (i.e. first row) • The type of arr is int(*)[2] • Type of ptr is int(*)[2] • ptr is initialized with the starting address of row 1 • ptr+1 will point to the next row • As types of arr and ptr are same and both refer to the same address, the expression ptr[1][0] is equivalent to the expression arr[1][0]
2	#include<stdio.h>		
3	main()		
4	{		
5	int arr[2][2]={{2,1},{3,5}};		
6	int (*ptr)[2]=arr;		
7	printf("Address of row 1 is %p\n",arr[0]);		
8	printf("Address of row 2 is %p\n",ptr+1);		
9	printf("1st element of row 1 is %d\n", arr[0][0]);		
10	printf("1st element of row 2 is %d\n",ptr[1][0]);		
11			
12	}		

Program 4-21 | A program that illustrates the creation and usage of a pointer to an array

4.7 Advantages and Limitations of Arrays

The direct indexing supported by arrays is their biggest advantage. **Direct indexing** means the time required to access any element in an array of any dimension is almost the same irrespective of its location in the array.

The limitations of arrays are as follows:

1. The memory to an array is allocated at the compile time.
2. Arrays are static in nature. The size of an array cannot be expanded or cannot be squeezed at the run time.
3. The size of an array has to be kept big enough to accommodate the worst cases. Therefore, memory usage in case of arrays is inefficient.

4.8 Summary

1. An array is used to store homogeneous data, i.e. data of the same type.
2. All the elements of an array have the same name, i.e. the array name. They are distinguished on the basis of their locations in the array. Locations are specified by using an integer value known as an index or a subscript.

3. Arrays are also known as indexed variables or subscripted variables.
4. Array index in C starts with 0.
5. C does not provide array index out-of-bound check.
6. Arrays are stored in contiguous (i.e. continuous) memory locations.
7. Arrays are classified as single-dimensional arrays and multi-dimensional arrays.
8. Subscript operator is used to access the elements of an array.
9. The array name refers to the address of the first element of the array and is a constant object.
10. An array cannot be assigned to or initialized with another array.
11. If an array is equated with another array, it always evaluates to false.
12. A pointer is a variable that holds the address of a variable or a function.
13. Restricted arithmetic can be applied on pointers. Arithmetic on pointers is governed by pointer arithmetic.
14. Addition of two pointers, addition of a float or a double value to a pointer, application of multiplication and division operators on pointers are not allowed.
15. void pointer is a generic pointer and can point to any type of object.
16. Dereferencing a void pointer and applying pointer arithmetic to it is not allowed.
17. Null pointer is a special pointer that does not point anywhere.
18. Dereferencing a null pointer leads to run time error.
19. An n-D array is an array of (n-1)-D arrays.
20. The expression of form E1[E2] is implicitly converted to an expression of the form *(E1+E2).
21. In C language, multi-dimensional arrays are stored in the memory by using row major order of storage.

Exercise Questions

Conceptual Questions and Answers

1. *What is a pointer? Where is it used?*

   A pointer is an object➲ that holds the address of another object. A pointer is used to indirectly manipulate the value of an object to which it points.

Forward Reference: Object (Chapter 7).

2. *I know about basic data types in C language but what is pointer type?*

   Apart from basic data types, the C language allows to derive types from the basic data types. These types are called **derived data types**. A pointer type is one of the derived data types.

Backward Reference: Refer Section 4.4 for a detailed description on pointer type.

3. *What will the output of the following piece of code be?*

```
main()
{
   int *a;
```

```
   float *b;
   char *c;
   printf("%d
}
```

The output
piler used.
same code
piler, it out
An importa
ing upon th
pointers to
types is nei
in the type

4. *Why does th*
instead of 2 2
outputs 4 1 in

```
main()
{
   char *a,b;
   printf("%d
}
```

The code a
a pointer st
comma sep
serve as a p
Thus, in the
is declared

5. *The bitwise*
of code, these
it possible?

```
main()
{
   int a=10, b=
   int *ptr;
   ptr=&a;
   printf("The
   ptr=&b;
   printf("The
}
```

The & symbo
symbol * ca
instance of
used. The co

1. Numbe
2. Type of

```
    float *b;
    char *c;
    printf("%d %d %d", sizeof(a),sizeof(b),sizeof(c));
}
```

The output of the given piece of code is dependent on the execution environment and the compiler used. If the code is executed using Borland TC 3.0 compiler for DOS, it outputs 2 2 2. If the same code is executed using Borland TC 4.5 compiler for Windows or Microsoft VC++ 6.0 compiler, it outputs 4 4 4.
An important point to be noted here is that all pointers take 2 or 4 bytes in the memory (depending upon the execution environment and the compiler used), irrespective of whether they are pointers to int, float, char or some other data type. The difference between pointers of different data types is neither in the representation of the pointer nor in their values. The difference, rather, is in the type of the object being addressed.

4. *Why does the following piece of code on execution using Borland TC 3.0 compiler for DOS outputs 2 1 instead of 2 2 and if executed using Borland TC4.5 compiler for Windows or Microsoft VC++ 6.0 compiler outputs 4 1 instead of 4 4?*

```
main()
{
    char *a,b;
    printf("%d %d",sizeof(a),sizeof(b));
}
```

The code actually gives a correct output. The syntactic rule concerned with the declaration of a pointer states that '**A pointer is declared by prefixing an identifier with punctuator *. In a comma separated declaration list, the punctuator * must precede each identifier intended to serve as a pointer**'.
Thus, in the declaration statement `char *a,b;`, a is declared as 'pointer to an object of type char' and b is declared as 'data object of type char' and not as a pointer.

5. *The bitwise AND operator (&) and multiplication operator (*) are binary operators. In the following piece of code, these operators are used with only one operand. Even then the code compiles successfully. How is it possible?*

```
main()
{
    int a=10, b=20;
    int *ptr;
    ptr=&a;
    printf("The object to which ptr points has value %d",*ptr);
    ptr=&b;
    printf("The object to which ptr points now has value %d",*ptr);
}
```

The & symbol can be used as a bitwise AND operator and as a reference operator. Similarly, the symbol * can be used as a multiplication operator and as a dereference operator. The particular instance of a symbol corresponds to which operator depends upon the context in which it is used. The context can be determined by looking at:

1. Number of operands
2. Type of operands

The following are the possible combinations:

Symbol	Number of operands	Type of operands	Meaning of arithmetic, scalar and pointer type	Operator
&	Two	Arithmetic type	Integer, float and character	Bitwise AND operator
	One	Scalar type	Arithmetic type and pointer type	Reference operator
*	Two	Arithmetic type	Integer, float and character	Multiplication operator
	One	Pointer type	Pointer to a data type	Dereference operator

The symbol & when used as a reference operator should appear as a prefix unary operator and should be applied on the operands of scalar type that have l-values. The symbol * when used as a dereference operator should appear as a prefix unary operator and should be applied on the operands of the pointer type. In the mentioned piece of code: & symbol refers to the reference operator and * symbol refers to the dereference operator, which are unary operators. Hence the code compiles successfully.

6. *Why does the following piece of code not compile successfully?*

```
main()
{
    int *ptr=10;
    printf("The value pointed to by pointer is %d",*ptr);
}
```

The mentioned piece of code does not compile successfully because of illegal initialization statement whereby a pointer variable ptr is tried to be initialized with an integer value 10. The compiler gives 'Cannot convert int to int*' error because the types int and int* are incompatible and the compiler will not carry out int to int*conversion implicitly. This error can however be removed by making use of explicit type casting and writing the statement as int ptr=(int*)10;. In this statement, the programmer has forcefully converted int to int* and will himself or herself be responsible for the results. This type of explicit type conversion is not recommended.

7. *Can* const *qualifier be used with pointer types like it can be used with basic data types?*

Yes, const qualifier can be used with pointer types. It is important to understand the use of const qualifier when it is mixed with pointer type. const qualifier can be mixed with pointer type in the following ways:

S.No	Use of const qualifier with pointer type (Column 2)	Meaning of statements in Column 2	What is constant?
1.	const int *ptr	ptr is a pointer to an integer constant	Integer object pointed to by ptr
2.	int const *ptr	ptr is a pointer to a constant integer (same as declaration at S.No. 1)	Integer object pointed to by ptr
3.	int *const ptr	ptr is a constant pointer to an integer	ptr is constant
4.	const int *const ptr	ptr is a constant pointer to an integer constant	Both ptr and the integer object pointed to by ptr are constant
5.	int const *const ptr	ptr is a constant pointer to a constant integer (same as declaration S.No. 4)	Both ptr and the integer object pointed to by ptr are constant

8. *What is pointer arithmetic?*

Backward Reference: Refer Section 4.4.1.4 for a description on pointer arithmetic.

9. *In the expression* `*pointer++`*, which entity gets incremented: pointer or the value to which the pointer points?*

Dereference operator * and the increment operator ++ are unary operators and unary operators are right-to-left associative. The expression `*pointer++` will be interpreted as `*(pointer++)`. Thus, in this expression, the value of the pointer instead of the value pointed by the pointer gets incremented.

10. *I want to print the memory address to which a pointer points. Which format specifier should I use to print it?*

The format specifier used for printing pointers (addresses) is `%p`. The printed format depends upon which memory model is used. It will either be `XXXX:YYYY` (segment:offset) or `YYYY` (offset only).
Consider the following piece of code:

```
main()
{
   int a=10;
   int *ptr=&a;
   printf("The value of pointer is %p",ptr);
}
```

If the code is executed using Borland TC 3.0 compiler for DOS and `small` memory model, it prints `FFF4` (offset address only). If worked with `huge` memory model, it prints `900E:00FE` (segment and offset address).

11. *What is array type and how is it declared?*

Backward Reference: Refer Sections 4.2 and 4.3 for a description on array type.

12. *I want to store an integer value, a* `float` *value and a character value in an array. Is it possible?*

No, arrays can only be used for storage of homogeneous data (i.e. data of the same type). Arrays cannot be used for storage of heterogeneous data (i.e. data of different types). For storage of heterogeneous data, structures and unions➲ are used.

Forward Reference: Structure and unions (Chapter 9).

13. *How is the declaration* `int * a[10]` *different from* `int (*a) [10]`*?*

While reading C declarations remember that [] binds➲ more tightly than *. In the declaration statement `int *a[10]`; the identifier name `a` is bound to [] instead of * and it is read as '`a` is an array of 10 integer pointers'. In the declaration statement `int (*a)[10];`, () is used to bind `a` to *. Hence, the declaration is read as '`a` is a pointer to an array of 10 integers'.

Forward Reference: Refer Section 5.3.1.1.8 on pointers to functions to see that () also binds more tightly than *.

14. *How is an expression involving a subscript operator internally represented?*

The general form of an expression involving a subscript operator is E1[E2], where both E1 and E2 are sub-expressions. One of the sub-expressions E1 or E2 must be of array type or pointer type and the other expression must be of integer type. Every expression of the form E1[E2] automatically gets converted to an equivalent expression of the form *(E1+E2). Hence, the expression E1[E2] is internally represented as *(E1+E2).

Consider the following piece of code:

```
main()
{
    int array[4]={4,5,6,7};
    int *pointer=array;
    printf("%d %d %d\n", array[0],pointer[1],*(array+2), *(pointer+3));
}
```

E1 is of array type
E1 is of pointer type
Transformed form of subscript operator

The mentioned code on execution outputs:

```
4 5 6 7
```

15. *Are the expressions* arr *and* &arr *same, if* arr *is an array of type* T*?*

No, **the expressions arr and &arr are not the same.** The expression &arr yields 'a pointer to an array of type T' and the expression arr yields 'a pointer to type T'. The expression arr refers to the address of first element of the array and the expression &arr refers to the base address of the entire array. To understand the difference between arr and &arr, consider the following piece of code:

```
main()
{
    int arr[5]={1,2,3,4,5};
    printf("The base address of array is %p or %p\n",arr,&arr);
    printf("After incrementing by one they point to %p and %p",arr+1,&arr+1);
}
```

The code on execution outputs:

```
The base address of array is 186F:223A or 186F:223A
After incrementing by one they point to 186F:223C and 186F:2244
```

Increment of one in arr increments it by 2-bytes as it is of type int* while increment of one in &arr increments it by 10-bytes as its type is int(*)[5] (i.e. pointer to an array of 5 integers).

16. *Does the C language provide array index out-of-bound check?*

No, the C language does not provide compile-time or run-time array index out-of-bound check. If an array is declared as T array[size], the maximum valid index is size-1, as array index in C language starts from 0. Nothing stops a programmer from stepping across an array boundary and accessing the array with an index greater than size-1. The program having array index out-of-bound will compile and execute but will access to the memory location that does not belong to the array. This illegal memory access may be fatal and may even crash the program.

17. *Why does the following piece of code on execution give a garbage value?*

```
main()
{
    int array[3]={1,2,3};
    printf("The last element of array is %d",array[3]);
}
```

The mentioned piece of code gives a garbage value because the array index is out-of-bound. The maximum valid array index is 2. Since the array is indexed with 3, reference has been made to the memory location that does not belong to the array (i.e. garbage field). Hence, the code on execution gives a garbage value. Note that in some cases the program may even crash, i.e. terminate.

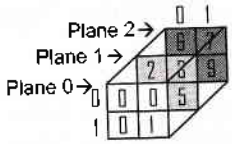
18. *How will you visualize a multi-dimensional array?*

Backward Reference: Refer Section 4.6.1 for a description on multi-dimensional arrays.

19. *How are multi-dimensional arrays stored in C?*

Backward Reference: Refer Section 4.6.1 for a description on multi-dimensional arrays.

Suppose a three-dimensional array is declared as `int a[3][2][2]={0,0,0,1,2,3,4,5,6,7,8,9};`. It can be visualized as:



The shown 3-D array will actually be stored in the physical memory as:

a

[0][0][0]	[0][0][1]	[0][1][0]	[0][1][1]	[1][0][0]	[1][0][1]	[1][1][0]	[1][1][1]	[2][0][0]	[2][0][1]	[2][1][0]	[2][1][1]
0	0	0	1	2	3	4	5	6	7	8	9
2000-01	2002-03	2004-05	2006-07	2008-09	2010-11	2012-13	2014-15	2016-17	2018-19	2020-21	2022-23

20. *What would be the result of a* `sizeof` *operator, when it is applied on an array type?*

When the `sizeof` operator is applied on an operand of array type, the result is the total number of bytes occupied by the array.

21. *What is null pointer? Is null pointer same as uninitialized pointer?*

Backward Reference: Refer Section 4.4.3 for a description on null pointer.

No, null pointer is not the same as uninitialized pointer. A null pointer does not point to any object or function, while an uninitialized pointer might point anywhere. The declaration statements `int *ptr=0;` and `int *ptr=NULL;` create a null pointer, named `ptr`, and the declaration statement `int *ptr;` creates an uninitialized pointer.

22. *What is a* `void` *pointer?*

Backward Reference: Refer Section 4.4.2 for a description on `void` pointer.

23. *Why is pointer arithmetic not applicable on* `void` *pointers?*

Pointer arithmetic is not applicable on `void` pointers because the compiler does not know what kind of object the `void` pointer is really pointing to. Before applying an arithmetic operator on `void*`, explicitly type cast `void*` to a pointer to a specific type.

24. *Given the declaration statement,* int array[10],i=2; *what are the types of expressions* array, &array, *array, array[i]*?*

The types of expressions:

array is int*	(i.e. pointer to the first element of array)
&array is int(*)[10]	(i.e. pointer to the entire array)
*array is int	(i.e. value of first element of the array)
array[i] is int	(i.e. value of $(i+1)^{th}$ element of the array)

25. *Given the declaration statement,* int array[10][10],i=2,j=2; *what are the types of expressions* array, &array, *array, array[i], **array, array[i][j]*?*

The types of expressions:

array is int(*)[10]	(i.e. pointer to the first row of array)
&array is int(*)[10][10]	(i.e. pointer to the entire array)
array is int	(i.e. pointer to first element in the first row of array)
array[i] is int*	(i.e. pointer to first element in $(i+1)^{th}$ row of the array)
**array is int	(i.e. value of first element in first row of the array)
array[i][j] is int	(i.e. value of element in $(i+1)^{th}$ row and $(j+1)^{th}$ column of the array)

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

26.
```
main()
{
    char*p1,p2;
    printf("%d %d",sizeof(p1),sizeof(p2));
}
```

27.
```
main()
{
    printf("%d %d %d",sizeof(char*),sizeof(int*),sizeof(float*));
}
```

28
```
main()
{
    char far *p1, near *p2, huge *p3;
    printf("%d %d %d",sizeof(p1),sizeof(p2),sizeof(p3));
}
```

29.
```
main()
{
    int a=10;
    int *ptr=&a;
    printf("%d %d",++*ptr,*ptr++);
}
```

30.
```
main()
{
    int a=10;
    int *ptr=&a;
    printf("%d %d",*ptr++,++*ptr);
}
```

31.
```
main()
{
    int a=10;
    const int *p
    *ptr=50;
    printf("The
}
```

32.
```
main()
{
    int a=10,b=2
    int *const p
    *ptr=20;
    printf("The
    ptr=&b;
    *ptr=10;
    printf("The
}
```

33.
```
main()
{
    int a=10,b=2
    const int *co
    *ptr=20;
    printf("The c
    ptr=&b;
    *ptr=10;
    printf("The c
}
```

34.
```
main()
{
    int *ptr=10;
    printf("The va
}
```

35.
```
main()
{
    int *ptr=0;
    printf("The va
}
```

36.
```
main()
{
    int *ptr1=0;
    int *ptr2=NUL
    if(ptr1==ptr2)
        printf("ptr
    else
        printf("ptr
}
```

```
main()
{
    int a=10;
    const int *ptr=&a;
    *ptr=50;
    printf("The changed value of pointed object is %d",*ptr);
}
```

```
main()
{
    int a=10,b=20;
    int *const ptr=&a;
    *ptr=20;
    printf("The changed value of pointed object a is %d",*ptr);
    ptr=&b;
    *ptr=10;
    printf("The changed value of pointed object b is %d",*ptr);
}
```

```
main()
{
    int a=10,b=20;
    const int *const ptr=&a;
    *ptr=20;
    printf("The changed value of pointed object a is %d",*ptr);
    ptr=&b;
    *ptr=10;
    printf("The changed value of pointed object b is %d",*ptr);
}
```

```
main()
{
    int *ptr=10;
    printf("The value of pointer is %p",ptr);
}
```

```
main()
{
    int *ptr=0;
    printf("The value of pointer is %p",ptr);
}
```

```
main()
{
    int *ptr1=0;
    int *ptr2=NULL;
    if(ptr1==ptr2)
        printf("ptr1 becomes a NULL pointer");
    else
        printf("ptr1 does not become a NULL pointer");
}
```

37.
```
main()
{
    int arr[ ];
    arr[0]=arr[1]=arr[2]=5;
    printf("%d %d %d",arr[0],arr[1],arr[2]);
}
```

38.
```
main()
{
    int size=3;
    int arr[size];
    arr[0]=arr[1]=arr[2]=5;
    printf("%d %d %d",arr[0],arr[1],arr[2]);
}
```

39.
```
main()
{
    int a[]={1,2,3};
    printf("%d %d %d",a[0],a[1],a[2]);
}
```

40.
```
main()
{
    int a[2]={1,2,3};
    printf("%d %d %d",a[0],a[1],a[2]);
}
```

41.
```
main()
{
    int arr[6]={1,2,3,4};
    int i;
    for(i=0;i<6;i++)
    printf("%d ",arr[i]);
}
```

42.
```
main()
{
    int arr[3]={1,2,3};
    printf("%d %d %d",arr[1],arr[2],arr[3]);
}
```

43.
```
main()
{
    int arr[]={1,2,3};
    arr[0,1,2]=10;
    printf("%d %d %d",arr[0],arr[1],arr[2]);
}
```

44.
```
main()
{
    int arr[]={1,2,3,4,5},i;
    arr[1+2]=10;
    for(i=0;i<5;i++)
        printf("%d ",arr[i]);
}
```

45.
```
main()
{
    int arr[]={1
    arr[2.5+1.5
    for(i=0;i<5;
        printf("
}
```

46.
```
main()
{
    int array[]=
    printf("The
}
```

47.
```
main()
{
    int a=10,b;
    int arr[]={1,
    printf("Assig
    b=a;
    printf("Assig
    brr=arr;
    printf("Conte
}
```

48.
```
main()
{
    int arr[]={1,2
    if(arr==brr)
        printf("Co
    else
        printf("Co
}
```

49.
```
main()
{
    int a[]={1,2,3,4
    int *ptr=a;
    printf("%d %d
}
```

50.
```
main()
{
    int arr[]={1,2,3
    printf("%p %p
    printf("%p %p
}
```

51.
```
main()
{
    int arr[]={1,2,3
    printf("%p %p
    printf("%p %p
}
```

```
main()
{
    int arr[]={1,2,3,4,5},i;
    arr[2.5+1.5]=10;
    for(i=0;i<5;i++)
        printf("%d ",arr[i]);
}
main()
{
    int array[]={1,2,3,4};
    printf("The number of elements in array are %d",sizeof(array)/sizeof(array[0]));
}
main()
{
    int a=10,b;
    int arr[]={1,2,3}, brr[3];
    printf("Assigning the content of a to b\n");
    b=a;
    printf("Assigning the contents of one array to another\n");
    brr=arr;
    printf("Contents of brr are %d %d %d",brr[0],brr[1],brr[2]);
}
main()
{
    int arr[]={1,2,3},brr[]={1,2,3};
    if(arr==brr)
        printf("Contents of array arr and brr are same\n");
    else
        printf("Contents of array arr and brr are not same");
}
main()
{
    int a[]={1,2,3,4,5};
    int *ptr=a;
    printf("%d %d\n%p %p",*a,*ptr,a,ptr);
}
main()
{
    int arr[]={1,2,3,4,5};
    printf("%p %p\n",arr,&arr);
    printf("%p %p",++arr,++&arr);
}
main()
{
    int arr[]={1,2,3,4,5};
    printf("%p %p\n",arr,&arr);
    printf("%p %p",arr+1,&arr+1);
}
```

```
52. main()
    {
        int a[]={1,2,3,4,5};
        printf("%d %d %d %d %d",*a,*(a+0),*(0+a),a[0],0[a]);
    }
53. main()
    {
        int *ptr;
        int arr[]={1,2,3,4};
        ptr=arr;
        printf("%d %d",arr[2],ptr[2]);
    }
54. main()
    {
        int arr[ ]={2,8,3,4,4,6,7,5};
        int j,*ptr=arr;
        for(j=0;j<5;j++)
        {
            printf(" %d ",*ptr);
            ++ptr;
        }
    }
55. main( )
    {
        int j=20;
        int arr[ ] = {10,j,30,40,50},i,*ptr;
        ptr = arr;
        for(i=0; i<5; i++)
        {
            printf("%d " ,*ptr);
            ptr++;
        }
    }
56. main()
    {
        int arr[2][3]={1,2,3,4};
        printf("%d %d %d %d %d %d",arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
    }
57. main()
    {
        int arr[2][3]={{1,2},{3,4}};
        printf("%d %d %d %d %d %d",arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
    }
58. main()
    {
        int arr[][]={1,2,3,4};
        printf("%d %d %d %d %d %d",arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
    }
```

```
main()
{
    int arr[2][][]={1,2,3,4,5,6,7,8};
    int i, j, k;
    for(i=0;i<2;i++)
        for(j=0;j<2;j++)
            for(k=0;k<2;k++)
                printf("%d",arr[i][j][k]);
}

main()
{
    int arr[][3]={1,2,3,4};
    printf("%d %d %d %d %d %d",arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
}

main()
{
    int arr[2][2]={1,2,3,4};
    printf("%p %p\n%p %p",&arr[0][0],&arr[0][1],&arr[1][0],&arr[1][1]);
}

main()
{
    int arr[2][3]={1,2,3,4,5,6};
    printf("%d %d %d",arr[1][2],1[arr][2],*(*(arr+1)+2));
}

main()
{
    int arr[2][3]={1,2,3,4,5,6};
    printf("%d %d %d",arr[1][2],1[arr][2],1[2[arr]);
}

main( )
{
    int a[ ] = {0,1,2,3,4};
    int *p[ ] = {a,a+1,a+2,a+3,a+4};
    int **ptr = p;
    printf("%d %p %p %p %p %p\n",**ptr,&ptr,*ptr,*p,p,a);
}

main()
{
    int a[2][2][2]={1,2,3,4,5,6,7,8};
    printf("%p %p %p\n",a,a[0],a[0][0]);
    printf("%p %p %p\n",a,a[1],a[1][1]);
    printf("%d %d",a[0][0][0],a[1][1][1]);
}

main()
{
    void a,b;
    void *ptr;
```

```
    ptr=&a;
    printf("ptr points to a\n");
    ptr=&b;
    printf("ptr now points to b");
}
```

67.
```
main()
{
    int a=10;
    int* i_ptr=&a;
    void* v_ptr=i_ptr;
    *i_ptr++;
    *v_ptr++;
    printf("The value of objects pointed to by pointers are %d %d",*i_ptr,*v_ptr);
}
```

68.
```
main()
{
    int arr[]={1,2,3,4,5};
    int *ptr=arr;
    ptr=ptr+1;
    printf("The value pointed by ptr is %d",*ptr);
}
```

69.
```
main()
{
    int arr[]={1,2,3,4,5};
    int *ptr1=arr;
    int *ptr2=arr+3;
    printf("The result of ptr2-ptr1 is %d",ptr2-ptr1);
}
```

70.
```
main()
{
    int array[]={1,2,3,4,5};
    int *ptr1=array;
    int *ptr2;
    ptr2=ptr1*2;
    printf("The value of ptr2 is %p",ptr2);
}
```

Multiple-choice Questions

71. Arrays are used to store the elements of
    a. The same type
    b. Different types
    c. Multiple types
    d. None of these

72. Array index in C language starts from
    a. 1
    b. 0
    c. Any integer value
    d. None of these

73. The size specifier in the array declaration must be
    a. An expression
    b. A constant expression
    c. A constant expression of integral type
    d. A constant expression of integral type having a value greater than zero

74. In C language, elements of two-dimensional arrays are stored in
    a. Random order
    b. Column major order
    c. Row major order
    d. None of these

75. The elements of an array are stored in
    a. Contiguous memory locations
    b. Discontinuous memory locations
    c. Randomly allocated memory locations
    d. None of these

76. If one of the operands of subscript operator is of array type, the other operand of the subscript operator can be
    a. An expression
    b. An expression of integral type
    c. An integral constant only
    d. None of these

77. If arr is an array of integers, which of the following expression(s) is equivalent to the expression arr[0]?
    a. *arr
    b. *(arr+0)
    c. 0[arr]
    d. All of these

78. Given the declaration statement int arr[5];, the type of expression arr is
    a. int*
    b. int(*)[5]
    c. int*[5]
    d. None of these

79. Given the declaration statement int arr[5];, the type of expression &arr is
    a. int*
    b. int(*)[5]
    c. int*[5]
    d. None of these

80. Given the declaration statement int arr[5][7];, the linear offset from the beginning of the array to any given element arr[2][3] can be computed as
    a. 2*7+3
    b. 2+3*5
    c. 2*5+3*7
    d. None of these

81. Given the declaration statement int array[3][2][2]={1,2,3,4,5,6,7,8,9,10,11,12};, what is the value of array[2][1][0]?
    a. 3
    b. 5
    c. 7
    d. 11

82. Given the statement int a[8]={0,1,2,3};, the definition of a explicitly initializes its first four elements. Which one of the following describes how the compiler treats the remaining four elements?
    a. The remaining four elements are initialized to zero
    b. It is illegal to initialize only a portion of the array
    c. C standard defines the particular behavior as implementation dependent
    d. None of these

83. In the C language, pointer is
    a. Address of a variable
    b. An indication of the variable to accessed next
    c. A variable for storing address
    d. None of these

84. Which of the following is a derived type?
    a. Pointer type
    b. Array type
    c. Function type
    d. All of these

85. Which of the following is a correct way to declare two integer pointers a and b?
    a. int* a,b;
    b. int *a,*b;
    c. int* a,int* b;
    d. None of these

86. A null pointer points to
    a. No object
    b. Null value
    c. Null character stored at the end of string
    d. None of these

87. Pointer arithmetic cannot be performed on
    a. void pointers
    b. Uninitialized pointers
    c. Dangling pointers
    d. None of these

88. Which of the following conversions is carried out implicitly by the compiler?
    a. Conversion of void pointer to any other pointer type on assignment
    b. Conversion of integer constant zero into null pointer of desired type on assignment
    c. Conversion of pointer of one type to the pointer of another type on assignment
    d. None of these

89. Given the declaration statement int* a[2][3][4];, which of the following definitions and initialization of p is valid?
    a. int* (*p)[3][4]=a;
    b. int ****p=a;
    c. int* (*p)[2][3][4]=a;
    d. None of these

90. Given the declaration statement int const* ptr;, which of the following objects is constant?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given declaration is not valid

91. Given the declaration statement const int* ptr;, which of the following objects is constant?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given declaration is not valid

92. Given the declaration statement int* const ptr;, which of the following objects is constant?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given declaration is not valid

93. Given the declaration statement int const* const ptr;, which of the following objects is constant?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given declaration is not valid

94. In the expression ++*ptr, the value of which entity gets incremented?
    a. ptr
    b The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given expression is not valid

95. In the expression *ptr++, the value of which entity gets incremented?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given expression is not valid

Outputs and Explanations to Code Snippets

26. 41 (If executed using Borland TC 4.5 for Windows or Microsoft VC++ 6.0)
21 (If executed using Borland 3.0 for DOS)

Explanation:

Backward Reference: Refer to the explanations given in Answer numbers 3 and 4.

27. 444 (If executed using Borland TC 4.5 for Windows or Microsoft VC++ 6.0)
222 (If executed using Borland 3.0 for DOS)

Explanation:

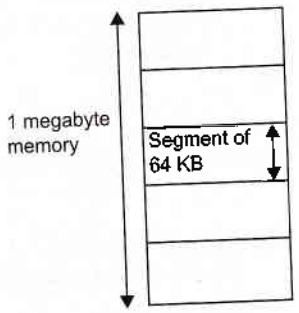
Backward Reference: Refer to the explanation given in Answer number 3.

sizeof operator yields the size of its operand in bytes. The operand can be an expression or parenthesized name of a type. In the given code, the operands of sizeof operators are parenthesized name of the derived types (i.e. char*, int* and float*).

28. 474 (If executed using Borland 3.0 for DOS)

Explanation:

In DOS, the total amount of memory accessible is 1 MB, i.e. 1 megabyte. The entire block of memory is divided into various segments that are 64 K, i.e. 64 kilobytes in size. There are various segments like Code Segment (CS), Data Segment (DS), Extra Segment (ES), etc.
The type of pointer to be used for accessing the memory location depends upon whether the memory location to be accessed lies in the same segment or different segments. If the memory location to be accessed lies in the same segment, the access is called **intra-segment access** and if it lies in a different segment then it is called **inter-segment access.**
If intra-segment access is to be made, pointer of 16-bits is sufficient to refer to all the memory locations (as 2^{16}= 64 K). The 16-bit (2 bytes) pointer that is used for intra-segment access is known as a **near pointer.**
However, if inter-segment access is to be made, the pointer of 16-bits falls short of its memory addressing capability. Hence, a bigger pointer of 32-bits is used to make inter-segment access. The 32-bit (4 bytes) pointers that are used for inter-segment access are known as **far pointers and huge pointers.**



The far pointer contains a 16-bit segment address and a 16-bit offset address (i.e. address within a segment) while the near pointer has only a 16-bit offset address. huge pointers are essentially far pointers but in a normalized form.
The concept of far, near and huge pointers is available in DOS, which has less memory accessible. It is not a part of the C standard and is an extension to the language provided by some of the compilers (e.g. Borland Turbo C 3.0). Refer to the compiler documentation before using these non-standardized qualifiers as they might not be supported by all the compilers (e.g. Borland Turbo C 4.5 and MS-VC++ 6.0 compilers do not support these non-standardized extensions).

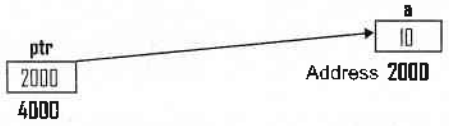
29. Garbage Value 10

Caution:

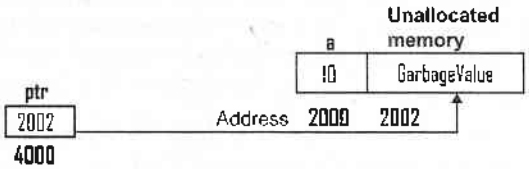
Program may even abnormally terminate.

Explanation:

Suppose variable a gets allocated at the memory address 2000 and variable ptr gets allocated at the memory address 4000. The variable ptr is initialized with the address of variable a. This can be illustrated as:



The arguments of printf functions are evaluated from right to left. So, the expression *ptr++ will be evaluated first and will be interpreted as *(ptr++). Due to post-increment, firstly the value of ptr will be used for the evaluation of expression and then the value of ptr will be incremented. The expression evaluates to 10 and the value of ptr becomes 2002. This can be illustrated as:

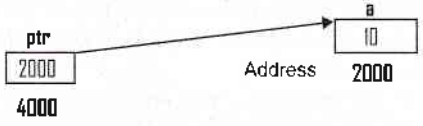


After evaluation of expression *ptr++, the expression ++*ptr will be evaluated. The expression will be interpreted as ++(*ptr). ptr being pointing to an unallocated memory location, i.e. 2002, the behavior of the operation ++(*ptr) is undefined. It will give a garbage value and in extreme cases the program may even terminate abnormally.

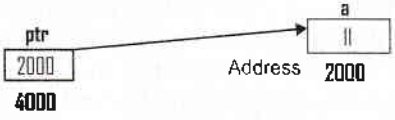
30. 11

Explanation:

Suppose variable a gets allocated at the memory address 2000 and variable ptr gets allocated at the memory address 4000. The variable ptr is initialized with the address of variable a. This can be illustrated as:



The expression ++*ptr will be evaluated first and will be treated as ++(*ptr). This expression makes the value pointed to by the pointer ptr to increment by 1. This can be shown as:

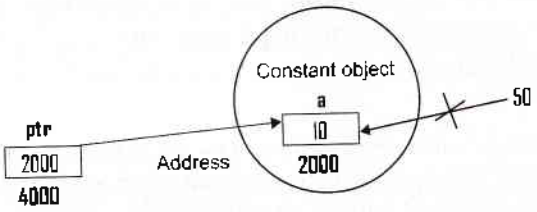


After the evaluation of expression ++*ptr, *ptr++ starts evaluation. The expression *ptr++ will be treated as *(ptr++). Being post-incremented, the value of ptr used for the evaluation of expression will be 2000. The expression evaluates to 11 and the value of ptr becomes 2002.

III. Compilation Error (Cannot modify a constant object)

Explanation:

In the declaration statement const int *ptr=&a;, ptr is declared as 'pointer to a constant integer'. The object to which pointer ptr points is constant and cannot be modified.

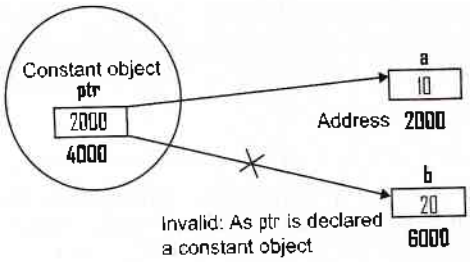


Hence, writing *ptr=50; is not valid and leads to a compilation error.

IV. Compilation Error (Cannot modify a constant object)

Explanation:

The declaration statement int *const ptr=&a; declares ptr as 'constant pointer to integer'. Pointer is constant and must point to the same object throughout. It cannot be made to point to a different object throughout the execution of the program.

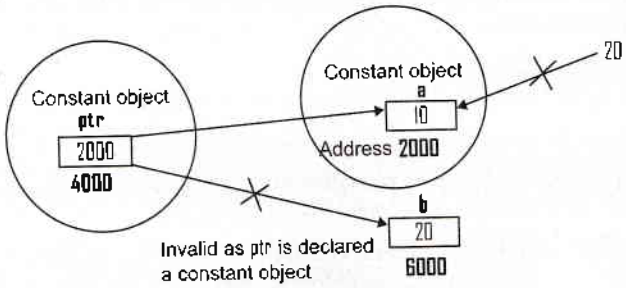


Hence, writing ptr=&b; is invalid and leads to a compilation error.

V. Compilation Error (Cannot modify a constant object)

Explanation:

The declaration statement const int* const ptr=&a; declares ptr as 'constant pointer to a constant integer'. Both the pointer and the object to which the pointer points are constant.



Hence, both the statements *ptr=20; and ptr=&b; are invalid and lead to a compilation error.

34. Compilation error (Cannot convert int to int*)

Explanation:

An integer value 10 is assigned to a pointer variable of type int*. This is not a standard conversi and the compiler will not be able to carry it out implicitly and gives a compilation error 'Ca not convert int to int*'. The error can be removed by explicitly type casting int to int* by writi int*ptr=(int*)10;. However, this conversion is not recommended.

35. The value of pointer is 0000:0000

Explanation:

The conversion of integer value zero to pointer type is standard conversion and is carried out i plicitly by the compiler. When a variable or expression of pointer type is initialized, assigned compared with 0, the constant 0 is implicitly converted into correctly typed null pointer. Hen there will be no compilation error as in Question number 34.

36. ptr1 becomes a NULL pointer

Explanation:

The integer constant zero is implicitly converted to null pointer of the correct type. NULL predefined macro that specifies null pointer value. Hence, in the given code both ptr1 and become null pointers. As two null pointers always compare equal, the if condition evaluate true and 'ptr1 becomes a NULL pointer' gets printed.

Forward Reference: Macros and symbolic constants (Chapter 8).

37. Compilation Error (Size of 'arr' is unknown)

Explanation:

Memory to an array is allocated at the compile time. To allocate the memory, the compiler shoul able to determine the number of bytes to be allocated. To determine it, the compiler needs to kno

1. The element type of array
2. The size of array

The size information can be provided by giving:

1. Size specifier (it should be a compile time constant expression), and/or
2. Initialization list (number of initializers in the initialization list determines the size of ar

Since in the declaration statement int arr[]; both the size specifier and the initialization list are given, the compiler will not be able to determine the number of bytes to be allocated to arr. leads to a compilation error.

38. Compilation error (Constant expression required)

Explanation:

Backward Reference: Refer the explanation given in Sections 4.2 and 4.3.

Although, size is initialized with a literal constant, size itself is a non-constant object (i.e. variable). Access to its value can only be accomplished at the run time, so it is illegal to use an array size specifier. To remove this error, make size a qualified constant by using const qual and write it as const int size=3; instead of int size=3;.

39. 1 2 3

Explanation:

The compiler us initialize the arr

40. Compilation error (To

Explanation:

The number of i fier. They can be

41. 1 2 3 4 0 0

Explanation:

If the number of leading array loc izers. The rest of array) and '\0', i.

42. 2 3 Garbage Value

Explanation:

In C language, a However, if the compilation erro accessed with an program may ter

43. 1 2 10

Explanation:

To access an arra used with a subs 0,1,2 is an expressi tion of a comma assigned to arr[2].

44. 1 2 3 10 5

Explanation:

Backward Ref

45. Compilation error (illeg

Explanation:

An expression of valid and leads to

46. The number of element

Explanation:

When the sizeof op of bytes allocated one element of th

39. 11

Explanation:

The compiler uses the initializers in the initialization list to determine the size of the array and to initialize the array locations.

40. Compilation error (Too many initializers)

Explanation:

The number of initializers in the initialization list cannot be more than the value of the size specifier. They can be less than or at most equal to the value of the size specifier.

41. 11400

Explanation:

If the number of initializers in the initialization list is less than the value of the size specifier, the leading array locations equal to the number of initializers get initialized with the values of initializers. The rest of the array locations get initialized to 0 (if it is an integer array), 0.0 (in case of float array) and '\0', i.e. null character (if array is of character type).

42. 11 Garbage Value

Explanation:

In C language, an array index starts from 0 and the maximum value of the valid index is size-1. However, if the index value greater than the maximum valid value is used, there will be no compilation error as C language does not provide an array index out-of-bound check. If array is accessed with an out-of-bound index, the result will be a garbage value. In the extreme case, the program may terminate.

43. 10

Explanation:

To access an array location, an expression of array type and an expression of integral type are used with a subscript operator. In the statement arr[0,1,2]=10;, arr is an expression of array type and 0,1,2 is an expression of integer type. The expression 0,1,2 evaluates to 2 as the result of the evaluation of a comma operator is the result of evaluation of the right-most sub-expression. Hence, 10 is assigned to arr[2].

44. 11105

Explanation:

Backward Reference: Refer to the explanation given in Answer number 43.

45. Compilation error (illegal use of floating point)

Explanation:

An expression of float type cannot be used with a subscript operator. Hence, writing a[2.5+1.5] is not valid and leads to a compilation error.

46. The number of elements in array are 4

Explanation:

When the sizeof operator is applied on the operand of an array type, the result is the total number of bytes allocated to the array. So, sizeof(array) results in 8. However, sizeof(array[0]) gives the size of one element of the array, i.e. 2. Hence, sizeof(array)/sizeof(array[0]) results in 4.

47. Compilation Error (L-value required error)

Explanation:

The name of an array refers to the address of the first element of an array and does not have a modifiable l-value. Since it does not have a modifiable l-value, it cannot be placed on the left side of the assignment operator. Hence, writing brr=arr is not valid and leads to a compilation error.

48. Contents of array arr and brr are not same

Explanation:

Suppose, the array arr gets allocated at the memory location 2000 and brr gets allocated at the memory location 4000. This is shown in the figure below:

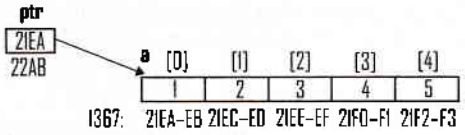
Since, the name of the array refers to the address of the first element of the array, arr refers to [illegible] and brr refers to 4000. The addresses are not equal (in fact they can never be) and hence, the printf statement of the else body gets executed to produce the mentioned result.

49. 11
1367:21EA 1367:21EA

Explanation:

Backward Reference: Refer to the explanations given in Answer numbers 15 and 20.

The name of type 'array of type T' is implicitly converted to pointer of type 'pointer to T' (with two exceptions). The pointer refers to the address of the first element of the array. Hence the initialization statement int *ptr=a; initializes ptr with the address of the first element of the array.



Therefore, *a and *ptr give the value of the first element of the array, a and ptr give the address of the first element of the array (i.e. base address of the array).

The mentioned addresses are in hexadecimal number system.

50. Compilation error (L-value required error)

Explanation:

Backward Reference: Refer to the explanation given in Answer number 15.

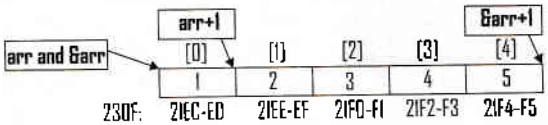
The name arr refers to the address of the first element of the array and &arr refers to the base address of the entire array. Both arr and &arr are constant objects and do not have a modifiable l-value. The increment operator can operate only on operands that have a modifiable l-value. Hence, the expressions ++arr and ++&arr are erroneous.

230F:21EC 230F:21EC
230F:21EE 230F:21F6

Explanation:

Backward Reference: Refer to the explanation given in Answer number 15.

Both the expressions arr and &arr refer to the starting address of the array arr, say 230F:21EC. The expression arr+1 evaluates to 230F:21EE as the type of arr is int* and the sizeof(int*) is 2. The expression &arr+1 evaluates to 230F:21F6 as the type of arr is int(*)[5] and the sizeof(int(*)[5]) is 10.



1111

Explanation:

All the expressions *a, *(a+0), *(0+a), a[0] and 0[a] are equivalent and refer to the first element of array, i.e. 1.

33

Explanation:

Both the expressions arr[2] and ptr[2] are equivalent to *(arr+2) and *(ptr+2), respectively. Since arr has been assigned to ptr, *(ptr+2) is equivalent to *(arr+2), i.e. 3.

23465

Explanation:

Since initializers in the initialization list of an integer array are of float type, the initializers will be demoted before initializing array locations. As ptr is initialized with arr, it points to the first element of the array arr. During every iteration of the loop, the value of element pointed to by ptr is printed, and ptr is made to point to the next element of the array arr. In this way, the entire array gets printed.

0 20 30 40 50

Explanation:

Initializers in the initialization list can be pre-defined variables. Hence, writing int arr[]={10,j,30,40,50} is valid as j is a predefined variable having a value of 20.

123400

Explanation:

As the number of initializers in the initialization list is less than the value of the size specifier, the leading array locations equal to the number of initializers get initialized with the values of initializers. The rest of the array locations get initialized to 0. Since multi-dimensional arrays in C are stored in row major order, elements of the array are initialized row by row. Thus, the contents of initialized array will be: