STRUCTURED PROGRAM DEVELOPMENT USING C PROGRAMMING LANGUAGE

WORKBOOK

WORKBOOK STRUCTURE

WORKBOOK STRUCTURE

WORKBOOK STRUCTURE

WORKBOOK STRUCTURE

WORKBOOK STRUCTURE

WORKBOOK STRUCTURE

Lesson Objectives
In this Lesson you will learn :   - *the Genesis of C Language*   - *the Stages in the Evolution of C Language*

THE STAGES IN THE EVOLUTION OF C LANGUAGE

C has evolved through various stages and has been standardized by the American National Standards Institute (ANSI). This standardized version of C is called ANSI C.

The Genesis

C is middle-level language having features of both second and third generation languages. C is originated from Basic Combined Programming Language (BCPL) at Bell Laboratories. Dennis Ritchie, a systems programmer at Bell Laboratories, was primarily involved in the development of C.

Addition of Basic Data Types

In 1971, C was in its embryonic stage. The version of C in its embryonic stage offered the two basic data types, `int` and `char`, representing integers and characters, respectively.

Addition of Arrays of integer and character data types

The version of C in its embryonic stage also allowed the use of arrays of integers and characters. An array is a group of items that are of the same data type and share a common name.

For example, the employee IDs of the employees of an organization can be stored as an array of integers. A group of characters, such as the name of an employee, can be stored as an array of characters. An array of integers or characters has a fixed size.

Addition of Pointer data types

To implement arrays in situations where the size of an array can change, a pointer to an integer and a character was introduced in embryonic C. A pointer to an integer or a character is a variable that contains the memory address where the integer or the character is stored. Therefore, a pointer allows you to access memory locations directly.

Introduction of Logical Operators

In 1972, C was in its neonatal stage. The two logical operators AND `(&&)` and OR `(||)` were introduced in neonatal C. The logical operators allow you to combine two or more conditions that compare values of variables.

Introduction of Preprocessor Directives

In early 1973, the preprocessor was introduced in C. The preprocessor is a utility that identifies and manipulates some special statements in a program called the preprocessor directives.

Introduction of Portable I/O Package

In addition to the preprocessor, a portable I/O package was developed by Mike Lesk. This portable I/O package was later reworked to provide the standard I/O functions in C.

Introduction of Casting

In 1977, a notation for type conversion called cast was invented. You can use cast to convert the data from one type to another. For example, you can convert a real number to an integer by using casts.

The White Book of C by K & R

In 1978, the first description called The C Programming Language was published by Brian Kernighan and Dennis Ritchie. This book is also called K&R C or the White Book. It served as the C language reference.

The Widespread of C Compilers

During the 1980s, the use of C became widespread. Therefore, various vendors came up with different compilers for almost every computer architecture and operating system. Different vendors implemented C differently in their compilers.

Standardization of C

Formal standardization was required to consolidate the features of the various version of C. Therefore, the X3J11 committee was constituted by ANSI to standardize C. The goal of the X3J11 committee was to retain the common and existing features of C. The committee also aimed at retaining the features that promote portability of user programs across C-language environments.

During 1989 and 1990, the C language was formally standardized by the X3J11 committee. This standardized version of C was called ANSI C.

In addition of retaining the portability feature, the ANSI version of C extensively described the preprocessor and the C library. Therefore, ANSI C standardized preprocessor directives and the C library functions.

LESSON 2 : STRUCTURE OF A ‘C’ PROGRAM

Lesson Objectives

In this Lesson you will learn :

- *the Structure of a C Program*

THE STRUCTURE OF A ‘C’ PROGRAM

C programs have a specific structure that you should follow for obtaining the desired results. In this lesson, you will learn about the structure of a C program.

C is a case-sensitive language. A ‘C’ program can be divided into two building blocks, preprocessor directives and the `main` function. Study the sample program given below.

Listing 2.1 : Printing a Line of Text

```
1   // Listing 2.1 : Text-printing program
2
3   #include <stdio.h>
4
5   main() // function main begins program execution
6   {
7
8       printf (“This is a sample C program”);
9
10  }
```

Detailed Explanation

Line 1 : Deals with Comments

```
// Listing 2.1 : Text-printing program
```

- The // symbol precedes single-line and multi-line comments. However, for multiline comments, each line of the comment should be preceded by the `//` symbol.
- A comment can also be enclosed within slash-asterisk `(/*)` and asterisk-slash `(*/)` in single-line and multiline comments. The beginning and end of a comment are represented by `/*` and `*/`, respectively.
- Comments are optional in a program because they are ignored by the compiler during compilation. You can use comments in a program for documentation and clarity. It is a good programming practice to place appropriate comments in your programs for future reference.

Line 3 : Deals with Preprocessor Directive

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

- The first block contains preprocessor directives. A preprocessor directive is a collection of special statements that are executed at the beginning of program compilation.
- Compilation is the process of translating of C program into its machine-language equivalent. Compilation is done by a piece of software called a compiler.
- A preprocessor directive begins with the hash `(#)` symbol. The highlighted preprocessor directive is the `#include` directive contains a file name, which is enclosed in angle brackets, next to `#include`.
- The angle brackets in a `#include` directive indicate that the specified file is located in a directory, `include`, which is a part of the C software files.
- Quotation marks can be used instead of angle brackets in a `#include` directive. Quotation marks indicate that the specified file in the `#include` directive could be located in either the current directory or the directories where the C software files are stored.
- The file specified in a `#include` directive usually has the extension `.h`. A file having the extension `.h` represents a header file. The file `stdio.h` is an example of a header file. Header files are precompiled.
- The `stdio.h` header file contains the representations of functions called prototypes that are defined in the C library. The functions defined in the C library for the `stdio.h` header file carry out standard I/O operations.
- The preprocessor identifies a preprocessor directive. The preprocessor inserts the contents of the header file into the program at the point where the preprocessor directive is written.
- A preprocessor directive is applicable to the code that follows it. Therefore, it is preferable to place the preprocessor directive at the beginning of a program.
- There can be more than one preprocessor directive in a program. In such a situation, each directive must be written on a separate line.

Line 5 : Deals with `main` *function*

```
main() // function main begins program execution
```

- The second block of a C program contains the main function definition. A C program cannot run without a `main` function. The `main` function block contains the function name, which is `main`, followed by a pair of parentheses and a pair of braces.
- The `main` function definition that is displayed on the screen does not accept any value. However, the function returns the value 1 or 0 to the operating system depending on the successful or erroneous execution of a program.

Lines 6 and 10 : Opening and closing the `main` *function*

- The set of braces that follows the `main` function name contains the statements of the `main` function. These statements constitute the body of the `main` function. Usually, a C statement ends with a semicolon.

Line 8 : Deals with displaying the message

```
printf (“This is a sample C program”);
```

- The body of the `main` function may contain calls to user-defined functions are usually placed after the closing brace of the `main` function. The highlighted statement is a call to the library function `printf`.

In a nutshell, C program contains preprocessor directives followed by the `main` function block. To ensure that the desired results are obtained, write the programs that conform to this structure.

LESSON 3 : PROGRAM DEVELOPMENT CYCLE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

PROGRAM DEVELOPMENT LIFE CYCLE

Consider an example in which you have to create an application in C. This application calculates the gross salary of the employees of an organization. To create an application, you follow the program development cycle.

The program development cycle has four steps: C*reating, Compiling, Linking, and Executing* a program.

- Create
- Compile
- Link
- Execute

Step 1 : Create

The first step in the program development cycle is writing the source code.

You write source code in C by using a text editor. A text editor is a software application that is used to create and modify text. The source code written using a text editor contains the preprocessor directives and the main function block that has statements specific to a program. After creating source code, save it as a file with the extension `.c`. The syntax used to write source code is decipherable by humans. However, computers can read only machine code. Therefore, source code is meaningless to a computer.

Step 2 : Compile

To make source code readable by a computer, source code has to be translated to object code. This translation process is called compilation and is performed by a program called a compiler. The header files specified in the `#include` directive contain the prototypes of C library functions. A prototype specifies information such as the arguments of a function. The header files need to be simply added to object code because C library functions are precompiled. The header files specified in the `#include` directive contain C library function prototypes.

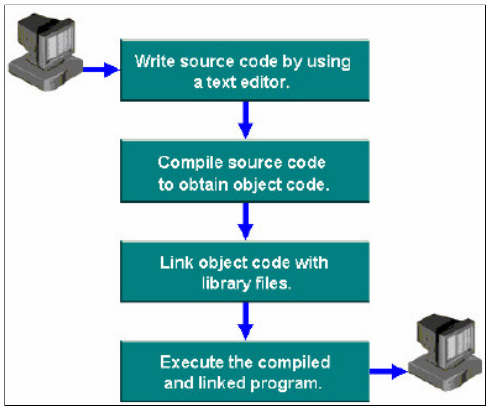
Step 3 : Link

The process of adding the precompiled library files to the object code is called linking. This process is performed by a program called Linker. After linking, an executable file is created, which is ready for execution. You can combine linking with compilation using a single command to both compile the source code and link the object code.

When compiling and linking are combined, first, object code is generated from source code. Next, the contents of header files are added to object code to create an executable file.

Step 4 : Execute

Source code that has been compiled and linked is ready to be run. The process of running a program is called execution. A program is executed by entering the name of the executable file at the command prompt and pressing Enter.


Write source code by using a text editor.
Compile source code to obtain object code.
Link object code with library files.
Execute the compiled and linked program.

You can obtain from a program by executing it. If there are any syntactic or logical errors, repeat the program development cycle from the first step onward. You must repeat the cycle until the desired result is obtained.

Self Review Exercise 2.1

1. The function of a compiler is to:

A. Translate source code is object code
B. Combine code of library functions with object code
C. Translate object code to source code
D. Translate source code *for library functions.*

LESSON 4 : COMPILING A C PROGRAM

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

COMPILING A C PROGRAM

After writing source code in C, the source code is compiled to its machine-language equivalent. The compiled code can then be executed to obtain the output. Here, you learn to compile C source code by using the gcc compiler of the GNU foundation.

GNU represents the recursive acronym for GNU's Not Unix. The gcc compiler and other related files are located in the djgpp subdirectory in drive C. To use the gcc compiler, you set the DJGPP variable. In addition, the path of the gcc compiler should also be set.

DJGPP represents DJ's GNU programming platform and DJ represents DJ Delorie who is the originator and principal maintainer of DJGPP. The DJGPP variable is set by using the details specified in the file djgpp.env, which is located in the djgpp subdirectory.

The DJGPP variable and the path of the compiler can be set as follows:

```
set DJGPP=:\djgpp\djgpp.env
set PATH=C:\djgpp\bin;%path%
```

The source code contained in a file try.c is displayed on the screen. This code is written to print the message 'This is C programming'. The file try.c is located on drive C.

Program to display a custom message

Listing 4.1 : Display a custom message

```
// Listing 4.1 : The sample code will display the message “This is C programming.”

#include <stdio.h>

main()

{

printf (“This is C programming. “);

}
```

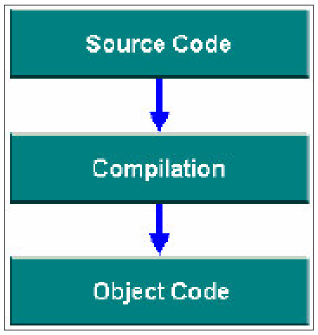
Steps to Compile a Program

- The command used to compile the file try.c is gcc try.c –o try. The command denotes that the source code contained in the file try.c is to be compiled. The –o try option in the command specifies that the executable file is also named try.

- To compile the source code contained in the file `try.c`, type `gcc try.c -o try` and press Enter.
- The source code contained in the file `try.c` has been compiled. This indicates that the `try.exe` file has been created. To run the executable program, type `try` and press Enter.
- The output of the source code contained in the file `try.c` is displayed on the screen.

Special Note :

- *Compiling enables you to translate source code into object code.*
- *Executing a program enables you to obtain the output of a program.*



LESSON 5 : LEXICAL ELEMENTS OF C

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

LEXICAL ELEMENTS

The basic units of the C programming language are called lexical elements. Lexical elements are used to develop statements, which are used to construct a program.

There are five types of lexical elements:

- Keywords
- Identifiers
- Constants
- Operators
- Punctuators

Keywords

The first type of lexical elements is the group of keywords in C. Keywords are the reserved words in C. If keywords are used for purposes other than those specified, the compiler generates error messages. The words `continue`, `int`, `short`, `sizeof`, `union`, and `while` are some of the reserved words in C. All keywords in C are written in lowercase letters.

Identifier

The second type of lexical elements is identifier. An identifier is a unique name for a variable or a function in a C program. An identifier consists of a series of letters, digits, or special characters. Identifiers are the variable elements of a program. Some examples of identifiers that you can create for variables are `Tax_Rate` and `price`. In addition to the identifiers that you can create, the functions in the C library, such as `printf` and `scanf`, also represent identifiers.

Constant

The third type of lexical elements is constant. You can use constants to name those elements of a program whose values do not change in the entire program. There are various types of constants, such as integer, floating-point, character, and string. Integer constants are numbers without decimal points, such as `17`, `91`, and `523`. Floating-point constants are numbers with decimal points, such as `1.2356`, `3.144`, and `896.234`. Character constants are always written within single quotation marks. Therefore, `‘a’`, `‘b’`, `‘6’`, and `‘z’` are examples of valid character constants.

String constants form another type of constant and include a series of characters enclosed within quotation marks. The sequence of characters can also contain blank spaces. You can create string constants according to the requirements of your program. Some examples of string constants are `“name of student”` and `“color”`.

Operators

The fourth type of lexical elements is operator. Operators are of the types unary, binary, and ternary. These specify evaluations to be performed on one, two, and three operands, respectively. Some operations in C are addition `(+)`, subtraction `(-)`, division `(/)`, multiplication `(*)`, equal to `(==)`, and assignment `(=)`.

Punctuators

The fifth type of lexical elements in C is punctuator. Punctuators indicate syntactic meaning to the compiler but do not specify an operation that evaluates to a value. For example, the punctuator semicolon `(;)` at the end of a line of code indicates that the line is a statement. If punctuators are missing in a program, the compiler generates errors. Some punctuators in C are `{,}`, `(,)`, `[,]`, semicolon, and comma. The punctuators `[,]`, `{,}`, `(`, and `)` should always appear in pairs. For every opening bracket, brace, or parenthesis, there should be a corresponding closing bracket, brace, or parenthesis.

Lexical Elements of C at a Glance

Keywords	continue, int, short, sizeof, union, while
Identifiers	Tax_Rate, price, printf, scanf
Constants	17, 91, 523, 1.2536, 3.144, 869.256, ‘a’, ‘b’, ‘6’,‘z’, ”name of student”, “color”
Operators	+, -, /, *, ==, =
Punctuators	{, }, (,), [,], :, ,

Self Review Exercise 5.1

1. Select the set that contains C keywords.

A. *scanf, rate, and char*
B. *int, while, and count*
C. *char, Int, and While*
D. *int, while, and union*

2. Select the set that includes constants in C.

A. *“Car_Code”, while, 1500*
B. *100.26, year, ‘C’*
C. *10, “name”, 51.236*
D. *time, ‘M’, 45*

LESSON 6 : DATA TYPES IN C

Lesson Objectives
In this Lesson you will learn : • *the Genesis of C Language* • *the Stages in the Evolution of C Language*

DATA TYPES IN C

In C programs, you use characters, whole numbers, or real numbers. To store these varied types of data, C provides different data types. There are four basic data types in the C language: `int`, `float`, `double`, and `char`. The words `int`, `float`, `double`, and `char` are keywords. Therefore, these words should be written in lower case.

Data Types in C at a Glance

`int`	*used to store numbers that do not have decimal points, and they are not fractions. (eg : 24 , 45 etc)*
`float`	*used to store numbers with decimal points (eg : 29.23 , 34.22 etc)*
`double`	*used to store floating-point numbers with double the precision of the* `float` *data type (eg: 29.2323423 etc)*
`char`	*The* `char` *data type is used to store a single character. (eg: 'a' , 'b' , 'c' etc)*

THE INTEGER DATA TYPES IN C

The `int` data type represents an integer. This data type can store both positive and negative whole numbers or integers. Integers do not have decimal points, and they are not fractions. The numbers `123`, `5236`, and `8569` are some examples of integers.

The C compiler allows a specific range of values that the `int` data type can store depending on the number of bytes that an integer occupies. The range of values that the `int` data type stores differs from one computer system to another. If an integer occupies `2` bytes, the range of values that the `int` data type can store is from `-32768` through `+32767`.

Modifiers to the `int` *data type*

You can increase or decrease the maximum value or minimum value stored by the `int` data type:

- To increase the maximum value that the `int` data type can store, use the modifiers `unsigned` or `long`.
- To decrease the range of values stored by the `int` data type use the modifiers `short`.

Modifiers to the* int *data type	***Purpose***
`unsigned`	`Used to increase the maximum value of an integer`
`long`	`Used to increase the maximum value of an integer`
`short`	`Used to decrease the maximum value of an integer`

Modifiers	*Detailed Explanation*
`unsigned int`	• The unsigned modifier allows only positive values to be stored in the int data type. • The `unsigned int` data type can store the values from `0` through `65535.`
`long int`	• The storage of the `long int` data type in memory is two times larger than that of the `int` data type. • The range of values that the `long` data type can store is from `-2,147,483,648` through `+2,147,483,647.` • The `long int` data type can stores an integer in 4 bytes regardless of the number of bytes required to store the `int` data type. However, in systems that store an integer in 4 bytes, the `long int` data type is not effective.
`short int`	• In systems that use 4 bytes to store an integer, the `short int` data type occupies half the memory space that the `int` data type occupies. Therefore, the `short int` data type saves memory.

Combining Modifiers

C allows a combination of modifiers, such as `long unsigned int` and `short unsigned int`. In the hierarchical order of the range of values that the int data type can store, an integer is represented as :

```
short int--> short unsigned int--> int--> unsigned int--> long int-->long unsigned int.
```

Some of examples for combining modifiers are:

`short int`	`unsigned int`
`short unsigned int`	`long int`
`int`	`long unsigned int`

THE FLOATING POINT DATA TYPES IN C

Real numbers are also called floating-point numbers. Some floating-point numbers are 1.23456, 7.321, and 100.0.

The `float` *Data Type*

To store numbers with decimal points, use the `float` data type. The `float` data type stores real number. The modifiers `long`, `short`, `signed`, and `unsigned` are invalid with the `float` data type. However, to store floating-point numbers with precision that is higher that of the `float` data type, use the `double` data type.

The `double` *Data Type*

The `double` data type stores floating-point numbers with double the precision of the `float` data type. The `float` data type stores only 6 digits after the decimal point, but the `double` data type stores 12 digits after the decimal point.

Data type	*Number of Digits After the Decimal Point*
`float`	6
`double`	12

Special Note : Use the `double` *data type only to store numbers that need high precision. The* `double` *data type occupies double the memory as the* `float` *data type.*

`long double`

To increase the range of values stored by the `double` data type, use the modifier `long`. You specify `long double` numbers with the suffix L. The `long double` data type occupies `8` or `10` bytes depending on the system. Therefore, the allowed range of values for the `long double` data type varies from one computer system to another.

In the hierarchical order of the range of values that the `float` data type can store, the floating-point numbers are represented as :

```
float--> double--> long double
```

`float` data type at a glance

Data type	*Number of Digits After the Decimal Point*
`float`	6
`double`	12
`long double`	Depends on the system

THE CHARACTER DATA TYPES IN C

The `char` data type is used to store a single character. The value that the `char` data type stores are enclosed within single quotation marks. Some examples of values that the `char` data type can store are `‘c’`, `‘a’`, and `‘t’`. Values such as `‘c’`, `‘a’`, and `‘t’` are called character constants. Uppercase characters are interpreted differently from lowercase characters because C is a case-sensitive language. You can also store single digits as characters by enclosing them within single quotation marks. Therefore, `‘5’`, `‘8’`, and `‘4’` are valid character constants. The `char` data type has an equivalent integer interpretation. Usually, the `char` data type has a range of values from -128 through +127.

Modifiers to the `char` data type

C allows the use of modifiers `signed` with the `char` data type. The data types `signed char` and `char` are the same. Therefore, the range of values for both the data types is the same. You can change the range of values for the `char` data type. To increase the range of values for the `char` data type, use the modifier `unsigned`. The `unsigned char` data type has a range of values from `0` through `255`.

Modifier	*Range*
`signed`	-128 to +127
`unsigned`	0 to 255

Selecting the correct data types to store values in your programs enables you to conserve memory.

Self Review Exercise 6.1

1. *The memory allocated to store an integer is 2 bytes. The value stored in a variable height is 1234500005. The data type of the variable height should be:*

   A. short int
   B. int
   C. unsigned int
   D. long int

2. *A floating-point number is to be stored in the variable weight. The value of weight has two digits after the decimal. The data type of the variable weight should be:*

   A. float.
   B. double.
   C. long double.
   D. long float.

LESSON 7 : THE NAMING RULES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*

THE NAMING RULES

Some of the Lexical Elements of C like the identifiers and constants are governed by the naming rules and conventions.

The Variable Naming Rules

The different values that are used in an operation can be represented by a variable. As the name suggests, the value of a variable can change during the execution of a program. Variables provide you with a means of storing and referring to values repetitively in a program.

The Naming Rules for Variables are :

- A program can have a number of variables. Variables are named according to the naming rules for variables. According to the first rule, the names of all variables must begin with a letter. However, to represent system variables, you start a variable name with an underscore.
- According to second rule, the first character may be followed by a series of letters or digits.
- Third, uppercase letters are treated differently from lowercase letters. Therefore, the variable names tag, Tag, and TAG represent different variables.
- Fourth, keywords cannot be used as variable names. Therefore, the reserved word int cannot be used as a variable name.
- Finally, special characters, such as a blank space, a period (.), a semicolon (;), a comma (,), and a slash (/), are not allowed in variable names.

Some Examples of Variable Names

Valid Variable Names	*Invalid Variable Names*	
`Profit`	`union`	`This is a keyword.`
`taxrate`	`#category`	`Hash is a special character.`
`rate99`	`last name`	`A blank space is not allowed`
`emp_code12`	`address.`	`Period is a special character.`
`quant2005`	`$range`	`Dollar is a special character`

Variables enable you to store and reuse values of various data types in your C programs. If you use variables, you need not type long cumbersome values repeatedly.

The Constants Naming Rules

As the name suggests certain values do not change during the program execution. These are called constants.

The Naming Rules for Constants are :

- All rules applicable for naming variables are applicable for naming constants
- Conventionally, constants are named in uppercase letters.
- `PI` and `MAX_RANGE` are same examples of constants.

LESSON 8 : DECLARING VARIABLES AND CONSTANTS

Lesson Objectives

In this Lesson you will learn :

- *the Structure of a C Program*

VARIABLES AND CONSTANTS --> THEIR PURPOSE

Variables and constants make C programs flexible and programmer-friendly. Consider a program that calculates the installment payable at a specific rate of interest on different loan amounts. The amount of the loan varies from person to person. Therefore, the amount should be represented by a variable. However, the rate of interest remains the same in the program and can be represented by a constant.

The Variable Declaration

- Before using a variable or a constant in a program, you must declare it.
- A declaration associates a variable with a specific data type.
- You must declare all variables before using them in an executable statement.
- A declaration consists of a data type followed by one or more variable names and ends with a semicolon.

The declaration for two float variables named amount and total is given below.

```
float amount, total;
```

In variable declaration, an initial value can be assigned to the variable. This means that you can declare a variable and simultaneously assign a value to it. The declaration that assigns an initial value to variable consists of a data type followed by a variable name, the equal to symbol, and an appropriate value for the data type.

For example, an `int` variable should be assigned only an integer. This means that the `int` variable `var1` cannot be assigned the value `10.5`. This is because the `int` data type stores only whole numbers.

In systems that use `2` bytes of memory to store an integer, assigning an integer to a `float` variable is a waste of memory. This is because the `float` data type needs `4` bytes of memory. A variable declaration statement is terminated with a semicolon. A statement that assigns the values `55.75` and `100.55` to the `float` variables `amount` and `total`, respectively, is shown below.

```
float amount = 5.75, total = 100.55;
```

Consider an example in which a floating –point number, which has `12` digits of precision, occurs `10` times may increase the chances of introducing an error in your source code. Use a constant in such a solution. The declaration of a constant is similar to that of a variable. The floating-point constant `CLASS` has the value `12.123456`. The declaration of `CLASS` is given below.

```
constant CLASS = 12.123456;
```

The declaration of a variable enables you to use different values for an entity in a program. The declaration of a constant enables you to use a constant value for an entity in a program.

Self Review Exercise 8.1

1. The rate of interest on a loan sanctioned by a bank depends on the duration of the repayment period. The interest is represented by the identifier rate, and its initial value is 10.5. Identify the declaration and initialization of the rate.

```
A.   float rate=10.5;
B.   int rate=10.5;
C.   rate=10.5;
D.   float rate;
```

Lesson Objectives
In this Lesson you will learn :   • *the Genesis of C Language*   • *the Stages in the Evolution of C Language*

AN EXPRESSION

An expression is a combination of constants, variables, and operators. Every expression has a value. If the expression consists only of a constant, the value of the expression becomes the value of the constant. variables, the value of the expression is evaluated according to the functionality of the operators. In an expression, operators should be correctly placed and keywords should not be used. Valid expression enable you to obtain error-free results.

If an expression consists of :	
a variable	*The value of the expression is the value of the variable*
operators in addition to constants and variables	*The value of the expression is evaluated according to the functionality of the operators*

An Expression can contain

- Constants
- Variables
- Operators

Some examples involving Expressions :
▪ For example, `x` and `y` are two variables. The expression `x+y` has the addition operator. This expression evaluates to the sum of the values assigned to the variables `x` and `y`.   ▪ In another example, the expression `p=r-s` has the subtraction operator, the assignment operator, and three variables `p`, `r`, and `s`. This expression evaluates to the difference of `r` and `s`, which is assigned to the variable `p`. The operators should be correctly placed in an expression. Otherwise, the expression will evaluate to an erroneous value. Consider the expression `x (8y-6)` where `x` and `y` are two variables.   ▪ In an expression `c(8y-6),` an operator between the variable `x` and the opening parenthesis is missing. Similarly, an operator between `8` and `y` is missing. Therefore, the expression `x(8y-6)` is not valid in C. The correct expression is `x*(8*y -6)`. The C keywords cannot form an expression. Therefore, the expression `double*10` is not valid.

Self Review Exercise 9.1

1. Select a valid expression in C.

A. *`sum + 15`*
B. *`100(average)`*
C. *`Double-100.568`*
D. *`$age+16`*

LESSON 10 : IMPLICIT CONVERSIONS

Lesson Objectives
In this Lesson you will learn :   ☛ *the Genesis of C Language*   ☛ *the Stages in the Evolution of C Language*

MIXED EXPRESSIONS

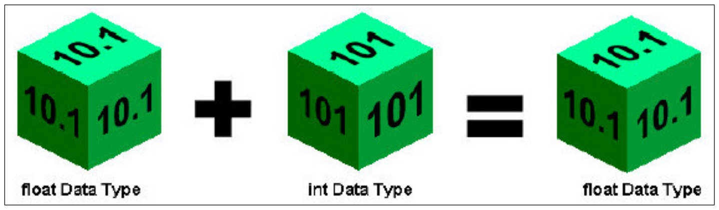
If the operands in an expression are of different data types, the expression is called a mixed expression. In mixed expressions, the type of one operand is converted to that of the other before the expression is evaluated.

IMPLICIT CONVERSIONS

The conversion of the data type of an operand is called type conversion or implicit conversion. Implicit conversion is performed automatically by the compiler. Implicit conversion enables you to mix the variables of different data types in expressions. The type conversion of the operands of different data types is performed according to certain conversion rules.

Usually, the final value of a mixed expression is expressed in the highest precision possible according to the data types of the operands in the expression. The final value in a mixed expression is always expressed in the highest precision.

For example, a mixed expression consists of the addition of a floating-point number and an integer. The result of the expression is expressed as a floating-point number because it has higher degree of precision.



The Implicit Conversions Rules for Expressions are :	
Rule 1	If one of the operands in an expression is of the `long double` data type, other operands of the expression will be converted to `long double` and the result will also be of the `long double` data type. `long double` *Data Type + Any Data Type Converted to* `long double` *DataType* = `long double` *Data Type*
Rule 2	If the highest precision of the operands in a mixed expression is the `double` data type, the other operands of the expression will be converted to the `double` data type. The result will also be of the `double` data type. `Double` *Data Type + Any Data Type Converted to* `double` *Data Type* = `double` *Data Type*
Rule 3	If the highest precision of one of the operands in an expression is of the `float` data type, the other operands will be converted to the `float` data type. The expression will also evaluate to the `float` data type. `float` *Data Type + Any Data Type Converted to* `float` *Data Type* = `float` *Data Type*

Casting

We can explicitly convert the value of an expression to a specific data type. This type of conversion is called casting, and the type of expression is called a cast.

The syntax to cast an expression:

```
(data type) expression
```

Limitations of Casting

If you use casting to convert a value of higher degree of precision to a value of lower degree of precision, you may lose data. This means that the original value might be truncated. For example, in the expression `(x+y)`, `x` is an integer variable having the value `7` and `y` is a floating-point variable having the value `8.5`. The cast expression converts the sum of `x` and `y` to an integer value. The sum of `x` and `y` is `15.5`. The floating-point value `15.5` becomes `15` when it is cast as an integer. Therefore, the expression `(int)(x+y)` evaluates to `15`.

Self Review Exercise 10.1

1.In the expression `length*breadth`, length is of the int data type and breadth is of the float data type. What will be the data type of the result of the expression?

A. float
B. double
C. long float
D. int

LESSON 11 : INPUT FUNCTIONS —> CHARACTER BASED

Lesson Objectives

In this Lesson you will learn :

- *the Structure of a C Program*

CHARACTER - BASED INPUT FUNCTIONS

Consider that the user of your program is to type a single-character input or multiple-character input. Based on character-input the program performs some calculations. For example, the user needs to type a character code to determine the availability of a book. To accept input for a program, one of the character-based input functions can be used. These functions are a part of the I/O functions of the C library.

There are five character - based input functions :

`getchar()`	*accepts buffered character input from the keyboard*
`getch()`	*accepts unbuffered character input from the keyboard*
`getche()`	*accepts unbuffered character input from the keyboard and it echoes the character on the screen*
`getc()`	*accepts a character from a file*
`gets()`	*accepts a sequence of characters with embedded spaces*

The `getchar()` *Function*

The `getchar()` function accepts a single character from the keyboard, as we specify the end of input by pressing Enter because the `getchar()` function accepts buffered input. This means that the input value is stored in a variable only after Enter is pressed. If more than one character is typed before pressing Enter, only the first character is accepted by the `getchar()` function. The rest of the input characters remain in the buffer while the first input character is returned by the `getchar()` function.

Listing 11.1 : Accepts a character using getchar() function

```
// Listing 11.1 : this sample code will accept a value for the variable answer.
// The value accepted as input for the variable answer will also be displayed.

#include <stdio.h>

main()
{
    char answer;

    answer = getchar();

    putchar(answer);
}
```

Special Note : To use one of these input functions in your programs, include the `stdio.h` *file in a* `#include` *directive.*

Detailed Explanation

- Consider the program in which the declaration for the variable `answer` is entered for you. To enable the variable `answer` to accept a character by using `getchar()` function, we have to instruct `answer = getchar();`.
- The character returned by the `getchar()` function is stored in the variable `answer`. The character stored in the variable `answer` can be displayed using the `putchar()` function.
- In addition, the program will also be compiled and executed. If the user typed character `y` when the program was executed.
- The output indicates that the `getchar()` function returns the character that was typed by the user.

The* `getch()` *Function

The `getch()` function accepts a single character from the keyboard. The `getch()` function accepts unbuffered input. This indicates that input is directly stored in a variable. Therefore, there is no need to press Enter to specify the end of input.

Listing 11.2 : Accepts a character using getch() function

```
// Listing 11.2 : The code will accept a value for the variable answer by using the getch
// function. The code will also display the value accepted for the variable answer.

#include <stdio.h>

main()
{
    char answer;

    answer = getch();

    putchar(answer);
}
```

In the example program, user input is to be accepted as a single character. The single-character input should be stored in the variable `answer`, which is of the `char` data type.

Detailed Explanation

- The declaration for the variable `answer` is entered. To accept a character from the keyboard by using the `getch()` function, type `answer=getch();` and press Enter.
- The character returned by the `getch()` function is stored in the variable `answer`. The character stored in `answer` can be displayed using the `putchar()` function.
- The statement for the `putchar()` function to display the value of the variable `answer` is given. In addition, the program will be compiled and executed.
- Let's suppose character `n` was typed by the user when the program was executed. Notice that the user-input will not echoed on the screen.

- The output of the program indicates that the `getch()` function returns the character that was typed by the user.

The* `getche()` *Function

The character-based input function `getche()` also accepts unbuffered input as a single character similar to the `getch()` function. However, input to the `getche()` function is echoed on the screen. The `getche()` function returns the character accepted from the keyboard.

A statement that uses the `getche()` function is given below. The value returned by the `getche()` function is stored in a character variable, `var1`.

```
var1 = getche();
```

The* `getc()` *Function

The `getc()` function is to accept a value for a character variable from a file. The file can be a disk file or the standard input device, which is the keyboard. A statement that uses the `getc()` function is given below. The function accepts the argument `stdin`, which represents standard input. The value accepted by the `getc()` function is assigned to the character variable `choice`.

Listing 11.3 : Accepts a character using getc() function

```
// Listing 11.3 : This code will accept a value for the variable choice by using the
// getc function. The code will also display the value accepted for the variable choice.

#include < stdio.h>

main()

{
    char choice;

    choice = getc(stdin);

    putchar(choice);
}
```

Detailed Explanation

- To accept a value for the variable `choice` by using the `getc()` function, type `choice=getc (stdin);`
- The character returned by the `getc()` function is stored in the variable `choice`. The character stored in the variable `choice` can be displayed using the `putchar()` function.
- The statement using the `putchar()` function to display the value of the variable `choice` is used. In addition, the program will be compiled and executed.
- Let's suppose that the input given is `Y`. Output `Y` indicates that the `getc()` function returns the character that was typed the user.

The `gets()` *Function*

The `gets()` function accepts a sequence of characters with embedded spaces. A sequence of characters with embedded spaces is called a string. The end of the string is specified by pressing Enter.

A statement that uses the `gets()` function is given below. The `gets()` function accepts a single argument, and the argument of the `gets()` function is represented by the variable `str1`. The variable `str1` stores the string accepted by the `gets()` function from the keyboard.

```
gets(str1);
```

Listing 11.4 : Accepts a string using gets() function

```
//Listing 11.4 : The code will accept a string of up to 30 character byl using the gets
// function. The code will also display the string.

#include <stdio.h>

main()

{
    char answer[30];

    gets(answer);

    puts(answer);
}
```

Detailed Explanation

- To accept a value for the variable `answer` by using the `gets()` function, we instruct `gets(answer);`.
- The string returned by the `gets()` function is stored in the variable `answer`. The string stored in the variable `answer` can be displayed using the `puts()` function.
- The `puts()` function to display the value of the variable `answer` is used.
- Let's suppose that the input string be `C Programming`. You will notice that the `gets()` function returns the string that was inputted.

Special Note : The `getchar()`, `getch()`, `getche()`, *and* `getc()` *functions return the character typed by the user, whereas the* `gets()` *function returns the string typed by the user.*

LESSON 12 : INPUT FUNCTIONS —> scanf()

Lesson Objectives
In this Lesson you will learn : ☛ *the Structure of a C Program*

THE INPUT FUNCTION —> scanf()

If a user of your program needs to type the values of any data type during program execution, the scanf() function is used. For example, the user may have to type a bank account number based on which the program displays the balance.

The scanf() function can accept character or numeric input. The scanf() function is a part of the I/O functions in the C library. To use the scanf() function in our program, we need to include the stdio.h file in a #include directive. The scanf() function enables you to type values for variables of any data type during the execution of a program.

The structure of* scanf() *function :	
scanf(control string, &arg1);	▪ The control string contains information about the format in which the value for the variable arg1 should be accepted. ▪ The control string is always enclosed within quotation marks. ▪ It contains the percent (%) symbol followed by a conversion character.
scanf("%conversion character", &arg1);	▪ Conversion characters are special characters that are used to accept values for specific data types. ▪ The conversion character for the exponential notation of a float data type is e. The exponential notation displays a floating-point number with an exponent. For example, the floating-point number 2575.080 in the exponential form is represented as 2.57508e+03. ▪ The conversion character for a string is s. The usage of the ampersand symbol in the scanf() function that uses conversion character s to accept a string is prohibited. The variable name that stores the string itself specifies the address of the memory location. ▪ Conversion characters are usually separated by a single space to differentiate variables. In addition, the argument list of the scanf() function should also contain all the variable separated by commas and prefixed with the ampersand symbol.

Conversion Character	Used for Accepting
c	character
d	integer
f	floating-point Number

The structure of the scanf() function is given below. The arguments of the function are a control string and the variable arg1 prefixed by the ampersand (&) symbol. The ampersand symbol represents the memory address allocated to the variable arg1.

Special Note : The scanf() function accepts a combination of numerical values and single characters from the keyboard.

Listing 12.1 : Reading value into a variable

```
1   // Listing 12.1 : The code will accept a value for the variable price by using the scanf()
2   //function.
3
4   #include <stdio.h>
5
6   main()
7   {
8
9         float price;
10        scanf("%f", &price);
11
12  }
```

Detailed Explanation

- Specify conversion character `f` in the `scanf()` function, as in line 10, because the data type of `price` is `float`. To specify the `scanf()` function that accepts a value for the variable `price`, code `scanf("%f",&price);`.
- You specified a `scanf()` function. This function has conversion character `f` for the variable `price`. This will enable the user to specify a floating-point number for the variable `price` when the program containing this statement is executed.
- The `scanf()` function can accept values for a number of variables simultaneously. To enter values for multiple variables, specify the conversion character for each variable in the control string of the `scanf()` function.
- Consider the variable `x`, `y`, and `z` of the `int`, `float`, and `char` data types, respectively. The statement for entering the values of `x`, `y`, and `z` by using the `scanf()` function is `scanf("%d %f %c", &x, &y, &z);`.

Self Review Exercise 12.1

1. The syntax of the `scanf()` function contains a :

A. Variable name suffixed with the ampersand symbol
B. Conversion character suffixed with the percent symbol
C. Variable name enclosed within quotation marks
D. Control string enclosed within quotation marks

2. Identify the code for the `scanf()` function that accepts values for the variables `Emp_Code` and `age`, which are of the data types `char` and `int`, respectively.

A. `scanf("%e%d", &Emp_Code, &age);`
B. `scanf("%c%d", &Emp_Code, &age);`
C. `scanf("%c%f", &Emp_Code, &age);`
D. `scanf("%d%c", &Emp_Code, &age);`

LESSON 13 : OUTPUT FUNCTIONS —> CHARACTER-BASED

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

CHARACTER - BASED OUTPUT FUNCTIONS

You may have to display single or multiple characters by using a program. The characters that are displayed on the screen can be messages or prompts to the users of the program. For example, in a program, a message has to be displayed to the user about the type of cars that are exclusively available in a showroom. In such situations, use one of the character-based output functions. These functions are a part of the I/O functions in the C library.

There are three character-based output functions.

`putchar()`	*accepts buffered character input from the keyboard*
`puts()`	*accepts unbuffered character input from the keyboard*
`putc()`	*accepts unbuffered character input from the keyboard and it echoes the character on the screen*

To use the character - based output functions in your programs, include the `stdio.h` header file in the `#include` directive.

The `putchar()` *Function*

The `putchar()` function displays a single character. A statement that uses the `putchar()` function is given below. The `putchar()` function accepts the `var1` argument, which is of the `char` data type.

Listing 13.1 : Displaying a value stored in a variable

```
1   // Listing 13.1 this sample program will display the value stored in the variable var1.
2
3   #include <stdio.h>
4
5   main()
6   {
7       char var1= 'y';
8       putchar(var1);
9   }
```

Detailed Explanation

- Line 7 : Declaration for the variable `var1`. The variable `var1` stores the value `y`.
- Line 8 : To specify the code that displays the value stored in the variable `var1` by using the `putchar()` function, code `putchar(var1);`.

The `puts()` *Function*

The `puts()` function is used to display a string.

In the example given below, the variable `str1` is of the `char` data type. The `str1` variable can store a string.

Listing 13.2 : Using puts() function

```
//Listing 13. 2 : this sample code displays a string by using the puts function.

#inclued <stdio.h>

main()
{
       char str1[25] = “c programming”;
       puts(str1);
}
```

Detailed Explanation

- The statement indicates that the `puts()` function accepts one argument, which is the variable `str1`.
- The screen displays the declaration of the variable `str1`. to specify the code that displays the value of the variable `str1`, type `puts(str1);`.
- The output of the compiled program containing the `puts(str1);` statement is given now.

The `putc()` *Function*

The `putc()` function is used to place a character to a stream. A stream can be a disk file or the standard output device, which is the monitor.

The listing displays an example of the `putc()` function. In this example, the `putc()` function accepts two arguments, `var1` and `stdout`.

The compiled program containing the code for the `putc()` function is given below.

Listing 13.3 : Using putc() function

```
//Listing 13.3 :  this sample code will display the value of the variable var1 by using
//putc() function.

#include <stdio.h>

main()

{
    char var1 = ‘F’;
    putc(var1, stdout);
}
```

Detailed Explanation

- The argument `var1` represents the character variable whose value is displayed by the `putc()` function. The argument `stdout` represents the standard output device. The output of the compiled program containing the code for the `putc()` function is given below.
- The `putchar()`, `puts()`, and `putc()` functions enable you to display messages consisting of single or multiple characters.

Self Review Exercise 13.1

1. A string, which is stored in the variable year, is to be displayed on the screen. Select the statement for the puts function that displays the value of the variable year.

A. `year =  puts();`
B. `puts(year);`
C. `puts() = year;`
D. `year = puts;`

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

`printf()` FUNCTION

As a part of a program, data in the form of numbers or a combination of numbers and characters is to be displayed on the screen. For example, you display the balance in the account of a customer who has a specific account number.

To display messages either as number or characters, use the `printf()` function.

The `printf()` function is used to display a combination of numerical values and single or multiple characters on the screen. The `printf()` function copies data to the screen. The `printf()` function returns the number of values or characters it displays.

Structure of the `printf()` function is as follows:

```
printf(control string, arg1);
```

- The control string in the structure of the `printf()` function represents information about the format in which the value of the variable `arg1` is to be displayed.

Conversion characters are special characters that are used to display the values of various data types. Conversion characters for a character, an integer, and a floating-point number are `c`, `d`, and `f`, respectively. The conversion character for a string is `s`. Conversion character `e` displays a floating-point number with an exponent.

Data Type	Conversion Character
`char`	`c`
`int`	`d`
`float`	`f`
`float(with exponent)`	`e`

In addition to the conversion character, the control string of the `printf()` function can also contain certain special characters, such as `\n` and `\t`. These special characters are called escape sequences. The escape sequence `\n` represents the newline character.

The newline character `(\n)` introduces a new line in the output of the `printf()` function that contains the newline character.

The listing displays a program containing the statement `printf("%d\n%d",x,y);`.

Listing 14.1 : A program using printf() function

```
//Listing 14.1: this sample code introduced a new line in the output of the printf()
//statement

#include <stdio.h>
main()
{
    int x = 7,y = 8;
    printf("%d \n %d", x, y);
}
```

Detailed Explanation

- The statement `printf("%d\n%d",x,y);`, enables you to display the values of variables `x` and `y` on two different lines on the screen.
- In the control string of the `printf()` function, the escape sequence `\t` represents the tab. The tab character introduces a fixed number of spaces in the `printf()` function output that contains the tab character.
- In the structure of the `printf()` function, `arg1` represents a variable, a constant, or an expression.

In the following example, the declaration for the `float` variable distance is typed. Use conversion character `f` in the `printf()` function because the variable distance is of the `float` data type.

Listing 14.2 : A program using printf() function

```
//Listing 14.2 : this code will display the value stored in the variable distance by
// using the printf function.

#include <stdio.h>
main()
{
    float distance = 7.5;
    printf("%f", distance);
}
```

Detailed Explanation

- To specify the code of the `printf()` function that displays the value of the variable `distance`, type `printf("%f", distance);`.

You can use the `printf()` function to display more than one value. In the next example , the variables `val1` and `val2` are integers.

Listing 14.3 : A program using a printf() function

```
//Listing 14.3 : this code will display the value stored in the variables val1, val2, and
//the expression val1 + val2 by using the printf function.

#include <stdio.h>
main()
{
    int val1 = 1;
    int val2 = 2;
    printf("%d %d %d', val1, val2, val+val2);
}
```

Detailed Explanation

- Line 4 and Line 5 gives the values of the variables `val1`, `val2`, and the expression `val1+val2`.
- You can use the `printf()` function not only for displaying the values of more than one variable but also for displaying the variables with the preferred number of spaces between them.

Displaying the variables with Spaces between them

The following code has the variables `ICode`, `PartNo`, and `cost` that are of the data types `char`, `int`, and `float`, respectively. The statement shows the values of `ICode`, `PartNo`, and `cost`.

Listing 14.4 : A program using printf() function

```
//Listing 14.4 : The sample code will display the values of variable with spaces between
//them

#include <stdio.h>

main()
{
    char ICode = ’T’;
    int ParNo =15;
    float cost = 15.6;
    printf(“%c %d  %f”, ICode, PartNo, cost);
}
```

Detailed Explanation

- The output of the program containing the `printf()` function has a single space between the values stored in the variable `ICode`, `PartNo`, and `cost`.
- The single space between the values stored in the variables `ICode`, `PartNo`, and cost corresponds to the single space between the conversion characters in the control string.
- You can have as many spaces between the conversion characters in the control string as you need between the values displayed by them in the output of a `printf()` function.

Listing 14.5 : A program using printf() function

```
//Listing 14.5 : the code will display the message “Learn about the printf function.”

#include <stdio.h>

main()
{
    printf(“Learn about the printf function”);
}
```

You can also display a message by using the `printf()` function. To do this, write the message within quotation marks as the argument of the `printf()` function. This function enables you to display numbers and characters in the preferred format on the screen.

Self Review Exercise 14.1

1. *The syntax of the* `printf()` *function contains:*

   A. a ampersand symbol prefix with the variable name.
   B. a conversion character suffix with the percent symbol.
   C. a control string enclosed within quotation marks.
   D. a single argument representing the variable that contain the value to be displayed.

2. *The variable* `Emp_Code` *and age the value* “`D`” *and* 45, *respectively. Select the code for the* `printf()` *function that displays the value of* `Emp_Code` *and* `age`.

   A. `printf(“%C %d”,Emp_Code, age);`
   B. `printf(“%c %d”,Emp_Code, age);`
   C. `printf(“%c %d”,&Emp_Code, &age);`
   D. `printf(“%c %d”,age, Emp_Code);`

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

OPERATORS

Operators define the action to be performed on the operands in an expression. Depending on the number of operands on which an operator acts, operators are classified as unary, binary, and ternary operators.

UNARY OPERATORS

Unary operators act on a single operand. Binary and ternary operators act on two and three operands, respectively.

Minus Operator

The minus operator is represented by the minus (-) symbol. The minus operator acts on a variable or a constant that follows the operator. The minus operator indicates that the value of the operand on which the operator acts is negative. This means that if x is an integer variable having the value 6, -x represents -6.

Increment Operator

The increment operator is represented by the double plus (++) symbol. The increment operator causes its operand to be increased by the value 1. The increment operator can be placed before or after an operand.

Pre-Incrementation

If the increment operator precedes an operand, then the value of the operand will be increased by the value 1 before the operand is used by a statement in a program. This type of increment is called a preincrement. In the example below, the preincrement operation is performed on variable k, which has the value 1. The printf() statements in the program display the values of k before and after preincrementation.

Listing 15.1 : A program to illustrate unary operator usage

```
//Listing 15.1 : the code will display the value of the operand k before and after
//preincrementation.

#include < stdio.h>

main()
{
    int k = 1;
    printf(“%d\n”,k);
    printf(“%d\n”, ++k);
    printf(“%d\n”,k);
}
```

Detailed Explanation

- The initial value of `k` is `1`. The value of `k` in the first `printf()` statement in the highlighted area is `2`. This is because the value of `k` is incremented before the value of `k` is displayed. The second `printf()` statement in the highlighted area displays the final value of `k`.

Post-Incrementation

If the increment operator follows an operand, then the value of the operand will be altered after the operand is utilized in a specific statement in a program. This type of increment is called a postincrement.

In the example given below, the postincrement operation is performed on the variable `j`, which has the value `1`. The `printf()` statements in the program display the values of `j` before and after postincrementation.

Listing 15.2 : A program to illustrate unary operator usage

```
//Listing 15.2 :the code will display the value of the operand k before and after
//postincrementation.

#include < stdio.h>
main()
{
    int j = 1;
    printf("%d\n",j);
    printf("%d\n", j++);
    printf("%d\n",j);
}
```

Detailed Explanation

- The initial value of `j` is `1`. The value of `j` is `1` in the first `printf()` statement. This is because the value of `j` is incremented after the value of `j` is displayed. The second `printf()` statement in the highlighted area displays the final value of `j`, which is `2`.

Decrement Operator

The decrement operator is represented by the double minus `(-)` symbol. The decrement operator causes its operand to be decreased by the value `1`. The decrement operator can be placed before or after an operand to perform predecrementation or postdecrementation.

The operation of the decrement operator is similar to the increment operator. However, the effect of the decrement operator is to decrease the value of the operand on which it acts.

Self Review Exercise 15.1

1. *The value stored the variable* `total` *is* 4. *The value of the expression* `100+(++total)` *is:*

   A. 104
   B. 105
   C. 106
   D. 107

LESSON 16 : ARITHMETIC OPERATORS

Lesson Objectives

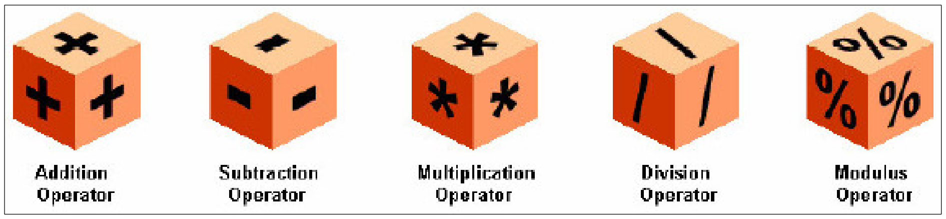
In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ARITHMETIC OPERATORS

For performing arithmetic operations in your programs, use arithmetic operators in expressions. The expressions that involve these operators are called arithmetic expressions. To obtain the required result, you follow a specific order and sequence in evaluating arithmetic expressions.

There are five arithmetic operators: `+`, `-`, `*`, `/`, and `%`, which represent addition, subtraction, multiplication, division, and modulus operations, respectively.



The operands that are acted upon by arithmetic operators must have numeric values. Therefore, operands can be integers or real numbers.

Points to remember :

- When the division of two integers results in a fraction, the fraction is truncated. For example, the expression `3/2` results in the value `1.5`. However, the compiler returns the result of `1` because the result of the division of two integers is an integer.
- Division of two integers that result in a fraction truncates the function.
- The modulus operation returns an integer remainder after the division of two integers. The modulus operator can be used only with integers and not with floating-point numbers.
- Use the modulus operator only with integers.

Consider an example of two floating-point variable, `a` and `b`. The variables `a` and `b` have the values `5.5` and `3.25`, respectively. The expressions `a+b` and `a-b` evaluate to `8.75` and `2.25`, respectively.

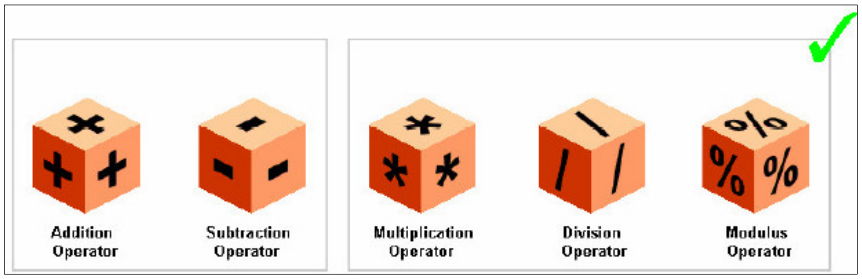
Input Variables	Expression	Output Value
`a=5.5, b=3.25`	`a+b`	`8.75`
`a=5.5, b=3.25`	`a-b`	`2.25`

In another example, the variables `x` and `y` have the values `9` and `3`, respectively. The expressions `x*y` and `x/y` evaluate to `27` and `3`, respectively. If the values of `x` and `y` are `10` and `3`, respectively, the expression `x%y` evaluates to `1`.

Input Variables	Expression	Output Value
`x=9, y=3`	`x*y`	`27`
`x=9, y=3`	`x/y`	`3`
`x=10, y=3`	`x%3`	`1`

Precedence of Arithmetic Operators

The arithmetic operators have a specific order of evaluation. This order of evaluation is called precedence. The multiplication, division, and modulus operators belong to one precedence group.

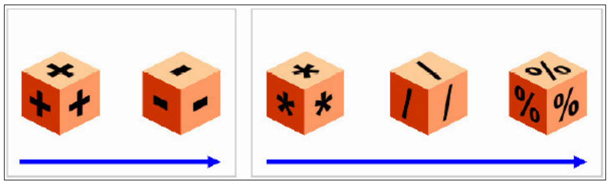


The addition and subtraction operators belong to another precedence group. The group of the multiplication, division, and modulus operators has precedence over the addition and subtraction operators.

Associativity of Arithmetic Operators

When evaluating arithmetic expressions, the order of consecutive operations within the same precedence group is called associativity.

Within each precedence group, the associativity of arithmetic operators is from left to right. Therefore, consecutive addition and subtraction operations and consecutive multiplication, division, and modulus operations are performed from left to right.



- For example, the expression `15+6-3` contains the addition and subtraction operators. Both the operators belong to the same precedence group, and their associativity is left-to-right.
- In the expression `15+6-3`, addition is performed first. Next, subtraction is performed. Therefore, the expression evaluates to the value `18`.
- The consecutive multiplication, division, and modulus operations are carried out from left to right. In the example `12/4*2`, division is performed first and then multiplication. Therefore, the expression evaluates to `6`.
- You can after the precedence of arithmetic operators by using parentheses. Parentheses have higher precedence over all arithmetic operators. You can also have nested parentheses in an expression.
- In expressions having nested parentheses, the innermost parentheses have precedence over the outer parentheses. In the expression, the addition of `x` and `y` is carried out first and the sum is then multiplied with the value `8`.

```
(8 * ( x + y ) )
```

- The expression displayed on the screen may appear to evaluate to `9`. however, according to the precedence of arithmetic operators, the multiplication and division operators have higher precedence over the addition operator.

```
1 + 2 * 6 / 2
```

- In addition, arithmetic operators associate from left to right. Therefore, the expression `1+2*6/2` is equivalent to the highlighted expression. This expression actually evaluates to `7`.

```
1 + 2 * 6 / 2 is equivalent to 1 + ( ( 2 * 6) / 2)
```

Self Review Exercise 16.1

1. *In the expression* `var1 % var2` *are integers. The modulus operator divides* `var1` *by* `var2` *and returns the :*

A. Quotient as an integer
B. Quotient as a floating - point number
C. Remainder of the Integer division
D. Remainder as a floating - point number

Lesson Objectives
In this Lesson you will learn : • *the Genesis of C Language* • *the Stages in the Evolution of C Language*

To assign a value to a variable, use the assignment operator. For example, a variable, `qty`, stores the maximum count of a product in a supermarket. To set the value of the variable `qty` to `10`, the assignment operator is used.

The assignment operator is represented by the equal to `(=)` symbol. An expression that contains an assignment operator is written in the form `var = expression`. The variable `var` stores the value of the expression.

- The expressions `age=25`, `area=50.236`, and `total=a+b` use the assignment operator.
- In `age=25`, `25` is assigned to age.
- In `area=50.236, 50.236` is assigned to `area`. In `total=a+b`, the sum of `a` and `b` is assigned to `total`.
- Such an assignment is called single assignment.

In addition to single assignment, multiple assignments are also possible in C. This means that two or more variables can be assigned the same value by using the assignment operator. The following expression assigns the value `10` to the variable `var1` and `var2`.

```
variable1 = variable2 = expression
var1 = var2 = 10
```

In addition to the assignment operator, C has other assignment operators, such as compound assignment operators. These are `+=`, `-+`, `*=`, `/=`, and `%=`. The compound assignment operators enable you to write short assignment expressions.

Compound Assignment Operators
+=
-=
*=
/=
%=

The assignment expression given below for variable, `var1`, and an expression, `exp`, contains the compound assignment operator for addition. The expression is equivalent to `var1 = var1+exp`.

```
var1 +=exp
```

The assignment expression given below for the variable `var1` and the expression `exp` contains the compound assignment operator for subtraction. The expression is equivalent to `var1 = var1-exp`.

```
var1 -= exp
```

The following expression contains the assignment operator for multiplication. This expression is equivalent to `var1 = var1*exp`.

```
var *=exp
```

The following expression contains the assignment operator for division. This expression is equivalent to `var1 = var1/exp`.

```
var1 /=exp
```

The following expression contains the assignment operator for modulus. This expression is equivalent to `var1 = var1 % exp`.

```
var1 %=exp
```

Assignment operators have a right-to-left associativity. This indicates that in the expressions that contain assignment operators, the expression is evaluated from right to left.

Consider two integers, `a` and `b`, that have the values `1` and `5`. In the expression that is given, evaluate `a+3` first. Next, subtract the sum of `a` and `3` from the value of `b`. Therefore, the expression evaluates to `1`.

```
a=1   b=5
b -= a+3
```

LESSON 18 : RELATIONAL OPERATORS

Lesson Objectives
In this Lesson you will learn :   • *the Genesis of C Language*   • *the Stages in the Evolution of C Language*

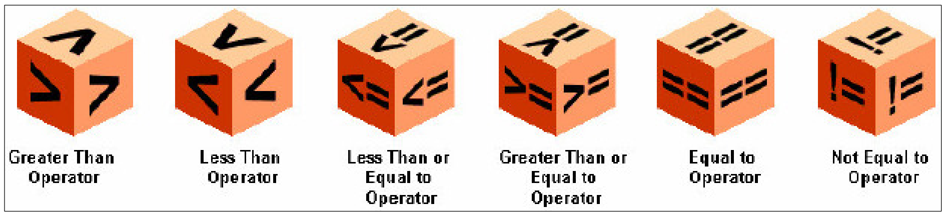
RELATIONAL OPERATORS

In C, you can use relational operators for comparing conditions. For example, a program calculates the grades of the students in a class. The `grade` of a student is calculated according to the `score` obtained by the student.

- If the `score` is below `75`, a student is awarded grade `B`.
- If the `score` is above `75`, the student is awarded grade `A`.
- To determine the `grade` of a student, the `score` of the student has to be compared with `75`.
- If the `score` of a student is `60` and is stored in the variable `student_score`, the expression, `student_score < 75`, compares to the value of `student_score` with `75` by using a relational operator.

An expression containing a relational operator is either `true` or `false`. Relational operators take two operands and yield either the integer value `1` representing a `true` condition or the integer value `0` representing a `false` condition.

There are six relational operators in C: greater than `(>)`, less than `(<)`, less than or equal to `(<=)`, greater than or equal to `(>=)`, equal to `(==)`, and not equal to `(!=)`.



The equal to and not equal to operators are called equality operators.

Consider the two integers, `i` and `j`, having the values `5` and `10`, respectively. The expression `(i < j)` evaluates to `1` because `5` is less than `10`. Therefore, the value of the expression `(i<j)` is true and the integer value of true is `1`.

```
 i=5      j=10
(i < j) evaluates to 1
```

For the floating-point variable `var1` that has the value `12.5`, the expression `(var1>100)` evaluates to 0 because `var1` is less than `100`. Therefore, the expression `(var1>100)` is false.

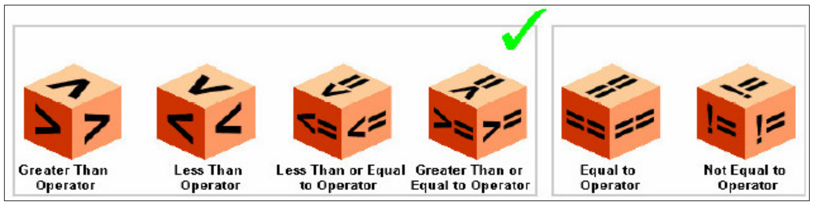
```
var1=12.5
(var1 > 100) Evaluates to 0
```

LESSON 18 : RELATIONAL OPERATORS

Relational operators have precedence that determines the order in which they are evaluated in a multiple-operator expression. Use parentheses to modify the precedence of relational operators in expressions.

Precedence of Relational Operators

It is always a good practice to use parentheses in expressions to clearly specify the precedence of relational operators. The operators greater than, less than, less than or equal to, and greater than or equal to fall into one precedence group.



The equality operators equal to and not equal to fall into another precedence group. The equality operators have lower precedence than other relational operators. In an expression containing the equal to operator and the greater than operator, the greater than operator is evaluated first and then the equal to operator is evaluated.

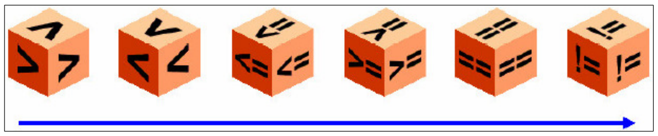
Consider two integers, `var1` and `var2`, having the values `1` and `2`, respectively. In the expression that is given below, the values of `var1<7` and `var2>6` are evaluated first.

```
Var1=1 var2=2
Var1<7!=var2>6
```

The expressions `var1<7` and `var2>6` evaluate to `1` and `0`, respectively. Finally, the values `1` and `0` are compared. The value `1` is not equal to `0`. Therefore, the expression `1!=0` is true, which is `1`.

Associativity of Relational Operators

The associativity of relational operators is from left to right. Therefore, in an expression that contains more than one relational operator, evaluate the expression from left to right.



The relational operators enable you to compare two expressions. The associativity of relational operators is from left to right.

Lesson Objectives

In this Lesson you will learn :

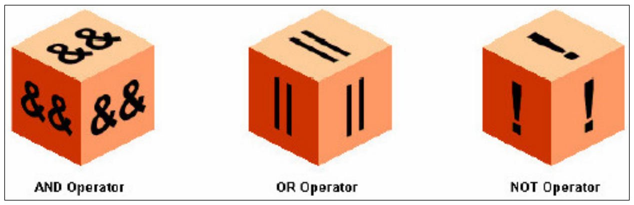
- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

LOGICAL OPERATORS

Logical operators enable you to combine two conditions. Consider the example in which a program calculates the allowance for the employees of an organization depending on their base pay. The base pay of the employees is represented by the variable `Base_Pay`.

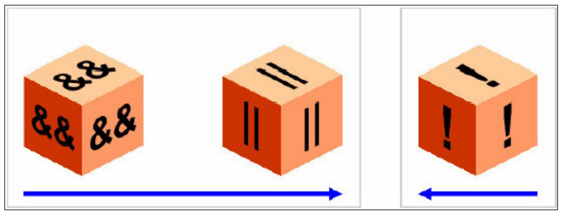
If `Base_Pay` of an employee is below `2,000` and above `1,000`, the allowance is `10%` of `Base_Pay`. When these two conditions are satisfied, an employee acquires an allowance of `10%` of the base pay. The two conditions that have to be verified are that the `Base_Pay` should be below `2,000` and above `1,000`. In this situation, you can use logical operators to combine the two conditions.

There are three logical operators in C : AND, OR and NOT. The AND, OR, and NOT operators are represented by `&&`, `||`, and `!`, respectively.



- The AND operator and the OR operator are binary operators. This means that these operators operate on two operands.
- The NOT operator is a unary operator. This means that the NOT operator operates on a single operand.
- The associativity of the AND and OR logical operators is from left to right. The associatively of the NOT operator is from right to left.

Associativity of Logical Operators



Using* AND *Logical Operator

Case 1:

In the following examples, If the value of `var1` is `7`, then the subexpression `(var1 > 5)` is `true`. The subexpression `(var1 < 10)` is also `true`. Hence, the condition is `true`.

Case 2:

```
var1 = 7
((var1 > 5) && (var1 < 10))
```

If the value of `var1` is `3`, then the subexpression `(var1 > 5)` is `false`. However, the subexpression `(var1 < 10)` is `true`. Hence, the condition is `false`.

If two expressions, `expr1` and `expr2`, are combined using the AND operator, the result is true only if both `expr1` and `expr2` are individually `true`. The result of the expression that combines the expressions `expr1` and `expr2` by using the AND operator is `false` if either `expr1` or `expr2` is `false`.

Values of :		
expr1	expr2	expr1 && expr2
False	False	False
False	True	False
True	False	False
True	True	True

Using* OR *Logical Operator

The expression below contains OR operator.

```
var1 = 5 var2=4
((var1!=10) || (var2 > 5))
```

Case 1:

If the values of `var1` and `var2` are `5` and `4`, respectively, then the subexpression `(var1 !=10)` is `true`.

Case 2:

The value of `var2` is `4`. Therefore, the subexpression `(var2 > 5)` is `false`. However, the expression is `true` because one of the subexpressions is `true`.

In the below expression, consider the values of `var1` and `var2` as `10` and `3`, respectively. Both the subexpressions `(var1 !=10)` and `(var1 > 5)` are `false`. Therefore, the highlighted expression is `false`.

```
var1 = 10 var2=3
((var1!=10) || (var2 > 5))
```

If two expressions, `expr1` and `expr2`, are combined using the OR operator, the result is `true` if either `expr1` or `expr2` is `true`.

Values of :		
expr1	expr2	expr1 || expr2
False	True	True
True	False	True
True	True	True
False	False	False

The result of the expression that combines the expressions `expr1` and `expr2` by using the OR operator is `false` only if `expr1` and `expr2` are `false`.

Using **NOT** *Logical Operator*

The NOT operator negates the value of an expression. This indicates that if an expression, `expr` is `true` originally, the NOT operator alters the result to `false`.

Values of :	
expr	!expr
False	True
True	False

Similarly, if an expression, `expr`, is originally `false`, the NOT operator alters the result to `true`.

The expression below contains the NOT operator. If the value of `var1` is `3`, then the expression `(var1 > 6)` is `false`. The negation of `false` is `true`. Therefore, the expression is `true`.

The logical operators enable you to combine conditions. The AND and OR operators associate from left to right and the NOT operator associates from right to left.

```
Var1 = 3
!(var1 > 6) Evaluates to 1
```

Self Review Exercise 19.1

1. *The expression* `(exp1)` *AND* `(exp2)` *evaluates to* `true` *if:*

   A. `exp1` and `exp2` evaluate to true individually.
   B. `exp1` evaluates to `true` and `exp2` evaluates to `false`.
   C. `exp2` evaluates to `true` and `exp1` evaluates to `false`.
   D. `exp1` and `exp2` evaluates to `false` individually.

LESSON 20 : `sizeof` OPERATOR

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

`sizeof` OPERATOR

The `sizeof` operator is used to determine the number of bytes occupied by an object.

An object can be a data type such as a `float`, or an expression, `x+y`, containing variables of any data type. An expression, `sizeof(object)`, containing the `sizeof` operator returns an integer that represents the number of bytes needed to store the object in RAM. The `sizeof` operator is a unary operator.

The value returned by the `sizeof` operator can be displayed using the `printf()` function. The `sizeof` operator always returns an integer. Therefore, the conversion character is `d`.

```
printf("%d", sizeof(object));
```

Consider the program `size.c` that determines the number of bytes occupied by the `char`, `int`, `float`, and `double` data types. The preprocessor directive and the `main()` function statement have been typed.

Listing 20.1 : A program to display the no. of bytes occupied by primitive data types

```
1   //The sample code will display the number of bytes occupied by the char, int, float,
2   //and double data types.
3
4   #include <stdio.h>
5
6   main()
7   {
8
9       printf("character: %d\n", sizeof(char));
10      printf("integer: %d\n", sizeof(int));
11      printf("float: %d\n", sizeof(float));
12      printf("double: %d\n", sizeof(double));
13
14  }
```

Detailed Explanation

- To display the number of bytes occupied by a character, `printf("character:%d\n", sizeof (char));` in line 9.
- To display the number of bytes occupied by an integer, `printf("Integer: %d\n", sizeof (int));` in line 10.
- To display the number of bytes occupied by a `float`, `printf("Float: %d\n", sizeof(float));` in line 11.
- To display the number of bytes occupied by a `double`, `printf("Double: %d\n", sizeof (double));` in line 12.

- You completed the program to determine the number of bytes occupied by a character, an integer, a floating-point number, and a number with `double` precision.

`sizeof` operator enables you to determine the number of bytes occupied by a data type or an expression.

LESSON 21 : OPERATOR PRECEDENCE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

OPERATOR PRECEDENCE

When using various operators, such as relational and logical operators, in expressions, you should follow the precedence and associativity of operators. This ensures the correct output. In C, different operators are grouped according to varying degree degrees of precedence. Within a precedence group, the operators are evaluated in a specific order.

Descending Order of Precedence

Operators	Their Associativity
`-, ++, --, !, sizeof`	Right to Left
`*, /, %`	Left to Right
`+, -`	Left to Right
`<, >, <=, >=`	Left to Right
`==, !=`	Left to Right
`&&`	Left to Right
`||`	Left to Right
`=, +=, -=, /=, %=`	Right to Left

The order of evaluating consecutive operators within the same precedence group is called associativity.

Detailed Explanation

- First, unary operators have the highest precedence. The unary operators are minus (-), increment (++), decrement (--), NOT (!), and sizeof.
- The arithmetic operators multiply (*), divide (/), and modulus (%) are next in the hierarchy of operators. The operators add (+) and subtract are placed next in hierarchy.
- Next in the hierarchy are relational operators. The relational operators are less than (<), greater than (>), less than or equal to (<=), and greater than and equal to (>=).
- Equality operators follow the relational operators in the hierarchy of operators. The equality operators are equal to (==) and not equal to (!=).
- The logical AND (&&) and OR (||) operators follow the equality operators. The AND operator has a higher priority over the OR operator.

- Assignment operators are placed at the bottom of the hierarchy of the operators.

It is a good programming practice to set the associativity of operators in long expressions by using parentheses. The use of parentheses clearly defines the order of evaluation.

The unary and assignment operators associate from right to left. However, the arithmetic, relational, equality, and logical operators associate from left to right.

In the expression below, `var2` is decremented and the decremented value is assigned to `var1`. This is because `var2` is decremented before the value of `var2` is assigned to `var1`.

```
var1 = --var2
```

Consider the expression that is given below. In this expression, the value of var1 is 10. The multiplication operator and the modulus operator have the same precedence.

```
var1 = 10
40-((var1*2)+(var % 3))
```

- Arithmetic operators associate from left to right. Therefore, `(var1*2)` and `(var1%3)` are evaluated. The result of `(var1*2)` and `(var%3)` are added, and this sum is finally subtracted from 40.
- The expressions `(var1*2)` and `(var1%3)` evaluate to 20 and 1, respectively. The sum of 20 and 1 is 21, and the difference of 40 and 21 is 19. Therefore, the expression evaluates to 19.

```
var1 = 10
40-((var1*2)+(var % 3))
(var1 * 2) evaluates to 20
(var2 % 3) evaluates to 1
```

By following the precedence and associativity of operators, you can obtain correct results.

LESSON 22 : `if` STATEMENT

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

DECISION MAKING

In certain situations, it may be necessary to write a code sample to perform various actions based on the user's input. For example, you are creating a program to accept numbers between `0` and `9` from the user.

If the number typed is between `0` and `9`, it is to be displayed. Otherwise, a message that the typed number is invalid is to be displayed. To handle such situations, C provides certain decision-making statements, such as the `if` statement.

The `if` *Statement*

The `if` statement has three forms. The syntax of the `if` statement is displayed below.

In the syntax, the expression part represents the condition.

```
If(expression)
 If_body;
```

The execution of the `if` statement involves the evaluation of the condition. If the condition evaluates to true or nonzero, the `if`-body part of the `if` statement is executed.

```
If(expression)    True
 If_body;
```

The simple form of the `if` statement does not provide an alternative for a situation where the expression part evaluates to `false`. Therefore, another `if` statement needs to be used to handle the `false` value of the expression.

The simple form of the `if` statement is inefficient because the flow of control will execute two `if` statements to handle the `true` and `false` states of an expression.

Write the code to accept a number between `0` and `9` from a user. If the number is between `0` and `9`, display the number. Otherwise, display a message that the number is not between `0` and `9`. The code to accept a number is as follows.

Listing 22.1 : A program using simple if statement

```
1   //Listing 22.1 : the code sample will accept a number between  0 and 9. if the  number
2   // is between 0 and  9, the number will be displayed otherwise a message that the number
3    //is not between 0 and 9 will be displayed.
4
5   #include <stdio.h>
6   main()
7   {
8       int num;
9       printf(“Type a number between 0 and 9: “);
10      scanf(“%d”, &num);
11      if(num >=0 && num <-9)
12        printf(“you typed %d.”, num);
13      if(num > 9 || num < 0)
14        printf(“you type a number between 0 and 9. “);
15  }
```

Detailed Explanation

- In the code, an integer type variable, `num`, is declared. Next, the `printf()` function is used to prompt the user to type a number between `0` and `9`. the `scanf()` function is used to accept a number from the user in the `num` variable.
- After accepting the number, evaluate whether it is between `0` and `9`. As in Line No `7 if (num>=0&&num<=9)`.
- If the number typed is between 0 and 9, display the number. The `printf()` function to do this is used as in Line No `8`.
- To handle the situation when the number typed is not between `0` and `9`, write another `if` statement. `if (num > 9 || num < 0)`. As in Line 9.
- If the number typed is not between `0` and `9`, display a message that the number is not between `0` and `9`. The `printf()` function to display this message is used in Line No - `10`.
- The program prompts you to type a number between `0` and `9`. Enter `9`.
- When a number is typed, the computer evaluates it to check whether or not it is between `0` and `9` in the first `if` statement. The number typed was `9`. Therefore, the condition evaluated to `true`. As a result, the output of the program was `you typed 9`.
- When the number `56` was typed, the computer evaluated the first `if` statement. This `if` statement evaluated to `false`. Then, the computer evaluated the second `if` statement. The second `if` statement is evaluated regardless of whether the first is `true` or not.
- The second `if` statement evaluated to `true`. Therefore, the output you did not type a number between `0` and `9` was obtained.
- If you want more than one statement to be executed in the `if_body` part of the `if` statement, enclose the statements within braces. If you use multiple statements in the `if_body` part without braces, you will not get the intended output. The compiler treats the `if_body` part as having a single statement, which is the first statement. The rest of the statements are executed regardless of the `if` statement.

```
if (expression)

{
 statement;
 statement;
}
```

A code sample that uses multiple statements within braces is given below.

Listing 22.2 : A program using multiple statements in if statement

```
//Listing 22.2 : The code sample will accept a number between  0 and 9. if the  number is
//0 and  9, the number will be displayed. Another number between 0 and 9 will be accepted
//and between displayed. If the first number typed is not between 0 and 9. A message
//that the typed number is not between 0 and 9 will be displayed.

#include <stdio.h>

main()
{
    int num;
    printf("Type a number between 0 and 9: ");
    scanf("%d", &num);
    if(num >=0 && num <-9)
    {
     printf("you typed %d.", num);
     printf("\n Type another between 0 and 9: ");
     scanf("%d", &num);
     printf("you typed %d. \n", num);
    }
    if(num > 9 )
     printf("you type a number between 0 and 9. ");
}
```

Detailed Explanation

- The same code sample is rewritten without enclosing the multiple statements in the if_body part within braces. If the numbers typed are 23 and 3, notice that the output is different from the intended output of the code.
- The last three lines of code after the first if statement should not have been executed. They are executed because the computer assumes that the first if statement is terminated when it encounters the semicolon after the first printf function.
- If a semicolon is added after the condition of the if statement, the statement after the if condition is not considered to be a part of the if statement. Therefore, even if the condition is false, the statement after the if statement is executed.
- The code sample given below should display the typed number if the user types a number between 0 and 9. If the typed number is not between 0 and 9, the program should terminate without displaying any message.

Listing 22.3 : A program positioning a semicolon after an if statement

```
1   //Listing 22.3 : code sample to demonstrate the effect of using a semicolon after
2   //the condition in the if statement
3
4   #include <stdio.h>
5
6   main()
7   {
8       int num;
9       printf("Type a number between 0 and 9: ");
10      scanf("%d", &num);
11      if(num > = 0 && num <= 9);
12       printf("you typed %d", num);
13  }
```

The program displays the message that the typed number is `12` even when it is not between `0` and `9`. This indicates that the `printf` function in Line `10` is not considered part of the `if` statement because of the use of the semicolon after the condition.

Self Review Exercise 22.1

1. *You learned about the simple* `if` *statement. Observer the sample that uses the simple* `if` *statement and four possible results of the code. If the number typed in the beginning of the program is* `22`*, choose the correct choice.*

```
#include<stdio.h>
main()
{
int Odd_Number;
printf("Type an odd number between 1 and 50: ");
scanf("%d", &Odd_Number);
if(Odd_Number %2 ! =0)
{
printf("%d is an odd number", Odd_Number);
}
if(Odd_Number % 2 ==0)
{
printf("%d is an even number.", Odd_Number);
printf("\n Type another number between 1 and 50 : ");
scanf("%d", &Odd_Number);
printf("You typed %d. ", Odd_Number);
}
}
```

A. `22` *is an even Number.*
Type another number between `1` *and* `50`*:* `22` *You typed* `22`*.*
B. `22` *is an even Number.*
Type another number between `1` *and* `50`*:*
C. `22` *is an odd Number.*
Type another number between `1` *and* `50`*:* `22` *You typed* `22`*.*
D. `22` *is an odd Number.*
Type another number between `1` *and* `50`*:*

LESSON 23 : `if..else` STATEMENT

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

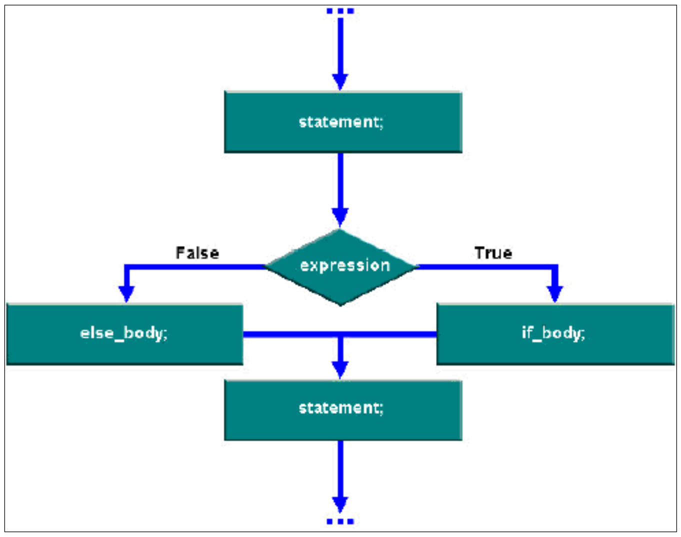
THE `if-else` STATEMENT

The `if-else` statement is another form of the `if` statement. This form can handle both the `true` and `false` values of an expression.

Observe the syntax of the `if-else` statement. If the expression evaluates to true, the `if_body` part of the `if` statement is executed. If the expression evaluates to false, the `else_body` part is executed.

```
if(expression)
 if_body;
else
 else_body;
```

The flow diagram displayed on the screen illustrates the flow of control in the `if-else` statement. If the expression is true, the control executes the `if_body`. If the expression is false, the control executes the `else_body`.



The `if-else` statement is more efficient than the simple `if` statement. This is because in the `if-else` statement, only the `if_body` part or the `else_body` part of the code is executed. Both the parts are not executed.

The following code will accept a number between `0` and `9` from the user. If the number typed is in the specified range, it will display the number.

Listing 23.1 : A program demonstrating the complexity of decision making in a simple if statement

```
1   // Listing 23.1 : the code will accept a number between 0 and 9. If the typed number is
2   // between 0 and 9.The number will be displayed otherwise a message that the typed
3   // number is not between 0 and 9 will displayed.
4
5   #include < stdio.h>
6
7   main()
8   {
9
10      int num;
11      printf("Type a number between 0 and 9: ");
12      scanf("%d", &num);
13
14      if(num >=0 && num <=9)
15              printf("you typed %d.", num);
16
17      if(num >9 || num <0)
18              printf("you did not type a number between 0 and 9");
19
20  }
```

Detailed Explanation

- If the number typed is not between 0 and 9, a message that you did not type a number between 0 and 9 will be displayed. To do this, two simple if statements are used in the code. This code is not efficient because both the if statements will always be executed.
- You can rewrite the same code by using the if-else statement. In the existing code, replace the line 17 with the word else. The resulting code will generate the same output more efficiently.
- The reason for efficiency of the code is that only the statements in the if part or only the statements in the else part will be executed. Both the if and else parts will not be executed.

Listing 23.2 : Rewriting the previous listing with simple if - else statement

```
1   // Listing 23.2 : the code will accept a number between 0 and 9. If the typed number is
2   // between 0 and 9.The number will be displayed otherwise a message that the typed
3   //number is not between 0 and 9 will displayed.
4
5   #include < stdio.h>
6
7   main()
8   {
9
10      int num;
11      printf("Type a number between 0 and 9: ");
12      scanf("%d", &num);
13
14      if(num >=0 && num <=9)
15              printf("you typed %d.", num);
16
17      else
18
19              printf("you did not type a number between 0 and 9");
20
21  }
```

Detailed Explanation

- The typed number is evaluated to check whether or not it is between `0` and `9`. The number `9`, is in the specified range. Therefore, the output is `you typed 9`.
- To check how the program will operate if the typed number is not in the specified range, type `69`.
- The typed number is evaluated to check whether or not it is in the specified range. The number, `69`, is not in the range `0` to `9`. Therefore, the output is `you did not type a number between 0 and 9`.

To execute multiple statements in the `if-else` statement, enclose the statements between the braces. The presence or absence of braces changes the output of a code sample drastically.

```
if(expression)
{

 statement;
 statement;

}

else
{

 else_body;

}
```

For example, the code given below will accept a number between `0` and `9`. If the number specified is in this range, the program will display the typed number and again prompt you to type a number.

Listing 23.3 : A program demonstrating execution of multiple statements in an if - else statement

```
// Listing 23.3 : The code will accept a number between 0 and 9. If the number typed is
// in this range, the program will display the typed number and again prompt you to type
// another number after accepting the second number, the program will display that
// number. If the first number is not between 0 and 9, the program will display a
// message.

#include<stdio.h>

main()
{

    int num;
    printf("Type a number between 0 and 9: ");
    scanf("%d", &num);

    if(num>=0 num <=9)
            {

                    printf("you typed %d.", num);
                    printf("\n Type another number between 0 and 9: ");
                    scanf(%d", &num);
                    printf("you typed %d. \n", num);

            }

    else
                    printf("you did not type a number between 0 and 9. ");
}
```

Detailed Explanation

- After accepting the second number, the program will display that number. Notice that braces are used to group all the statements in the `if` statement.
- If the number typed is `6`, the output of the code is "`You typed 6`". On the next line, the program displays the message "`Type another number between 0 and 9:`".
- If the number typed is `8`, the output is "`You typed 8`". All the statements in the `if` part of `if-else` statement are executed.

If the same code is without enclosing the statements in the `if` part within braces. This code will generate an error when it is compiled. The code is compiled. The compiler generates an error indicating that there is a parse error before `else`. A parse error is generated by the compiler when it finds misplaced or missing punctuators in source code.

If you use multiple statements in the `else` part of the `if-else` statement without braces, the will not generate an error. In addition, the code will not generate the required output. As an example, a code sample is given below.

Listing 23.4 : Demonstration of multiple statements without braces in the else part of an if-else statement

```
// Listing 23.4 : The code sample will accept a number between 0 and 9. If the number
// typed is not between 0 and 9, the program will display a message indicating  this. If
// the number typed is between 0 and 9 , the program will display the typed number and
// again prompt you to type  another number. After accepting the second number. the
// program will display that number.

#include < stdio.h>

main()
{

    int num;
    printf("Type a number between 0 and 9: ");
    scanf("%d", &num);

    if(num < 0 || num > 9)
            printf("you did not type a number between 0 and 9. ");

    else
            printf("you typed %d",num);
            printf("\n Type another number between 0 and  9: ");
            scanf("%d", &num);
            printf("you typed %d \n", num);

}
```

Detailed Explanation

- If the number typed is `23`, the output is `"You did not type a number between 0 and 9"` and `"Type another number between 0 and 9"`.

When you use multiple statements without enclosing them within braces, the computer assumes that the `else` part has only one `printf` function. The remaining `printf` functions are treated as separate lines of code.

Within the `if-else` statement, you can use other `if` statements. This feature is called nesting. You can use the nested `if -else` statements to perform actions based on multiple conditions at one time.

Using many nested `if-else` statements should be avoided because it can make the complicated to read. In addition, braces should be used carefully with the nested `if-else` statements. This is because the output of a code sample can change based on the presence or absence of braces.

A sample code to accept two numbers between `0` and `9` and `100` and `1000` is given below. If the first number is between `0` and `9`, check if the second number is greater than `1000` or less than `100`. If this is `true`, inform the user that the second number should be between `100` and `1000`. If the second number is less than `1000` and greater than `100`, terminate the program. If the first value typed is greater than `9`, display a message that the `first number is greater than 9`.

Listing 23.5 : Demonstration of if - else statement

```
// Listing 23.5 : The code will accept the number between 0 and 9 and 100 and 1000. if
// the first number is between 0 and 9, the program will check if the second number is
//greater than 1000 or less than 100. if this is true, the program will display a
// message that the second number should be between 100 and 1000. If the second number
// is less than 1000 and greater than 100. the program will be terminated. If the first
// number typed is greater than 9, the program will display a message that the first
// number is greater than 9.

#include<stdio.h>

main()
{

    int num1, num2;
    printf("Type a number between 0 and 9: ");
    scanf("%d", &num1);
    printf("Type another number between 100 and 1000: ");
    scanf("%d", &num2);

    if(num1 >=0 && num1 <= 9)
            {
                    if(num2 > 1000 || num2 <100)
                    printf("Your second number should be between 100 and 1000. ");

            }

    else
                    printf("Your first number should be between 0 and 9. ");
}
```

Detailed Explanation

- The code is compiled and executed. If the first number typed is `7` and the second number typed is `1200`, the output is "`Your second number should be between 100 and 1000`".
- If the first number typed is `67` and the second number typed is `1200`, the output is "`Your first number should be between 0 and 9`".

If the same code is written without using braces in the nested `if-else` statement, the output will be different.

```
if(num1 >= 0 && num1 <= 9)

 if(num2 > 1000 || num2 <100)
        printf("you second number should be between 100 and 1000. ");
else
 printf("your first number should be between 0 and 9. ");
```

Detailed Explanation

- If the first and second numbers typed are `67` and `1200`, respectively, the program does not generate output. This is different from what was expected.
- The reason for no output is the absence of braces in the first `if` statement. In the case of nested `if` statements, the `else` of the outermost `if` statement is paired with the nearest `if` statement in the absence of braces.

Self Review Exercise 23.1

1. *The following code uses the* `if-else` *statement and four possible results of the code. If the letter typed in the beginning of the program is* a*,*

```
#include <stdio.h>
main()
{
        char vowel;
        printf("Type any vowel : ");
        vowel = getche():
        if(vowel == 'a' || vowel == 'e' || vowel == 'i' || vowel == 'o' || vowel == 'u')
                {
                        printf("\n%c is a vowel. ", vowel);
                }
        else
        printf("\n%c is a consonant. ", vowel);
        printf("\nType any vowel : ");
        vowel = getche();
        printf("\nYou typed %c . ", vowel);
}
```

A. *a is vowel.*

B. *a is vowel.*
Type any vowel: a

C. *a is vowel.*
Type any vowel:

D. *a is vowel.*
a is consonant.
Type any vowel: a
You typed a.

Self Review Exercise 23.2

2. *The following sample code uses a nested* `if-else` *statement and four possible results of the code. If the first and second numbers typed are* 2 *and* 3, *respectively, find the right option of the nested* `if-else` *statement.*

```
#include <stdio.h>
main()
{
        int num1, num2;
        printf("Type an even number: ");
        scanf("%d", &num1);
        printf("Type an odd number: ");
        scanf("%d", &num2);
        if(num1 % 2 == 0)
                {
                        printf("Your first number is an even number.");
                        if(num2 % 2 != 0)
                          printf("Your second number is an odd number.");
                }
        else
                        printf("Your first number is not an even number.");
}
```

A. Your first number is an even number.
B. Your first number is an even number.
Your second number is an odd number.
C. Your first number is an even number.
D. The compiler generates a parse error.

LESSON 24 : `if..else..if` STATEMENT

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE `if - else - if` STATEMENT

To perform various actions based on multiple conditions at one time, you can use the `if-else-if` statement. This statement is another form of the if-statement. The syntax of the `if-else-if` statement is given below. In the syntax, `expression1` and `expression2` indicate different conditions.

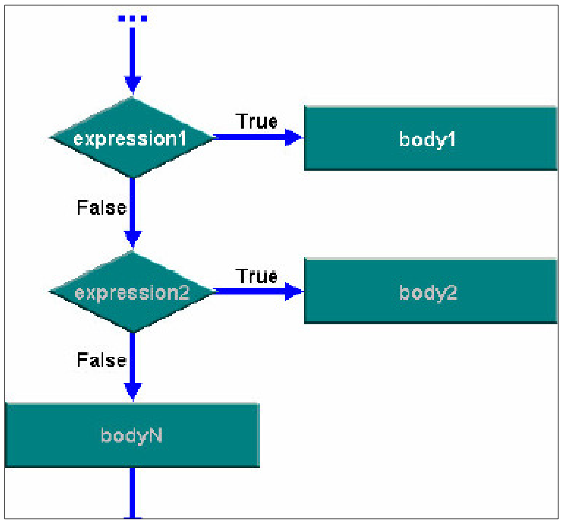
Syntax of the `if - else - if` *statement*

```
if(expression1)
 Body1
else if(expression2)
 Body2
.........
else
 BodyN
```

Detailed Explanation

- The `body1`, `body2`, and `bodyN` parts of the `if-else-if` statement indicate the statements that will be executed if the respective conditions evaluate to true.

The flow diagram on the screen indicates the direction in which the control moves in the `if-else-if` statement. The `if` statement evaluates `expression1`. If it is true, the statement in the `body1` part is executed.



Detailed Explanation

- If `expression1` is false, `expression2` is evaluated. If `expression2` is true, the statement in the `body2` part is executed. If `expression2` is false, the statement in the `bodyN` part is executed.

In the `if-else-if` statement, the last `else` is paired with the if in the line above it. In the `if-else-if` statement, only the statement corresponding to the expression that evaluates to true is executed and the remaining construct is skipped.

A program to accept a number between `0` and `5` and display the typed number in words. If the number is not between `0` and `5`, display a message that the number is not between `0` and `5`.

Listing 24.1 : A program implementing if - else - if statement

```
//Listing 24.1 : A simple if - else - if block sample
//The code sample will accept a number between 0 and 5 and display
//number in words. If the typed number is not between 0 and 5, a
//message that the number is not between 0 and 5 will be displayed.

#include< stdio.h>

main()
{

    int num1;
    printf(“Typed a number 0 and 5 : “);
    scanf(“%d”, &num1);

    if(num1==0)
            printf(“zero”);

  else if(num1==1)
            printf(“one”);

  else if(num == 2)
            printf(“two”);

  else if(num1==3)
            printf(“three”);

  else if(num1==4)
            printf(“four”);

  else if(num1==5)
            printf(“five”);

    else
            printf(“Number is not between 0 and 5”);

}
```

Detailed Explanation

- In the code, an integer variable, `num1`, is declared. The `printf` function will prompt the user to type a number between `0` and `5`. The `scanf` function will accept the number in the `num1` variable.
- Next, compare the typed value with the numbers between `0` and `5` to check if they are equal. To compare the value in `num1` with `0`, type `if(num1==0)` as in line `15`.
- If the value in the `num1` variable is equal to `0`, display `0` in words. The `printf` function to do this is used in line `16`.
- If the value in `num1` is not equal to `0`, compare `num1` with `1` to check if they are equal. Use `else if(num1==1)` as in line `18`.
- If the value in the `num1` variable is equal to `1`, display in words. The `printf` function to do this is used. Similarly, the code to compare the value of `num1`with the numbers from `2` to `5` is added in the lines `21` to `31`.
- The code to compare the value of `num1` with the numbers from `2` to `5` is displayed. If the typed number is not equal to any number between `0` and `5`, display a message as in lines `33` and `34`.
- The program prompts you to type a number between `0` and `5`. Type **`4`.**
- The output of the code is `four`.
- If you enclose the `else if` statements between the fist `if` and the last `else` within braces, the compiler will generate a parse error. The code sample given contains braces.
- The code is compiled. The compilation error that is generated indicates that there is a parse error before `else`.
- If you want to use multiple statements in the different body parts of `if-else-if` statement, enclose the statements within braces. Otherwise, the compiler will generate a parse error.

The followed code uses multiple statements in the body part without enclosing these statements within braces.

Listing 24.2 : A program dealing with multiple statements inside an if - else - if block

```
//Listing 24.2 : Multiple statements inside an if - else - if block
//The code sample to demonstrate the error generated due to the use of
//multiple statement within if-else-if statement without enclosing
//then within braces

#include< stdio.h>

main()
{

    int num1;
    printf("Typed a number 0 and 5 : ");
    scanf("%d", &num1);

    if(num1==0)
      {

       printf("zero");
         printf("zero multiple by 2 is %d .", num1 * 2);

      }

  else if(num1==1)
             printf("one");

  else if(num == 2)
             printf("two");

  else if(num1==3)
             printf("three");

  else if(num1==4)
             printf("four");

  else if(num1==5)
             printf("five");

    else
             printf("Number is not between 0 and 5");

 }
```

Detailed Explanation

- When the code is compiled, the compiler generates an error. The error indicates that there is a parse error before else in the code. This error indicates that statements following the if statement should be placed within braces.
- The same code sample will work properly if braces are used.
- The code is compiled and executed. if the number typed is `0`, the output is displayed in two lines. The two lines are `zero and zero multiplied by 2 is 0`.

This lesson covered the syntax of the `if-else-if` statement, how to use this statement in a code sample, and the use of braces in this statement. The `if-else-if` statement is useful in situations where various actions are to be performed based on multiple conditions.

Self Review Exercise 24.1

1. *You learned about the `if-else-if` statement. Given below is a code sample that uses the `if-else-if` statement and four possible results of the code.*

```
# include <stdio.h>
main()
{
int num1;
printf("Type a number between 10 and 14:");
scanf("%d", &num1);
if(num1 == 10)
    printf("ten");
else if(num1 == 11)
    printf("eleven");
else if(num1 == 12)
    printf("twelve");
else if(num1 == 13)
    printf("thirteen");
else if(num1 == 14)
    printf("fourteen");
else
    printf("Number is not between 10 and 14");
}
```

A. Parse error before else
B. Twelve
C. Ten eleven twelve thirteen fourteen
D. Ten eleven twelve thirteen fourteen number is not between 10 and 14

LESSON 25 : CONDITIONAL OPERATOR

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

CONDITIONAL OPERATOR

The C language provides an operator to handle situations where different actions are performed based on the evaluations of a condition. This is the conditional operator.

- The conditional operator is also called the ternary operator, and it is represented by a question mark and a colon`(?:)`;
- The conditional operator accepts three operands.
- Each of these operands is an expression.

The syntax of the conditional operator is

```
expression1 ? expression2 : expression3.
```

The `expression1` part is the condition. If this condition is `true`, `expression2` is evaluated. If the condition is `false`, `expression3` is evaluated.

The example `z=(a>b) ? a:b`; demonstrates the use of the conditional operator. If the value of `a` is greater than the value of `b`, the value of `a` is assigned to `z`. otherwise, the value of `b` is assigned to `z`.

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

The conditional operator is an efficient alternative to a simple `if-else` statement. This is because it is smaller in size compared to a simple `if-else` statement. It reduces the size of the code and increases the speed of execution.

Two code samples that will generate similar result are given below. The first sample uses the conditional operator, and the second sample uses the `if-else` statement. Notice that the first code sample is smaller than the second code sample.

Conditional Operator

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

`if-else` ***Statement***

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

LESSON 25 : CONDITIONAL OPERATOR

Create a program that accepts a number between `0` and `9` from users. If the typed number is between `0` and `9`, display the number. Otherwise, display a message that the number is not between `0` and `9`. The program can be created using the conditional operator. The code to accept a number is as follows:

Listing 25.1 : Using conditional operator

```
// Listing 25.1 : The code will accept a number between 0 and 9. If the number is between
// 0 and 9,the number will be displayed. Otherwise, a message that the number is not
// between 0 and 9 will be displayed.

#include< stdio.h>

main()
{

    int num;
    printf("Type a number between 0 and 9: ");
    scanf("%d", &num);

    (num >= 0 && num <= 9) ? printf("you typed %d,", num) : printf("Number is not between
    0 and 9");

}
```

Detailed Explanation

- If the condition is `true`, the typed number is displayed.
- If the condition is `false`, display a message indicating that the number is not between `0` and `9`.
- If the number typed is `5`, the output is You typed `5`.
- After the number was typed, the computer evaluated the condition to check whether or not the typed number was between `0` and `9`. The number, `5`, is in the specified range. Therefore, the output is `You typed 5`.
- If the number typed is `10`, the output is displayed as `Number is not between 0 and 9`.

The typed number is compared with the condition `(num>=0&&num<=9)`. The number, `10`, is not in the specified range. Therefore, the output is displayed as `Number is not between 0 and 9`.

Conditional operator helps in reducing the size of code, which increases the execution speed of the program.

Self Review Exercise 25.1

1. *The code that uses this operator and four possible results of the code are given below. Choose the correct output of the code now.*

```
#include <stdio.h>
main( )
{
int num1 = 10, num2 = 100, x = 0;
x = (num1 > num2) ? num1 - num2; num2 - num1;
printf("\n\n%d", x);
}
```

A. 90
B. -90
C. 0
D. Parse error

LESSON 26 : switch STATEMENT

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE switch STATEMENT

The `switch` statement is a conditional statement branching to various alternative actions. You can use this statement to perform various actions based on the evaluation of a simple expression. The `switch` statement can be used instead of the `if-else-if` statement if the condition evaluates to an integer value.

The syntax of the `switch` statement is shown below. The statement starts with the `switch` keyword and is followed by an expression within parentheses.

```
switch (expression)
{
case value 1:
         statement;
         statement
         break;

case value 2:
         statement;
         statement
         break;

case value N:
         statement;
         statement
         break;
default:
         statement;
         statement
         break;
}
```

Detailed Explanation

- The `expression` can be the value of a variable, an arithmetic expression, a logical comparison, a bitwise expression, or the `return` value of a called function. The `expression` within parentheses must evaluate to an integer.
- The `switch` statement contains a sequence of labels within braces. Each label consists of the `case` keyword and a constant integral value. This value can be an integer or a character, but not a floating point number or a character string. The value cannot be an expression.
- The constant integral value must be different in various `case` labels. The `case` label terminates with a colon. Under each `case` label, you can write multiple related statements.
- You can conclude the statements used in a `case` label with a `break` statement. The `break` statement transfers the flow of control out of the `switch` statement.

- If a break statement is not provided, each time a match is found in the switch statement, the statements for all the remaining case labels will be evaluated sequentially. The presence or absence of the break statement can change the output of a code sample.
- The switch statement can also contain the default case. The default case contains the default keyword terminated with a colon. Under the default case you can write multiple related statements. The break statement can also be used with this keyword. The statements mentioned with the default keyword are executed when none of the case values match the expression. The default keyword is optional.

Consider an example to understand how the switch statement works. Write a code sample to accept two numbers from a user. Based on whether the user wants to add, subtract, multiply, or divide the numbers, display the result.

The code to accept two numbers and the type of operation is shown below. In the code, two integer variables, num1 and num2, and a character variable, Compute_Code, are declared.

Listing 26.1 : A complete switch demonstration

```
//Listing 26.2 : The code will accept two numbers. Based on whether the users wants to
//add,subtract, multiply, or divide the numbers, the result will be displayed.
#include< stdio.h>
main()
{
    int num1, num2;
    char Compute_code;
    printf (“Type the first number: “);
    scanf (“%d”,&num1);
    printf (“Type the second number: “);
    scanf (“%d”, &num2);
    printf(“Do you to add, subtract, multiply, or divide the numbers? type a, s, m, or d:”);
    Compute_Code=getche();
    switch (Compute_Code)
    {
    case ‘a’:
            printf(“\n The sum of the two numbers is %d”, num1 + num2);
            break;
    case ‘s’:
            printf(“\n The difference of the two numbers is %d”, num1- num2);
            break;
    case ‘m’:
            printf(“\n The product of the two numbers is %d”, num1 * num2);
            break;
    case ‘d’:
            printf(“\n The division of the two numbers is %d”, num1 / num2);
            break;
    default:
            printf(“\nout of range”);
            break;
    }
}
```

Detailed Explanation

- Three printf functions are coded to prompt the user to type two numbers and the type of operation to be performed on the numbers. The two scanf functions will accept the two numbers in the num1 and num2 variables.

- The `getche` function will accept a character into `Compute_Code` which indicates the operation the user wishes to perform.
- After accepting the value for `Compute_Code`, determine the type of operation specified by the user. To do this, you can use the `switch` statement. As in Line `14`.
- All the `case` labels in the `switch` statement are enclosed within opening and closing braces.
- To determine if the value of `Compute_Code` is one of the characters `a, s, m, or d`, use the `case` labels of the `switch` statement. To check if the value typed is `a`, use `case 'a':`.
- If the type of operation specified is `a`, display the sum of the two numbers typed by the user. The `printf` function is used to display this message.
- After the sum of the two numbers is displayed, the execution of the `switch` statement should stop. To do this, you can use the `break` statement.
- The rest of the code to compare the value of `Compute_Code` with the characters `a, s, m or d` to determine the kind of operation will be entered for you.
- The program above displays the complete code to show the results of the various types of operations performed on the two numbers.
- The code is compiled and executed. Two numbers typed are `12` and `2`. The type of operation specified is `d` for divide. The output of the code displayed on the screen indicates that the quotient of the two numbers is `6`.
- If you write the same code displayed on the screen without `break` statements, the output will be different. This is because the `case` label that matches the passed variable and the `case` labels following it will be executed sequentially if a `break` statement is not used.
- Notice that the statements under all the labels including the `default` label will be executed. This is because of the absence of `break` statements.
- If the numbers typed are `4` and `2` and the operation type is `s`, indicating subtraction, the output is different. Notice that the statements under all the labels starting from label's and ending with the `default` label will executed.
- If the numbers typed are `4` and `2` and the operation type is `m`, indicating multiplication, the output displays the result of the division operation and the `default` label.
- If the operation type is `d` and the numbers typed are `4` and `2`, the output has the result for the division and `default` cases. This indicates that in the absence of `break` statements, all the `case` labels are executed starting from the label that matches the `switch` expression.

There can be some situations where the same action is performed for multiple values. In that case, multiple `case` labels can be followed by a single set of instructions. A code sample that uses this concept is given below.

Listing 26.2 : A program involving multiple cases inside a switch statement

```
//Listing 26.3 : code sample to demonstrate the use of one set of instructions
//to be performed for multiple case values

#include <stdio.h>
main()
{
    char alpha;
    printf (“Type a vowel now: \n”);
    scanf (“%c”, &alpha0;
    switch (alpha)
    {
            case ‘a’:
            case ‘e’:
            case ‘i’:
            case ‘o’:
            case ‘u’:
                    printf (“you typed the value %c.”, alpha);
                    break;
            default:
                    printf (“you did not type a vowel.”);
    }
}
```

Detailed Explanation

- The code is compiled and executed.
- If the value of alpha matches with one of the characters `a`, `e`, `i`, `o`, or `u`, the `printf` statement under label `u` is executed.
- Otherwise, the statement under the `default` label is executed.

LESSON 27 : THE while STATEMENT

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE while STATEMENT

To perform an action repeatedly, the C language provides certain loop statements. One of the loop statements is the `while` statement. The syntax of the `while` statement is given below. The `while` statement contains a condition and a `loopbody`. The condition is enclosed within parentheses. All the variables used in the condition of the `while` statement must be initialized at a point before the `while` loop is reached.

```
while(condition)
 Loopbody
```

Detailed Explanation

- The values of the variables used in the condition must be changed in the `loopbody`. Otherwise, the condition will always remain `true` and the loop will never terminate. You can write a single statement or multiple statements in the `loopbody`. Multiple statements should be enclosed within braces. Otherwise, the code may generate an unexpected output.
- If the condition of the `while` statement is `true`, the `loopbody` is executed. After the `loopbody` is executed, the condition is evaluated again. The sequence of evaluating the condition and executing the `loopbody` continues until the condition is `false`.

Let's write a code program that accepts a number and a letter from the user and displays the letter in a certain number of rows, which is indicated by the number typed by the user. The letter is displayed the number of times as indicated by the row number. The letter should be repeated in each row as many times as the row number. This can be done using the `while` statement. The code to accept a number and a letter is given below.

Listing 27.1 : A program to illustrate the usage of while loop

```
1   // Listing 27.1 : The code accepts a number and a letter and displays the letter in as
2   // many rows as the typed number. It also displays the letter in each row as many times as
3   // the row number.
4
5   #include <stdio.h>
6
7   main()
8   {
9       char alpha;
10      int count, y, x;
11
12      y=1;
13      x=1;
14      printf("Type a number: ");
15      scanf("%d", &cout);
16      printf("Type a letter: ");
17      alpha=getche();
18
19      while(x<=cout)
20              {
21                      y-x;
22                      printf("\n%c", alpha);
23
24                      while(y>1)
25                      {
26                              printf("%c", alpha);
27                              y--;
28                      }
29              x++;
30              }
31  }
```

Detailed Explanation

- In the code, a character variable, `alpha`, and three integer variables, count, `y`, and `x`, are declared. The variables `y` and `x` are assigned the value `1`.
- Two `printf` functions are written to prompt users to type a number and a letter. The `scanf` function will accept the number in the `count` variable. The `getche` function will accept a letter in the `alpha` variable.
- The typed letter should be displayed in as many rows as the typed number. You can specify this condition by using the while statement `(while(x<=count())` as in line `19`.
- The code is complete. Line `20` and Line `30` are the opening and closing braces of the outer `while` loop. In Line `21`, the value of the `x` variable is assigned to the `y` variable. In Line `22`, the value of the `alpha` variable is given.
- In Line `24`, another `while` loop is started. The condition of this `while` loop is that the value of the `y` variable is greater than `1`. Line `25` and Line `28` are the opening and closing braces of this `while` loop.
- Line `26` will display the value of the `alpha` variable. In Line `27`, the value of the `y` variable is decremented by `1`. In Line `29`, the value of the `x` variable is incremented by `1`.
- The code is compiled and executed. The program prompts you to type a number. `Type 5`.
- The program again prompts you to type a letter. `Type w`.

- Letter w is displayed in five rows. In each row, the letter is displayed as many times as the row number.
- The code given will help you understand how this output is obtained. In Line 19, the condition is that the value of x should be less than or equal to the value of the count variable.
- The number typed was 5. Therefore, the value of the count variable is 5. The value of x is 1, which is less than the value of count. Therefore, the condition is true and the flow of control moves inside the while loop.
- Inside the loop, the y variable is assigned the value of the x variable, which is 1. Next, the value of the alpha variable, w, is displayed in the first row.
- In Line 24, the value of y is checked whether or not it is greater than 1.
- The value of y is 1. Therefore, the condition becomes false. The flow of control skips this while loop and executes Line 29. As a result, the value of x becomes 2.
- The flow of control moves back to Line 1 where the value of x is again compared with the value of count. The value of x is 2. Therefore, it is still less than the value of count, which is 5.
- The condition evaluates to true, and the flow of control again moves inside the while loop. The value of y becomes 2 in Line 21. In Line 22, the printf function displays the value of alpha, w, in the second row.
- In Line 24, the value of y is compared with 1. The value of y is 2. Therefore, it is greater than 1. As a result, the condition becomes true and the control moves inside the second while loop. The value of alpha, w, is displayed again in the second row.
- In Line 27, the value of y is decremented by 1. Therefore, the value of y becomes 1. The control moves to Line 24 where the value of y is again compared with 1. The value of y is 1. Therefore, the condition becomes false and the control moves to Line 24.
- In Line 24, the value of x becomes 3. The flow of control moves back to Line 19. This looping continues until the value of x becomes 6. The flow of control moves out of the first while loop, and the program terminates.

If a semicolon is added after the condition in the while statement in a program, the program will result in an infinite loop. This is because the control never comes out of the while statement. A code sample given below.

Listing 27.2 : A program listing several forms of a while loop

```
// Listing 27.2 : sample code  to demonstrate the effect of using a semicolon after
// the condition in the while statement

#include <stdio.h>

main()
{
    int num = 5;
    while(num > 0);
    {
            printf("Hello World!\n");
            num--;
    }
}
```

You can use a `while` statement to perform an action repeatedly depending on a condition.

Self Review Exercise 27.1

1. *You learned about the* `while` *statement. A code sample that uses this statement and four possible results of the code are given below. Trace the correct output of the code now.*

```
#include <stdio.h>
main( )
{
int number = 5;

while(number < 10)
{
printf("%d\n", number);
number ++ ;
}
}
```

A	*B*	*C*	*D*
Infinite loop	5	5	1
	6	6	2
	7	7	3
	8	8	4
	9	9	5
		10	

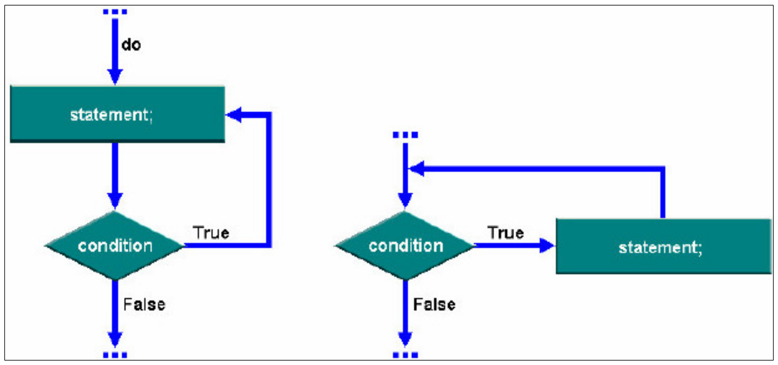
Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE do-while STATEMENT

The do-while statement is another loop statement provided by C. The statement helps to perform certain repetitive actions depending on a condition. The major difference between the do-while statement and the while statement is that in the do-while loop, the loopbody is executed once without the condition being evaluated.



do-while Statement **while Statement**

After the loopbody is executed once, the condition is checked. If the condition evaluates to true, the loopbody is executed repeatedly until the condition becomes false. You can use the do-while loop in situations where an action must be performed at least once without evaluating the condition.

The syntax of the do-while statement is given below. The do-while statement starts with the do keyword followed by the loopbody. In the loopbody, you can write a single statement or multiple statements enclosed within braces.

```
do
   {

    loopbody

   } while (condition);
```

After the closing brace, the while statement followed by the condition enclosed in parentheses is written. In the do-while statement, add a semicolon after the condition. The semicolon indicates that the do-while loop has come to an end.

Write a program that accepts a number and a letter from the user and displays the letter in the number of rows indicated by the typed number. In the first row, display the letter the number of times indicated by the typed number. In the subsequent rows, the letter should be displayed one less time than in the previous row. Next, check if the user wants to quit.

If the user wants to continue, repeat the process of accepting a number and a letter until the user wants to quit. If the user does not want to continue, terminate the program. You can create this program by using the `do-while` statement.

Listing 28.1 : A program illustrating do - while statement

```
//Listing 28.1 : demonstration of do - while
#include <stdio.h>
main()
{
char response;
char alpha;
int count, y, x;
do
{
     y=1;
     x=1;
     printf (“Type a number: “);
     scanf (“%d”, &count);
     printf (“Type a letter: “);
     alpha=getche();
     while (x <= count)
              {
                       printf(“\n”);
                       y=x;
                       while (y<=count)
                                {
                                         printf (“%c”, alpha);
                                         y++;
                                }
                       x++;
              }
     printf(“\n Do you want to continue (y/n)?: “);
     response=getche();
     printf(“\n”);
}while (response == ”y”);
}
```

Detailed Explanation

- Two character variables, `response` and `alpha`, and three integer variables, `count`, `y` and `x`, are declared
- In the program the brace after the `do` keyword is the opening brace of the `do-while` loop. Next, the `y` and `x` variables are initialized to `1`. The `printf` functions will prompt the user to type a number and a letter.
- The `scanf` function will accept a number from the user in the `count` variable. The `getche` function will accept a letter in the `alpha` variable.
- The code to display the typed letter in different rows is given. The `while` statement will evaluate the condition to check if the value of `x` is less than or equal to the value of `count`.

- If the condition is `true`, the flow of control will execute the statements in the outer `while` loop. The highlighted brace is the opening brace of the outer `while` loop. The `printf` function in the outer `while` loop will place the cursor on the next line in the output.
- Next, the value of `x` is assigned to `y`. The code `while (y <= count)` will be evaluated to check if the value of `y` is less than or equal to the value of `count`. If the condition evaluates to true, the flow of control will execute the statements in the inner `while` loop.
- The brace is the opening brace in Line No `20` of the inner `while` loop. The `printf` function in the inner `while` loop will display the value of the `alpha` variable. The next line of code will increment the value of the `y` variable by `1`.
- The brace is the closing brace in Line `26` of the inner `while` loop. In the next line, the value of `x` is incremented by one. The brace in the next line is the closing brace of the outer `while` loop.
- The `printf` function in the code will display a message whether the users want to continue or terminate the program. The `getche` function will accept the user's input in the `response` variable.
- The last `printf` function in the outer `while` loop will place the cursor on the next line in the output.
- After the user's response is accepted in the `response` variable, specify the condition to check if the `response` is equal to `y`. using `while (response == "y");`. As in Line `30`.
- The closing brace of the `main` function will be added. The completed code will be compiled and executed for you.
- If the number and the letter typed are `2` and `v`, respectively, then the `response` to the question is `n`.
- The values typed are `2` and `v`. Therefore, the value of count is `2` and the value of `alpha` is `v`. In Line `15`, the value of `x` is compared with the value of `count`.
- In Line `18`, `y` is assigned the value of the `x` variable. Therefore, `y` becomes equal to `1`. In Line `19`, the value of `y` is compared with the value of `count`. The value of `y` is `1`, and the value of `count` is `2`. Therefore, the condition evaluates to true.
- The `printf` function in the `while` loop is executed. As the result, the value of `alpha`, which is `v`. Next, the value of `y` is incremented by one. Therefore, the value of `y` becomes `2`.
- When the closing brace is encountered the flow of control moves back to Line `19`. In this line, the value of `y` is again compared with the value of `count`. The value of `y` is `2`, and the value of `count` is also `2`. Therefore, the condition evaluates to true.
- The statements inside the inner `while` loop are executed again, and another `v` is displayed in the same row. The value of `y` is incremented to `3`.

- When the flow of control moves back to Line `19`, the condition becomes `false` because the value of `y` is more than the value of `count`. The flow of control skips the inner `while` loop and executes Line `23`.
- The value of `x` becomes `2`. When the closing brace of the outer while loop is encountered, the flow of control moves to Line `15`. The value of `x` is again compared with the value of `count`.
- The condition evaluates to true because both the `x` and `count` variables have the same value, `2`. The statements inside the outer `while` loop are executed again.
- The value of `x` becomes `3`, and the flow of control skips the outer `while` loop because the condition becomes false. As a result, `v` is displayed only once in the second row.
- The program also prompts the users to indicate whether or not they want to continue with the program. The `response n` is compared with `y` in Line `28`. The condition evaluates to false, and the program terminates.

You can use this statement when a series of statements that are based on a condition are to be executed at least once without evaluating the condition.

Self Review Exercise 28.1

1. *Identify the correct statement about the* `do-while` *statement.*

   A. The `do-while` *statement starts with the* `do` *keyword followed by the* `while` *and the condition.*
   B. The `loopbody` *is executed at least once without the condition being evaluated.*
   C. The `do-while` *statement is used to execute only multiple statements.*
   D. The `loopbody` *is executed until the condition becomes* `true`.

2. *The following code that uses* `do-while` *statement and four possible results of the code. Choose the correct output of the code.*

```
#include<stdio.h>
main()
{
int num=0;
do
{
printf("The value stored in num is %d.\n", num);
}while(num==0)
}
```

   A. The program generates a parse error.
   B. The value stored in `num` *is* `0`.
   C. The value stored in `num` *is* `0`.
   The value stored in `num` *is* `0`.
   D. The program results in an infinite loop.

3. *The following code sample use the* `do-while` *statement and four results of this code. Choose the correct output of the display code now.*

```
#include<stdio.h>
main()
{
int x=1;
do
{
x- -;
printf("The number stored in x is %d,\n", x);
}while(x==0)
}
```

   A. The number stored in `x` *is* `0`.
   The number stored in `x` *is* `-1`.
   B. The number stored in `x` *is* `1`.
   The number stored in `x` *is* `0`.
   C. The number stored in `x` *is* `0`.
   D. The program results in an infinite loop.

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE `for` STATEMENT

The `for` statement is another statement that is used to handle iterative actions. The `for` statement creates a loop in which a set of statement is repeatedly executed until the test condition becomes `false`. The syntax of the `for` statement is given. The `for` statement starts with the `for` keyword. This statement has three elements enclosed within parentheses.

```
for (initialization expressions; test condition; update expressions)
        loopbody;
```

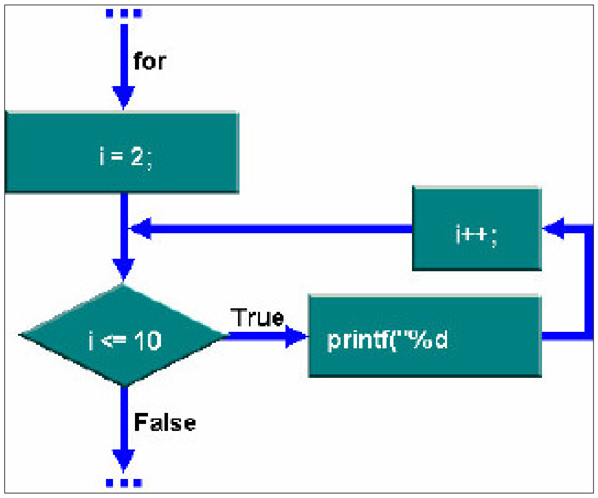
Detailed Explanation

- The first element of the `for` statement is initialization expressions. The `initialization expressions` element is usually an assignment, which is done only once before the `for` statement begins execution. The `test condition` is evaluated before each iteration of the `for` statement. The `test condition` determines whether the statement should continue or terminate.
- The `update expressions` change the values of the variables used in the test condition or any other associated value. The `update expressions` are executed at the end of each iteration after the `loopbody` is executed. The elements of a `for` statement are separated by semicolons.
- You can use multiple statements in the initialization and update expressions parts of the `for` statement by using the comma operator. The multiple statements separated by the `comma` operator are evaluated from left to right.

```
for (x=1,y=2; test condition; x++, y++)
                loopdody;
```

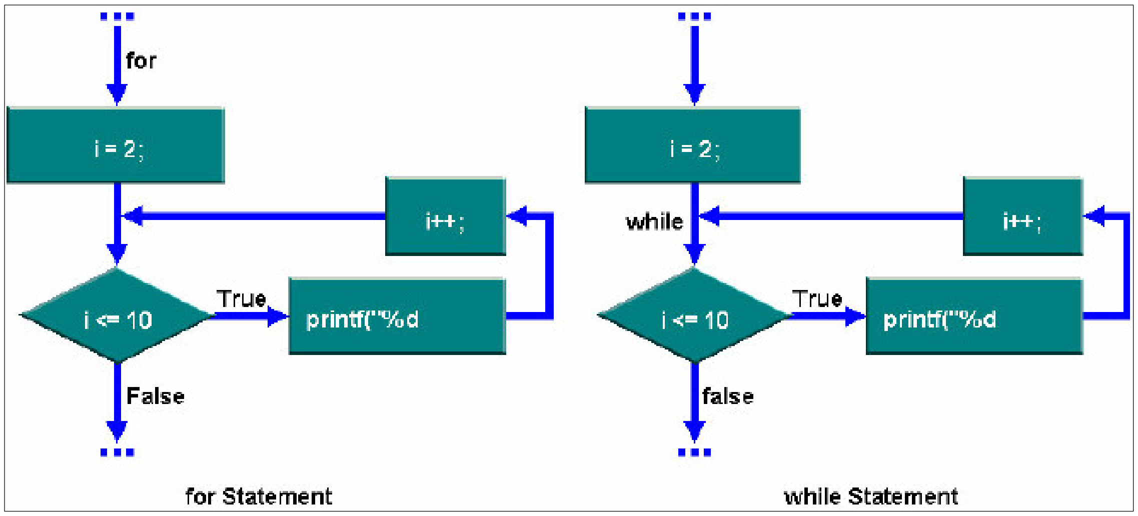
- In the `for` statement, you can omit the initialization and update expressions parts but you must provide two semicolons inside parentheses. If you also omit the `test condition` from the `for` statement, you program will result in an infinite loop.
- The `for` statement also has a `loopbody` in which you can use a single statement or multiple statements enclosed within braces.

Sequence in which the various parts of the `for` loop are executed is shown in the following diagram.



- In the `for` loop, first the variable `i` is initialized to `2`. Next, the value of `i` is compared with `10`. Next, the value of `i` is incremented by `1`.
- After the value of `i` is incremented, the test condition is evaluated again. If the `test condition` is true, the `printf` function is executed again. If the test condition is `false`, the flow of control comes out of the `for` loop.

As compared to the `while` statement, the `for` statement creates a similar loop and is more compact. Therefore, to reduce the size of the code, you use the `for` statement.



Two different samples of code, which will generate the same output, are given below. The first code (Listing 29.5) sample uses the for statement, and the second code (Listing 29.6) sample uses the while statement. Notice that the first code sample is more compact than the second.

```
for (i = 2; i <= 10; i++)

 printf ("%d

i=2;

while (i<=10)
 {
         printf ("%d
         i++;
 }
```

A program to display the even numbers between 2 and 10.

```
Listing 29.1 : A program illustrating for loop usage

//The code sample will display the even numbers between 2 and 10.
#include <stdio.h>
main()
{
    int i, x;
    for (i=2, x=0; i<=10; i++)
            {
              x - I % 2;
              if (x == 0)
              printf ("%d\n", i);
            }
}
```

Detailed Explanation

- In the code, two integer variables, i and x, are declared.
- Next, initialize the i and x variables. Specify the test condition for displaying the even numbers between 2 and 10. To do all this in one line of code, Enter for (i=2, x=0; i<=10; i++).
- The logic used to find the even numbers between 2 and 10 involves storing the result of i modulus 2 in the x variable. Then check if the value in x is equal to 0 or not.
- If the value of x is 0, it indicates that the value of i is an even number because even numbers are completely divisible by 2. To store the result of i modulus 2 in the x variable, type x=i%2;.
- Next, compare the value of x with 0. Type if (x==0)
- If the value of x is equal to 0, print the value of i. using printf(%d\n",i);
- Type the closing brace } of the for loop.
- Type the closing brace } of the main function
- The code is complete. It will be compiled and executed.
- The output of the program shows that all the even numbers between 2 and 10 are displayed.

- The flow of control enters the `for` loop. In Line `8`, the value of `i` modulus `2` is stored in `x`. the value of `i` is `2`. Therefore, the output of `i%2` is `0`. Consequently, the value of `x` is `0`.
- In Line `9`, the value of `x` is compared with `0`. The value of `x` is equal to `0`. Therefore, the test condition to `true` and the value of `i` is `2`.
- When the closing brace is encountered, the flow of control moves back to the update section of the `for` statement. In this section, the value of the `i` variable is incremented by one. Therefore, the value of `i` becomes `3`.
- The value of `i` is again compared with `10`. The value of `i` is less than `10`. Therefore, the test condition evaluates to `true`. The flow of control again moves into the body of the `for` loop.
- In the `loopbody`, the result of `i%2` is assigned to the `x` variable. The value of `i`, which is `3`, is not completely divisible by `2`. Therefore, the value of `x` is greater than `0`.
- The test condition in the `if` statement becomes `false`, and the value of `i` is because it is not an even number. In this way, the `for` loop iterates until the even numbers between `2` and `10` are displayed.

If for is used instead of the `while` statement, the `for` statement reduces the size of the code because it is more compact than the `while` statement.

Self Review Exercise 29.1

1. *Identify the correct syntax of the* `for` *statement.*

   A. `for(initialization expression; test condition; update expressions)`
   `Loopbody;`
   B. `for(initialization expression, test condition, update expressions)`
   `Loopbody;`
   C. `for(update expressions; test condition; initialization expression)`
   `Loopbody;`
   D. `for(test condition; initialization expression; update expressions)`
   `Loopbody;`

2. *The code sample uses statement and four possible result of the code are given. Choose the correct output of the code now.*

```
#include<stdio.h>
main()
{
int number;
for (number=10; number <15; number++0
{
printf("%d\n", number);
}
}
```

```
A. infinite loop    B. 10    C. 10    D. 14
                       11       11       13
                       12       12       12
                       13       13       11
                       14       14       10
                       15
```

LESSON 30 : break STATEMENT

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE `break` STATEMENT

In several situations, such as an error or incorrect input from the user, it may be necessary for the flow of control to come of a loop statement. For this purpose, the C language provides certain `jump` statements. One of the `jump` statements is `break`. The `break` statement forces the control to come out from a `switch` statement or from a `while`, `do-while`, or `for` loop.

After the forced exit from a `switch` statement or various loop statements, the control moves to the statement after the loops or the `switch` statement. In nested loops, the `break` statement functions in a different way. A `break` statement used in an inner loop of nested loops will transfer the flow of control to the immediate outer loop and not out of the entire nested loop.

The `break` statement is written as `break`; without any expressions or statements.

Write a program that accepts the product of 10 multiplied by 10. Give the user three attempts to provide the correct answer. If the answer is correct, display a message and terminate the program. The program should not prompt the user for an answer again if the user provides a correct answer in the first or second attempt. If the answer is incorrect, display a message and again prompt the user for an answer if it is not the user's third attempt.

Listing 30.1 : Using break statement

```
// Listing 30.1 : The code will accept the product of 10 multiplied by 10. The program will
// give users three attempts to provide the correct answer. If the answer is correct, a
// message will be displayed and the program will be terminated. For an incorrect answer, a
// message will be displayed and the user will be prompted again for the correct answer if
// it is not the users third attempt.

#include<stdio.h>

main()
{
    int i, num;
    num = 0;
    for(i=0;i<3;i++)
            {
                    printf("what is the product of 10 *10?");
                    scanf("%d",&num);
                    if(num==100)
                            break;
                    else
                            printf("wrong answer!\n");
            }
    if(i<3)
            {
                    printf("you typed the correct answer.");
            }
}
```

Detailed Explanation

- If the user provides the correct answer, the flow of control will be forced out of the `for` loop to the `if` statement after the loop. The code will be compiled and executed.
- If the number typed is `100`, the output of the code will be displayed as `You typed the correct answer`.

You cannot use the `break` statement without the `switch` statement or a loop statement such as `while` or `do-while`. The code sample below uses the `break` statement without a loop statement or a `switch` statement.

Listing 30.2 : Using break statement

```
// Listing 30.2 : Code sample to demonstrate the error generated when the break statement
// is used without the switch statement or loop statements

#include <stdio.h>

main()
{
    int num;
    printf("Type a number between 10 and 1000: ");
    scanf("%d", &num);
    if(num >= 10 && num <=1000)
            printf("you typed %d.", num);
    else
            break;
}
```

Detailed Explanation

- When the code is compiled, the compiler generates an error that the `break` statement is not within a loop or a `switch` statement.
- You can use the `break` statement to force the flow of control to exit from the loop or `switch` statements.

Self Review Exercise 30.1

1. *The* `break` *statement:*

   A. causes immediate exit from the switch statement and various types of loops.
   B. transfers control out of a program.
   C. transfers control out of the entire nested loop.
   D. is written as break();.

Self Review Exercise 30.2

1. *The following code sample uses the break statement and four possible results of the code. If the number typed is 12, choose the correct result of the remaining code now.*

```
#include <stdio.h>
main( )
{
  int i, num;
  num = 0;
for(i = 0; i < 0; i ++)
{
  printf("Type a number between 10 and 1000: “);
  scanf(“%d”, &num);
  if(num >= 10 && num <= 1000)
  {
   break;
   num = num + 1;
  }
  else
   printf(“Out of the specified range\n”);
}
printf(“You typed %d,” , num);
}
```

A. You typed 12.
B. The program does not generate output.
C. You typed 12.
You typed 12.
You typed 12.
D. You typed 13.

Lesson Objectives

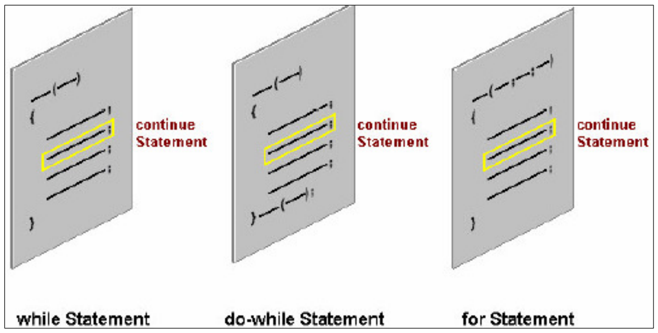
In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE `continue` STATEMENT

The `continue` statement stops the current iteration of a loop and immediately starts the next iteration of the same loop. When the current iteration of a loop stops, the statements after the `continue` statement are not executed. The `continue` statement is written as `continue;` in a code sample.

You can use the `continue` statement in the `while`, `do-while`, and `for` statements but not in the `switch` statement.



The `continue` statement causes the flow of control to move to the condition part of the `while` and `do -while` loops.

In the case of the `for` statement, the `continue` statement causes the flow of control to move to the update expressions part of the `for` statement before moving to the test condition part.

To understand how the `continue` statement can be used in a situation, evaluate the numbers from `1` to `10` and display the even numbers in this range.

If the value is an odd number, it should be skipped. The code sample to evaluate and display the even numbers and skip the odd numbers between `1` and `10` by using the `continue` statement is as follows.

LESSON 31 : `continue` STATEMENT

Listing 31.1 : Using the continue statement

```
1   //Listing 31.1 : code sample to display the even numbers between 1 and 10 by using the
2   // continue statement.
3
4   #include <stdio.h>
5   main()
6   {
7       int i, x;
8       For (i=1, x=0; i<=10; i++)
9               {
10                      x =  i % 2;
11                      if (x !=0)
12                              continue;
13                      else
14                              printf ("%d is an even number\n", i);
15              }
16  }
```

Detailed Explanation

- In the above program, two integer variables, `i` and `x`, are declared. The variables `i` and `x` are initialized to `1` and `0`, respectively, in the `for` statement. The test condition is that the value of variable `i` should be either less than or equal to `10`.
- In the initialization section of the `for` statement, the value of `i` is assigned the value `1`. Within the braces, the value of the expression `i` modulus `2` is assigned to `x`.
- In the `if` statement, the value of `x` is evaluated to check if it is not equal to 0. If the condition becomes true, the `continue` statement will skip the current iteration and execute the update part of the `for` loop.
- If the value of `x` is `0`, the flow of control will execute the `else` part of the `if` statement and the value of `i` will be displayed on the screen.
- The closing brace of the `for` loop will cause the flow of control to move to the update part of the `for` loop. Next, the test condition will be evaluated. If the test condition evaluates to true, the statements in the body of the `for` loop will be executed again.
- The `loopbody` of the `for` statement is executed repeatedly until the value of `i` becomes greater than `10` and all the even number between `1` and `10` are displayed on the screen.
- In Line `8`, variable `i` has the value `1` and variable `x` has the value 0. Therefore, the test condition `i<=10` evaluates to true. In Line 10, the value of `x` is `1`.
- In Line `11`, the test condition evaluates to true because the value of `x` is not equal to `0`. The flow of control moves to line `12` of the code.
- The `continue` statement stops the current iteration and moves the flow of control to the `i++` part of the `for` statement in Line `7`. The value of `i` becomes `2`. Next, the value of `i` is compared with `10`.
- The value of `i` is still less than `10`. Therefore, the condition evaluates to true and the flow of control executes Line `10` again. In Line `10`, the value of `x` becomes 0 because the output of `2% 2` is `0`.

- In Line 11, the condition becomes false and the `printf` function in the else part of the code is executed. As a result, 2 is displayed on the screen.
- The closing brace in Line 15 sends the flow of control to the update part of the `for` statement. In this way, the `for` loop iterates until the value of `i` becomes greater than 10 and all the even number between 1 and 10 are displayed.

Self Review Exercise 31.1

1. The continue statement:

A. transfers control out of a for loop.
B. transfers control out to the next loop in a nested loop construction.
C. transfers control to the beginning of the program.
D. stops the current iteration of a loop and immediately starts the next iteration of the same loop.

Self Review Exercise 31.2

2. *The code below uses continue statement. Four possible results of the code are also displayed on the screen. Choose the correct result of the code now.*

```
#include <stdio.h>
main()
{
int count, i, x;
for(count = 1, x=0, i = 0; count < = 4; count++, i++){
x=i%2;
if(x==0)
continue;
else
{
printf("%d is an odd number \n", i);
continue;
}}
printf("Only odd number");
}
```

A. infinite loop
B. 1 is an odd number.
  3 is an odd number.
  Only odd numbers
C. Only odd numbers
D. 0 is an odd number
  1 is an odd number.
  3 is an odd number.
  Only odd numbers

3. *The follows code sample uses statement and four possible results of the code. Choose the correct result now.*

```
#include<stdio.h>
main()
{
int i, x;
for (i =10, x = 0; i <= 15; i++){
x = i%2;
if(x!=0)
printf("%d is an number. \n", i);
else
continue;
}}
```

A. 11 is an odd number.
B. 11 is an odd number.
  13 is an odd number.
  15 is an odd number.
C. The program generates a syntax error.
D. The program terminates abruptly.

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

FUNCTIONS IN C

A function is a named and independent section of a C program. A function performs a specific task and may return a value to the calling function. When a function invokes another function, it is called a calling function.

A function can also accept arguments. Functions are an integral part of the programs that are written using the C language. There is no specific limit on the number of different functions that may occur in a program.

Types of Functions

There are two types of functions in C, library and user-defined.

Library Functions

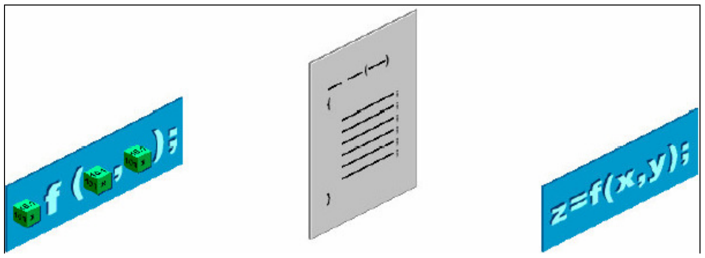
Library functions are predefined, ready - to - use functions made available to the program for their use. For example,

- `printf` function
- `scanf` function

User - Defined Functions

The functions created by programmers are called user-defined functions.

The user-defined functions will be referred to as functions. The use of a function in a piece of code involves writing a function prototype for the function and defining and calling the function.



Function Prototype

A function prototype enables a compiler to check whether or not a function is called correctly. A function prototype specifies the details, such as arguments, of a function to the compiler. Therefore, when the function is called, the compiler is aware of its details.

LESSON 32 : FUNCTION PROTOTYPE

The syntax of a function prototype is given below. A function prototype can contain three elements.

```
return_type name(parameter_list);
```

The first element : `return_type`

It indicates the data type of the value that is returned by the function. If the `return_type` element is omitted from a function prototype, the data type of the value that is returned by a function is assumed to be `int`. You can use the term `void` as the `return_type` to indicate that the function will not return a value.

```
return_type name(parameter_list);

int

void
```

The second element : `function_name`

The third element : `parameter_list`

In the `parameter_list` element, you specify the data types of the arguments that are accepted by the function. If the `parameter_list` element is omitted from a function prototype, the compiler does not check the data types of the arguments that are passed to the function.

In addition, the compiler automatically converts the data type of an argument passed to the data type that is specified in the function definition. You can use the term void in the `parameter_list` element to indicate that the function does not accept parameters. If you try to pass a parameter to the function, the compiler will generate an error. A function prototype must be terminated with a semicolon.

An example of a function prototype is given below. In this example, `int` is the data type of the value that is returned by the function. The name of the function is sum. The second `int` in the displayed function prototype is the data type of the argument that will be accepted by the `sum` function. The function prototype indicates that the `sum` function accepts an integer value as an argument and returns an integer value.

```
int sum(int);
```

You can also specify identifiers with the data types in the `parameter_list` element of a function prototype. In the function prototype displayed , `num1` is an identifier. Usually, a function prototype is written before the `main` function in a program. However, it can also be written within the `main` function.

```
int sum(int num1);
```

Some examples of function prototypes are:

- The function prototype `float abc(void);` indicates that the function `abc` will not accept an argument and will return a `float` value.
- The function prototype `void clear(void);` indicates that the clear function will not accept any argument and will not return a value.
- The `long xyz(int, int, int);` function prototype indicates that the function `xyz` accepts three integer arguments and returns a value of the `long` data type.
- The function prototype `long speed(int distance, int time);` indicates that the `speed` function accepts two integer arguments and returns a `long` value. The identifiers `distance` and `time` for the two arguments are also specified in the function prototype.
- In the `multiply(double num);` function prototype; the return type of the multiply function is not declared explicitly. Therefore, the data type of the value returned by the function is assumed to be `int`. The function prototype indicates that the `multiply` function accepts an argument of the data type `double`. The identifier `num` is also specified for the argument that is accepted by the function.
- In the `float divide(),` function prototype, the data type of the argument is not specified. Therefore, the compiler will not check the data types of arguments passed to the `divide` function. In addition, the function will return a value of the data type `float`.

A function prototype is an essential component of a C program. Therefore, always write function prototypes for the functions used in a program. This will help you avoid errors.

Self Review Exercise 32.1

1. *Identify the correct statement about a function prototype.*

   A. *If the* `return_type` *element is not specified in a function prototype, the data type of the value returned by the function is assumed to be* `void`.
   B. *A function prototype enables a compiler to check whether or not a function is called correctly.*
   C. *A function prototype consists of a fixed return type,* `int`*, which indicates the data type of the value returned by the function.*
   D. *A function prototype consists of more than three elements.*

2. *Identify the correct syntax of a function prototype.*

   A. name return_type(parameter_list);
   B. return_type name(parameter_list)
   C. return_type name(parameter_list);
   D. return_type name parameter_list;

3. *Identify the statement that describes the function prototype* `float divide(float,float);` *most appropriately.*

   A. *The function* `divide` *will accept two arguments of the* `float` *data type and will return a* `float` *value.*
   B. *The function prototype is syntactically incorrect.*
   C. *The function* `divide` *will accept two arguments of the* `float` *data type and will return multiple* `float` *values.*
   D. *The function* `divide` *will accept one argument of the* `float` *data type and will return two* `float` *values.*

LESSON 33 : FUNCTION DEFINITION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

FUNCTION DEFINITION

An important component required for using functions in a C program is the function definition. A function definition contains the statements that perform specific operations. A function definition has two parts, the header and the body. The format of a function definition is given below.

```
Header    return_type function_name(parameter_list)
          {
                  statement;
    Body          statement;
          }
```

The Header Section

The header consists of the `return_type` element, the `function_name` element, and the `parameter_list` element. The `return_type` element specifies the data type of the value returned by the function. A function header does not end with a semicolon.

The `return_type` element of a function can be any of the data types used in C, such as `char`, `int`, `float`, or `double`. You can also define a function that does not return a value by using the term `void` as the return type.

```
char, int, float, double, void

return_type function_name(parameter_list)
```

The next component in the header of a function is the `function_name` element. The `function_name` element should be unique and meaningful. For example, if you are creating a function to calculate the sum of two numbers, the name of the function should be `sum` and not `xyz`.

The third component of the header of a function is the `parameter_list` element. The `parameter_list` element is specified within parentheses. In the `parameter_list` element, you specify a data type and a variable name for each argument that is passed to the function when it is invoked. Multiple parameters having different data types can be specified for the `parameter_list`. The parameters should be separated by commas.

The statement float `surface_area(float x, float y)` is an example of a function header. The header indicates that the function `surface_area` accepts two float values in the `x` and `y` variables and returns a `float` value.

```
float surface_area(float x, float y)
```

The Body Section

The next part of a function definition is the body of the function. The body is enclosed within braces and immediately follows the header of the function.

The body of a function contains all the statements of that function. When a function is called, execution starts at the beginning of the body and terminates when either the `return` statement or the closing brace of the function is encountered. You cannot define another function within the body of a function. A function definition can be written either before or after the main function in a program.

Self Review Exercise 33.1

1. *Identify the format of the function header.*

   A. return_type function_name(parameter_list)
   B. return_type function_name(parameter_list);
   C. function_name(parameter_list) return_type
   D. return_type function_name parameter_list

2. *Identify a feature of the body of a function.*

   A. Consists of return type of the function
   B. Consists of a unique function name
   C. Contains the definition of another function
   D. Encloses its components within braces

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE `return` STATEMENT

The `return` statement explicitly terminates the execution of the function and passes the control to the calling function. You can use the `return` statement in a function to pass a value to the calling function. However, the `return` statement cannot pass more than one value.

The syntax of the `return` statement is displayed on the screen. The expression can be a value or a complex expression that returns a value. This value is returned to the calling function.

```
return expression;
```

The data type of the value to be returned to the calling function must match the return type specified in the function header.

A code sample that uses the `return` statement is given below. The `return` statement in the code passes the sum of the numbers stored in the `a` and `b` variables to the `main` function.

Listing 34.1 : Using return statement

```
// Listing 34.1 : The program accepts two numbers and display their sum. The use of
// the return statement is demonstrated in the code sample.

#include<stdio.h>

main()
{
    int num1, num2, total;
    printf("Type two numbers: ");
    scanf("%d %d", &num1, &num2);
    total = sum(num1, num2);
    printf("The sum of the two numbers is %d", total);
}

int sum(int a, int b)

    {
            return a+b;
    }
```

Detailed Explanation

- In the `main` function, the value returned by the `sum` function is assigned to the `total` variable. Next, the value of the `total` variable is displayed using the `printf` function.
- The code is compiled and executed. The two numbers typed are `24` and `33`, which are stored in `num1` and `num2` respectively. When the function `sum` is invoked, these numbers are copied to `a` and `b`. The function returns the sum of `a` and `b`, which is stored in `total`.
- The `return` statement can also be used without an expression. It is a good practice to use the `return` statement in a function even if the function does not return a value.

A program that uses the `return` statement that does not pass a value to the calling function is given below.

Listing 34.2 : A program to demonstrate return statement passing no value

```
// Listing 34.2 : The program accepts two numbers and display their sum. The use of
// the return statement that does not pass any value to the calling
// function is demonstrated in the code sample.

#include<stdio.h>

main()
{
    int num1, num2, total;
    printf("Type two numbers: ");
    scanf("%d %d", &num1, &num2);
    sum(num1, num2);
}

void sum(int x, int y)
{
    int total;
    total= x+y;
    printf("The sum of the two numbers is %d", total)
    return;
}
```

Detailed Explanation

- The `main` function in the code invokes the `sum` function, which displays the sum of two numbers.
- The `return` statement used in the code does not pass any value to the `main` function. It only transfers the control to the `main` function after all the statements in the `sum` function are executed.
- The code is compiled and executed. If the two numbers typed are `100` and `200`, the output is displayed as "`The  sum of the two numbers is 300`".

A function can contain multiple `return` statements but only one of them will be executed. This is because when the `return` statement is executed, the control is immediately transferred to the calling function.

The code below prompts you to type a number between `0` and `9`. The `main` function involves the `check` function to verify whether or not the typed number is between `0` and `9`.

Listing 34.3 : A program to demonstrate the use of multiple return statements

```
// Listing 34.3 : The program demonstrates the use of multiple return statement.

#include <stdio.h>

int check(int);
main()
{
    int num1, status;
    printf("Type a number between 0 and 9: ");
    scnanf("%d", &num1);
    status = check(num1);
    if(status ==1);
            printf("you typed %d", num1);
    else
            printf("The number typed is out of the specified range.");
}
check(int num1)
{
    if(num1 >=0  && num1<=9)
            return 1;
    else
            return 0;
}
```

Detailed Explanation

- The `check` function `returns 1` if the number typed is between `0` and `9`. If the typed number is not in the specified range, `0` is returned. Notice that multiple `return` statements are used to pass either `1` or `0` to the `main` function.
- The code to demonstrate the use of multiple return statements is compiled and executed. If the number typed is `5`, the control executes the `if` part of the `if-else` statement in the `check` function. Therefore, the function returns `1` to the `main` function, which is stored in status.
- If the number typed is `22`, the control executes the `else` part of the `if-else` statement in the `check` function. Therefore, the `check` function returns `0` to the `main` function, which is stored in status.

The `return` statement can be used in a called function a pass a value to the calling function.

Self Review Exercise 34.1

1. *The* `return` *statement:*

   A. passes two values to the calling function.
   B. transfers the control out of a loop statement.
   C. terminates the execution of a function and passes the control to the calling function.
   D. transfers the control out of a switch statement.

2. *A code sample having multiple* `return` *statements is given below . Find the incorrect place of the* `return` *statement*

```
#include <stdio.h>
void sum(int, int);
main()
{
       int input1, input2;
       printf("Type two numbers: “);
       scanf(“%d%d”, &input1, &input2);
       sum(input1, input2);
} return; // 1

void sum(int a, int b)
{
       return; // 2
       printf(“The sum of the two numbers is %d. “, a + b);
       return; // 3
       return; // 4
}
```

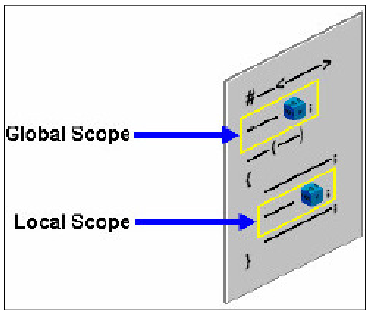
Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

SCOPES OF VARIABLES

The accessibility of a variable depends on its scope. The scope of a variable refers to that of a program in which the variable can be referenced or accessed. Scope can be of two types, global and local. A variable that is declared outside a function has global scope. However, a variable that is declared inside a function has local scope.



Global Scope

There are some rules related to the scope of variables. A variable with global scope can be used anywhere in a program after its declaration.

Local Scope

In contrast, a variable with local scope can be used only within the function in which it is declared.

The code sample given below illustrated the concept of local and global variable. In the code, a global integer variable `num` is declared and initialized to `5` outside the functions.

LESSON 35 : SCOPES OF VARIABLES

Listing 35.1 : Using global and local variables

```
//Listing 35.1 : Code sample to demonstrate the use of local and global variables

#include <stdio.h>

int num =5;
void display(char);

main()
{

char letter='x';

display(letter);

}
void display(char letter)
{
    int w=28;
    for(; num > 0; num--)

            printf ("%c \n", letter);
            printf ("The value of the local variable w is %d.\n",w);
}
```

Detailed Explanation

- The function prototype `void display(char);` indicates that the display function accepts an argument of the character data type and does not return a value.
- A local character variable, `letter`, is declared and initialized to `x` inside the main function. Next, the `display` function is invoked and the variable `letter` is passed to it.
- In the `display` function, a local integer variable, `w`, is declared and initialized to `28`. The `for` statement displays the value of the variable `letter` until the value of the global variable `num` becomes `0`.
- The second `printf` function in the `display` function displays the value of variable `w`.
- The code is compiled and executed. The output of the code is the value `x` of the variable `letter` is displayed once in each of five rows. This is because the `for` loop is executed five times.

Output
X
X
X
X
X

The value of the variable w is `28`.

- The value `x` is displayed as many times as the value of the global variable `num`. This is because the scope of the variable `num` is global and it is accessible in the entire program.
- The sentence in the output displays the value of the local variable `w`. The scope of variable `w` is local. Therefore, it can be used within the `display` function.
- If you try to use the local variable `w` outside the `display` function, the compiler will generate an error.

To illustrate the error generated by the compiler, the local variable `w` is used in the `main` function, `int` the program given below.

Listing 35.2 : A program using local and global variables

```
//Listing 35.2 : Code sample to demonstrate the use of local and global variables

#include <stdio.h>

int num =5;
void display(char);

main()
{

char letter='x';
display(letter);
printf ("The value of the local variable w is %d.\n", w);
}

void display(char letter)
{
    int w=28;
    for(; num > 0; num--)
            printf ("%c \n", letter);
            printf ("The value of the local variable w is %d.\n", w);
}
```

- During compilation, the compiler generates the error message that variable `w` is not declared in the `main` function.

There is another rule related to the scope of variables. If a global variable and a local variable share the same name, the function in which the local variable is declared uses the value of the local variable and not that of the global variable.

To illustrate the concept of global and local variable, a code sample is given below.

Listing 35.3 : A program demonstrating the use of local and global variables

```
//Listing 35.3 : Code sample to demonstrate the use of local and global variables

#include <stdio.h>

int num =5;

void display(char);

main()
{

char letter='x';
display(letter);
}

void display(char letter)
{
    int num = 2;
    for(; num > 0; num--)
            printf ("%c \n", letter);
}
```

- In the code, a global integer variable, `num`, is declared and initialized to `5`.
- In the `display` function, a local integer variable with the same name `num` is declared. This local variable is initialized to the value `2`.
- The code is compiled and executed. Letter `x` is displayed once in each of two rows. Although the value of the global variable `num` is `5`, `x` is displayed only two times because the value of the local variable `num` is `2`.

Another rule related to the scope of variables is that the name of a local variable can be used to declare another local variable of the same data type in another function. Similar names will not affect the scope of the two local variables.

The code sample given below uses two local variables that have the same name.

Listing 35.4 : A program using local variables

```
//Listing 35.4 : Code sample to demonstrate the use of the two local variable that
//have similar names

 #include <stdio.h>

int display(char);

main()
{

 char letter= 'x';
 int w = 2000;
 display(letter);
 printf("The value of the local variable w declared in the main function is %d.\n", w);
}

void display(char letter)
    {
    int w=28;
    for(;num>0;num--)
     printf (%c\n", letter);
    printf ("The value of the local variable w declared in the display function is %d.\n", w);
    }
```

Detailed Explanation

- in the `main` function, a local integer variable, `w`**,** is declared and initialized to `2000`.
- Another local integer variable with the same name `w` is declared and initialized to `28` in the `display` function.
- The code is compiled and executed. Letter `x` is displayed once in each of two rows. The global variable `num` has the value `2`**.** Therefore, the loop is executed twice.
- The output also displays the different values `28` and `2000` of both the local variables `w`**.** This indicates that having similar names does not affect the access rules of the multiple local variables declared in different functions.
- To conclude, you learned about the rules related to the scope of variables. This will help you use variables in programs according to their scope.

Self Review Exercise 35.1

1. *A variable that is declared:*

   A. outside a function has global scope.
   B. Outside a function has local scope.
   C. Inside a function has global scope.
   D. With global scope can be used only in the main function.

2. *A code sample with its four possible results are given below. Choose the right choice.*

```
#include<stdio.h>
int num=5;
int display(char);

main()
{
char letter = ‘n’;
display(letter);
printf(“The value of the local variable alpha is %d,\n”, alpha);
}
int display(char letter)
{
int alpha  = 1000;
while(num == 5)
{
printf(“%c\n”, letter);
num++;
}
}
```

   A. The compiler will generate an error that the alpha variable is not declared in the main function
   B. w
   C. w
   The value of the local variable alpha is 1000.
   D. The compiler will generate an error that the num variable is not declared in the display function.

Self Review Exercise 35.2

3. *You learned about using two variables with the same name. You also learned about using a local variable and a global variable with the same name. A code sample and its four possible results are give below. choose the correct choice.*

```
#include<stdio.h>
int display(char);

main()
{
char letter = 'w';
display();
printf("The value of the variable letter in the main function is %c.\n", letter);
}
void display()
{
char letter = 'a';
printf("The value of the variable letter in the display function is %c.\n", letter);
}
```

A. The value of the variable letter in the display function is a.
The value of the variable letter in the main function is w.
B. The compiler displays a warning that the same name is used for two variable.
C. The value of the variable letter in the display function is w.
The value of the variable letter in the main function is a.
D. The value of the variable letter in the display function is w.
The value of the variable letter in the main function is w.

LESSON 36 : BLOCKS IN A FUNCTION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

BLOCKS OF CODE

A block consists of one or more lines of code enclosed within opening and closing braces. A block can contain many nested blocks. In addition, a block can contain parallel blocks. Parallel blocks are the blocks that come after one another.

There are various rules related to the scope of the variables that are declared inside nested and parallel blocks.

- First, you learn about the rules related to the scope of the variables declared in nested blocks. A variable declared in an outer block can be accessed in inner blocks if it is not declared again in the inner block.
- Another rule is that if a variable is declared in an outer block and another variable with the same name and any data type is declared in an inner block, the outer block variable is masked in the inner block.
- When the flow of control returns from the inner block to the outer block, the variable that was redeclared retains its original value.

The code sample given illustrates the scope rules for nested blocks.

Listing 36.1 : A program to demonstrate the scope rules for nested blocks

```
//Listing 36.1 : code sample to demonstrate the scope rules for nested blocks

#include<stdio.h>

void number();
main()
{
    number();
}
void number()
{
    int b=5, c=4;
  printf("\n\n Outer loop: The value of b is %d-", b);
  printf("\n\n Outer loop: The value of c is %d-", c);
    {

    int b=10;
    printf("\n\n Inner loop : The value of b is %d-", b);
    printf("\n\n Inner loop: The value of c is %d-", c);

    }
  printf("\n\n Outer loop again: The value of b is %d-", b);
  printf("\n\n Outer loop again: The value of c is %d-", c);

}
```

Detailed Explanation

- In the numbers function, two integer variables, `b` and `c`, are declared and initialized in the outer block.
- The highlighted `printf` functions will display the values of the `b` and `c` variables.
- The variable `b` is reinitialized to `10` in the inner block. The two highlighted `printf` functions will display the values of the `b` and `c` variables again.
- The last two `printf` functions in the outer block will display the values of the `b` and `c` variables again.
- In line `17` of code, the values of the variables `b` and `c` are `5` and `4`, respectively. The `printf` functions in Line `18` and Line `19` display these values.
- In Line `22`, the variable `b` is reinitialized to `10`. Therefore, the value of `b` in the inner block is `10`. The `printf` function in Line `23` displays `10` as the value of the variable `b`.
- The value `10` is displayed in Line `23`. This is because if variable `a` is declared in the outer block and another variable with the same name is declared in the inner block, the outer block variable is masked in the inner block.
- The `printf` function in Line `24` displays the value of the variable `c`. The value of `c` is `4`. This indicates that the values of the variables declared in the outer block can be accessed in the inner block.
- The `printf` function in Line `27` displays the initial value of the variable `b`, which is `5`. Notice that the value of `b` is displayed in Line `27` is different from the value of this variable displayed in Line `6`.
- The reason is that after the control returns from the inner block to the outer block, the value of the variable `b` that was declared in the outer block is displayed.
- The `printf` function in Line `10` displays `4` as the value of the variable `c`.

The scope rule for a parallel blocks is different from the rules for nested blocks. If an outer block has two inner parallel blocks, a variable declared in the first inner block is not available to the second inner block or the outer block.

Similarly, a variable declared in the second inner block is not available to the first inner block or the outer block.

The code given below has the numbers function.

Listing 36.2 : Demonstration of scope rule for parallel blocks

```
//Listing 36.2 code sample to demonstrate the scope rule for parallel blocks

#include<stdio.h>

void members();
main()
{
    numbers();
}
void numbers()
{
   //outer block
{

   int x=5;          //block
   printf("\n\n block1: The value of z is %d.",z);

}

{                    //block
   int z=2000;
   printf("\n\n block2: The value of x is %d", x);
}
}
```

Detailed Explanation

- This function contains two parallel inner blocks, `block1` and `block2`. In `block1`, an integer variable, `x`, is declared and initialized to `5`. Another integer variable, `z`, is declared and initialized to `2000` in `block2`.
- The first block, `block1`, contains a `printf` function to display the value of the variable `z`, which is declared in `block2`. The second block, `block2`, also contains a `printf` function to display the value of the variable `x`, which is declared in `block1`.
- When the code is compiled, the compiler generates an error indicating that `the variable z and x are undeclared`. This is because although the two variables are declared, they are being used in the blocks other than those in which they are declared.
- The scope rule for parallel blocks states that the variables declared in parallel blocks are not available to each other.

You learned about scope rules for the variable used in nested and parallel blocks. Using variables according to their scope in nested and parallel blocks will help you avoid errors in program.

Self Review Exercise 36.1

1. *You learned about the way in which the scope rules for the variable used in nested blocks affect the output of a piece of code. A code sample and its four possible results are given below. Find the correct choice.*

```
# include <stdio.h>
void letters();
main()
{
    Letters();
}
void letters()
{
Char alpha1 = ‘b’, alpha2 = ‘c’;
printf(“\nOuter block : The value of alpha1 is %c.”, alpha1);
printf(“\nOuter block : The value of alpha2 is %c.”, alpha2);
{
int alpha2 = ‘z’;
printf(“\nInner block : The value of alpha1 is %c.”, alpha1);
printf(“\nInner block : The value of alpha2 is %c.”, alpha2);
}
printf(“\nOuter block again : The value of alpha2 is %c.”, alpha2);
printf(“\nOuter block again : The value of alpha1 is %c.”, alpha1);
}
```

Outer block : The value of alpha1 is b. *Outer block : The value of alpha2 is c.* *Inner block : The value of alpha1 is b.* *Inner block : The value of alpha2 is z.* *Outer block again : The value of alpha2 is c.* *Outer block again : The value of alpha1 is b.* *A*	*Outer block : The value of alpha1 is b.* *Outer block : The value of alpha2 is c.* *Inner block : The value of alpha1 is b.* *Inner block : The value of alpha2 is z.* *Outer block again : The value of alpha2 is z.* *Outer block again : The value of alpha1 is b.* *B*
Outer block : The value of alpha1 is b. *Outer block : The value of alpha2 is c.* *Inner block : The value of alpha1 is b.* *Inner block : The value of alpha2 is c.* *Outer block again : The value of alpha2 is c.* *Outer block again : The value of alpha1 is b.* *C*	*Outer block : The value of alpha1 is b.* *Outer block : The value of alpha2 is c.* *Inner block : The value of alpha1 is z.* *Inner block : The value of alpha2 is b.* *Outer block again : The value of alpha2 is c.* *Outer block again : The value of alpha1 is b.* *D*

Self Review Exercise 36.2

1. *You learned about the way in which the scope rule for the variables used in parallel blocks affects the output of a piece of code. A code sample and its four possible results are given below. Choose the correct choice.*

```
# include <stdio.h>
void letter();
main()
{
     Letter();
}
void letter()
{         //Inner block1
     char store1 - 'a';
     printf("\n\nInner block1 : The value of the store2 variable is %c.", store3);
}
{         //Inner block2
     char store2 - 'v';
     printf("\n\nInner blick2 : The value of the store1 variable is %c.", store1);
}
}
```

A. Inner block1 : The value of the store2 variable is a.
Inner block2 : The value of the store1 variable is v.
B. Inner block1 : The value of the store2 variable is v.
Inner block2 : The value of the store1 variable is a.
C. The compiler generates an error that variables store2 and store1 are not declared.
D. Inner block2 : The value of the store1 variable is a.

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

CALLING A FUNCTION

After defining a function, you can call it to perform the intended action. A function call is an expression that can be used either as a separate statement or within other statement. A function is invoked using the call by value method. In this method, a copy of an argument is passed to the function.

FUNCTION CALL BY VALUE

The value of the argument can be altered inside the called function. However, the original value of the argument in the calling function will not be affected. A function is called using its name followed by its arguments enclosed within parentheses. The number and type of arguments being passed to the function being called must be similar to those specified in the function prototype and the function definition.

If a function returns a value and you want to store the value, then the call statement is written in a specific way. For example, to call the function text while passing variable a and storing the return value of the function in the `x` variable, uses `x=text(a);` as the call statement. Write a piece of code to accept two numbers from users and display their sum.

Listing 37.1 : A program implementing a simple call by value mechanism

```
//Listing 37.1 : A simple program implementing call by value
//The code sample accepts two numbers and displays their sum.  The code
//demonstrates the method to call a function by using the call by value method.

#include< stdio.h>

int sum(int, int);

main()
{

        int num1, num2, result;
        printf("Type two numbers:");
        scanf("%d", &num1);
        scanf("\n%d", &num2);
        result = sum(num1, num2);
        printf("The sum of the two numbers %d and %d is %d.", num1, num2, result);

}

sum(int x, int y)
{

        int d;
        d=x + y;
        return d;

}
```

Detailed Explanation

- In the main function, three integer variables, `num1`, `num2`, and `result`, are declared.
- The `printf` function in Line 13 will display a message to type two numbers. The next two `scanf` functions will accept the two numbers in the `num1` and `num2` variables.
- The `printf` function in Line 17 will display the sum of the two numbers specified by the users. The `sum` function will accept two integer variables and return an integer value.
- The `sum` function will store the two typed numbers in the `x` and `y` variables and return their sum to the calling function, which is the main function.
- An integer variable, `d`, is declared in the `sum` function. In Line 25, the sum of the values stored in the `x` and `y` variables is assigned to variable `d`.
- In line 26, the `return` statement will pass the value of variable `d` to the calling function, which is the `main` function.
- The `sum` function can return the sum of the two typed numbers in the `main` function only if it is called in the `main` function. The value returned by the `sum` function is being displayed in the `main` function. Therefore, this value must be trapped.
- You can write the call statement to call the `sum` function while passing the two numbers typed by the user as arguments and store the value returned by the function in the result variable.
- To call the `sum` function, type `result=sum(num1,num2);`
- The code sample to accept two numbers from a user and display their sum is completed. It is compiled and executed.
- If the two numbers typed are 23 and 11, the output of the code is displayed as
  ```
  The sum of the two numbers 23 and 11 is 34.
  ```
- In Line 14, the first number typed 23, is stored in the `num1` variable. The second number typed, 11, is stored in the `num2` variable in Line 15.
- In line 16, the `sum` function is called and the `num1` and `num2` variable are passed as arguments. The control moves to the `sum` function. In this function, the two typed numbers are stored in the `x` and `y` variables.
- In Line 25, the sum of the values of the `x` and `y` variables is assigned to variable `d`.
- In Line 26, the `return` statement transfers the value of variable `d` to Line 16 of the `main` function. In Line 16, the value of variable `d` is assigned to the `result` variable. In Line 17, the sum of the two numbers is displayed.

The call statement for a function that does not return any value is written in a specific manner. For example, `text(a);` is the call statement to invoke the function `text` while passing variable **a.**

Write a piece of code to accept two numbers from users and display their sum.

Listing 37.2 : A program to demonstrate call by value mechanism

```
// Listing 37.2 : A program demonstrating call by value mechanism
// code sample accepts two numbers and display their sum. The code
//demonstrates the method to call a function by using the call by value method.

#include<stdio.h>

void sum(int,int);

main()
{

      int num1,num2,result;
      printf("type two numbers :");
      scanf("%d",&num1);
      scanf("\n%d",&num2);
      sum(num1,num2);

}

sum(int x, int y)
{

printf("The sum of the numbers %d and %d is %d"x,y,x+y);

}
```

Detailed Explanation

- Three integer variables, `num1`, `num2`, and `result`, are declared in the `main` function.
- The `printf` function will prompt the users to type two numbers. The two `scanf` functions will accept the values of the two numbers to be stored in the `num1` and `num2` variables.
- The `sum` function in the code will accept two integer variables and display the sum of their values. The `sum` function does not return a value to the calling function.
- The `sum` function will display the `sum` of the two typed numbers only when it is called in the `main` function. Call the `sum` function and pass the variables `num1` and `num2` as arguments to this function. `sum(num1,num2);`
- The code to accept two numbers from users and display their sum is completed. The code will be compiled and executed.
- If the number typed are `23` and `11`, the output of the code is displayed as `The sum of the numbers 23 and 11 is 34`. The last line of output was printed when the control was in the `sum` function.
- After the numbers `23` and `11` are entered, the `sum` function is called from the `main` function and the two numbers are passed to the called function. As a result, the control moves to the `sum` function.

- In the `sum` function, the two typed numbers are copied to the `x` and `y` variables. Next, the `printf` function displays the sum of the values stored in the `x` and `y` variables.

In the call by value method, a copy of a variable is passed as an argument to the called function.

Self Review Exercise 37.1

1. *Identify the statement that describes the call by value method of invoking a function.*

   A. A copy of an argument is passed to a function.
   B. The data types of arguments passed to the called function may be different from the data type of arguments specified in the function prototype and the function definition.
   C. The value of an argument passed to a function can not be altered inside the called function.
   D. If the value of an argument passed to a function is altered inside the function, the original value of the argument is affected in the calling function.

2. *You learned to call a function by using the call by value method. The program below has four possible options to call the multiply function. Find the correct choice*

```
#include <stdio.h>

int multiply(int, int);

main()
{

int number1, number2, output;
printf("Type two numbers: ");
scanf("%d", &number1);
scanf("\n%d", &number2);
// . . . . . . . . . . . . . . .
printf("The product of %d and %d is %d.", number1, number2, output);

}
```

   A. Output = multiply();
   B. Multiply(number1, number2);
   C. Output = multiply(number1, number2);
   D. Multiply(number1, number2) = output;

LESSON 38 : FUNCTION RECURSION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

FUNCTION RECURSION

Recursion is a feature of the C language in which a function calls itself either directly or indirectly. You can replace recursion used in a piece of code with a loop statement. You may prefer to use recursion instead of loop statements.

Listing 38.1 : Using recursion

```
1   // Listing 38.1 : This program calculates factorial of a given number.
2   #include <stdio.h>
3   int f, x;
4   int factorial(int a);
5
6   main( )
7   {
8       printf("Type an integer value between 1 and 8: “);
9       scanf(“%d”, &x);
10
11      if(x > 8 || x < 1)
12      {
13              printf(“Number is out of the specified range”);
14      }
15      else
16      {
17              f = factorial(x);
18              printf(“The factorial of %d is %d.\n”, x, f);
19      }
20  }
21
22  int factorial(int a)
23  {
24      if(a == 1)
25      return 1;
26      else
27      {
28              a *= factorial(a - 1);
29              return a;
30      }
31  }
```

This is because the code that uses recursion is compact as compared to the code that uses loop statements to perform the same action. In addition, it is easier to write some programs by using recursion. For example, you can use recursion to create a program that calculates the factorial of any number between `1` and `8`.

Detailed Explanation

- In the code, two global integer variables, `f` and `x`, are declared. The third line is the function prototype of the factorial function.
- The `printf` function in the `main` function will prompt you to type a number between `1 and 8`**.** The scanf function will store the typed number in variable `x`**.**

- The if statement in Line `11` will evaluate whether the typed number is greater than `8` or less than `1`. If the typed number is not between `1` and `8`, the `printf` function will display the message that the number is out of the specified range.
- If the typed number is between `1 and 8`, the lines of code in the else part of the `if` statement will be executed. In Line `17`, the `factorial` function will be called and variable `x` will be passed to it as an argument.
- The value returned by the `factorial` function will be stored in `the f` variable. In Line `18`, the `printf` function will display the number typed by the user and its `factorial` stored in the `f` variable.
- Line `22` contains the header of the `factorial` function definition. In Line `24`, the `if` statement will compare the value of variable `a` with `1`. If the value of `a` is equal to `1`, the `return` statement in Line `29` will pass `1` to the `main` function.
- If the value of `a` is not equal to `1`, the `else` part of the `if` statement will be evaluated. To calculate the factorial of the typed number `a*=fact(a-1);`
- To pass the value of a to the `main` function, type `return a;`
- The closing braces for the `else` block and the `factorial` function will be entered for you.
- The code is completed. It will be compiled and executed.
- The program prompts you to type a number between `1` and `8`. Type `5`.
- In Line `11` the typed number is evaluated to check if it is greater than `8` or less than `1`.
- The number specified is `5`. Therefore, the if condition evaluates to false. As a result, Line `17` is executed. The factorial function is called and the typed number is passed to this function.
- In the `factorial` function, the typed number is copied to the `a` variable. In Line `24`,the value of `a` is compared with `1` to check if they are equal. The value of `a is 5`. Therefore, the `if` condition evaluates to false.
- The flow of control moves into the `else` part of the `factorial` function. Line `28` is executed. In this line, the `factorial` function is called again and passed the value `4`**.**
- The value returned by the `factorial` function will be multiplied by the value of `a`**,** and the prod-uct will be assigned to variable `a`**.**
- When the `factorial` function is called again in Line `28`, the control moves to Line `22`. The value of `a` is `4`. After Line `22` is executed, the control moves to line `24`. in this line, the `if` condition is evaluated to `false`.
- As a result, the control moves to the else part of the code. In Line `28`, the `factorial` function is called again and passed the value `3`. Therefore, the flow of control moves to Line `22` again and the value of `a` becomes `3`.

- The condition in Line 24 is still `false`. Therefore, Line 28 is executed again. The `factorial` function is called and passed the value 2.
- The control again moves back to Line 22, and the value of `a` becomes 2. The `if` condition evaluates to false, and the control moves to the `else` part. The `factorial` function is called again and passed the value 1 in Line 28.
- The flow of control again starts execution at Line 22, and the value of `a` becomes 1. The `if` condition evaluates to true because the value of `a` is equal to 1. in Line 25, the `return` statement returns the value 1 to Line 17.
- In Line 28, the value of `a` becomes 2. The `return` statement in line 29 passes this value to the `factorial` function that was called the second to the last time.
- As a result, the value of `a` becomes 6. This value of `a` is again returned to the `factorial` function that was called with the value 3. Therefore, the value of `a` becomes 28.
- The value 24 is again passed to the factorial function that was called with the value 4. The value of a finally becomes 120. This value of a is returned to the `main` function in Line 17.
- In Line 17, the value returned by the `factorial` function is assigned to the `f` variable. This value is the factorial of 5. The `printf` function in Line 15 displays this value of the `f` variable on the screen.

The use of recursion instead of loping statements in piece of code will reduce the size of the code.

LESSON 39 : STRUCTURED PROGRAMMING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

STRUCTURED PROGRAMMING

An efficient method to solve a problem is to break the problem into small and manageable pieces.

A structured program uses independent functions to perform individual tasks. A C program that uses individual functions for performing individual tasks is an example of structured programming.

There are many advantages of structured programming. Structured programming simplifies program debugging. The structured design of a program helps you to isolate a problem to a specific section of a piece of code.

In structured programming, you can reuse a function to perform a similar task in different programs. This saves time. You can reuse a function to perform similar subtasks in various tasks. This not only saves time but also reduces the code size and makes the program efficient.

In a structured program, it is easy to maintain the code. If you need to modify a specific functionality in a program, it is easy to locate the function that incorporates the functionality in a structured program.

ADVANTAGES OF STRUCTURED PROGRAMMING

- Simplifies Program Debugging
- Help in Isolating a Problem
- Enables the Reuse of Functions
- Helps in Reducing the Code Size
- Simplifies the Code Maintenance

A structured program uses independent functions to perform individual tasks in a program.

Self Review Exercise 39.1

1. *Select an advantage of structured programming.*

   A. Reduce the number of modules used in a piece of code.
   B. Simplifies program debugging.
   C. Difficult to modify as compared to non-structured programming.
   D. Reduces compilation time, regardless of the code size

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ARRAYS

Programmers often find it difficult to handle a large number of variables in an application. One efficient way to reduce the number of variables used in an application is to use arrays.

An array is defined as a regular and imposed grouping or arrangement of the same data types. This topic details the features of an array.

FEATURES

A feature of an array is that it is a contiguous memory location, which is referred to by a common name. An array is a special data type. The variable of an array data type can store more than one value at the same time. This is an advantage that fundamental data type variables do not provide.

For example, an integer variable can store only one number at a specified time. You must declare five different variables of the type integer to store five numbers whereas an array can store all five numbers at the same time.

Another feature of an array is that it represents a group of similar objects. For example, you can use an array to store the name of all the employees of an organization.

However, a single array cannot contain both the name and basic salaries of the employees of an organization. This is because names and basic salaries are different types of objects. The names and basic salaries of employees must be stored in two arrays.

The objects of an array are called elements. The size of an array is defined by the number of elements in that array. Each element of an array can be referred to by using the array name and a subscript.

A subscript is also known as an index. A subscript uniquely identifies an element. The value of the first subscript is 0, and the values of the subsequent subscripts increase by one. Therefore, the value of the last subscript is one less than the array size.

Arrays can be of many dimensions. An array can be described as a table with rows and columns. An array with a single row or column is called a one-dimensional array. An array that consists of more than one row and one column is called a multidimensional array.

Arrays are contiguous memory locations that store more than one value at one point of time. If an application needs many values of the same data type and at the same time, you can use an array to store these values.

Self Review Exercise 40.1

1. *An array is a:*

   A. contiguous memory location.
   B. user-defined data type.
   C. simple data type.
   D. representation of different types of objects.

2. *Select a feature of an array.*

   A. It is a disjoined memory location.
   B. It is a group of similar objects.
   C. It is a fundamental data type.
   D. It can store integer and character values at the same time.

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ARRAY DECLARATION

An array allows you to work with a collection of values of the same data type. Like any other data type, an array needs to be declared before it can be used in an application.

The syntax to declare a one-dimensional array is given below. A one-dimensional array declaration statement starts with the data type of the values that the array will store.

```
data type arrayname[n];
```

The data type is followed by the name of the array. The value `n` in the array declaration defines the number of elements. The number of elements of an array represents the maximum number of elements in a given dimension. In a one-dimensional array, `n` can be replaced by any positive finite integer or any constant expression that results in a positive finite integer.

The number of elements of an array represents the maximum number of elements in a given dimension. In a one-dimensional array, `n` can be replaced by any positive finite integer or any constant expression that results in a positive finite integer.

Consider a situation in which five integer values need to be stored. You can declare five integer variables to store five integer values. Alternatively, you can declare an integer array of the size 5 to store five integer values. This is illustrated by the example given below.

Listing 41.1 : Declaring one - dimensional array

```
//Listing 41.1 : This program declares a one-dimensional array.

#include <stdio.h>
main()
{
      int a, b, c, d, e;
      int number[5];
      int number1[100];
}
```

Detailed Explanation

- Similarly, to store `100` integers, you can declare either `100` integer variables or an integer array of the size `100`.
- If you need to store many similar values, such as `100` integer values, at the same time, it is better to declare an array of `100` elements. This is because declaring and managing `100` different integer variables is very difficult.

- If an array consists of `n` elements, the array starts at the index `0` and ends at the index `n-1`. Therefore, the largest valid index in the array `a[100]` that contains `100` elements is `99`.

Consider another integer one-dimensional array declaration statement given below. The array can store `10` integers. The integers will be stored in the elements from `number[0]` through `number[9]`.

Listing 41.2 : Declaration of one - dimensional array

```
//Listing 41.2 : This program declares a one-dimensional array.

#include <stdio.h>

main()
{
    int number[10];
}
```

A one-dimensional float type array is declared in the same way as a one-dimensional integer array is declared. Consider the example shown below. This statement declares a float array of 10 elements, called `mynumber`.

Listing 41.3 : Declaration of one - dimensional array

```
// Listing 41.3 : This program declares a one-dimensional array.

#include <stdio.h>

main()
{
     float mynumber[10];

     char mychar[10];
}
```

Detailed Explanation

- A one-dimensional character array can be declared just as integer and float type arrays are declared. In C, there is no separate data type called `string`. A `string` is represented as an array of characters.
- Consider the example given. The array `mychar` is an array of `10` characters. The `10` characters will be stored in the elements from `mychar[0]` through `mychar[9]`.
- If the last element of a character array is the `NULL (\0)` character, the array can be classified as a `string`. Without it, the array is just an array of characters. The `NULL` character is also referred to as the string terminator.

Self Review Exercise 41.1

1. *You learned about array declaration. Specify the maximum number of valid elements for the one-dimensional array declaration statement given below. Enter the number of valid element.*

```
#include<stdio.h>
main()
{
int number[51];
}
```

2. *You learned about array declaration. Specify the maximum number of valid elements for the one-dimensional array declaration statement given below. Enter the number of valid element.*

```
#include<stdio.h>
main()
{
char Name[85];
}
```

LESSON 42 : ONE-DIMENSIONAL ARRAYS --> INITIALIZATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ARRAY INITIALIZATION

Initialization is a process in which values are assigned to variables for the first time in an application. It is a good practice to initialize an array. Otherwise, it can contain superfluous values from any previous application.

An array can be directly initialized. This implies that while declaring the array, you can specify the values of some or all elements or the array. Alternatively, you can declare an array and then initialize it.

The direct initialization of an array is possible in two ways. In the first way, individual values are assigned to the array. In this method, it is essential that the string terminator or the `\0` character be explicitly assigned.

Listing 42.1 : Initializing one - dimensional array

```
// Listing 42.1 : This program initializes a one-dimensional array.

#include<stdio.h>

main()
{
   Char str1[6] = {‘T’, ‘O’, ‘P’, ‘I’, ‘C’, ‘\0’};
}
```

Detailed Explanation

- In the above example, each value is separately assigned to the character array.
- Each value in the example must be enclosed within single quotation marks. The `\0` character also needs to be enclosed within single quotation marks.

In the second way, all the values are assigned as a single string of characters. In this case, the string terminator or the `\0` character is attached automatically to the array by the compiler.

In the example given below, the array `str2` is initialized with the string of characters `TOPIC`. In this case, the value must be enclosed within a set of double quotation marks.

Listing 42.2 : A program to initialize one - dimensional array

```
// Listing 42.2 : This program initializes a one-dimensional array.

#include<stdio.h>

main()
{
   Char str1[6] = {“TOPIC”};
}
```

In the following example, an integer array is initialized. The integer and floating - point arrays are initialized in the same way. To initialize an integer array or a floating - point array, individual elements must be assigned to the array.

Listing 42.3 : Initializing one - dimensional array

```
// Listing 42.3 : This program initializes a one-dimensional array.

#include<stdio.h>

main()
{
    int Age[] = {19, 23. 24, 27, 29};
}
```

Detailed Explanation

- In the example, the dimension of the array is not specified. The compiler automatically calculates the dimension of the array based on the initialization.

An array can also be initialized inside a function. After an array is declared inside a function, the array can be initialized by assigning values to the elements of the array.

Arrays are initialized to assign values for the first time in an application. This process eliminates the possibility of storing unwanted values in an array.

Self Review Exercise 42.1

1. *Write the code to initialize the one-dimensional array* `str1` *with the string* `COMPUTER`.

LESSON 43 : ONE-DIMENSIONAL ARRAYS --> ACCESSING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ACCESSING ARRAYS

An array consists of elements. An element of an array is accessed by specifying the name of the array followed by the subscript or the index.

Subscripts are positive finite integers or the expressions that generate positive finite integers. Therefore, an element of an array can be accessed using an integer variable.

To access the first element of an array, the expression `arrayname[0]` is used. Similarly, to access the eight element of the array, the expression `arrayname[7]` is used. The positive integer numbers can be replaced by `integer` variables that store these values.

Consider the example given below. An integer array of the size 10 is declared. An array can contain fewer elements than the array size. In this example, the array is initialized with six elements. An integer variable, `i`, is also declared and is initialized to 0.

Listing 43.1 : Initialising and printing an array

```
//Listing 43.1 : This program initializes an integer array and prints the first
//element of the array.

#include <stdio.h>

main()
{
    int number[10] = {1, 2, 3, 4, 5, 6};
    int i = 0;
    printf (“%d”, number[i]);
}
```

Detailed Explanation

- After the array number is initialized, the `printf` statement is used in the example to print the first element of the array. Notice that the subscript is replaced by integer variable `i`. Currently, `i` stores the value 0. Therefore, the `printf` statement will print 1.

You learned to access an element of an array. A piece of code below teaches you accessing and printing the sixth element of the array.

Listing 43.2 : Initializing and printing an array

```
// Listing 43.2 : Initializes and prints an element of an array
#include<stdio.h>
main()
{
int number[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int i=0;
printf(“%d”, number[5]);
}
```

Consider another example given below. In this piece of code, the `for` loop statement is used to access the elements of an array. The `for` loop is controlled using integer variable `i`.

Listing 43.3 : Initializes and prints an element of an integer array

```
//Listing 43.3 : This program initializes an integer array and prints the first
//element of the
//array.

#include <stdio.h>

main()
{
    int a[5];
    int i;
for (i = 0; i < 5; i++)
    {
        a[i]=i;
    }

for (i = 0; i < 5; i++)
    {
                printf (“a[%d]-%d\n”, i, a[i]};
            }
}
```

Detailed Explanation

- Notice that when `i` is equal to 0, 0 is assigned to the element `a[0]`. Then, `i` is incremented by 1. The new value of `i`, which is 1, is less than 5. Therefore, 1 is assigned to the element `a [1]`. In this way, the complete array is initialized.
- In the example, the array a stores the values 0, 1, 2, 3, and 4. To access the third element of the array, you can type `a[2]`. This is because the subscript of the third element is 2.
- Next, in the example, another for loop statement is used to print the values stored in the array. When the value of `i` is 0, the output of the `printf` statement is `a[0]=0`.
- When the displayed piece of code is executed, it prints `a[0]=0, a[1]=1, a[2]=2, a[3]=3,` and `a[4]=4`.

You learned to access an element of an array. A piece of code is given below to teach you how to print the last element.

Listing 43.4 : Initializing and printing an array

```
//Listing 43.4 : initializing and printing an integer array
#include<stdio.h>
main()
{
    int number[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i=0;
    printf(“%d”, number[9]);
}
```

An element of an array is accessed by specifying the array name and the associated subscript.

LESSON 44 : ONE-DIMENSIONAL ARRAYS --> FUNCTION ARGUMENTS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ARRAYS AS FUNCTION ARGUMENTS

To better structure applications, functions are used. Variables of all the fundamental data types can be passed as arguments to functions. Similarly, an array can also be used as a function argument.

The syntax of the function declaration to pass an array to a function as an argument is displayed. The `return type` of the function is followed by the name of the function. The argument, which is an array, is enclosed in parentheses. The type of the array is also mentioned.

```
function return type function name(array type[ ]);
```

Notice that the array type is followed by a set of brackets. The bracket notation is used to indicate that the type of the argument is an array.

The sample code that adds the elements of an array is as follows. In the program, the function `sum` is declared. An integer array is passed as an argument to this function.

Listing 44.1 : A program to pass an array as an argument to a function

```
// Listing 44.1 : This program is used to pass an array as an argument to a function.
#include <stdio.h>
void sum(int [ ]);
main( )
{
    int num[ ] = {3, 5, 7, 8, 9};
    sum(num);
}
void sum(int snum[ ])
{
    int i, s = 0;
    for(i = 0; i < 5; i++)
    {
            s = s + snum[i];
    }
    printf("%d, s);
}
```

Detailed Explanation

- In the `main` function, an integer array called `num` is declared and initialized. The function `sum` is called, and the array `num` is passed to this function.
- In the next part of the example, the function `sum` is defined. In the header of the function definition, the array `num` is passed as an argument to the function `sum` and is stored in the local array `snum`.
- In the function definition, two integer variables, `i` and `s`, are declared and variable `s` is initialized to `0`.

- A `for` loop statement is used to calculate the sum of the values of the elements of the array `num`. The sum of the values of the elements is stored in variable `s`. The `printf` statement will print the sum of the values of the elements.

Declare a function named `myfunction` that accepts an integer array as the argument. To do this, type `void myfunction(int[]);`.

Listing 44.2 : A program passing an array as a function argument

```
#include<stdio.h>

void nyFunction(int [ ]);

main( )
{
    int array1[ ] = {3, 5, 7, 8, 9};
    myfunction(array1);
}

void myFunction(int myarr[ ])
```

Detailed Explanation

- `myfunction` is a function that accepts an integer array as the argument.
- The `main` function is added for you. Next, call the function `myfunction`. This function accepts an array as an argument. Pass the array `array1` to the function `myfunction`. You can call this function as `myfunction(array1);`.
- Specify the header of the definition of the function `myfunction`. This function receives an integer array and stores the array in a local array named myarr. You can do this by `void myfunctio(int myarr[])`.

LESSON 45 : strcpy()

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

USING `strcpy()`

In C, assignment statements are used to assign a value to a variable. However, string variables can also be assigned values by using the string-handling function `strcpy`.

The `strcpy` function is also used to initialize strings in C. Besides the `strcpy` function, C provides a programmer with other string-handling functions, such as `strcmp`, `strcat`, `strlen`, `sscanf`, and `sprintf`.

The syntax of the `strcpy` function is given below. The syntax starts with the function name `strcpy`.

```
strcpy(string1, string2);
```

The arguments to the function `strcpy` are placed within parentheses as given in the syntax. The arguments are separated by a comma. The syntax ends with a semicolon.

The arguments can be two string variables or a string variable and a string constant. The `strcpy` function copies the second string to the first string. As a result, the original content of the first argument is lost.

In the example below, the `strcpy` function is used to copy values to two string variables. The fourth statement in this example is used to include the `string.h` header file. This file must be included to be able to use the standard string-handling functions.

Listing 45.1 : Using strcpy()

```
//Listing 45.1 : This program copies the content of one string into another by using the
//strcpy function.

#include <stdio.h>

#include <string.h>

main()
{
     char s1[100], s2[100];
     strcpy(s1, “hello”);
     strcpy(s2, s1);
}
```

Detailed Explanation

- In the `main` function of the program given below, two strings of the size `100` are declared.
- After the declaration statement, the `strcpy` function is used to copy the value `hello` to the `s1` string. Therefore, the `s1` string stores the value `hello` in the first five spaces and the sixth space is reserved for the `NULL` character. The remaining `94` spaces are not initialized.

- In the Line no `12` statement, the value of the `s1` string is copied to the `s2` string. Currently, `hello` is stored in the `s1` string. Therefore, `hello` is copied to the `s2` string. The `s1` and `s2` string contain the same values.

You can use this function to assign a value to a string variable or copy a value of a string variable to another string variable.

Self Review Exercise 45.1

1. *You learned how to use the standard string-handling function strcpy. A piece of code that uses the strcpy function is given below. Choose the option that displays the output of this code now.*

```
#include<stdio.h>
#include<string.h>
main()
{
char s1[100],s2[100];
strcpy(s1,"world");
strcpy(s2,"earth");
strcpy(s2,s1);
printf("%s",s2);
}
```

A. `World`
B. `earth`
C. `s1`
D. `NULL`

LESSON 46 : strcmp()

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

USING `strcmp()`

C programs use logical operators, such as <, >, and ==, to compare two variables or a variable with a constant to generate a result. However, logical operators cannot be used to compare two strings. To do this, use the standard string-handling function `strcmp`.

The syntax of the `strcmp` function is given below. The two strings that are compared are passed as arguments to the `strcmp` function. The syntax must end with a semicolon.

```
strcmp(string1, string2);
```

The `strcmp` function returns an integer that indicates the result of the comparison. The integer value that is returned by this function can be stored in an integer variable. If the `strcmp` function returns the value `0`, it indicates that the values of the strings that are passed as arguments to the `strcmp` function are equal.

If negative value is returned by the `strcmp` function, it indicates that the first string is less than the second string. Alternatively, a positive value indicates that the first string is greater than the second string.

Listing 46.1 : Using strcmp()

```
//Listing 46.1 : This program compares two strings by using the strcmp function.

#include<stdio.h>
#include<string.h>

main()
{
    char s1[100], s2[100];
    printf("\n Enter the first string");
    gets(s1);
    printf("\n Enter the second string");
    gets(s2);
    if (strcmp(s1, s2) == 0)
       printf("Strings are equal \n");
    else
    if(strcmp(s1, s2) < 0)
       printf("s1 lee than s2 \n");
    else
        printf("s1 greater than s2 \n");
}
```

Detailed Explanation

- The header files `stdio.h` and `string.h` are included. Within the `main` function, two strings, `s1` and `s2`, of the size `100` are declared.
- After the declaration statement, the input function gets is used to obtain from the keyboard for the two string variables in the example given.

- In the first `if` statement, the integer value that is returned by the `strcmp` function is compared to `0`. If `s1` and `s2` contain the same value, `strcmp` will return the value `0` and the first `printf` statement will be executed. Therefore, the program will print `Strings are equal`.
- If the `strcmp` function does not return `0`, the second if statement is executed. It checks whether the value returned by the function is a negative value. If `s1` is less than `s2`, `strcmp` will return a negative value. Therefore, the program will print `s1` less than s2.
- If the second `strcmp` function is executed and returns a positive value, the last `printf` statement will be executed.
- A positive value will be returned by the `strcmp` function if `s1` is greater than `s2`. Therefore, the program will print `s1` greater than `s2`.

The `strcmp` function is used to compare two strings; and it returns 0, a positive value, or a negative value.

```
integer variable = strcmp(string1, string2);
```

Self Review Exercise 46.1

1. *You learned about the standard string-handling function `strcmp`. A sample of code that uses the `strcmp` function is given. Choose the option that displays the output of this code given below.*

```
#include<stdio.h>
#include<string.h>
main()
{
    char s1[100], s2[100];
    strcpy(s2, “Basic”);
    strcpy(s1, s2);
    if (strcmp(s1, s2) == 0)
        printf(“Strings are equal\n”);
    else if (strcmp(s1, s2) > 0)
         printf(“String1\n”)
    else
         printf(String2\n”);
}
```

A. String1
B. Strings are equal
C. String2
D. Basic

LESSON 47 : `strcat()`

Lesson Objectives
In this Lesson you will learn : - *the Genesis of C Language* - *the Stages in the Evolution of C Language*

USING `strcat()`

The values of two integer variables can be added to generate another integer variable value by using the addition `(+)` operator. However, two string variables cannot be added by using the `+` operator. To add two string variables, the `strcat` function is used. The `strcat` function concatenates two strings. It adds the value of one string to the value of another string.

The syntax of the `strcat` function is given below. The syntax begins with the function name `strcat`.

```
strcat(string2, string1);
```

Within the parentheses in the syntax, the two strings that will be concatenated are passed as arguments. The syntax ends with a semicolon.

The `strcat` function is different from the `strcpy` function. The `strcpy` function copies the value of one string into another. This result in two strings with the same content.

In contrast to the `strcpy` function, the `strcat` function adds the value of the second string to the value of the first string. The value of the second string is added at the end of the first string. The `strcat` function also moves the `\0` character to the end of the resultant string.

For example, the statement in line `10` will add the string `ABC` to the current contents of the string `str1`. Therefore, if the original value of `str1` was `XYZ`, the new value of `str1` is `XYZABC`. The `\0` character is placed at the end of `XYZABC`.

Listing 47.1 : A program to add the values of two strings

```
1   //Listing 47.1 :  This program adds the value of one string to the value of another string by
2   // using the strcat function.
3
4   #include<stdio.h>
5   #include<string.h>
6   main( )
7   {
8   char str1[100];
9   strcpy(str1.”XYZ”);
10  strcat(str1,”ABC”);
11  printf(“%s”,str1);
12  }
```

The `strcat` function adds the value of the second string to the value of the first string.

Self Review Exercise 47.1

1. *You learned about the standard string-handling function* `strcat`. *A sample code that uses the* `strcat` *function is given below. Choose the option that displays the output of this code now.*

```
#include<stdio.h>
#include<string.h>
char s1[100] = “Programming in”;
char s2[100] = “the C Language”;
main( )
{
strcat(s1, s2);
printf(“%s”, s1);
}
```

A. the C Language

B. Programming in

C. Programming in the C Language

D. the C LanguageProgramming in

LESSON 48 : strlen()

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

USING `strlen()`

In C programs, strings may be declared without specifying their sizes. A string without a specific initial size can store a value of any length. To determine the length of the value stored in a string, the `strlen` function is used.

The syntax of the `strlen` function is given below. The syntax begins with the function name `strlen`.

```
integer variable = strlen(string);
```

In the given syntax, the value of a string variable or a string variable itself is passed as the argument.

The `strlen` function returns the numbers of characters currently stored in the string that is passed to the function as an argument.

For example, in the piece of code given below, a string variable, `str`, of the size `10` is declared.

Listing 48.1 : A program to calculate the length of a string

```
// Listing 48.1 : This program calculates the length of a string by using the strlen
// function.

#include<stdio.h>
#include<string.h>
main( )
{
    int y;
    char str[10];
    strcpy(str, “1234”);
    y = strlen(str);
    printf(“%d”, y);
}
```

Detailed Explanation

- The value `1234` is assigned to the string `str`. When this string is passed as the argument to the `strlen` function, the function returns the value `4`. This is because the string `str` currently stores four characters.
- Although the string can store `10` characters, the current length of the string is `four`, which is the value returned by the `strlen` function.
- The `printf` statement will print `9` because the size of the string name is `9`. This value is stored in variable `i`, which is returned by the `strlen` function.
- This function returns an integer value that specifies the length of a string.

Self Review Exercise 48.1

1. *You learned about the standard string-handling function* `strlen`. *A piece of code that uses the* `strlen` *function is given. Enter the output of this piece of code.*

```
#include<stdio.h>
#include<string.h>
char name[100] = {“Don Allen”};
main( )
{
        int i;
        i = strlen(name);
        printf(“%d”, i);
}
```

LESSON 49 : sscanf()

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

USING sscanf()

A string can store numbers. However, the number stored in a string cannot be used for any mathematical operation.

To use a number stored in a string in a mathematical operation, it must be converted to an integer. The conversion can be done using the standard string-handling function `sscanf`.

The `sscanf` function reads from a string variable in the memory and stores the data in different variables specified as its arguments. The `sscanf` function can also be used to transfer data between variables in a specific format. This function assigns the contents of a string to different variables

The syntax of the `sscanf` function is given below. The syntax starts with the function name. The arguments are passed within parentheses. The arguments are separated by commas.

```
sscanf(string, format-specification, &variable, …);
```

- The first argument is the string that stores the values that will be assigned to different variables.
- The second argument is the format in which the string will be converted. For example, you use the format `%d` to convert a string into an integer.
- The last argument in the syntax is the address of the variable that will store the value of the string that is passed as the first argument

An example using the `sscanf` function is displayed on the screen. The header files `stdio.h` and `string.h` are included in this function. A string named number of the size `100` is declared. This string is initialized with the value `1234`.

Listing 49.1 : Using sscanf()

```
//Listing 49.1 : This program converts a string to an integer by
//using the sscanf function.

#include<stdio.h>
#include<string.h>

    char number[100] = “1234”;

main()
{
   int newnumber;
   sscanf(number, “%d”, &newnumber);
   printf(“%d”, newnumber);
}
```

Detailed Explanation

- In the `main` function, an integer variable, `newnumber`, is declared.
- Next, the `sscanf` function is used to convert the value stored in the string number into an integer. This value will be stored in the `newnumber` variable. Therefore, `1234` will be stored in `newnumber`. The value `1234` can be used in mathematical operations.

The `sscanf` function transfers the value of the string to the integer variable. The output is `2468` because the `sscanf` function has transferred the value `2468` to the integer variable.

Self Review Exercise 49.1

1. *You learned about the standard string-handling function* `sscanf.` *Sample code that uses the* `sscanf` *function is given below. Enter the output of the piece of code.*

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

char str[10]-{“2468”};

main()
{

int N1-2;
sscanf(str, “%d”, &N1);
printf(“%d”,N1);

}
```

LESSON 50 : sprintf()

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

USING `sprintf()`

Standard string-handling functions, such as `strcpy` and `strcat`, are used to manipulate data and format the output. However, these functions cannot accept an integer variable or a `float` variable. To solve this problem, C provides the standard string-handling function `sprintf`.

Functions of `sprintf()`

- It converts the contents of different data type variables into a string.
- It writes to a variable in the memory.
- It is used to transfer data between variables.

The `sprintf` function performs three functions. It converts the contents of different data type variables into a string. The `sprintf` function also writes to a variable in memory. In addition, this function is used to transfer data between variables in a specific format.

The syntax of the `sprintf` function is given below. The syntax starts with the function name `sprintf`.

```
sprintf(string, specific-format, data, …);
```

- In the given syntax, the arguments enclosed in parentheses are passed. The arguments are separated by commas.
- The first argument is the string that will store the value that is currently stored in a variable.
- The second argument is the format in which the new value will be converted.
- The last argument is the variable that stores the value that will be converted into a string.

Consider the example given below. Three integer variables, `mm`, `dd`, and `yy`, store three different values. The string named date will store the values of the variables `mm`, `dd`, and `yy`.

Listing 50.1 : A program that converts an integer into a string

```
// Listing 50.1 : This program converts an integer into a string by using the sprintf
// function.

#include <stdio.h>
#include <string.h>

main()
{
    int mm = 10, dd = 23, yy = 1;
    char date[9];
    sprintf(date, “%d/%d/%d”, mm, dd, yy);
    printf(“%s”, date);
}
```

Detailed Explanation

- In the example given, the `sprintf` function is used to convert the integer values into a string. The value that will be stored in the string variable date is also formatted.
- When the piece of code displayed on the screen is executed the string date will store the new value 10/23/1. This is because the `sprintf` function converted the three integer values into a string.
- The `sprintf` function also added the special character `/` after the values of variables `mm` and `dd`. In this way, the new value is stored in the string `date`.

`sprintf` function is used to convert variables of different data types into a string.

Self Review Exercise 50.1

1. *You learned to use the standard string-handling function* `sprintf`*. A sample code that uses the* `sprintf` *function is given. Choose the option that displays the output of this code.*

```
#include<stdio.h>
#include<string.h>
        char String[10] = {“C”};
main()
{
        char Newstr[100];
        int n1 = 4;
        int n2 = 5;
        sprintf(Newstr, “%d%s%d”, n2, String, n1);
        printf(“%s”, Newstr);
}
```

A. 4C5
B. 5C4
C. C54
D. 54C

LESSON 51 : MULTI DIMENSIONAL ARRAYS —> DECLARATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

MULTI DIMENSIONAL ARRAYS

An array can be represented as a table of elements. A one-dimensional array is a table consists of a single row with multiple columns. The C language also supports multidimensional arrays. A multidimensional array consists of one or more dimensions. The simplest form of a multidimensional array is the two-dimensional array.

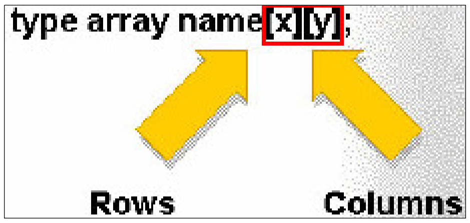
A two-dimensional array is a table that consists of multiple rows with multiple columns. A two -dimensional array is actually an array of one-dimensional arrays.

The syntax to declare a multidimensional array is given below. The syntax is similar to the declaration of one-dimensional arrays. The only difference is in the number of indexed or subscripts used with the array name.

```
type array name[x][y][z]….
```

The number of subscripts represents the dimension of an array. If there are three subscripts used with the array name, the array is a three-dimensional array. Similarly, an array name with four subscripts indicates that the array is four-dimensional.

The syntax for declaring a two-dimensional array is given below. Notice that the syntax contains the name of the array with two subscripts.



Two-dimensional arrays store elements in a row-column matrix. The first index indicates the row, and the second index indicates the column.

Consider the example of the integer type two-dimensional array declaration statement . This statement indicates that the array number is a two-dimensional array consisting of 5 rows and 10 columns.

Listing 51.1 : Declaring a two - dimensional array

```
// Listing 51.1 : This program declares a two-dimensional array.

#include<stdio.h>
main()
{
    int number[5][10];
}
```

The two-dimensional array declaration statement also implies that the array is an array of 5 one -dimensional arrays, where each of the 5 one-dimensional arrays is an array of 10 integers.

Consider the example of an integer type three-dimensional array declaration statement as given below. This statement indicates that the array is three-dimensional and consists of 5 tables. Each table consists of 10 rows and 2 columns.

Listing 51.2 : Declaration of two - dimensional array

```
1   //Listing 51.2 : This program declares a two-dimensional array.
2
3   #include<stdio.h>
4   main()
5   {
6       int EmpCodeBasic[100][2];
7       char EmpName[5][10];
8   }
```

Consider Line 6 statement. It declares a two-dimensional array, `EmpCodeBasic`, which will store the employee IDs of 100 employees and their corresponding basic salaries.

In the two-dimensional array `EmpCodeBasic`, the first subscript is 100, which implies that the array consists of 100 rows. In the same way, the second subscript, which is 2, implies that the array consists of 2 columns.

Consider Line 7. It declares a character two-dimensional array, `EmpName`. This array consists of 5 rows and 10 columns.

Declare an array name `Details` to store the points scored by 100 students in 2 different semesters. To do this, type `int Details[100][2];`.

```
#include<stdio.h>
main( )
{
 int Details[100][2];
}
```

Declare another two-dimensional character array, `Customer`, that stores the names of 10 customers. A name should not exceed 50 characters. To declare the Customer array, type `char Customer[10][51];`.

```
#include<stdio.h>
main()
{
 char Customer[10][51];
}
```

While declaring a multidimensional array, state the type and name of the array. The number of subscripts in the declaration determines the dimension of the array.

Self Review Exercise 51.1

1. *Identify the statement that declares a two-dimensional integer array,* `Array1`*, consisting of* 10 *one-dimensional arrays of the size* 5.

A. `int Array1[10][5];`
B. `int Array1[5][10];`
C. `int Array1[5][5];`
D. `int Array1[10][10];`

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

INITIALIZATION

Initialization is a process in which you first assign values to variables in an application. A multidimensional array, such as a two-dimensional array, is initialized in the same way as a one -dimensional array is initialized. A two-dimensional array can be declared and initialized at the same time.

The general syntax to initialize a two-dimensional array is given below. The first row of the array will store the set of values `n_11` and `n_12`, the second row in the array will store the values `n_21` and `n_22`, and the last row will store `n_x1` and `n_xy`.

```
type array name[x][y] = {{n_11, n_12}, {n_21, n_22}, …{n_x1, n_xy}};
```

Consider the example given below. In this example, the two-dimensional array `squares` is declared and initialized at the same time.

Listing 52.1 : initializing a two - dimensional integer array

```
// Listing 52.1 : This program initializes a two-dimensional integer array.

#include<stdio.h>

main()
{
    int squares[5][2] = {{1, 1}, {2, 4}, {3, 9}, {4 ,16},{5, 25}};
}
```

Another example is given below. In this example, a two-dimensional character array is declared and initialized in the same statement. The character array books can store `100` strings of `100` characters each.

Listing 52.2 : initializes a two - dimensional character array

```
// Listing 52.2 : This program initializes a two-dimensional character array.

#include<stdio.h>
main()
{
    char books[100][100] = {“C Language Made Easy”, “Programming in C”, “C Programming”};
}
```

- The array `books` stores the names of three books, and each name is a string of characters that is one-dimensional character array.

The example given below illustrates the procedure for initialization of a three-dimensional array. The array matrix consists of two tables, at each table consists of two rows and two columns.

Listing 52.3 : Initializing a three - dimensional integer array

```
// Listing 52.3 : This program initializes a three-dimensional integer array.

#include<stdio.h>
main()
{
    int matrix[2][2][2] = {{{2, 4}, {6, 8}, {3, 6}, {9, 12}}};
}
```

➥ In a three-dimensional array, the values of the first table and then those of the second table are initialized.

A two-dimensional array is an array of one-dimensional arrays.

Self Review Exercise 52.1

1. *Identify the correct two-dimensional array initialization statement that initializes the two-dimensional integer array* `Array1` *consisting of two one-dimensional arrays of the size* 3.

A. `int Array1[2][3] = {{0, 1, 1}, {2, 3, 5}};`
B. `int Array1[3][2] = {{0, 1}, {1, 2}, {3, 5}};`
C. `int Array1[2][3] = {{0, 1}, {1, 2}, {3, 5}};`
D. `int Array1[2][3] = {{0, 1, 1}, {2, 3, 5}};`

LESSON 53 : MULTI DIMENSIONAL ARRAYS —> ACCESSING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

DECLARATION

An array consists of elements. An element of an array is referred to by the name of the array followed by the subscript or the index. To access a specific element in a two-dimensional array, the subscripts are specified.

Consider the example given below. An integer two-dimensional array that consists of three rows and four columns is declared and initialized.

Listing 53.1 : Demonstration of two - dimensional array access

```
//Listing 53.1 : A program to access an element from a two - dimensional array

#include<stdio.h>

int sales[3][4] = {{143,210},{337, 653, 35},{47}};

main()
{

    printf("%d", sales[0][0]);
    printf("%d", sales[1][0]);
    printf("%d", sales[1][2]);

}
```

Detailed Explanation

- To access the first element of the array, you need to type `sales [0][0]`. Both the subscripts are `0` because `0` is always the first element of an array.
- Similarly, to access the value `337`, you need to type `sales[1][0]`. This is because `337` is the first element of the second one-dimensional array.
- If you type `sales[1][2]`, you can access the value `35` because `35` is the third element of the second one-dimensional array.

Accessing a character two-dimensional array is different from accessing a two-dimensional array of the type integer or `float`. In a character two-dimensional array, you can access a single character or a string of characters.

To access a string of characters, specify only the row subscript because a string is a one-dimensional array of characters. However, the individual elements of a character two-dimensional array are accessed by specifying both the row and column subscripts.

The example is given below illustrates the procedure for accessing a string of characters and a single character from a character two-dimensional array.

Listing 53.2 : Illustration of two - dimensional array access

```
// Listing 53.2 : A program to access an element from a two - dimensional array

#include<stdio.h>

char Language[5][13] = {“BASIC”, “COBOL”, “C”, “C++”, “Visual Basic”};

main()
{

    printf(“%s”, Language[1]);
    printf(“%c”, Language[4][3]);

}
```

Detailed Explanation

- To access the string `COBOL`, you need to type `Language[1]`. However, the statement `Language [4][3]` will return an individual element, which is character `u`, from the string `Visual Basic`.

The only difference in accessing a three-dimensional array element as compared to accessing a two -dimensional array element is that you need to specify three subscripts instead of two.

Listing 53.3 : Demonstration of three - dimensional array access

```
// Listing 53.3 : A program to access an element from a three - dimensional array

#include<stdio.h>

int sales[2][2][2] = {{{1, 8},{2, 42}}, {{3,34}, {4, 10}}};

main()
{

    printf(“%d”, sales[0][0][0]);
    printf(“%d”, sales[1][0][1]);

}
```

Detailed Explanation

- To access the first element in the displayed example, specify `sales[0][0][0]`. Similarly, you can access the element `34` of the array by typing `sales[1][0][1]`.

Self Review Exercise 53.1

1. ***Identify the output of the piece of code given. The two-dimensional array Language is declared and initialized. Choose the option that displays the output.***

```
#include<stdio.h>
char Language[5][13] = {“BASIC”, “COBOL”, “C”, “C++”, “Visual Basic”};
main()
{
        printf(“%c”, Language[0][2]);
}
```

A. A
B. F
C. S
D. B

REVIEW EXERCISE

1. Initialize the array `str` *with the string* `Hello, World`*. Choose the option that contains the correct code to initialize the one-dimensional array* `str` *with the string* `Hello, World`*.*

A.
```
#include<stdio.h>
main()
{
   char str[11] = {“Hello, World”};
}
```
B.
```
#include<stdio.h>
main()
{
   char str[13] = {“Hello, World”};
}
```
C.
```
#include<stdio.h>
main()
{
   char str[10] = {‘H’, ’e’, ’l’, ’l’, ’o’, ‘,’, ‘W’, ‘o’, ‘r’, ‘l’, ’d’, ‘\0’};
}
```
D.
```
#include<stdio.h>
main()
{
   char str[10] = {“Hello, World”};
}
```

2. Identify the output of the piece of code that uses the string-handling function `sprintf()`*. Four options are given below. Choose the option that displays the output of the code.*

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

main()
{
       int mm = 11, dd = 12, yy = 99;
       char date[9];
       sprintf(date, “%d/%d/%d”, mm, dd, yy);
       printf(“%s”, date);
}
```

A. 12/11/99
B. 11/99/12
C. 11/12/99
D. 99/12/11

3. Specify a piece of code to pass an array as an argument to a function. Give a function prototype for a function that accepts an integer array as an argument. The function name is `function2()`*, and the return type is* `void`*.*

4. Type the line of code to call the function `function2()`*, which accepts the array* `MyArray2`*.*

5. Type the line of code that defines the header of the definition of the function `function2()`*, which receives the integer array* `MyArray2` *as the argument.*

6. Identify the output of the displayed piece of code that uses the string-handling function `strcmp()`. *Four options are given below. Choose the option that displays the output of this piece of code.*

```
#include<stdio.h>
#include<string.h>
main()
{
       char s1[100], s2[100];
       strcpy(s1, “machine”);
       strcpy(s2, “computer”);

       if (strcmp(s1, s2) ==0)
              printf(“equal\n”);
       else if (strcmp(s1, s2)<0)
              printf(“s1 less than s2\n”);
       else
              printf(“s1 greater than s2\n”);
}
```

A. equal
B. s1 greater than s2
C. s1 less than s2
D. compile-time error

7. Identify the features of an array.

A. It is a user-defined data type in C.
B. It can be referred to by a common name.
C. It is a group of objects belonging to the same data type.
D. It is a fundamental data types in C.
E. Its elements are referred to by an array name and indexes.
F. It is a contiguous area in memory.

8. Identify the output of a set of statements used to access a two-dimensional array element. The two-dimensional array terms is declared and initialized. Choose the option displaying the correct output.

```
#include<stdio.h>

char terms[][140] = {“C language”, “Strings”, “Arrays”, “Computers and Peripherals”};

main()
{
        printf(“%c”, terms[3][0]);
        printf(“%s”, terms[1]);
        printf(“%c”, terms[2][4]);
}
```

A. CStringsy
B. AC languagei
C. lArraysg
D. SArraysg

9. Specify the number of elements that are valid for the one-dimensional array declaration statement given below. Enter the number of elements.

```
#include<stdio.h>
main()
{
       char array1[600];
}
```

REVIEW EXERCISE

10. Identify the output of the piece of code that uses the string-handling function `strcat()`*. Four options are given below. Choose the option that displays the output of this code.*

```
#include<stdio.h>
#include<string.h>
        char s1[100] = “Programming”;
        char s2[100] = “In C”;
main()
{
       strcat(s2, s1);
       printf(“%s”,s2);
}
```

A. In C Programming
B. ProgrammingIn C
C. Programming In C
D. In CProgramming

11. Specify the output of the given piece of code that uses the string-handling function `sscanf()`*. Enter the output.*

```
#include<stdio.h>
#include<string.h>
        char str[100] = {“1002”};
main()
{
       int number;
       number = 100 * 2;
       sscanf(str, “%d”, &number);
       printf(“%d”, number);
}
```

12. Identify the output of the piece of code that uses the string-handling function `strlen()`*. Four options are given below. Choose the option that displays the output of this code.*

```
#include<stdio.h>
#include<string.h>
        char s1[100] = {“Debbie”};
main()
{
       int i;
       i = strlen(s1);
       printf(“%d”, i);
}
```

A. 100
B. 7
C. 5
D. 6

13. Write the statement to declare a two-dimensional character array, `name`*, that stores the names of five users. Each name can be 49 characters long.*

14. Write the statement to declare a two-dimensional integer array, `sale`*, that stores the profits of five Salesperson IDs across four regions.*

15. Identify the output of the piece of code that uses the string-handling function `strcpy()`. *Four options are given below. Choose the option that displays the output of this code.*

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100], s2[100];
        strcpy(s1, “world”);
        strcpy(s2, “hello”);
        strcpy(s1, s2);
        printf(“%s”, s1);
}
```

A. *world*
B. *hello*
C. *s1*
D. *NULL*

16. Initialize the array `squares` *with the first eight natural numbers and their squares. Choose the option that correctly initializes the array.*

A.
```
#include<stdio.h>
     int squares[8][2] = {{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49},
                          {8, 64}};
main()
{
}
```

B.
```
#include<stdio.h>
      int squares[2][8] = {{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49},
                           {8, 64}};
main()
{
}
```

C.
```
#include<stdio.h>
      int squares[8][2] = {{1, 1, 2, 4, 3, 9, 4, 16), {5, 25, 6, 36, 7, 49, 8, 64}};
main()
{
}
```

D.
```
#include<stdio.h>
      int squares[2][8] = {{1, 1, 2, 4, 3, 9, 4, 16), {5, 25, 6, 36, 7, 49, 8, 64}};
main()
{
}
```

17. Specify the statement to access an element from a one-dimensional array. The piece of code to declare and initialize the one-dimensional array `age` *is displayed on the screen. Write the statement to access the third element.*

```
#include<stdio.h>
main()
{
         int age[5] = {15, 25, 35, 45, 55};
}
```

18. *Write the statement to access the first element.*

REVIEW EXERCISE

19. Write the statement to access the last element.

20. Initialize the array `str` *with the two string* `Hi` *and* `There`*. Choose the option that correctly initializes the array.*

A.
```
#include<stdio.h>
#include<string.h>
    char str[2][10] = {“Hi”, “There”};
main()
{
}
```

B.
```
#include<stdio.h>
#include<string.h>
    char str[10][2] = {“Hi”, “There”};
main()
{
}
```

C.
```
#include<stdio.h>
#include<string.h>
    char str[2][10] = {“Hi, There”};
main()
{
}
```

D.
```
#include<stdio.h>
#include<string.h>
    char str[2][2] = {“Hi, There”};
main()
{
}
```

21. Write the statement to declare a two-dimensional integer array, `emp`*, that stores the employee IDs of 50 employees and their years of joining. Write the statement.*

22. Write the statement to declare a two-dimensional character array, `spring`*, that stores the names of five books and each name is* `9` *characters long. Type the statement.*

23. Initialize the array `str` *with the string* `Hi, There`*. Choose the option that contains the correct code to initialize the one-dimensional array* `str` *with the string* `Hi, There`*.*

A.
```
#include<stdio.h>
main()
{
    char str[10] = {“Hi, There”};
}
```

B.
```
#include<stdio.h>
main()
{
    char str[7] = {“Hi, There”};
}
```

C.
```
#include<stdio.h>
main()
{
 char str[9] = {‘H’, ‘i’, ‘,’, ‘T’, ‘h’, ‘e’, ‘r’, ‘e’, ‘\0’};
}
```

D.
```
#include<stdio.h>
main()
{
             char str[8] = {“Hi, There”};
}
```

24. Identify the output of the given piece of code that uses the string-handling function `strcmp()`*. Four options are given below. Choose the option that displays the output of this piece of code.*

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

main()
{
       char s1[100], s2[100];
       strcpy(s1, “bad”);
       strcpy(s2, “good”);

       if (strcmp(s1, s2) ==0)
               printf(“equal\n”);

       else if (strcmp(s1, s2)<0)
               printf(“s1 less than s2\n”);

       else
               printf(“s1 greater than s2\n”);
}
```

A. equal
B. s1 less than s2
C. s1 greater than s2
D. compile-time error

25. Initialize the array cubes with the first five natural numbers and their cubes. Choose the option that correctly initializes the array.

A.
```
#include<stdio.h>
     int cubes[2][5] = {{1, 1}, {2, 8}, {3,27}, {4, 64}, {5, 125}};
main()
{
}
```

B.
```
#include<stdio.h>
     int cubes[5][2] = {{1, 1}, {2, 8}, {3,27}, {4, 64}, {5, 125}};
main()
{
}
```

C.
```
#include<stdio.h>
     int cubes[2][5] = {{1, 1, 2, 8, 3}, {27, 4, 64, 5, 125}};
main()
{
}
```

D.
```
#include<stdio.h>
     int cubes[5][2] = {{1, 1, 2, 8, 3}, {27, 4, 64, 5, 125}};
main()
{
}
```

26. *Initialize the array* `str` *with the three strings* `Hello`, `World`, *and* `There`. *Choose the option that correctly initializes the array.*

A.
```
#include<stdio.h>
#include<stdio.h>
char str[3][20] = {“Hello”, “World”, “There”};
main()
{
}
```

B.
```
#include<stdio.h>
#include<stdio.h>
char str[20][3] = {“Hello”, “World”, “There”};
main()
{
}
```

C.
```
#include<stdio.h>
#include<stdio.h>
char str[3][2] = {“Hello”, “World”, “There”};
main()
{
}
```

D.
```
#include<stdio.h>
#include<stdio.h>
char str[2][20] = {“Hello”, “World”, “There”};
main()
{
}
```

27. *Identify the output of the piece of code that uses the string-handling function* `strcpy()`. *Four options are given below. Choose the option that displays the output of this code.*

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

main()
{
       char s1[100], s2[100];

       strcpy(s1, “Hello”);
       strcpy(s2, “World”);
       strcpy(s1, s2);

       printf(“%s”, s2);
}
```

A. World
B. Hello
C. s2
4. NULL

28. Identify the output of the piece of code that uses the string-handling function `strcat()`. *Four options are given below. Choose the option that displays the output of the code.*

```
#include<stdio.h>
#include<stdio.h>
        char s1[100] = "Hello, World";
        char s2[100] = "I am Here";
main()
{
       strcat(s2, s1);
       printf("%s", s2);
}
```

```
A. I am Here Hello, World
B. I am HereHello, World
C. Hello, World I am Here
D. Hello, WorldI am Here
```

29. Identify the output of a set of statements used to access a two-dimensional array element. The two-dimensional array terms is declared and initialized. Choose the option displaying the correct output.

```
#include<stdio.h>
char devices[][40] = {"keyboard and mouse", "monitor", "printer", "other peripherals"};
main()
{
       printf("%c\n", devices[1][3]);
       printf("%s\n", devices[2]);
       printf("%c\n", devices[3][8]);
}
```

```
A.  i             B.  i             C.  r             D.  r
    monitor           printer           monitor           printer
    r                 r                 i                 i
```

30. Identify the output of the piece of code that uses the string-handling function `strlen()`. *Four options are given below. Choose the option that displays the output of the code.*

```
#include<stdio.h>
#include<stdio.h>
       char s1[100] = "Hello, World";
main()
{
       int i;
       i = strlen(s1);
       printf("%d", i);
}
```

```
A. 100
B. 13
C. 12
D. 11
```

REVIEW EXERCISE

31. Identify the output of the piece of code that uses the string-handling function `sprintf()`. *Four options are given below. Choose the option that displays the output of the code.*

```
#include<stdio.h>
#include<string.h>
main()
{
       int mm = 2, dd = 4, yy = 1;
       char date[9];
       sprintf(date, “%d/%d/%d”, mm, dd, yy);
       printf(“%s”, date);
}
```

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

32. Specify the statement to access an element from a one-dimensional array. The piece of code to declare and initialize the one-dimensional `numbers` *is given. Write the statement to access the last element.*

```
#include<stdio.h>
main()
{
       int numbers[4] = {100, 200, 300, 400};
}
```

33. Write the statement to access the first element.

34. Write the statement to access the third element.

35. Specify the number of elements that are valid for the one-dimensional array declaration statement given below. Write the number of elements.

```
#include<stdio.h>
main()
{
       int array1[100];
}
```

36. Specify the number of elements that are valid for the one-dimensional array declaration statement. Write the number of elements.

```
#include<stdio.h>
main()
{
       char array2[90];
}
```

37. Specify the number of elements that are valid for the one-dimensional array declaration statement. Write the number of elements.

```
#include<stdio.h>
main()
{
       char array3[100];
}
```

LESSON 54 : POINTERS —> DECLARATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

POINTERS

Pointers are variables that store memory addresses. Using pointers in a program, you can access the address of a variable in the program.

Before using a pointer in a program, you must declare it.

All variables that are stored in memory are stored in bytes. Each byte or a group of bytes has a unique memory address. When a variable is declared in a program, the compiler assigns a unique memory address to the variable.

The value of a variable is stored in the address location that is assigned by the compiler when the variable is declared. This address remains constant throughout the execution of the program.

To access the data stored in memory, the address of the memory location is required. This address can be stored in a pointer variable. In other words, a pointer points to the memory address of a location where data is stored.

Declaration

The declaration of a pointer variable should be preceded by the asterisk `(*)` symbol, and the variable must be declared with the same data type as that of the variable to which it should point.

```
int *ptr;
```

As an example of a pointer declaration, the statement `int *ptr;` declares the pointer variable `ptr` of the integer data type. This means that `ptr` is a pointer variable that `ptr` is a pointer variable that points to an integer data type.

Partial code is given below. To declare a pointer, `pv`, to an integer variable, we use `int *pv;`.

Listing 54.1 : pointer declaration

```
// Listing 54.1 : declaring pointer variables
#include<stdio.h>
main()
{
    int *pv;
    char *alpha;
}
```

Detailed Explanation

- You declared a pointer, `pv`, to an integer. This pointer points to a variable of the integer data type.
- Similarly, a pointer to a character variable can be declared as `char *ptr;`. In this declaration, `ptr` is the pointer variable that pointer to a character data type. To declare a pointer, `alpha`, to a character variable, we use `char *alpha;`.
- You declared a pointer to a character variable. The pointer variable `alpha` points to a character data type.
- A pointer to a `float` variable can be declared as `float *ptr;`. This declaration indicates that `ptr` is a pointer variable that points to a `float` data type.

A pointer and a variable of the same data type can be declared together.

Listing 54.2 : pointer declaration

```
// Listing 54.2 : declaring pointers
#include<stdio.h>
main()
{
    int x, *y;
    char *pt1, *pt2;
}
```

Detailed Explanation

- The declaration `int x, *y;` indicates that `x` is an integer variable and `y` is a pointer variable that points to a variable of the integer data type.
- The declaration `char *pt1, *pt2;` indicates that the variables `pt1` and `pt2` are pointer variables. The declaration also indicates that `pt1` and `pt2` point to variables of the character data type.

Pointers are variables that store memory addresses. Using pointer variables in a program, you can access the memory addresses of other variables used in the program.

Self Review Exercise 54.1

1. ***Identify a valid pointer declaration.***

A. `int *ptr;`
B. `char *ptr;`
C. `*y;`
D. `int ptr;`

LESSON 55 : POINTERS —> INITIALIZATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

INITIALIZATION

It is good programming practice to initialize a pointer after it is declared in a program. This is because values from a previous program can reside in the memory due to a refresh problem with the memory. When a pointer is initialized in a program, it is assigned an address.

If the value assigned to a pointer contains a character value or an integer value, the program might stop responding because a pointer can store only addresses. If a pointer is not initialized, it may point to any area in the memory causing the program to stop responding.

Consider the statement given below. In this statement, x is an integer variable that has been assigned the value 10.

```
int x = 10, *ptr;
```

- In the above statement, ptr is declared as a pointer variable and it points to an integer data type. Notice that the pointer variable ptr is declared but not initialized.
- To access the value stored in x, the address of x can be used. To access the address of x, the address of x, the address operator ampersand (&) is used.
- The address operator returns the address of a variable. This address can then be stored in a pointer variable. When a value is stored in a pointer variable, the pointer variable is initialized.

```
ptr = &x;
```

- The statement ptr=&x; initializes ptr with the memory address of variable x. In this initialization, notice that the value that is stored in ptr is the address of x and not the value of x, which is 10. This is because x is preceded by the & symbol in the statement

A pointer variable can also be initialized at the time of its declaration. For example, the statement given below initializes the pointer variable ptr_to_i with the address of i. Notice that variable i must be declared before initializing the pointer variable ptr_to_i.

```
int i=10, *ptr_to_i = &i;
```

A pointer variable can contain the address of only a variable that is of the same data type as the pointer variable. Therefore, if x is initialized as int x=0;, the statement char *alpha=&x; is an invalid pointer initialization statement.

A sample program is given below, and integer variable `y` is declared. To initialize the pointer variable `ptr_to_y` with the address of `y`, enter `*ptr_to_y=&y;`

Listing 55.1 : pointer initialization

```
// Listing 55.1 : pointer initialization
#include<stdio.h>
main()
{
    int y = 10, *ptr_to_y = &y;
}
```

You initialized the pointer variable `ptr_to_y` with the address of integer variable `y` at the time of its declaration.

By initializing a pointer after it is declared in a program, you assign an address to the pointer. Declaring a pointer and not initializing it might cause a program to stop responding because the pointer can point to any memory location.

Self Review Exercise 55.1

1. ***Identify the correct pointer initialization statement for the specified pointer declaration.***

```
int x, y, *ptr;
char a;
```

A. `ptr - &a;`
B. `ptr - &x;`
C. `ptr – y;`
D. `a - &y;`

LESSON 56 : POINTERS —> MANIPULATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

MANIPULATION

Pointers can be manipulated in a program in many ways. For example, a pointer variable that stores the address of another variable can be used to return the value of the variable to which it is pointing. In addition, a pointer can be assigned the value of another pointer variable.

The address of a variable can be accessed with the help of the address operator. To access the value of a variable to which a pointer is pointing, the indirection operator (*) is used.

The indirection operator can be used only on pointers. A code sample is given below. Notice that the use of the indirection operator in the declaration statement `int q = 5,*p;` implies that pointer variable `p` is being declared.

Listing 56.1 : printing the address of a variable via a pointer

```
1   // Listing 56.1 : This program prints the address of q and the value contained in q.
2
3   #include<stdio.h>
4   main()
5   {
6       int q = 5, *p;
7       p = &q;
8       printf("%d" , p);        //Prints the address of q.
9       printf("\n%d" , *p);     //Prints the value contained in q
```

Detailed Explanation

- The statement in line `7` in the code initializes `p` with the address of `q`. Therefore, when the first `printf` statement in the code is executed, the output of the code is the address of `q`.
- When the indirection operator is used anywhere other than in the declaration, it refers to the value contained in the variable to which the pointer is pointing. As a result, when the second `printf` statement is executed, the output of the code is the value that is stored in `q`, which is `5`.

Another code sample is given below. In this code, three integer variables, `i`, `j`, and `k`, are declared and initialized. The pointer variable `ptr` that points to an integer data type is also declared.

Listing 56.2 : demonstration of pointer declaration and assignment

```
// Listing 56.2 : //This program prints the address of q and the value contained in q.

#include<stdio.h>
main()
{
    int i, j, k, *ptr;
   i =10;
    j = 20;
    k = 39;

    printf("i = %d, j = %d", i, j); //The output of this statement is: i = 10, j = 20.
    ptr = &i; //The address of variable i is assigned to the pointer variable ptr.
    j = *ptr; //Variable j is assigned the value of variable i to which variable ptr is
             //pointing.

    printf("\ni = %d, j = %d", i, j); //The output of this statement is : i = 10, j = 10.
    k = *ptr +1; //Variable k is assigned the value 11.
    ptr = &k;
    *ptr = 0; //Variable k is assigned the value 0.

/*rest of the program*/
}
```

Detailed Explanation

- When the first `printf` statement is executed, the output is `i=10, j=20`. This is because `i` and `j` were initialized with the values `10` and `20`, respectively, when they were declared.
- After the first `printf` statement is executed, the address of variable `i` is assigned to `ptr`. In other words, the pointer variable `ptr` is initialized with the address of variable `i`. In addition, the value of variable `i` to which `ptr` points is assigned to variable `j`.
- When the second `printf` statement is executed, the output is `i=10`, `j=10`. This is because both `i` and `j` currently contain the value `10`.
- The next statement assigns the value `11` to variable `k`. This is because the value of `*ptr` is `10` and when `1` is added to `*ptr`, the result is `11`. The statement `ptr=&k;` reinitializes `ptr` with the address of variable `k`.
- Notice that at this point in the program, the pointer `ptr` no longer points to variable `i`. The statement assigns the value `0` to variable `k` because `*ptr` points to the memory address of variable `k`.

A pointer can also be assigned to another pointer just as a variable is assigned another variable. In addition, the value pointed to by one pointer can be assigned to a variable pointed to by another pointer.

Listing 56.3 : demonstration of pointer to pointer assignment

```
// Listing 56.3 : This program is to demo assigning a pointer to another pointer.

#include<stdio.h>

main()
{
    int x, y, *ptr1, *ptr2;
    x = 10;
    y = 20;

    ptr1 = &x;
    ptr2 = &y;

    *ptr1 = *ptr2; //Variable x is assigned the value of y.
}
```

Detailed Explanation

- The code sample given declares two integer variables, `x` and `y`, and two pointer variables, `ptr1` and `ptr2`.
- The pointer variable `ptr1` is initialized with the address of `x`, and the pointer variable `ptr2` is initialized with the address of `y`.
- In the last statement, `*ptr1` is assigned the value of `*ptr2`. Notice that `*ptr1` refers to the value of the variable to which `ptr1` points, and `*ptr2` refers to the value of the variable to which `ptr2` points.
- When the statement `*ptr1=*ptr2;` is executed, variable `x` is assigned the value of variable `y`. This is because the pointer variable `ptr1` points to variable `x` and the pointer variable `ptr2` points to variable `y`.

By manipulating pointers, you can access the value of the variable to which a pointer is pointing. You can also assign the value of a pointer variable to another pointer variable and change the value of a variable by using pointers.

Self Review Exercise 56.1

1. ***Identify the output of the specified statements.***

```
int x=20, *y=&x;
*y=10;
printf("%d",*y);
```

A. 30
B. 20
C. 10
D. 40

2. ***Identify the output of the specified statements.***

```
int x=15;
int *y=&x;
*y=10;
printf("%d", *y);
```

A. *y
B. 15
C. &x
D. 10

3. ***Identify the output of the specified statements.***

```
int x=15;
int *y=&x;
*y=x+40;
printf("%d", x);
```

A. 15
B. 40
C. x
D. 55

LESSON 57 : THE RELATIONSHIP

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE ARRAY - POINTER RELATIONSHIP

There is a close relationship between arrays and pointers. The name of an array contains the address of the first element of the array. A pointer also contains an address. Therefore, it logical to say that an array name is actually a pointer because both contain an address.

It is possible to manipulate an array by using a pointer. A code sample is given below. The first statement in the code sample declares an integer array, `arr;` of `10` elements and a pointer to an integer name `ptr`.

Listing 57.1 : Demonstrating pointer - array relationship

```
// Listing 57.1 : This program demonstrates the assignment of array to a pointer

#include <stdio.h>
main()
{
int arr[10], *ptr;
ptr=arr; //This statement passes the address of arr to ptr
/* In this selection code
arr[0] is equivalent to *(ptr+0)
arr[1] is equivalent to *(ptr+1)
arr[2] is equivalent to *(ptr+2) */
}
```

Detailed Explanation

- The second statement in the code sample assigns `arr` to `ptr`. Notice that `arr` stores the address of the first element of the array `arr`. Therefore, the statement `ptr=arr;` initializes the pointer variable `ptr` with the address of the first element of the array `arr`.
- To access the first element of the array by using the array name and a subscript, you type `arr[0]`. The same element can be accessed using the pointer variable `ptr` because `ptr` stores the address of the first element of the array.
- To access the first element of the array by using `ptr`, you type `*(ptr+0) or *(ptr).`
- Similarly, you can access the second element of the array by typing `*(ptr+1)` and the third element by typing `*(ptr+2).`
- The second and third elements can also be accessed by typing `arr[1]` and `arr[2]`, respectively, when using the array name instead of a pointer.

Consider another code sample that brings out the relationship between an array and a pointer. Although the output of both the `printf` statements in the displayed code is the same, there is a difference between the pointers `str` and `ptr`.

Listing 57.2 : Demonstrating pointer - array relationship

```
// Listing 57.2 : This program to demonstrate pointers.

#include<stdio.h>
main()
{
    char str[] = {"John Barrett"};
    char *ptr;
    printf("My name is %s", str);     //The output of this statement is: My name
                                      // is John Barrett
    ptr = str;
    printf("\nMy name is %s", ptr);   //The output of this statement is: My name
                                      //is John Barrett
}
```

Detailed Explanation

- The pointer variable `str` is `static`. It contains the address of the first element of the array `str`. The value of `str` remains the same throughout the execution of the program.
- The pointer variable `ptr` is dynamic. It can point to any memory location. Currently, `ptr` points to `str`. The pointer variable `ptr` stores the address of the first element of the array `str`.
- A simple assignment like `ptr=str1;` can make ptr point to another array, `str1`. If this statement is executed, `ptr` holds the address of the first element of the array `str1`. Notice that the statement `str=ptr;` is an invalid statement because `str` is `static`.

A pointer can also be used to initialize a string. For example, the statement given initializes the string `John Barrett` and declares the pointer variable `myname` that stores the address of the letter `J`.

```
char *myname = "John Barrett";
```

You can also have pointers that point to an array. For example, the statement `int (*pointer) [10];` declares a pointer to an array of `10` integers.

```
int(*pointer)[10];
```

Self Review Exercise 57.1

1. ***Select the option to access the second element of the array `arr1` by using `ptr` for the statements.***

```
int arr1[15];
int *ptr;
ptr=arr1;
```

A. *(ptr +1)
B. *(ptr + 0)
C. *(ptr)
D. *(arr1 + 2)

2. ***Identify the correct statement for the specified declaration.***

```
char c[20], *cptr;
```

A. c - cptr;
B. c++;
C. cptr - c;

LESSON 58 : POINTER ARITHMETIC

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

POINTER ARITHMETIC

Arithmetic operations, such as incrementing and decrementing, can be performed on pointers. Performing these operations on pointers is called pointer arithmetic. A requirement of pointer arithmetic is that a pointer should be declared as pointing to a certain data type. When a pointer is declared as pointing to a data type and arithmetic operations are performed on it, the pointer moves by the appropriate number of bytes.

Consider the statement `ptr++;` in the program below. This statement does not necessarily mean that after `ptr++;` is executed, the pointer points to the next memory location. When the pointer is declared as a pointer to an integer, it moves by the size of type `int`.

Listing 58.1 : Pointer demonstration

```
//Listing 58.1 : This program to demonstrate pointers.

#include<stdio.h>
main()
{
    int arr[10], *ptr;
    ptr = arr;
    ptr++;
/*rest of code*/
}
```

Detailed Explanation

- The size of type `int` may be of `2` bytes or `4` bytes depending on the system on which the program is run.
- If the system requirements the size of type `int` as `2` bytes, the pointer ptr points to the third byte upon the execution of the statement `ptr++;`.
- If the system on which the program is run represents the size of type `int` as `4` bytes, the pointer `ptr` points to the `fifth` byte after the statement `ptr++;` is executed.
- If the pointer is declared as a pointer to a character, then `ptr` points to the next byte when the statement `ptr++;` is executed. This is because a character occupies `1` byte of space.

A code sample that involves pointer arithmetic is given below.

Listing 58.2 : Demonstrating pointer arithmetic

```
// Listing 58.2 : This program is to demonstrate pointer arithmetic.

#include<stdio.h>
main()
{
    char astr[] = “Arithmetic operators”;
    char *ptr;

    ptr = astr;
    printf(“%s”, astr); //The output of this statement is: Arithmetic operators
    printf(“\n%s”, ptr); //The output of this statement is: Arithmetic operators

    ptr++;
    printf(“\n%s”, astr); //The output of this statement is: Arithmetic operators
    printf(“\n%s”, ptr); //The output of this statement is: Arithmetic operators
}
```

Detailed Explanation

- In the code above, a character array, `astr`, is declared and initialized and `ptr` is declared as a pointer to a character. In addition, `ptr` is initialized with the address of the first element of `astr`.
- When the first `printf` statement is executed, the output of the code is Arithmetic operators. This is because `astr` stores the address of the first element of the array `astr` that is initialized with the string Arithmetic operators.
- Similarly, when the second `printf` statement is executed, the output of the code is Arithmetic operators. This is because `ptr` is initialized with the address of the first element of the array `astr` in the third statement of the code
- When the statement `ptr++;` is executed, the pointer variable `ptr` is incremented. The third `printf` statement also prints the string Arithmetic operators because `astr` is `static` and cannot change during the course of the execution of the program.
- When the last `printf` statement is executed, the output is `rithmetic operators`. This is because after the statement `ptr++;` is executed, `ptr` is incremented by one position or byte to point to the second element of the array, which is `r`. In other words, `ptr` stores the address of `r`.
- The statement `ptr++;` made `ptr` point to the next location or byte because `ptr` was declared as a pointer to a character. The next location for the pointer `ptr` was the location in which letter `r` is stored because a character is stored in a single byte.

Pointer arithmetic, such as incrementing and decrementing, allows you to move a pointer by a certain number of bytes depending on the data type of the pointer as stated in the pointer declaration.

Self Review Exercise 58.1

1. ***The statement*** `pv++;`***, in which*** `pv` ***is assigned the address of an array, makes*** `pv` ***point to the ___ element of the array.***

   A. second
   B. first
   C. third
   D. fourth

2. ***Consider a situation in which system represents the size of type*** `int` ***as*** `2` ***bytes. When the statement*** `ptr+=3;`***, in which*** `ptr` ***is a pointer to an integer, is executed, the new address of*** `ptr` ***will be __ bytes more than the old address.***

   A. 6
   B. 1
   C. 2
   D. 3

LESSON 59 : ARRAY OF POINTERS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ARRAY OF POINTERS

By definition, an array is a collection of similar data types. These data types can be any valid C data types, including pointers. An array of pointers is a collection of pointers in which each element points to a separate memory location.

The advantage of having an array of pointers is that each element in the array can point to data blocks of varying sizes. Unlike an array in which each element of the array is of the same size, the elements of an array of pointers can point to data blocks of varying sizes.

An example of declaring an array of pointers is given below.

In this example, the statement `char *name[5];` declares name as an array of pointers in which each element points to a character data type.

Listing 59.1 : Demonstrating arrays of pointers

```
1   // Listing 59.1 : this program demonstrates how an array of pointer works
2   #include<stdio.h>
3
4   main()
5   {
6
7    char *name[5];
8    name[0] = “David”; //Using the * symbol: *(name) = “David”;
9    name[1] = “Jim”; //Using the * symbol: *(name + 1) = “Jim”;
10   name[2] = “John”; //Using the * symbol: *(name + 2) = “John”;
11   name[3] = “Debbie”; //Using the * symbol: *(name + 3) = “Debbie”;
12   name[4] = “Pat”; //Using the * symbol: *(name + 4) = “Pat”;
13
14  printf(“%s”, name[2]); //Using the * symbol: printf(“%s”, *(name + 2));
15
16  }
```

Detailed Explanation

- Notice that the same declaration can be thought of as an array of string or a two-dimensional array of five elements in which each element is a single-dimensional array of characters.
- The array `name` can be considered as an array of five strings. You can also say that each element of the array is a string.
- Individual strings can be initialized by separate assignment statements because each element of the array is a pointer.
- The statement in Line No `7` assigns the string `David` to the first element of the array name. In other words, the first element of the array name is a pointer to the string `David`.

- The string `David` can also be initialized using the `*` symbol. The statement Line No `7` in assigns the string `David` to the first element of the array. As a result, the first element of the array `name`, which is a pointer, points to the string `David`.
- Similarly, the second, third, fourth, and fifth elements of the array `name` are initialized to the strings `Jim`, `John`, `Debble`, and `Pat`, respectively.
- After the initialization of the second, third, fourth, and fifth elements of the array `name`, the second element of the array points to `Jim`, the third element Point to `John`, the fourth element points to `Debble`, and the fifth element points to `Pat`.
- The third name in the array, which is `John`, can be printed by either of the cases as in line no `13`. In the first statement, `name[2]` itself is an array of characters and name stores the address `o` the first element of the array of pointers.
- In the Line No `13`, name stores the address of first pointer in the array and name `+2` is pointing to. The output is `John` because the third pointer points to `john`.

To declare an array, author, of three pointers to character types, code is given below.

Listing 59.2 : printing array elements using pointers

```
// Listing 59.2 : this program demonstrates the array elements access using pointers

#include<stdio.h>
main()
{
char *author[3];
*(author) = “Ton wilkins”;
*(author+1) = “Ron Floyd”;
*(author+2) = “Mary Peterson”;
printf(“%s”, *(author+1));
}
```

Detailed Explanation

- You declared an array of pointers named `author`. The array `author` contains three elements. Each element in the array `author` is a pointer to a character.
- The elements of the array `author` are initialized. To print the second element in the array `author`, enter `printf(“%s”,*(author+1));`
- You typed the statement to print the second element in the array.

An array of pointers is flexible. Using an array of pointers, you can make each element in the array point to data blocks of different sizes.

Self Review Exercise 59.1

1. ***To print the second string in the array of pointers*** `char *str[5];`, ***the statement ____________ will be used.***

   A. `printf(“%c”,*(str));`
   B. `printf(“%s”,*(str+1));`
   C. `printf(“%s”,*(str+2));`
   D. `Printf(“%c”,*(str+2));`

LESSON 60 : FUNCTION ARGUMENTS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

FUNCTION ARGUMENTS

When variables are passed as arguments to functions, their values are copied to the corresponding function parameters. As a result, any changes made to the parameters in the called function do not affect the actual arguments that were passed to the function.

The mechanism of copying values to corresponding function parameters is called call by value. However, you can pass the address of a variable to a function by passing a pointer as an argument. This mechanism is called call by reference.

When a pointer is passed to a function as an argument, the address of the variable to which the pointer is pointing is passed. Therefore, when changes are made in the called function, the actual argument is changed.

An example of a function that accepts a pointer as an argument is given below.

Listing 60.1 : Demonstration of pointer as an argument

```
//Listing 60.1 : In this program, the function swap accepts a pointer as an argument.

#include<stdio.h>

void swap(int *, int *); //int * indicates that the function arguments are of the
                         //pointer type.

main()
{

       int i = 5, j = 10;
       swap(&i, &j); //Instead of passing the value, the addresses of variables i and j
                     //are passed.
       printf("%d, %d", i, j);

}

void swap(int *p, int *q) //The addresses of variables i and j are passed to pointers p
                          //and q.
{

       int tmp;
       tmp = *p;
       *p = *q;
       *q = tmp;

}
```

Detailed Explanation

- The second statement in the code sample declares a function named `swap`.
- The keyword `void` in the function declaration indicates that the function does not return any value.

- Notice that in the function declaration, the parameters are specified as `int*` and `int*`.
- The `*` symbol in the declaration statement just after the data type indicates that the function arguments are of the pointer type. In other words, the arguments that the function `swap` receives are actually pointers to integers.
- In the function `main`, two integer variables, `i` and `j`, are declared and initialized with the values `5` and `10`, respectively.
- The second statement in the `main` function calls the `swap` function and passes the addresses of `i` and `j` to the function.
- The second statement in the `main` function calls the `swap` function and passes the addresses of `i` and `j` to the function.
- The header of the function definition, `void swap(int *p, int *q)`, does not have a semicolon at the end. This is because function declarations contain a semicolon at the `end while` headers in function definitions do not contain a semicolon at the end.
- An integer variable, `tmp`, is declared in the function `swap`. Next, the value of `i` is assigned to the variable `tmp` by using the statement `tmp=*p`**;.**
- The statement `*p=*q;` assigns the value of `j` to `i`. At this point, `i` holds the value `10`.
- The statement `*q=tmp;` assigns the value in `tmp` to variable `j`. this means that `j` currently stores the value `5` because `tmp` was assigned the value of `i` in the statement `tmp=*p;`.
- When the last statement in the `main` function is executed, the output of the function is `10, 5`. Notice that if `i` and `j` were passed by value, the swapping of the values in the called function `swap` would not have affected the values of `i` and `j` in the calling function `main`.

Partial code is given below.

Listing 60.2 : Demonstrating pointers as function arguments

```
// Listing 60.2 : A program demonstrating the concept of pointers as function arguments

#include<stdio.h>

void reset(int *, int *);

main()
{

    int a = 1;
    int b = 3;
    printf("\n\nBefore calling reset: a = %d, b = %d", a, b);
    reset(&a, &b);
    printf("\nAfter calling reset: a = %d, b = %d", a, b);

}

void reset(int *p, int *q)
{

    *p = 0;
    *q = 0;

}
```

Detailed Explanation

- To declare a function, reset, that receives two integer pointers as arguments, enter `void reset (int *, int *);`.
- In the declaration `void reset(int *, int *);`, the `*` symbol indicates that the function `reset` receives two pointers to the integer data type. The keyword `void` indicates that the function `reset` does not return a value.
- To call the function `reset` and pass the addresses of variables `a` and `b`, enter `reset(&a, &b);`.
- The `&` symbol in the function call indicates that the address of variable `a` and the address of variable `b` are passed to the function `reset`.
- To write the header of the definition for the function `reset` that receives the addresses of two integer variables and stores them in pointer variables `p` and `q`, enter `void reset(int *p, int *q)`.
- You declared a function that receives a pointer as an argument. You also typed the statements to call the function that receives a pointer as an argument and declare the header of the definition for the function.

By passing pointers as arguments to functions, you can change the values of the original variables passed to the function.

Self Review Exercise 60.1

1. *For the statements* `int i=25, j=10; swap(&i,&j);`*, the arguments that are passed to the function* `swap` *are:*

   A. `the addresses of i and j.`
   B. `the values of i and j.`
   C. `the address of j and the value of i.`
   D. `10 and 25.`

2. *To pass the address of integer variable* `x` *to function* `f`*, the __ statement is used.*

   A. `f(int &x);`
   B. `f(*x);`
   C. `f(&x);`
   D. `f(int *);`

LESSON 61 : RETURNING POINTERS FROM FUNCTIONS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

RETURNING POINTERS FROM FUNCTIONS

A function can return only one value. This value can be of any fundamental C data type. However, if you have a function that returns a pointer, the function can return more than one value

Consider the code sample given below. The code has the function `copy` that copies the string `str1` into another string, `str3`, and then returns `str3` to the string `str2`.

Listing 61.1 : Functions returning pointers

```
1   // Listing 61.1 : In this program, the function copy copies  a string into another string
2   // and returns the new string.
3
4   #include<stdio.h>
5
6   char* copy(char *);  // The * symbol indicates that the return type is a pointer to a
7                        // character.
8   main()
9   {
10
11      char *str2, *str1 = “Hello world”;
12      str2 = copy(str1);
13      printf(“%s”, str2);
14
15  }
16
17  char* copy(char *nstr1)
18  {
19
20      char *str3;
21      str3 = nstr1;
22      return str3;
23
24  }
```

Detailed Explanation

- In Line `5` the statement in the code is the declaration for the function `copy`. Notice that the return type is specified as `char *` in the declaration statement. This indicates that the function returns a pointer to a character.
- In the `main` function, `str2` is declared as a pointer to a character. In addition, `str1` is declared as a pointer to a character and is initialized with the string `Hello world`.
- In Line `12` the statement in the `main` function calls the function `copy` and passes `str1` as the argument to the function. In addition, the returned pointer is assigned to `str2`.
- In the header of the definition for the function `copy`, `str1` is stored in the pointer variable `nstr1`. In the definition for the function, the pointer variable `str3` to the character data type is declared.

- The next statement in the function definition assigns `nstr1` to `str3`. When this statement is executed, the address contained in `nstr1` is stored in `str3`. As a result, `str3` points to the string "`Hello, world`".
- Finally, `str3` is returned by the function.
- When the `printf` statement is executed, the output of the code is `Hello world`. This is because the value returned by the function is actually a pointer to the string `Hello world` and this pointer is assigned to `str2` in the second statement of the `main` function.

Partial code is given below. To declare a function, `func`, that accepts a pointer to a character and returns a pointer to a character, we use `char *func(char *);`.

Listing 61.2 : Returning a pointer to a character

```
// Listing 61.2 : This program declare a function, that accepts a pointer to a character
// and returns a pointer to a character

#include<stdio.h>

char *func(char *);

main()
{
    char *str2, str1 = "Hello world";
    str2 = func(str1);
    printf("%s", str2);
}

 char *func(char *st1)
{
    st1 = "Hi, There";
    return st1;
}
```

Detailed Explanation

- The `*` symbol before the function name in the function declaration indicates that the function `func` returns a pointer. The `return` data type, `char`, in the function declaration indicates that the pointer that is returned points to a character data type.
- To call the function `func` that returns a pointer to a character and store the pointer in the pointer variable `str2`, enter `str2=func(str1);`.
- In the statement `str2=func(str1);`, the function `func` is called. The pointer that is returned by the function is then assigned to the pointer variable `str2`.
- Write the header of the definition for the function `func`. This function stores the pointer passed to it the pointer variable `st1` and returns a pointer to a character. Enter `char *func(char *st1)`.
- You declared a function that returns a pointer to a character. You also typed the statements to call the function and declare the header of the definition for the function.
- When an integer pointer has to be returned from a function, the syntax remains the same expect that `char` is replaced with `int`.

- If a function that returns a pointer to an integer is declared as `int *abc();`, you need to enter `return &a;` to return the address of integer variable `a` from the function.

By returning pointers from functions, you can return more than one value from a function.

Self Review Exercise 61.1

1. *To declare a function* `somefunc` *that returns a pointer to an integer, the __statement is used.*

   A. `somefunc(char *);`
   B. `char *somefunc();`
   C. `somefunc(int *);`
   D. `int *somefunc();`

REVIEW EXERCISE

STUDY AREA : POINTERS

1. Select the correct pointer initialization statement for the pointer declaration `int a, b, *p; char c;`.

A. p = &a;
B. p = &c;
C. b = &c;
D. a = &b;

2. A piece of code that declares a character array and a pointer to a character is given. Four assignment expressions are listed below the piece of code. Identify the expression for the piece of code that correctly states the relationship between an array and a pointer.

```
#include<stdio.h>
main()
{
       int arr[10], *ptr;
}
```

A. arr = arr + 1;
B. arr = ptr;
C. ptr = arr;
D. arr = arr[2];

3. Identify the output of the specified set of statement involving pointer manipulation. Four options are given below. Choose the option that displays the output of the set of statements.

```
#include<stdio.h>
main()
{
       int x, y, *ptr1, *ptr2;
       x =10;
       y = 20;
       ptr1 = &x;
       ptr2 = &y;
       *ptr1 = *ptr2;
       *ptr2 = *ptr1;
       printf("%d %d", *ptr1, *ptr2);
}
```

A. 20 20
B. 10 20
C. 20 10
D. 10 10

4. Identify the output of a set of statements that implements an array of pointers. Four options are given below. Choose the option that displays the output of the set of statements.

```
#include<stdio.h>
main()
{
       char *city[4] = {"New York City", "Washington", "San Francisco", "Atlanta"};
       printf{"%s", *(city + 1));
}
```

A. New York City
B. Washington
C. San Francisco
D. Atlanta

REVIEW EXERCISE

STUDY AREA : POINTERS

5. A function that returns a pointer is given. Identify the output of the function. Four options are given below. Choose the option that displays the output of the function.

```
#include<stdio.h>
char* copy(char *);
main()
{
        char *str2, *str1 = “Hi There”;
        str2 = copy(str1);
        printf(“%s”, str2);
}

char* copy(char *newstr)
{
        char *str3;
        str3 = newstr;
        return str3;
}
```

A. newstr;
B. str2
C. Hi There
D. str3

6. Identify the output of a set of statements that involves pointer arithmetic. Choose the option that displays the output of the set of statements.

```
#include<stdio.h>
main()
{
        char arr[] = {‘A’, ‘B’, ‘C’, ‘D’}, *c;
        c = arr;
        c++;
        printf(“%c”, *c);
}
```

A. C
B. B
C. D
D. A

7. A function that accepts a pointer as an argument is given. Identify the output of the function. Four options are given below. Choose the option that displays the output of the function.

```
#include<stdio.h>
void change(int *, int *);
main()
{
        int x = 20, y = 25;
        change(&x, &y);
        printf(“%d %d”, x, y);
}
void change(int *p, int *q);
{
        *p = *q;
        *q = *p;
}
```

A. 25 25
B. 25 20
C. 20 25
D. 20 20

8. Which of the following are valid pointer declarations?

```
A. int *p;
B. float *ft;
C. int a, *b;
D. char *ch;
E. char *p1, *p2;
F. *ptr_to_char;
G. int ptr*;
```

9. Identify the output of a set of statements that implements an array of pointers. Four options are given below. Choose the option that displays the output of the set of statements.

```
#include<stdio.h>
main()
{
       char *wdays[7] = {"sunday", "monday", "tuesday", "wednesday", "thursday",
                         "friday", "saturday"};
       printf{"%s", *(wdays + 3));
}
```

```
A. wednesday
B. thursday
C. tuesday
D. friday
```

10. A function that accepts a pointer as an argument is given. Identify the output of the function. Four options are given below. Choose the option that displays the output of the function.

```
#include<stdio.h>
void afunc(int *, int);
main()
{
       int i = 10, j = 20;
       afunc(&i, &j);
       printf("%d %d", i, j);
}
void afunc(int *a, int b);
{
       b = b + 10;
       *a = *a + 10;
}
```

```
A. 10 10
B. 10 20
C. 20 10
D. 20 20
```

Lesson Objectives
In this Lesson you will learn : ☛ *the Genesis of C Language* ☛ *the Stages in the Evolution of C Language*

typedef —> PURPOSE

The C language provides some built-in data types, such as `int`, `char`, and `float`. These data types are called fundamental data types.

In addition to the fundamental data types, a user can define certain specific data types called user-defined data types. The user-defined data types can be used to increase the clarity of code.

The `typedef` keyword allows you to define user-defined data types based on any data type. A user-defined data type is defined using the `typedef` keyword to qualify the data type that you specify for a variable.

The syntax for defining a user-defined data type with the help of the `typedef` keyword is given below. It starts with the keyword `typedef`. This keyword is followed by the name of any data type and the name of the user-defined data type.

```
typedef datatype user - defined datatype;
```

For example, if you want to declare integer variables that store the salaries of two employees, you can use the declaration `int emp1, emp2;`. However, this declaration does not clearly indicate the type of data that will be stored in the integer variables `emp1` and `emp2`.

The `typedef` keyword can be used to indicate the type of data that will be stored in a variable. Consider the declaration that is given below. The first statement, `typedef int salary;`, defines salary as a user-defined data type based on the data type `int`.

```
typedef int salary;
```

Instead of the data type, you can use the salary data type to declare variables that will store integer data. The second statement displayed on the screen, `salary emp1, emp2;`, uses the user-defined data type `salary` to declare the variables `emp1 and emp2`.

```
salary emp1, emp2;
```

The statement `salary emp1, emp2;` clearly indicates that the integer variables `emp1` and `emp2` can be used to store the salaries of two employees. By using the `typedef` keyword, you can increase the clarity of code.

You can use the `typedef` keyword to declare arrays. The declaration defines sentence as a user-defined data type, which is a character array of `50` elements.

```
typedef char sentence[50];
```

Notice the two examples given. There are two statements in the first example, which collectively declare the two variables `header` and `footer` that are of the `sentence` data type.

In the second example, a single statement declares two arrays of elements each. Both examples declare two arrays of `50` elements each. However, the first example provides more clarity because it indicates the type of data that will be stored in the variables `header` and `footer`.

```
char header[50], footer[50];
```

The `typedef` keyword allows you to define user-defined data types based on any data type.

Self Review Exercise 62.1

1. *Identify the statement that implements the* `typedef` *keyword.*

   A. `typedef int distance;`
   B. `typedef empid int;`
   C. `typedef char[20] name;`
   D. `typedef name[20];`

LESSON 63 : enum --> FEATURES

Lesson Objectives
In this Lesson you will learn : ☛ *the Genesis of C Language* ☛ *the Stages in the Evolution of C Language*

`enum`

An enumerated data type is a user-defined data type. The `enum` keyword is used to declare an enumeration data type. Enumerated data types are used when you know in advance the finite list of values that a variable can store.

For example, you want to declare a data type that can only store the names of some specific products, such as `pen`, `pencil`, `eraser`, and `paper`. You can use a character array to store the names of these products.

However, a character array will also allow you to store the names of other products. By using the `enum` keyword, you can declare an enumerated data type that will only store the names of the products `pen`, `pen-cil`, `eraser`, and `paper`.

The syntax for declaring an enumerated data type is given below. It starts with the keyword `enum` and is followed by the name of the enumerated data type.

```
enum user-defined data type{member1, member2, …};
```

In the syntax, the specific values that can be stored are enclosed in braces. The declaration ends with a semicolon.

An example of an enumerated data type is given below. The `enum` keyword is used to declare the enumerated data type product. You can use it to declare the variables that will only store the names of the products `pen`, `pencil`, `eraser`, and `paper`.

The specific values stored by an enumerated data type variable are referred to by two names, members or predefined values.

The members of an enumerated data type are constants. For example, in the enumerated data type `product`, the members `pen`, `pencil`, `eraser`, and `paper` are all constants.

```
enum product{pen, pencil, eraser, paper};
Members are constants.
Members are unique.
```

The members of an enumerated data type must be unique. Therefore, an enumerated data type cannot contain two identical members.

In the example given the statement `enum product p1;` declares an enumerated data type variable, `p1`. Variable `p1` can be assigned only one of the members `pen`, `pencil`, `eraser`, or `paper`. The statement `p1=pen;` will assign the member `pen` to variable `p1`.

LESSON 63 : enum --> FEATURES

Listing 63.1 : Using enum data types

```
// Listing 63.1 : This program assigns a value to an enumerated data type variable.

#include<stdio.h>

main()
{
    enum product{pen, pencil, eraser, paper};
    enum product p1;
    p1 = pen;
}
```

The members of an enumerated data type cannot be directly used as input with an input function, such as `scanf`. For example, the member `pen` cannot be used as an input for variable `p1` in the `scanf("%d",&p1);` statement.

To use a member as input, an integer value is assigned to the member by the compiler. By default, the first member of any enumerated data type is assigned the value `0` by the compiler. The second member is assigned `1`, and the third member is assigned `2`.

The compiler assigns each member in the list a value that is one greater than the value of the preceding member.

In the example given below, the member pen, pencil, eraser, and paper are assigned the values `0, 1, 2, 3`, respectively.

```
enum product{pen, pencil, eraser, paper};

pen    = 0
pencil = 1
eraser = 2
paper  = 3
```

The integer values assigned are used as input for the enumerated data type variable `p1`. The `scanf("%d", &p1);` statement is used to specify a value for variable `p1`. You specify the value `0` so that the first member can be used as input for variable `p1`.

You can also override the default value `0`, assigned by the compiler. To this, you can assign an explicit integer value to a member in the declaration of the enumerated data type.

When you assign an explicit value to a member of an enumerated data type, the remaining members are automatically assigned a value by the compiler.

```
enum product{pen = 1, pencil, eraser, paper};

pen      = 1
pencil   = 2
eraser   = 3
paper    = 4
```

These values increase successively by one from the previous explicit assignment. In this example, `pencil`, `eraser`, and `paper` are automatically assigned the values `2, 3`, and `4`, respectively. These values cannot be changed anywhere in the program.

The enumerated data type

- is a user-defined data type.
- stores specific values called members.
- members are constants.
- members are unique.
- members cannot be directly used as input for the `scanf` function.
- members are assigned integer values by the compiler.
- members can be assigned explicit values.

Self Review Exercise 63.1

1. *An enumerated data type:*

   A. contains a list of variables.
   B. contains members that can be identical.
   C. is a fundamental data type.
   D. is used to store specific values.

2. *Identify the feature of an enumerated data type.*

   A. A member of an enumerated data type is directly used as input for the `scanf` function.
   B. The members of an enumerated data type can be assigned explicit values anywhere in the program.
   C. An enumerated data type has members that are addressed as integers.

LESSON 64 : enum --> IMPLEMENTATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

`enum` —> IMPLEMENTATION

Enumerated data types are used when you want to store specific values in a variable, such as the names of some specific products. These specific values are predefined in the declaration of the enumerated data type.

A piece of code is given below. When this piece of code is executed, it will accept an integer value and print the name of the corresponding product.

Listing 64.1 : Implementing an enumerator

```
1   // Listing 64.1 : This program prints the name of the product depending on the integer
2   // value specified
3
4   #include<stdio.h>
5
6   main()
7   {
8
9       enum products{floppy, keyboard, mouse, cd};
10      enum products p1;
11      printf("Type an integer between 0 and 3");
12      scanf("%d", &p1);
13
14  switch (p1)
15      {
16
17      case floppy:
18              printf("\n floppy");
19              break;
20
21      case keyboard:
22              printf("\n keyboard");
23              break;
24
25      case mouse:
26              printf("\n mouse");
27              break;
28
29      case CD:
30              printf("\n CD");
31              break;
32
33      default:
34              printf("\n Error");
35
36      }
37
38  }
```

Detailed Explanation

- Line `9` declares an enumerated data type, products. The members of the enumerated data type products are `floppy`, `keyboard`, `mouse`, and `CD`. A variable, `p1`, of the enumerated data type products is declared.

- When the input statement `scanf("%d", &p1);` is executed, it accepts an integer value. If the integer value typed is `0`, the first member of the enumerated data type is assigned to `p1`. If the integer value typed is `1`, the second member is assigned to `p1`.
- In the example given, the controlling expression in the `switch` keyword is variable `p1`. The value of variable `p1` is compared with each case statement. If the value of `p1` is `0`, the case statement case `floppy` is executed and the member `floppy` is printed.
- If the value of variable `p1` is `1`, the case statement case keyboard is executed and the member `keyboard` is printed.
- When the value of variable `p1` is greater than `3` or less than `0`, the `default` statement is executed and the error message `Error is printed`
- The keyword break in each case statement passes the control from the case statement to the end of the program.

Self Review Exercise 64.1

1.

2. *You learned to implement the keyword enum. Four blocks of code that implement the keyword enum are given below. Choose the correct block of code.*

```
#include<stdio.h>
main()
{
enum device
{dot_matrix, daisywheel, ink_jet, laser};
enum device d1;
d1=daisywheel;
}
```
A

```
#include<stdio.h>
main()
{
enum device
{dot_matrix, daisywheel, ink_jet, laser};
enum device d1;
d1=laser_jet;
}
```
B

```
#include<stdio.h>
main()
{
enum device
{dot_matrix, daisywheel, ink_jet, laser};
enum device d1;
d1="laser";
}
```
C

```
#include<stdio.h>
main()
{
enum device
{dot_matrix, daisywheel, ink_jet, laser};
dot_matrix=1;
}
```
D

LESSON 65 : STRUCTURES --> AN INTRODUCTION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

STRUCTURES

A structure is a user-defined data type that is used to aggregate variables of different data types in one unit. In C, structures are often used to read and write records in a file.

For example, you have to store information about an employee in a file. The information includes details such as `employee id`, `data of joining`, and `salary`. This information consists of different data types.

There is no data type in the C language that allows you to store different data types in one unit. However, you can store all this information as one unit in a structure and use the structure to write records in a file. The variables contained in a structure are called its members. The data types of these members can be different. For example, you can have a structure that has members of the data types `int`, `char`, and `float`. In addition, structures can contain arrays and pointers.

Unique Member Names

The member names within a structure are unique. However, the members in different structures can have the same name.

Unlimited Members

There is no limit to the number of members that a structure can contain. Structures are user-defined data types that can be used to aggregate variables of different data types in one unit. Structures are often use to read and write records in C.

A structure

- is a user-defined data type.
- aggregates different data types
- contains members.
- has unique member names.
- can have unlimited members.

Self Review Exercise 65.1

1.

2. *Identify a feature of a structure.*

A. It is a fundamental data type
B. It cannot contain integers
C. It has members that are unique
D. It contains three members

LESSON 66 : STRUCTURES --> DECLARATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

By declaring structures, you can create data types in C that are not built-in. For example, you can create a data type to store the complete record of an employee. The record may store information of different data types, such as `int`, `char`, and `float`.

The syntax to declare a structure is given below. It starts with the keyword `struct`. The keyword is followed by the name of the structure, which is known as its `tagname`.

```
struct tagname
{
 member1;
 member2;
  .
  .
};
```

After the tag name in the structure declaration, the members of the structure are specified. The structure members are enclosed in braces, and the declaration ends with a semicolon.

An example of a structure declaration in which the structure stores a date is given below. The members of this structure can include the `day`, the `month`, and the `year`.

Listing 66.1 : Structure declaration

```
// Listing 66.1 : This program to demonstrate structure declaration

#include<stdio.h>

struct date
{
    int day;
    char month[15];
    int year;
};

main()
{
}
```

Detailed Explanation

- In the given structure declaration, the members `day` and `year` are integer data types and the member `month` is a character array.
- The members `day` and `year` are integer data types because days and years can be stored as integers. The member month is a character array because a month can be stored as a string.

The declaration of a structure does not reserve any bytes in memory for the structure. The declaration only indicates the variable that can be stored in the structure.

Self Review Exercise 66.1

1. *You have learn to declare a structure. Write the code that declares a structure.*

```
#include <stdio.h>

struct product
{
       int x;
       char y;
};

main()
{
}
```

Lesson Objectives
In this Lesson you will learn : • *the Genesis of C Language* • *the Stages in the Evolution of C Language*

STRUCTURE VARIABLES DECLARATION

Structure variables are declared before they can be used in programs. These variable are declared so that they can be used to stored data in structures.

A structure declaration is given below. It declares a structure, `employee`. The members of the structure `employee` are `Empid` and `EmpName`.

Listing 67.1 : Illustration of syntax of a structure

```
//Listing 67.1 : A simple program giving the syntax of a structure

#include<stdio.h>

struct employee
{

    int EmpId;
    char EmpName[30];

};

main()
{

}
```

A structure variable can be declared in three ways.

- The first way is to declare it outside the declaration of the structure.
- The second way is to declare the structure variable within the declaration of the structure.
- The third way is to use the `typedef` keyword.

The syntax for declaring a structure variable outside the declaration of the structure is given below. It starts with the keyword `struct`. This is followed by the tag name and the structure variable name. The declaration ends with a semicolon.

An example of declaring structure variables outside the declaration of the structure is given below. In this example, `e1` and `e2` are structure variables. When a structure variable is declared outside the declaration of the structure, the tag name is essential.

```
struct tagname variablename;
```

Listing 67.2 : declaration of structure variables

```
// Listing 67.2 : A simple program declaring structure variables

#include<stdio.h>

struct employee
{

    int EmpId;
    char EmpName[30];

};

main()
{

  Struct employee e1, e2;

}
```

The syntax for declaring a structure variable within the declaration of the structure is given below. It starts with the keyword `struct`. When the structure variable is declared within the declaration of the structure, the tag name is optional.

```
struct
{
  member1;
  member2;
}variablename;
```

By default, the data type of the structure variable is that of the structure within which it is declared. The members are enclosed in braces and a semicolon is placed after the variable name.

An example of declaring a structure variable within the declaration of the structure is given below. Structure variables `e1` and `e2` are declared within the declaration of the structure. Notice that in the declaration, the tag name is not present

Listing 67.2 : declaration of structure variables

```
// Listing 67.3 : A program listing the declaration of structure variables

#include<stdio.h>

struct employee
{

    int EmpId;
    char EmpName[30];

}e1, e2;

main()
{

}
```

The method that does not use a tag name is used only when you know how many structure variables are needed. Without a tag name, structure variable cannot be declared later.

When a structure variable is declared, memory is allocated for the structure variable. In this example, when structure variable e1 and e2 are declared, memory is allocated for each variable.

Depending on the computer system, the compiler allocates 2 bytes or 4 bytes of memory for the integer variable EmpId and 30 bytes of memory for the character array EmpName. A total of 32 bytes or 34 bytes each are reserved for structured variables e1 and e2.

```
typedef struct
{
        member1;
        member2;
} user-defined data type;
```

The typedef keyword can also be used to declare structures. To declare a structure by using this keyword, you need to specify the typedef keyword followed by the struct keyword.

Next, the members and a user-defined data type are specified. The declaration ends with a semicolon.

In the example given below, the user-defined data type is specified as book. The members of this user -defined data type are author and pages.

Listing 67.4 : declaring a user - defined data type

```
//Listing 67.4 : A program using a user–defined data type

#include<stdio.h>

typedef struct
{

    char *auther;
    int pages;

}book;

main()
{

   book b1;

}
```

Self Review Exercise 67.1

1. *You learned to declare a structure variable outside the declaration of the structure. Four blocks of code are given below. Choose the block of code that declares a structure variable outside the structure declaration now.*

```
#include<stdio.h>
struct s
{
int a;
char c;
};
main()
{
s s1;
}
```
A

```
#include<stdio.h>
struct s
{
int a;
char c;
};
main()
{
struct s s1;
}
```
B

```
#include<stdio.h>
struct s
{
int a;
char c;
};
main()
{
struct  s1;
}
```
C

```
#include<stdio.h>
struct s
{
int a;
char c;
};
main()
{
struct s1 s;
}
```
D

2. *You learned to declare a structure variable by using the* `typedef` *keyword. Four blocks of code are given below. Choose the block of code that declares a structure variable by using the* `typedef` *keyword.*

```
#include<stdio.h>
typedef struct
{
int x;
int y;
}var1;
main()
{
struct var1 st1;
}
```
A

```
#include<stdio.h>
typedef struct var1
{
int x;
int y;
};
main()
{
var1 st1;
}
```
B

```
#include<stdio.h>
typedef struct
{
int x;
int y;
}var1;
main()
{
var1 st1;
}
```
C

```
#include<stdio.h>
typedef
{
int x;
int y;
}var1;
main()
{
var1 st1;
}
```
D

Self Review Exercise 67.2

3. *You learned to declare a structure variable within the declaration of the structure. Four blocks of code are given below. Choose the block of code that declares a structure variable within the structure declaration.*

```
#include<stdio.h>
struct
{
int i;
char j;
}var1;
main()
{

}
```

A

```
#include<stdio.h>
s struct
{
int i;
char j;
}var1;
main()
{

}
```

B

```
#include<stdio.h>
struct
{
int i;
char j;
}var1;
main()
{

}
```

C

```
#include<stdio.h>
{
int i;
char j;
}struct var1;
main()
{

}
```

D

LESSON 68 : STRUCTURE MEMBERS --> ACCESSING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

STRUCTURE MEMBERS —> ACCESSING

You can assign values to structure members and manipulate the data in structures by accessing the members of a structure. For example, in a structure that contains the points of a student, you can calculate the average of the points obtained by the student.

To calculate the average, the members of the structure have to be accessed. To access the members of a structure, the structure is declared and initialized.

In the example below, a structure, student, is declared. The members of this structure are name and age. The first member, `name`, is a character array, and the second member, `age`, is an integer. A structure variable, `st1`, is also declared and initialized.

Listing 68.1 : declaring and initializing a structure

```
1   // Listing 68.1 : This program declares and initializes a structure.
2
3   #include<stdio.h>
4   struct student //Declaration of the structure
5   {
6       char name[10];
7       int age;
8   };
9
10  main()
11  {
12      struct student st1 = {“Jerry”, 24}; //Initialization of the structure
13  }
```

Detailed Explanation

- In Line `12`, the first member, `name`, is initialized to `Jerry`, and the second member, `age`, is initialized to `24`.
- The initialization of the members must be in the same sequence in which they appear in the structure declaration.
- To access the members of a structure, the dot operator `(.)` is used. The name of the structure variable is followed by a dot and the name of the structure member. The statement `st1. name` access the first member of the structure `st1`.
- The accessed structure member is used with various statements to obtain the required output. For example, to print the value of the first member, the statement given can be used. The output of the statement is `Jerry`.
- The second member is accessed with the statement `st1.age`. The second member can be printed with the statement given. The output of the statement is `24`.

A structure member may also need to be accessed so that it can be assigned a value. For example, to assign the value `23` to the structure member `age`, the statement given is used.

The members of one structure variable can also be assigned to another structure variable. The members of one structure variable can be assigned to another structure variable with the dot operator. The statement `st2.age=st1.age`; assigns the value of the member `age` of the structure variable `st1` to the member `age` of the structure variable `st2`.

One structure variable can be assigned to another by using the assignment operator `(=)`. For example, if there are two student structure variables, `st1` and `st2`, the value of st1 can be assigned to `st2` by the statement `st2 = st1;`.

The members of a structure can be manipulated to obtain the desired result. The value of the member `age` is `23`. For example, to increase the value of the member `age` by `10`, the example is as follows.

Self Review Exercise 68.1

1. *You learned to access the members of a structure. The declarations of the structure student and structure variable s1 are given below. Choose the statement that should be used to access the structure member* `rollno` *now.*

```
#include<stdio.h>
struct student
{
      int rollno;
      char *name;
};
main()
{
      struct student s1;
}
```

A. `rollno.s1`
B. `s1.rollno`
C. `student.rollno`
D. `s1 rollno`

LESSON 69 : FUNCTION ARGUMENTS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

STRUCTURES AS FUNCTION ARGUMENTS

Structures can be used as function arguments to obtain the required output. For example, you can pass a structure as an argument to a function to print a date that is stored in the structure.

To print date, the structure that stores the date is passed as an argument to the function that prints the date. Entire structures can be passed as arguments to functions. Structure members can also be passed individually as arguments.

By default, a structure is passed as an argument to a function by value. When a structure is passed by value, a local copy of the structure variable is made in the called function. When a structure is passed by value to a function, any change made to the argument in the called function is not reflected in the calling function.

A program is given below. It illustrates the mechanism to pass the structure date by value as an argument to the function `PrintDate`. The function `PrintDate` will be used to print a date.

Listing 69.1 : accessing structure members

```
// Listing 69.1 : This program prints the date stored in a structure

#include<stdio.h>
struct date                                   // Step 1
{
    int month;
    int day;
    int year;
};

void PrintDate(struct date); //function declaration    // Step 2

main()
{
    struct date today = {4, 1, 1986};                  // Step 3
    PrintDate(today);//Function call                   // Step 4
}

void PrintDate(struct date dt) //Function definition   // Step 5
{
    printf("Month = %d", dt.month);                    // Step 6
    printf("Day = %d", dt.day);                        // Step 6
    printf("year = %d", dt.year);                      // Step 6
}
```

Detailed Explanation

- In the first step, the structure `date` is declared as given. This structure has three members; `month`, `day`, and `year`. All the members are of the integer data type.
- In the second step, the function `PrintDate` is declared. The argument to this function, `date`, is enclosed in parentheses.

- A structure variable, `today`, is declared and initialized in the third step. The value of the structure members `month, day`, and `year` are `4`, `1`, and `1986`, respectively.
- In the fourth step of the example given, the `PrintDate` function is called from the `main` function. The function `PrintDate` is called with today as the argument. The argument is enclosed in parentheses.
- In the fifth step, the header of the function definition contains the argument `dt`. This argument is a local copy of the argument today.
- The definition of the function `PrintDate` is displayed in the sixth step. It contains three `printf` functions. These statements print the value of the members `month, day`, and `year`. The output is `Month = 4 Day = 1 Year = 1986`.

Another example is given below. It illustrates that when a structure is passed by value, the change made in the called function are not reflected in the calling function.

Listing 69.2 : structure as a function argument

```
// Listing 69.2 : This program passes a structure as an argument to a function.

#include<stdio.h>
struct date
{
    int month;
    int day;
    int year;
};

void ChangeDate(struct date);

main()
{
    struct date today = {3, 4, 1999};
    ChangeDate(today);
    printf("Month = %d", today.month);
    printf("Day = %d", today.day); //Original value is printed because the
                                   //changed value is not reflected.
    printf("Year = %d", today.year);
}

void ChangeDate(struct date cdate)
{
    cdate.day = cdate.day + 10;//Value is changed in the local variable cdate.
}
```

- A structure, `date`, is declared initially. The structure date has the members `month, day`, and `year`. A function, `ChangeDate`, is also declared. This function will be used to change a date the argument to this function is the structure `date`.
- Next, the structure variable today is declared and initialized. The value of the members `month`, `day`, and `year` are initialized to `3,4,` and `1999`, respectively. The function `ChangeDate` is called with today as the argument.
- When the function `ChangeDate` is called, the value of the argument today is passed to the structure variable `cdate`. This is a local variable. The structure variable `cdate` is recognized only inside the function `ChangeDate`.

- In the next step, the function `ChangeDate` increases the value of the structure member day by `10`. The original value was `4`. The new value of this structure member is `14`.
- The original value, `4`, has changed only for the local variable `cdate`. This will not be reflected in the structure variable today.
- In the final step, three `printf` functions are used print the value of the structure members `month`, `day`, and `year`. The change in the value of `cdate` does not change the value of today. The output Is still `Month = 3 Day =4 Year =1999`.

When an entire structure is passed an argument, all the members of the structure are copied to the called function. As a result, a considerable amount of memory is used.

You may need only some members of the structure in the function to obtain the output. In such a case, the members of the structure are passed to the function as arguments.

An example that illustrates the passing of a structure member as an argument is given below. A structure, `employee`, is declared in the first step. The members of this structure are an integer code, a string `name`, and an integer `salary`.

Listing 69.3 : structure as a function argument

```
// Listing 69.3 : This program passes a structure member as an argument to a function.

#include<stdio.h>
struct employee
{
    int code;
    char *name;
    int salary;
};

int increase(int); // Step 2

main()
{
    struct employee  e1 = {10, “Nancy”, 2000};
    e1.salary = increase(e1.salary);
    printf(“%d”, e1.salary);
}

int increase( int e2)
{
    e2 = e2 + 500;
    return e2;
}
```

- To write a function to increase the salary of an employee, only the structure member `salary` is required in the function. Therefore, you do not pass the entire structure to the function. Only the structure member `salary` is passed as an argument.
- In the second step, it is declaration of the function `increase`. This function accepts an argument of the integer data type. The function increases return an integer value.

- After the function increases is declared, structure variable `e1` of the type employee is declared and initialized in the `main` function. The values assigned to the members `code`, `name`, and `salary` are `10`, `Nancy`, and `2000`, respectively.
- Next, the function increases is called. The parameter passed to this function is the structure member is the structure member `salary`. This member is referred to with the dot operator.
- The header of the `increase` function definition is displayed in this step. The argument for the function is an integer variable, `e2`. Variable `e2` obtain the value `2000` from the structure member `salary`.
- In the next step, the value of local variable `e2` is increase by `500`. The current value `e2` is `500`.
- After the value of local variable `e2` is increased, the value is returned to the main function.
- Finally, the returned value, `2500`, is stored in the member salary. The `printf` function prints the value of the member salary. The output is `2500`.

By default, the structures are passed by value to the functions. Either entire structures can be passed or individual members of the structure can be passed to manipulate the date in the structures.

Self Review Exercise 69.1

1. *The structure is declared before the function when it is passed as an argument to the function. Otherwise, the structure is not recognized by the function. Find the correct answer.*

   When a structure:

   A. is passed to a function, it is passed by reference by default.
   B. is passed to a function, the function is declared before the structure.
   C. is passed to a function by value, a local copy of the structure variable is made i the called function.
   D. is passed to a function by value, any changes made in the called function are reflected in the calling function.

Self Review Exercise 69.2

2. *A block of code in which a structure is passed as an argument to a function is given below. Choose option that contains the output of the function now.*

```
#include<stdio.h>
struct emp
{
      int EmpId;
      int BasicSal;
};
void DisplayRecord(struct emp);
main()
{
struct emp e1 = {45, 2000};
struct emp e2 = {35, 1500};
DispaRecord(e2);
}
void DispRecord(struct emp e3)
{
printf("%d %d", e3.BasicSal, e3.EmpId);
}
```

A. 45 2000
B. 35 1500
C. 1500 35
D. 2000 45

3. *A block of code in which a structure is passed as an argument to a function is given. Choose the option that contains the output of the block of code.*

```
#include<stdio.h>
struct product
{
      int ProCode;
      char *ProName;
};
void PrintDetail(struct product);
main()
{
      struct product p1 = {100, "printer"};
      PrintDetail(p1);
      printf("%s%d", p1.ProName, p1.ProCode);
}
void PrintDetail(struct product p2)
{
      p2.ProCode = 200;
      p2.ProName = "monitor";
}
```

A. printer 100
B. 100 printer
C. monitor 200
D. 200 monitor

Self Review Exercise 69.3

4. *A block of code in which a structure member is passed as an argument to a function is given below. Choose the option that contains the output of the block of code.*

```
//This program passes a structure member as an argument to a function.
#include<stdio.h>
struct employee
{
      int code;
      char *name;
      int salary;
};
int increase(int);
main()
{
      struct employee  e1 = {10, “Nancy”, 2000};
      e1.salary = increase(e1.salary);
      printf(“%d”, e1.salary);
}
int increase( int e2)
{
      e2 = e2 + 500;
      return e2;
}
```

A. 1500
B. 1000
C. 500
D. 10

LESSON 70 : RETURNING STRUCTURES FROM FUNCTIONS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

By default, structures are passed as arguments to functions by value. As a result, any change made to the structure variable in the called function is not reflected in the calling function.

By returning structures from functions, any changes that are made to the structure variable in the called function can be reflected in the calling function.

A program is given below. It illustrates the mechanism for returning a structure from a function. In this program, a function, `adjust`, is called. The argument to this function is the structure `BankAccount`. The function `adjust` also returns the structure `BankAccount`.

Listing 70.1 : Returning a structure

```
//Listing 70.1 : This program returns a structure from a function.

#include<stdio.h>

struct BankAccount                                                               //step1
{
    int ActNo;
    char *name;
    int balance;
};

struct BankAccount adjust(struct BankAccount);                                   //step2

main()
{
    struct BankAccount customer = {333, “David”, 3000);                          //step3
    printf(“%d %s %d\n”, customer.ActNo, customer.name, customer.balance);       //step4
                                                        //Original Values are printed.
    customer = adjust(struct BankAccount cust);                                  //step5
    printf(%d %s %d”, customer.ActNo, customer.name, customer.balance);//New values are
                                                                       //printed.
}

struct BankAccount adjust(struct BankAccount cust)                               //step6
{
    cust.ActNo = 645;                                                            //step7
    cust.name = “Debbie”;
    cust.balance = 4000;
    return cust; //Structure variable cust is returned.                          //step8
}
```

Detailed Explanation

- In the example, changes are made to the structure `BankAccount` by assigning new values to its members. These changes will be reflected in the calling function if you return the structure from the called function.
- In the first step for returning a structure from the function adjust, the structure `BankAccount` is declared. The members of this structure are `ActNo`, name and balance. The member `name` is a string, and the members `ActNo` and balance are integers.

- In the second step, the function `adjust` is declared. The declaration indicates that this function will return the structure `BankAccount`, which is passed to the function adjust as an argument.
- Third, a structured variable, `customer`, is declared and initialized in the `main` function. The value assigned to the structure members `ActNO`, name, and balance are `333`, `David`, and `3000`, respectively.
- Fourth, the `printf` function is specified. The `printf` function prints the values of each member of the structure. The output is `333 David 3000`.
- In the fifth step, the function `adjust` is called. The argument passed this function is the structure variable `customer`. The function `adjust` also returns a structure will be stored in the structure variable `customer`.
- In the sixth step, the header of the `adjust` function definition indicates that the function will return the structure `BankAccount`. The header also indicates that the value of the argument customer is copied in the local structure variable `cust`.
- In the seventh step, new values are assigned to the members of the structure variable `cust` in the function adjust. The values assigned to the members `ActNo`, name, and balance are `645`, `Debbie`, and `4000`, respectively.
- In the eighth step, the return statement returns the structure variable `cust`. The returned structure variable `cust` is stored in the structure variable customer by the statement `customer = adjust (customer);`.
- In the example, the `printf` function in Line No `20` again prints each member of the structure customer. The output is `645 Debbie 4000`. The values in the structure variable customer have changed because the function adjust returned the structure `BankAccount`.
- If the function adjust does not return the structure, the changes made to the structure members will not be reflected in the calling function. In that case, the output of the statement will still be `333 David 3000`.
- By returning structures from functions, any changes made to the structure variable in the called function are reflected in the calling function.

LESSON 70 : RETURNING STRUCTURES FROM FUNCTIONS

Self Review Exercise 70.1

1. *You learned to return a structure from a function. A block of code in which a function returns a structure is given below. Choose the option that contains the output of block of the code.*

```
#include<stdio.h>

struct product
{
int code;
 Char *name;
};

struct product DisRecord(struct product);

main()
{
struct product printer = {100, “laser”};
printer = DispRecord(printer);
printer(“%d %s”, printer.code, printer.name);
}
struct product DispRecord(struct product p1)
{
p1.code=200;
p1.name=“daisywheel”;
return p1;
}
```

A. 100 laser
B. 200laser
C. 200daisywheel
D. 100daisywheel

Lesson Objectives
In this Lesson you will learn :   • *the Genesis of C Language*   • *the Stages in the Evolution of C Language*

UNIONS

A `union` is a collection of variables. These variables are called the members of the `union`. The data types of the members can be different. For example, you can have a `union` that contains an integer, a character, and a `float`.

All the members of a union are not assigned values at the same time. The value of the first member is lost when a value is assigned to the second member because both the members occupy the same memory area.

If you do not keep track of the member that is assigned a value at any specified time, you may access incorrect data. When you assign a value to the first member, it occupies the memory. If you try to access the second member, the value of the first member will be retrieved.

A `union` might contain two members, one integer and one character. When a `union` contains members of different data types, memory is reserved so that the member that occupies the most memory can be accommodated.

An integer occupies `2` or `4` bytes of memory depending on the system, and a character occupies `1` byte. When the members are an integer and a character, the compiler reserves `2 or 4` bytes of memory because integers occupy more memory than characters.

Unions are used in applications that require multiple interpretations for a specific memory block. For example, to store the grade and points scored by a student, you can use a `union`. The grade can be stored as a character, and the points can be stored as an integer.

If the grade and points are stored in a structure, separate memory is allocated for each. However, if you require only the grade or points at a specific time in an application, unions can be used. The same memory is allocated for both the grade and points.

A `union` is a collection of variables that can be of different types. These variables are called members. All the members of a union occupy the same memory area. In this way, unions conserve memory.

A `union` :

- contains different data types.
- contains variables called members.
- contains members that occupy the same memory area.
- conserves memory.

LESSON 71 : `union` -->FEATURES

Self Review Exercise 71.1

1. *Identify a feature of a union.*

   A. It is a fundamental data type.
   B. The members of a union occupy the same memory area.
   C. It can store only one data type.
   D. The variables contained in a union are called tag names.

LESSON 72 : union --> DECLARATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A `union` is a user defined data type that helps to conserve memory.

The syntax to declare a `union` is given below. It starts with the keyword `union`. The keyword is followed by the name of the `union`, which is known as its `tag name`.

```
union tagname
{
 member1;
 member2;
  .
  .
};
```

In the declaration of a `union`, the members of the `union` are enclosed in braces and the declaration ends with a semicolon.

An example of a `union` declaration is given below. In the declaration, the `tagname` of the `union` is product and the `union` contains two members, `code` and `name`.

Listing 72.1 : Union declaration

```
// Listing 72.1 : This program demonstrate union declaration.

#include<stdio.h>

union product
{
    int code;
    char name[15];
};

main()
{
}
```

- The first member of the `union` product, `code`, is an integer. The second member, `name`, is a character array of `15` elements.
- The block of code that declares the `union` product is given. It starts with the keyword `union`. This is followed by the `tag name` product. The members `code` and `name` are enclosed in braces. The declaration ends with a semicolon.
- The declaration of a `union` does not allocate memory for the members of the `union`. The declaration only indicates the variables that will be stored in the `union`.
- `union` declaration are used to create user- defined data types that help in conserving memory.

Self Review Exercise 72.1

1. *You learned to declare a union. Four blocks of code are given below. Choose the code that declared a union.*

```
#include<stdio.h>
union u1
{
      int a;
      Float b;
};
main()
{
}
```
A

```
#include<stdio.h>
u1 union
{
      int a;
      Float b;
};
main()
{
}
```
B

```
#include<stdio.h>
union
{
      int a;
      Float b;
};
main()
{
}
```
C

```
#include<stdio.h>
u1
{
      int a;
      Float b;
};
main()
{
}
```
D

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

UNION VARIABLES —> DECLARATION

`union` variable are declared so that they can be used to store data in unions.

Consider the `union` declaration that is given below. It defines the `union product`. The members of the `union` are `code` and `name`. The member `code` is an integer, and the member `name` is a character array of `15` elements.

Listing 73.1 : Declaring union variables

```
// Listing 73.1 : union variables declaration
#include<stdio.h>
union product
{

    int code;
    char name[15];

};

main()
{
    union product var1, var2;
}
```

A `union` variable can be declared in three ways.

- Outside `union` Declaration
- Within `union` Declaration
- Using the `typedef` Keyword

Outside `union` *Declaration*

The syntax of declaring a `union` variable outside the declaration of the `union` is given below. It starts with the keyword `union`. This is followed by the `tag name` and the `union` variable name. The declaration ends with a semicolon.

```
union tagname variablename;
```

A `union` variable declaration is given in the example. In this example, `var1` and `var2` are the variables of the `union` product. The `union` variables are declared outside the declaration of the `union`.

LESSON 73 : union VARIABLES --> DECLARATION

Within Union *Declaration*

The syntax for declaring a union variable within the declaration of the union is given below. It starts with the keyword union.

```
union
{
 member1;
 member2;

}variablename;
```

When a union variable is declared within the declaration of the union, the tag name of the union is optional. The name of the variable is specified after the braces enclosing the members. A semicolon is placed after the variable name.

In the code that is given below, the variables var1 and var2 are declared within the declaration of the union. In this situation, it is not necessary to specify the tag name.

Listing 73.2 : union variables declaration

```
// Listing 73.2 : declaring union variables
#include<stdio.h>

union
{
    char color[12];
    int size;
}var1, var2;

main()
{

}
```

Using typedef *keyword*

The typedef keyword can also be used to declare unions. To declare a union by using the typedef keyword, this keyword is specified followed by the union keyword. Next, the members and the name of a user-defined data type are specified.

```
typedef union
{

  member1;
  member2;

}user-defined data type;
```

A `union` declaration that uses the `typedef` keyword is given below. The `union` contains the member's `eng` and `math`.

Listing 73.3 : union variable declaration using typedef

```
//Listing 73.3 : using typedef to declare a union
#include<stdio.h>

Typedef union
{

    int eng;
    int math;

}points;

main()
{

points m1;

}
```

Detailed Explanation

- A user-defined data type, point, is also declared with the `union` declaration. The points data type will be used to declare `union` variables.
- When a `union` is declared with the help of the `typedef` keyword, the keyword `union` is not used for declaring the `union` variable. Notice that `union` variable `m1` is declared without the keyword `union` because the `typedef` keyword was used to declare the `union`.
- A `union` variable can also be declared as a member of a structure. For more information on declaring a `union` variable as a member of structure.
- In addition to using the `typedef` keyword to declare a `union`, a `union` can also be declared as a member of a structure. An example illustrating this is given below. Alternatively, a structure can also be declared as a member of a `union`.

In the example below, the `union` product is declared. Its members are an integer, code, and a character array, `name`, A structure, `pen`, and a structure variable, `p1`, are also defined.

Structure variable `p1` will contain the members manufacturer and cost and the `union` variable description. The `union` data type may represent either a `code` or a `name`.

LESSON 73 : union VARIABLES --> DECLARATION

To make the declaration concise, the declarations of the `union` product and the structure `pen` can be combined. In the example given below, the `union` is declared inside the structure `pen`.

Listing 73.4 : declaring a union inside a structure

```
//Listing 73.4 : this program declares a union inside a structure
#include<stdio.h>
union product

struct pen
{
    char manufacturer[2];
    float cost;
 union
 {
    int code;
    char name[15];
}description;
};

main()
{
}
```

A `union` variable can also be declared as a member of a structure. When a `union` variable is declared, memory is allocated for the `union` variable. In the example below, there are two members of different data types.

Listing 73.5 : declaring union variables

```
//Listing 73.5 : declaration of union variables
#include<stdio.h>
union product

{
    int code;
    char name[15];
}var1, var2;

main()
{
}
```

The second member, `name`, of the union is a `15`- character string. It will occupy more storage area than the first member, `code`, which is an integer.

The memory occupied by the `15`-character string is the memory that is allocated to each `union` variable because the character string requires more memory.

A declared `union` variable is used for assigning values to its members. The syntax for assigning a value to a `union` member is given below. The name of the `union` variable is followed by a dot and by the name of the member.

```
variablename.membername = value;
```

To assign the value `20` to the `union` member code, the statement `var1.code=20;` is used. Only one member of the `union` variable is assigned a value at a time.

`Union` variables are declared so that they can be used to store data in unions.

Self Review Exercise 73.1

1. *You learned to declare a* `union` *variable outside the declaration of the* `union`*. Four pieces of code are given below. Choose the code declares a* `union` *variable outside the* `union` *declaration.*

A
```
#include<stdio.h>
union u
{
int x;
char y;
};
main()
{
union u u1;
}
```

B
```
#include<stdio.h>
union u
{
int x;
char y;
};
main()
{
union u1;
}
```

C
```
#include<stdio.h>
union u1
{
int x;
char y;
};
main()
{
u1,u2;
}
```

D
```
#include<stdio.h>
union u
{
int x;
char y;
};
main()
{
u1,u;
}
```

2. *Choose the code that declares a* `union` *variable within the* `union` *declaration.*

A
```
#include<stdio.h>
union
{
int x;
float y;
};var1
main()
{
}
```

B
```
#include<stdio.h>
{
int x;
float y;
};var1
main()
{
}
```

C
```
#include<stdio.h>
union
{
int x;
float y;
}var1;
main()
{
}
```

D
```
#include<stdio.h>
u union
{
int x;
float y;
}var1;
main()
{
}
```

Self Review Exercise 73.2

3. *Choose the code that declares a* `union` *variable by using the* `typedef` *keyword.*

```
#include<stdio.h>
typedef union
{
int a;
float b;
}v1;
main()
{
union v1,u;
}
```
A

```
#include<stdio.h>
typedef union
{
int a;
float b;
}v1;
main()
{
union v1,u;
}
```
B

```
#include<stdio.h>
typedef union v1
{
int a;
float b;
}v1;
main()
{
v1,u;
}
```
C

```
#include<stdio.h>
typedef
{
int a;
float b;
}v1;
main()
{
v1,u;
}
```
D

REVIEW EXERCISE

1. Which of the listed options are features of the structure?

A. It can point to a data type.
B. It is a user-defined data type.
C. It can contain pointers as members.
D. It is effectively used to write and read records in C.
E. It is used to aggregate different data types.
F. The names of the members of a structure are identical.
G. There is no limit to the number of members it can contain.

2. A block of code in which a function returns a structure is given. Choose the option that contains the output of the block code.

```
#include<stdio.h>
struct points
{
       int RollNo;
       int math;
       int  eng;
};
struct points ReportCard();

main()
{
       struct points m1, m2;
       m1 = ReportCard();
       printf("%d %d", m1.RollNo, m1.math + m1.eng);
}

struct points ReportCard()
{
       struct points m3;
       m3.RollNo = 100;
       m3.math = 60;
       m3.eng = 80;
       return m3;
}
```

A. 100 140
B. 100 80
C. 100 60
D. 80 60

3. Four blocks of code are given below. Choose the code that declares a structure.

A.
```
#include<stdio.h>
struct mystruct
{
    int SNo;
    char descrip[15];
};
main()
{
}
```

B.
```
#include<stdio.h>
mystruct struct
{
    int SNo;
    char descrip[15];
};
main()
{
}
```

C.
```
#include<stdio.h>
struct mystruct
{
    char ItemName[10];
    int ItemNo;
};
main()
{
}
```

D.
```
#include<stdio.h>
mystruct
{
    int SNo;
    char descrip[15];
};
main()
{
}
```

REVIEW EXERCISE

4. Four blocks of code are given below. Choose the code that declares a structure variable outside the declaration of the structure.

A.
```
#include<stdio.h>
struct svar
{
    int i;
    char c;
};

main()
{
    struct svar st;
}
```

B.
```
#include<stdio.h>
struct astruct
{
    int i;
    char c;
};

main()
{
    struct s1 astruct;
}
```

C.
```
#include<stdio.h>
struct myst
{
    int i;
    char c;
};

main()
{
    myst s3;
}
```

D.
```
#include<stdio.h>
struct
{
    int i;
    char c;
};

main()
{
    s3;
}
```

5. Four blocks of code are given below. Choose the code that declares a structure variable with the help of the `typedef` *keyword.*

A.
```
#include<stdio.h>
typedef struct
{
    int i;
    char c;
}s1;

main()
{
    struct s1 st;
}
```

B.
```
#include<stdio.h>
typedef
{
    int i;
    char c;
}s1;

main()
{
    s1 st;
}
```

C.
```
#include<stdio.h>
struct
{
    int j;
    char alpha;
};var1

main()
{
}
```

D.
```
#include<stdio.h>
typedef struct
{
    int i;
    char c;
}s1;

main()
{
    s1 st;
}
```

6. Four statements that implement the `typedef` *keyword are given below. Choose the option that displays the correct statement.*

A. typedef Empid int;
B. typedef Empid;
C. typedef char str;
D. typedef str[10] Empid;

7. Four statements that implement the `typedef` *keyword are given below. Choose the option that displays the correct statement.*

```
A. typedef int points;
B. typedef points int;
C. typedef array str[10];
D. typedef str[10] array;
```

8. Four blocks of code are given below. Choose the code that declares a `union`.

A.
```
#include<stdio.h>
union card
{
    char color[15];
    int number;
};

main()
{
}
```

B.
```
#include<stdio.h>
union
{
    char color[15];
    int number;
};

main()
{
}
```

C.
```
#include<stdio.h>
card
{
    char color[15];
    int number;
};

main()
{
}
```

D.
```
#include<stdio.h>
card union
{
    char color[15];
    int number;
};

main()
{
}
```

9. Four blocks of code are given below. Choose the code that declares a `union` *variable within the declaration of the* `union`.

A.
```
#include<stdio.h>
union
{
    char mem1;
    int mem2;
}u1,u2;

main()
{
}
```

B.
```
#include<stdio.h>
union
{
    char mem1;
    int mem2;
}union u1, u2;

main()
{
}
```

C.
```
#include<stdio.h>
union;
{
    char mem1;
    int mem2;
}u1, u2;

main()
{
}
```

D.
```
#include<stdio.h>
union;
{
    char mem1;
    int mem2;
}aunion;

main()
{
}
```

REVIEW EXERCISE

10. Four blocks of code are given below. Choose the code that declares a `union` *variable outside the declaration of the* `union`*.*

A.
```
#include<stdio.h>
union
{
    char mem1;
    int mem2;
};

main()
{
    u1;
}
```

B.
```
#include<stdio.h>
union aunion
{
    char mem1;
    int mem2;
};

main()
{
   aunion u1;
}
```

C.
```
#include<stdio.h>
union aunion
{
    char mem1;
    int mem2;
};

main()
{
   union aunion u1;
}
```

D.
```
#include<stdio.h>
union aunion
{
    char mem1;
    int mem2;
}aunion;

main()
{
   union u1 aunion;
}
```

11. Four blocks of code are given. Choose the code that declares a `union` *variable with the help of the keyword* `typedef`*.*

A.
```
#include<stdio.h>
typedef union
{
    char mem1;
    int mem2;
}u1;

main()
{
   unio u1
}
```

B.
```
#include<stdio.h>
typedef union
{
    char mem1;
    int mem2;
};

main()
{
   union u1;
}
```

C.
```
#include<stdio.h>
typedef union u1
{
    char mem1;
    int mem2;
};

main()
{
   u1 v1;
}
```

D.
```
#include<stdio.h>
typedef union
{
    char mem1;
    int mem2;
}vi;

main()
{
   v1 aunion;
}
```

12. A block of code in which a function accepts a structure as an argument is given below. Four options are given. Choose the option that contains the output of the block of code.

```
#include<stdio.h>
struct BankAct
{
       int ActNo;
       char *CustName;
       int bal;
};

void transfer(struct BankAct, struct BankAct);

main()
{
       struct BankAct cust1, cust2;
       cust1.CustName = “Jim”;
       cust1.ActNo = 100;
       cust1.bal = 20000;
       cust2.CustName = “Anne”;
       cust2.ActNo = 200;
       cust2.bal = 10000;
       transfer(cust1, cust2);
       printf(“%s %d %d”, cust2.CustName, cust2.ActNo, cust2.bal);
}

void transfer(struct BankAct ncust1, struct BankAct ncust2)
{
       ncust2.CustName = ncust1.CustName;
       ncust2.ActNo = ncust1.ActNo;
       ncust2.bal = ncust1.bal;
}
```

A. Jim 100 10000
B. Anne 200 10000
C. Jim 100 20000
D. Anne 100 20000

13. A union:

A. is a substitute for a structure.
B. contains members that are stored in the same memory area.
C. contains members that are not assigned values at the same time.
D. contains members that are constants.
E. is a user-defined data type.
F. contains members that are of different data types.

14. Select the features of an enumerated data type.

A. It contains members that are unique.
B. Its members are directly used as input for the `scanf` *function.*
C. It contains members that can be addressed as integers.
D. It contains members that are constants.
E. It has a possible range of values that are predefined.
F. It contains members that can be assigned explicit values.
G. It is a fundamental data type.

REVIEW EXERCISE

15. Four blocks of code that implement the enumerated data type are given below. Choose the correct code.

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

main()
{
   enum months{january, february, march, april, may, june, july, august, september,
               octomber, november, december};
   enum months m1;
   m1 = “january”;
}
```

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

main()
{
   enum months{january, february, march, april, may, june, july, august};
   enum months m1;
   m1 = december;
}
```

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

main()
{
   enum months{january, february, march, april, may, june, july, august, september,
               octomber, november, december};
   enum months m1;
   m1 = march;
}
```

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

main()
{
   enum months{january, february, march, april, may, june, july, august, september,
               octomber, november, december};
   enum months m1;
   january = 1;
}
```

16. A block of code is given below. In the four statements listed below it, choose the statement that should be used to access a member of the specified structure.

```
#include<stdio.h>
struct date
{
       int mon;
       int day;
       int year;
};

main()
{
       struct date d1 = {12,5,99};
}
```

A. d1 day
B. day.d1
C. d1.day
D. d1.date

17. A block of code in which a function accepts a structure as an argument is displayed on the screen. Four options are displayed. Choose the option that contains the output of the block of code.

```
#include<stdio.h>
struct dist
{
       int feet;
       int inch;
};
void AddDist(struct dist, struct dist);
main()
{
       struct dist d1, d2;
       d1.feet = 5;
       d1.inch = 3;
       d2.feet = 7;
       d2.inch = 4;
       AddDist(d1, d2);
       printf("%d %d", d2.feet, d2.inch);
}
void AddDist(struct dist nd1, struct dist nd2)
{
       nd2.feet = nd1.feet;
       nd1.inch = nd1.inch + nd2.inch;
}
```

A. 57
B. 75
C. 74
D. 53

18. Four blocks of code that implement the enumerated data type are given below. Choose the correct code.

A.
```
#include<stdio.h>
main()
{
   enum cards{diamonds, clubs, hearts, spades};
   enum cards card1;
   diamonds = 3;
}
```

B.
```
#include<stdio.h>
main()
{
   enum cards{diamonds, clubs, hearts, spades};
   enum cards card1;
   card1 = clubs;
}
```

C.
```
#include<stdio.h>
main()
{
   enum cards{diamonds, clubs, hearts, spades};
   enum cards card1;
   card1 = "clubes";
}
```

D.
```
#include<stdio.h>
main()
{
   enum cards{diamonds, clubs, hearts, spades};
   enum cards card1;
   card1 = hearts;
}
```

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE HEADER FILES

A file containing the declarations of functions is called a header file. To include a header file in a program, use the `#include` preprocessor directive.

Statements that begin with the `hash(#)` symbol are called preprocessor directives. The `#` symbol is followed by the directive name.

Pre Processor Directives

C uses the preprocessor and its preprocessor directives to extends its power and notation. The C preprocessor is a macro processor that is used by the C compiler to transform a program before it is compiled.

A preprocessor directive is effective from the point at which it is declared in a program either until the end of the program or until its effect is negated by another directive. For example, the effect of the `#define` directive is negated by the `#undef` directive.

Preprocessor directives are handled by the preprocessor. The remaining part of a program is handled by the compiler. Preprocessor directives that can be used in a program are predefined in C. Programs cannot define new preprocessor directives. Certain preprocessor directives use arguments. These arguments must be separated from the preprocessor directive name by a space.

A preprocessor directive does not end with a semicolon. A preprocessor directive is usually a single line of code unless the statement that contains the preprocessor directive is long. If the statement containing a preprocessor directive is long and cannot fit in a single line of code, the preprocessor directive can be split across multiple lines.

Comments containing newlines can also be used to split a preprocessor directive into multiple lines. However, the comments are changed to spaces before the preprocessor directive is interpreted by the preprocessor.

You can also split a preprocessor directive into multiple lines by using a string constant with a newline. However, the usual way of splitting a preprocessor directive across multiple lines is by using the backslash (\) character before the newline character.

The convention to name a heard file is to give it the `.h` extension. You should avoid special characters, such as `#` and `@`, in header file names because these characters may not be recognized by an operating system (OS).

When the characters used in a header file name are not recognized by an operating system, the portability of the file that contains the characters is reduced.

Types of Header Files

There are two types of header files, system and user-defined. The functionality provided by a system header file differs from the functionality provided by a user-defined header file.

System header files

System header files declare interfaces to parts of the operating system. These files are included in a program to supply the definitions and declarations that are needed to invoke system calls and libraries from the program. An example of a system header file is `stdio.h`.

User-defined header files

User-defined header files contain declarations for interfaces between the source files of a program.

When there is a group of related declarations and macro definitions that are used in various source files, it is a good practice to create a user-defined header file for the declarations and definitions.

Including a header file in a source code file produces the same result in C compilation as copying the header file into the source code file. Copying the entire header file into a source code file can be time consuming and leads to errors. It can also increase the size of the source code file. By including a header file in a source code file, the declarations of the functions that are used in the source code file appear in only one location in the file.

If the declarations of a function need to be changed, they can be changed only in the header file. When the header file is included in a source code file, all programs that include the header file will automatically use its new version when the programs are recompiled.

Including a header file also eliminates the risk of inconsistencies in the declarations used in a program or in multiple programs.

An inconsistency may arise in a program if there are different declarations for the same function in the program. Such inconsistencies are eliminated by including a header file. A header file serves as the central point for storing the declarations used in the programs.

The declarations in a header file are used to execute functions that do not have prototypes in a program. These functions can be directly used in the program without declaring or defining them.

To execute functions that do not have prototypes in a program, you only include the appropriate header file in the program. These declarations may be shared by several source code files.

Header files are of two types, system and user-defined. A header file contains declarations that are required to execute functions in a program and can be shared by several source code files.

Self Review Exercise 73.1

1. *Select the statement that is handled by the preprocessor.*

   A. `#define abc 1`
   B. `main()`
   C. `int i=1;`
   D. `printf("%d",abc);`

2. *A header file is included in a program to:*

   A. Supply the definitions and declarations that are needed to invoke system calls and libraries.
   B. Compile the source file into an object file.
   C. Provide declarations in multiple locations in a program.
   D. Define functions that have prototypes in the program.

LESSON 75 : #include DIRECTIVE --> USES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE #include DIRECTIVE

The #include directive is used to include a header file in a program. Header files provide the definitions and declarations that are needed to invoke system calls and library functions from a program.

The First Variant

- The #include directive has three variants. One of the variants is #include <filename>. This variant is commonly used to include standard header files, such as stdio.h, in a program. The parameter filename in this variant generally stands for a standard header file.
- The variant #include <filename> searches for a file named filename first in a list of directories specified in the compiler options and then in the directories specified by the include environment variable.
- The system directory \include is a part of the standard list of system directories that are specified by the compiler.

The Second Variant

- Another variant of the #include directive is #include "filename". This variant is commonly used to include user-defined header files in a program. The parameter filename in this variant represents a user-defined header file.
- The variant #include "filename" searches for a file named filename first in the current directory and then in the same directories that are used for header files with angle brackets. The current directory is the directory where the file containing the #include directive resides.
- The current directory is searched first because it is presumed to be the location of the files to which the current source code file refers. In the #include "filename" variant of the #include directive, you can also specify the complete path of the file named filename instead of only specifying the name of the file.
- The variant #include macroname of the #include directive is called a computed #include. In this variant, the parameter macroname represents a macro that is defined using the #define directive.
- An example of the variant #include macroname is given below. In the example, variant represents the macro that is defined using #define.

- A macro is an abbreviation that stands for a fragment of code. When a macro is found by the preprocessor in a program, the preprocessor replaces the macro with the corresponding fragment of code.

```
#define variant <stdio.h>
```

The Third Variant

- The variant `#include macroname` allows you to define a macro that specifies a file name to be used at a later point in a program.
- When processing the variant `#include macroname`, the preprocessor checks if the parameter `macroname` is defined using `#define`. If it is defined using `#define`, it is expanded.
- When the `macroname` is expanded, the result of the expansion must match one of the two variants `#include <filename> or #include ”filename”`.

```
#include <filename>
#include “filename”
```

- In particular, the expanded text must be enclosed in either angle brackets or quotation marks. The result of the expansion is `#include<stdio.h>`.

```
#define variant <stdio.h>

  #include variant

#include<stdio.h>
```

- One application of the variant `#include macroname` is to include a platform-specific configuration file specifies the names of the system header files to be used in a program.

A platform-specific configuration file for a program also contains specific details pertaining to the operating system on which the program is run.

You can use many platform-specific configuration files depending on the operating system on which you run the program. This helps in executing a program on various operating system.

The preprocessor directive `#include` is used to include the system and user-defined header files in a program. The `#include` directive has three variants: `#include<filename>, #include “filename”, and #include macroname`.

Self Review Exercise 75.1

1. *The variant ____ searches for a file named filename first in a list of directories specified in compiler options and then in the directories specified by the* `INCLUDE` *environment variable.*

   A. `#include <filename>`
   B. `#include ”filename”`
   C. `#define filename`
   D. `Include <filename>`

2. *The #include directive is used to:*

   A. insert the system and user-defined header files in a program.
   B. compile the source file into an object file.
   C. define a macro.
   D. convert a file before its compilation.

LESSON 76 : #define DIRECTIVE --> USES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The preprocessor directive `#define` allows the definition of identifiers to store values in a program. These identifiers may be constants or macros.

The `#define` directive can be used in two forms. One of the forms is `#define identifier code_string`. In this form, identifier is the name of a macro and code_string is the fragment of code that the macro represents.

```
#define identifier code_string
```

If this form of a `#define` directive occurs in a file, the preprocessor replaces every occurrence of identifier with `code_string` in the remainder of the file. In other words, the macro identifier expands to the fragment of code that the macro represents.

The other form of the `#define` directive is given below. In this form, `identifier(argument1,…, argumentn)` is the macro.

```
#define identifier code_string
#define identifier(argument1, …, argumentn) code_string
```

The parameter `code_string` is the fragment of code that the macro represents. The parameter `code_string` contains the arguments specified in the macro.

If this from of a `#define` directive occurs in a file, the preprocessor replaces every occurrence of `identifier (argument1,….,argumentn)` with `code_string` in the remainder of the file.

An example of the from `#define identifier(argument1,….,argumentn) code_string` is displayed on the screen. In the example, `MUL(x, y)` is the macro and `(x*y)` is the fragment of code that the macro represents.

```
#define MUL(x, y) (x*y)
```

You can declare an identifier that represents a `long` character string or a `long` decimal value by using the `#define` directive.

```
#define MAX 8.147683645
```

The identifier can then be directly used in the rest of the program wherever the string or the value is needed without writing the string constant or the decimal value. If the string or the value is to be changed, it needs to be changed in only the `#define` directive. This helps present errors or inconsistencies in reference to the same string or the same value in the program.

LESSON 76 : `#define` DIRECTIVE --> USES

The `#define` directive is also used to make a program portable. Using the `#define` directive, you can specify different blocks of code to be executed for different hardware platforms or operating systems. The `#define` directive allows you to define you to define a constant or a macro in a program.

Self Review Exercise 76.1

1. The #define directive is used to:

A. supply the definitions and declarations that are needed to invoke system calls and libraries.
B. include a system header file in a program.
C. include a user-defined header file in a program.
D. make a program portable.

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

PREDEFINED MACROS

A predefined macro is a macro that is defined in the C preprocessor. Each predefined macro provides a program-specific or a system-specific value, such as the system time at which the preprocessor is run. You can use a predefined macro without providing its definition.

Predefined macros are a part of the C preprocessor and are always available to a program. A predefined macro can be undefined in a program and then redefined. However, undefining and redefining a predefined macro is not a good programming practice.

Undefining a predefined macro gives a warming. Undefining a macro results in canceling the definition of the macro.

There are five predefined macro in ANSI

C:_DATA_FILE_,_LINE_,_STDC_, and_TIME_.

Each macro name includes two leading and two trailing underscore characters.

- The _DATE_ macro expands to a string constant that represents the date on which the preprocessor is run.
- The string constant containing 11 characters. For example, the string constant Jun 29 1999 represents the date on which the preprocessor was run. The string constant Apr 11 1999 indicates that the preprocessor was run on April 11,1999.
- The _FILE_ macro expends to the name of the current program file as a C string constant. The _FILE_ macro defines a string that is the name of the file currently being processed.
- The _LINE_ macro expands to the current line number as a decimal integer constant.
- The _FILE_ and _LINE_ macros are useful for generating error messages to report the line number or the input file in which an error has occurred.
- The _STDC_ macro expands to the constant 1 to signify that the compiler conforms to ANSI Standard C. The value of this macro remains constant.
- The _TIME_ macro expands to a string constant that describes the time at which the preprocessor is run in the 24_hour format. The string constant contains eight characters. An example of such a string constants is 23:59:01.

Self Review Exercise 77.1

1. *Select the predefined macro that represents the current position of the program control.*

   *A. _LINE_*
   *B. _DATE_*
   *C. _STDC_*
   *D. _TIME_*

LESSON 78 : MACROS WITH ARGUMENTS

Lesson Objectives
In this Lesson you will learn : • *the Genesis of C Language* • *the Stages in the Evolution of C Language*

MACROS WITH ARGUMENTS

Flexibility is added to macros when they accept arguments. Unlike simple macros that always represent the same text each time they are used, macros with arguments provide differing results depending on the arguments they accept.

Arguments are fragments of code that you supply each time a macro is used. According to the macro definition, these fragments are included in the expansion of the macro. A macro that accepts arguments is called a function-like macro because the syntax for using it is similar to a function call.

The syntax for defining macros with arguments is given below.

```
#define identifier(argument1, …, argumentn) code_string
```

- The macro name is `identifier`, and `argument1, …., argumentn` is the list of argument names specified within parentheses.
- The opening parentheses in the macro definition must immediately follow the name without any space.
- The argument names in `(argument1,…, argumentn)` may be any valid C identifiers, and the argument names can be separated by commas. Spaces between arguments are also allowed.
- The parentheses enclosing the argument names in a macro definition must balance. The parameter `code_string` is the fragment of code to which the macro expands each time the macro is used.

A code sample that involves a macro call with arguments is given below. In this code sample, the macro min returns the lower of the two numeric values stored in arguments `X` and `Y`.

Listing 78.1 : using macros using arguments

```
// Listing 78.1 : This program computes the minimum of two numeric values.
#include<stdio.h>
#define min(x, y) ((x) <(y) ? (x) : (y))
main()
{
    int i;
    int x = 3;
    int y = 4;
    i = min(x, y);
    printf("%d", i);
}
```

Detailed Explanation

- In the given code sample, integer variable `i` is declared to store the minimum of any two values. Integer variable `x` is declared and initialized to `3`. In addition, integer variable `Y` is declared and initialized to `4`.
- Identifiers `x` and `Y` in the `#define` statement are the arguments that are substituted in the program wherever the macro name occurs in the program.
- The number of arguments you specify must match the number of arguments that the macro accepts. With the arguments `3` and `4`, the macro call `min(X, Y)` expands to the fragment of code as given.

```
((3) < (4) ? (3) : (4))
```

- The result that is returned by the expanded code, which is `3`, is assigned to integer variable `i`. the last statement in the code sample displays the output of the code, which is the value that is stored in variable `i`.
- The expansion text of a macro with arguments depends on the values that are stored by arguments. Each argument name specified with the macro in a program is replaced with its corresponding value.
- The arguments of a macro can contain calls to other macros that may or may not have arguments. The arguments can also contain calls to the same macro.

```
#define min(x, y) ((x) <(y) ? (x) : (y))
min(min(a, b), c)
```

Another code sample that involves a macro call with arguments is given below. In this code sample, the macro `SQRPRT` computes the square of a numeric value and displays the result along with a constant string.

Listing 78.2 : using macros with arguments

```
// Listing 78.2 : This program computes the square of a numeric value.
#include<stdio.h>
#define SQRPRT(id) printf( "The square of  "#id" is %d \n", id*id)
main()
{
    int k = 10; SQRPRT(k);
}
```

Detailed Explanation

- In the code, integer variable `k` is declared and assigned the value `10`. The macro `SQRPRT` computes the square of variable `k`.
- The identifier id in the `#define` statement is the argument that is substituted in the program wherever the macro name occurs.

- When the statement `SQRPRT(k);` is processed, the statement is replaced with the string `printf (“the square of k is %d\n”, k*k);`. This is because the argument to the macro `SQRPRT` is `k`.
- When the program is executed, the output is `The square of k is 100`. This is because the value of `k` is `10` and `#id` indicates the argument passed to the macro, which is k.

You learned about macros with arguments. To create a macro that accepts arguments, the `#define` directive is used. Macros with arguments are flexible and can accept arguments that provide differing results in a program depending on the values contained in them.

Self Review Exercise 78.1

1. *Select the correct macro definition with arguments.*

   A. `#define SQR(x, y) (((x)*(x))-((y)*(y)))`
   B. `#include SQR(x, y) (((x)*(x))-((y)*(y)))`
   C. `#define SQR (x, y) (((x)*(x))-((y)*(y)))`
   D. `#define SQR(x, y) (((x)*(x))-((y)*(y)));`

2. *With arguments* 3 *and* 5*, the macro call* `ABC(x,y)` *expands to __ for the definition* `ABC(x,y) ((x) * (y))`.

   A. 4
   B. -2
   C. 8
   D. 14

LESSON 79 : #undef DIRECTIVE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

THE #undef DIRECTIVE

You can undefine a macro in a program when it is not required. For example, if you have a macro that has different purposes in different functions, you can undefine the macro in each function and then redefine it in another function. Undefining a macro cancels its definition.

A macro is undefined using the #undef directive, regardless of whether or not the macro contains arguments. The syntax of the #undef directive is #undef followed by the macro name to be undefined.

```
#undef macro name
```

A macro is undefined at a specific point in source code, and the macro is ineffective from that point. After that point in source code, the macro name is not interpreted as a micro by the preprocessor. The preprocessor treats the macro name as a variable.

```
Undefining a Macro
```

Consider the code sample given below. In this code, FOO is a macro that is defined using the #define directive and stores the value 4.

Listing 79.1 : the #define directive illustration

```
// Listing 79.1 : this program demonstrates the #define directive
#include<stdio.h>
#define FOO 4
    int x = FOO;

main()
{

#undef FOO;
    x = FOO;
/*rest of Cpde*/

}
```

Detailed Explanation

- The next statement in the code assigns the value of FOO, which is 4, to a variable named x. Variable x holds the value 4 until is assigned another value.
- The #undef directive in the main function checks if the macro name that is specified as its argument has been defined using the #define directive.
- If the macro name has been defined using the #define directive, the #undef directive undefines the macro.

- If the macro name specified as an argument to the `#undef` directive was not previously defined, the `#undef` directive has no effect. This is because the `#undef` directive has no effect on a name that is not currently defined as a macro.
- If `FOO` was declared as a variable and assigned a value before the `#define` statement, the final statement `x=FOO;` will result in `x` having the same value as `FOO`. This value is the original assignment that `FOO` had before it was declared as a macro.
- In this topic, you learned about the `#undef` directive. This directive is used to cancel the definition of a macro that is not required in a program. After the definition of a macro is canceled, the macro name is treated by the preprocessor as a variable.
- The Line 9 results in a compilation error. Notice that at this point, `FOO` does not hold the value that it was assigned during its definition because the `#undef` directive cancelled its definition. Therefore, `FOO` is treated as undeclared when this statement is compiled.
- If `FOO` was declared as a variable and assigned a value before the `#define` statement, the final statement `x=FOO;` will result in `x` having the same value as `FOO`. This value is the original assignment that `FOO` had before it was declared as a macro.

#define directive is used to cancel the definition of a macro that is not required in a program. After the definition of a macro is canceled, the macro name is treated by the preprocessor as a variable.

Self Review Exercise 79.1

1. *Select the option in which the* #undef *statement is effective.*

A. `#define MACRO 3.198`
```
      #undef MACRO
```
B. `#define “MACRO.h”`
```
      #undef MACRO
```
C. `#define Null 0`
```
              #undef MACRO
```
D. `#define MACRO 12;`
```
                      #undef MACRO
```

LESSON 80 : CONDITIONAL COMPILATION --> ADVANTAGES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

CONDITIONAL COMPILATION

Conditional compilation allows you to exclude program code from compilation or include program code for compilation base on certain conditions.

Conditional compilation is used for program development and for writing code. Conditional compilation is used to make easily portable from one system to another. Using conditional compilation, you can specify different blocks of code to be compiled for different hardware platforms or operating systems.

ADVANTAGES

- Another advantage of conditional compilation is that it allows a part of a program to be ignored during compilation based on certain conditions.
- Ignoring a part of a program during compilation based on a condition results in fast compilation. This is because only the part of the program that specifies the condition is compiled.
- The condition based on which a part of a program is ignored or included during compilation could be either an arithmetic expression or a condition that checks if a name is defined as macro.
- In some cases, the code for one operating system may be invalid for other operating systems. This is because the code may contain references to library routines that do not exist on the other operating systems.
- When code is system-specific for an operating system, it is not enough to avoid executing it. Including it in the program makes it impossible to compile and link the program. Using conditional compilation, you can prevent system-specific code from being compile and linked.
- Conditional compilation also allows you to compile a source file into two different programs. The difference between the programs could be that one program runs frequent time-consuming consistency checks on its intermediate data or prints the value of data for debugging while the other program does not do this.
- By compiling the source file into two programs, you can run either of the programs depending on your requirement.
- An example of compiling the source file into two programs is that if a source file contains two modules that perform different tasks, you can use conditional compilation to compile the modules into two separate programs.

- Conditional compilation allows you to retain the code to be executed from program compilation and execution for future use.
- To retain the code to be executed from program compilation for future use, a conditional that has a false condition is used. A conditional is a directive used for conditional compilation. If the condition is always false, the code will always be excluded from compilation.
- If developers want to alter source code, they can add the new code to the old code and use conditional compilation to compile the new code into the final program.
- If the new program does not perform as expected, it can be recompiled using the old code without rewriting it.

Self Review Exercise 80.1

1. *Conditional compilation allows you to:*

   A. ignore a part of a program during compilation based on certain conditions.
   B. cancel the definition of a macro.
   C. include a header file in a program.
   D. define a macro in a program.

2. *Conditional compilation enables you to:*

   A. remove inconsistencies while referencing long values in a program.
   B. include system header files in a program.
   C. exclude invalid code from a program.
   D. print a macro value in a program.

LESSON 81 : CONDITIONAL COMPILATION --> DIRECTIVES

Lesson Objectives
In this Lesson you will learn : • *the Genesis of C Language* • *the Stages in the Evolution of C Language*

DIRECTIVES

There are six directives that are available in C to control conditional compilation.

These directives are
`#if`
`#ifdef`
`#ifndef`
`#else`
`#elif`
`#endif`

The directives `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` are called conditional compilation directives because they delimit blocks of program text that are compiled only if specified condition is `true`.

Conditional compilation directives can be nested. When these directives are nested, the program text within each directive block can be arbitrary. The program text can consist of preprocessor directives or C statements.

Listing 81.1 : Illustrating Directives

```
//Listing 81.1 : this program illustrates the usage of Directives
#if OS == 1
    printf("Version 1.0");
#elif OS == 2
    printf("Version 2.0");
#else
    printf("Version Unknown");
#endif
main()
{
/*rest of code*/
}
```

Detailed Explanation

- The beginning of a block of text is marked by one of the conditional compilation directives, `#if`, `#ifdef`, or `#ifndef`
- Conditional compilation directives, such as `#if`, `#ifdef`, and `#ifndef`, allows you to include more than one definition of an identifier in a program.
- Using the `#if`, `#else`, or `#elif` directives, you can provide different definitions for an identifier to store different values in a program depending on certain conditions.

- Only one definition of the identifier will be passed to the compiler even if multiple definitions are provided for the identifier. This is because only one directive statement will be `true` depending on the condition.

```
#if expression newline
```

- The syntax of the `#if` conditional compilation directive is `#if` expression `newline` in which expression represents a condition.
- The condition expression of the `#if` directive evaluates to a nonzero value or the value `0` depending on whether the condition in expression is `true` or `false`.

The operand used in the condition must be a constant integer expression that does not contain any increment `(++)`, decrement `(--)`, pointer `(*)`, address `(&)`, `sizeof`, or `cast` operators.

Using* `#if` *Compilation Directive

The output of the code sample is `This is C Programming!` because the condition in the `#if` directive is true.

Listing 81.2 : The #if directive

```
//Listing 81.2 : using #if directive
main()
{

#if 1
    printf("This is C programming!\n");

}
```

Using* `#ifdef` *Compilation Directive

Another conditional compilation directive is `#ifdef`. The syntax of this directive is `#ifdef identifier nemline` in which identifier is either a constant or a macro that is defined using `#define`.

```
#ifdef identifier newline
```

The `#ifdef` directive checks if an identifier is currently defined in a program. An identifier can be defined using a `#define` directive or on the command line.

If identifiers have not been undefined before the `#ifdef` directive, they are treated as if they are currently defined.

An example of the `#ifdef` directive is given below. The directive checks if the identifier `DEBUG` is currently defined in the program. If it is defined, the statement This is a debug message is printed.

Listing 81.2 : The #ifdef directive

```
//Listing 81.3 : using #ifdef directive
#define DEBUG
#ifdef DEBUG
    printf("This is a debug message");
#endif
main()
{
/*rest of the code*/
}
```

LESSON 81 : CONDITIONAL COMPILATION --> DIRECTIVES

The preprocessor directive `#ifdef` is also a conditional compilation directive. It checks if an identifier is not currently defined in a program.

Using `#ifndef` *Compilation Directive*

The syntax of the `#ifdef` conditional compilation directive is `#ifndef identifier newline`.
The parameter identifier can be a constant or a macro.

```
#ifndef
#ifdef identifier newline
```

An example of the `#ifndef` directive is given below. The `#ifndef` directive checks if the identifier `DEBUG` is not currently defined in the program. If it is not currently defined, the statement This is not a debug message is printed.

Listing 81.4 : The #ifndef directive

```
//Listing 81.4 : using #ifndef directive
#define RELEASE
#ifndef DEBUG
    printf("This is not a debug message");
#endif
main()
{
/*rest of  the code*/
}
```

Using `#else` *Compilation Directive*

When conditional compilation directives are nested, an alternative block of text can begin with one of the two conditional compilation directives `#else` or `#elif`.

The `#else` conditional compilation directive delimits alternative source text to be compiled if the condition in the corresponding `#if` directive, the `#def` directive, or the `#ifndef` directive is `false`. The `#else` directive has the syntax `#else newline`.

```
#else
#else newline
```

The example below explains the #else conditional compilation directive. If variable `OS` is equal to `1`, the string `Version 1.0` is printed. Otherwise, the string `Version Unknown` is printed.

Listing 81.5 : Using #else directive

```
//Listing 81.5 : using #else directive
#if OS == 1
    printf("Version 1.0");

#else
    printf("Version Unknown");

#endif

main()
{
/*rest of code*/
}
```

The conditional compilation directive `#endif` ends the scope of the `#if`, `#ifdef`, `#ifndef`, `#else`, and `#elif` directives. The syntax of the `#endif` directive is `#endif newline`.

The `#elif` conditional compilation directive delimits the source code lines to be compiled if the conditions in the corresponding `#if`, `#ifdef`, and `#ifndef` directives are `false` and the condition in the `#elif` directive is true.

The example below explains the `#elif` conditional compilation directive. If variable `OS` is equal to `1`, `Version 1.0` is printed. If variable `OS` is equal to `2`, `Version 2.0` is printed. Otherwise, `Version unknown` is printed.

Listing 81.5 : Using #elif directive

```
//Listing 81.6 : using #elif directive
#if OS == 1
    printf("Version 1.0");

#elif OS == 2
    printf("Version 2.0");

#else
    printf("Version Unknown");

#endif

main()
{
/*rest of code*/
}
```

Using `#endif` *Compilation Directive*

The end of block of text is marked by the `#endif` conditional compilation directive. The conditional compilation directive `#elif` performs a task that is similar to the combined use of the `else-if` statements in C. Its syntax is `#elif constant-expression newline`.

An example of the `#endif` conditional compilation directive is given below. In this example, the directive ends the scope of the `#if` and `#elif` conditional compilation directives.

Listing 81.5 : Using #endif directive

```
//Listing 81.7 : using #endif directive
int main(void)
{
#if 1
    printf("Checkpoint1\n");

#elif 1
    printf("Checkpoint2\n");

#endif
return 0;
}
```

Self Review Exercise 81.1

1. *You use the ___ directive to include more than one definition of an identifier in a program.*

   A. `#if`
   B. `#define`
   C. `#include`
   D. `#undef`

2. *Select the directive that checks if an identifier is not defined in a program.*

   A. `#ifdef`
   B. `#ifndef`
   C. `#include`
   D. `#define`

REVIEW EXERCISE

1. Identify the predefined macro for a specific purpose. Type the predefined macro that expands to a string constant indicating the date on which the preprocessor is run.

2. Enter the predefined macro that expands to the name of the current program file in the form of a C string constant.

3. Enter the predefined macro that expands to the current line number in the form of a decimal integer constant.

4. Enter the predefined macro that expands to the constant `1`*, which signifies that the compiler conforms to ANSI Standard C.*

5. Enter the predefined macro that expands to a string constant describing the time at which the preprocessor is run.

6. Identify the uses of the `#define` *directive.*

A. It helps to prevent errors in references to long values in a program.
B. It includes system and user-defined header files.
C. It requests the use of a system `.dll` *file in a program.*
D. It makes a program portable.

7. Identify the conditional compilation directive to be used in a specific situation. The conditional compilation directives are listed on the left and the situations on the right.

A. `#if expression newline`		*A.*	*You want to check if a expression is currently defined in a program*
B. `#ifdef expression newline`		*B.*	*You need to check if expression is true or nonzero.*
C. `#ifndef expression newline`		*C.*	*You want to check if expression is not currently defined in a program.*

8. In the code samples, choose the code sample in which the `#undef` *directive is effective.*

A.
```
#include<stdio.h>
main()
{
        #define BUFFER 4
        x = BUFFER
        #undef NULL
}
```

B.
```
#include<stdio.h>
main()
{
        int x;
        #define BUFFER 4
        #define SIZE 1020
        x = BUFFER ;
        #undef NULL
}
```

C.
```
#include<stdio.h>
main()
{
        #define BUFFER 10
        x = BUFFER
        #undef STATE
        x = STATE
}
```

REVIEW EXERCISE

8. The uses of the `#include` *directive are to:*

A. include user-defined header files in a program.
B. include `system.dll` *files in a program.*
C. define functions.
D. include a file name specified by a macro.
E. include system header files in a program.

9. Conditional compilation helps to:

A. undefined a macro definition.
B. remove invalid code from a program.
C. include the contents of a library file in the source file.
D. write portable code.
E. compile a source file into two different programs.
F. ignore a part of a program during compilation.
G. retain the code to be excluded from a program for future use.

10. A code sample that involves a macro call with arguments is given below. Choose the output of the code.

```
#include<stdio.h>
#define PR(id) printf("The value of "#id" is %d\n", id)

main()
{
       int i = 10; PR(i);
}
```

```
A. The value of i is id.
B. The value of i is PR.
C. The value of i is 10.
D. The value of i is i.
```

11. Identify the advantages of including header files in a program.

A. Prototypes are not needed in the source code file if they are in the header file.
B. The declarations for functions can be changed in only one location.
C. A declaration for a function is changed at multiple locations.
D. The size of the source code file is reduced.
E. The effort of finding and changing all copies of declarations is eliminated.
F. A declaration for a function is available at multiple locations.
G. The declarations for functions appear in only one location.

12. A code sample that involves a macro call with arguments is displayed on the screen. Choose the correct output of the code.

```
#include<stdio.h>
#define SQ(x, y) ((x * x) < (y) ? (x) : (y))

main()
{
       int x = 3;
       int y = 2;
       printf("The output of the code sample is %d.", SQ(x, y));
}
```

```
A. The output of the code sample is 2.
B. The output of the code sample is 3.
C. The code sample will give an error.
D. The output of the code sample is 9.
```

LESSON 82 : ALLOCATING MEMORY

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ARRAY OF POINTERS: ALLOCATING MEMORY

A programmer can dynamically allocate memory for an element of an array of pointers by using the `malloc` function. Dynamic allocation helps in conserving memory because memory is allocated according to the number of bytes required, which can vary from time to time.

An example that does not use dynamic memory allocation is given. In the example, a fixed size array called `st` is created to store the names of students. The student names having varying lengths. This results in additional memory allocation.

```
char st[50];
```

The next example uses dynamic memory allocation. It is possible to calculate the length of each name when it is specified and allocate memory dynamically according to the length. Therefore, memory is conserved.

```
char *name;
```

Dynamic memory allocation can be done using pointers. When a pointer is created, it does not store any address. Therefore, memory can be allocated according to the requirement at run time. The address of the allocated memory can then be stored in a pointer.

The `malloc` function is used to store an address in a pointer at run time. The function accepts an argument that specifies the bytes of memory to be allocated. This argument can either be specified using an integer value or the `sizeof` operator.

Partial code that uses the method of passing an integer as an argument to the `malloc` function is given. In line `9`, the `malloc` function allocates 4 bytes of memory and assigns the address of the first byte of the allocated memory to the pointer `ptr`.

Listing 82.1 : Demonstrating memory allocation

```
1   // Listing 82.1 : This program demonstrate memory allocation using malloc function.
2
3   #include <stdlib.h>
4   #include <stdio.h>
5   main()
6   {
7       int *ptr;
8
9       ptr = malloc(4);
10  }
```

The above example should be used on machines in which an integer occupies 4 bytes of memory. However, this code will reserve additional memory if it is executed on a machine in which an integer occupies 2 bytes.

LESSON 82 : ALLOCATING MEMORY

The second method of allocating memory dynamically is to use the `sizeof` operator with the `malloc` function. The `sizeof` operator is used to find the size of a specified data type. For example, this operator can be used to find the size of a `float` data type.

Note in the code given below. The `sizeof` operator returns the number of bytes occupied by an integer on the computer on which the code is executed. The address returned by the `malloc` function is assigned to `ptr` by using the assignment `(=)` operator.

In the code given below, the use of `sizeof` operator makes the code appropriate for any computer, regardless of the number of bytes allocated for an integer on that computer.

A student name cannot exceed 20 characters. To allocate memory for a student name by specifying an integer as an argument, enter `cptr=malloc(20);`.

Listing 82.2 : memory allocation using malloc function

```
// Listing 82.2 : This program uses malloc to allocate memory

struct cust
{
    char *name;
    int accno;
    char acctype;
    float balance;
}*custptr;

custptr = malloc(sizeof(struct cust));
```

You specified the code to allocate `20` bytes of memory for a character array by using the `malloc` function. The `malloc` function returns a `void` pointer, which contains the address of the first byte of the allocated memory. A `void` pointer can store the address of any data type.

Some compilers report an error if you use the `malloc` function as in the code sample. Typecasting the return value of the `malloc` function can rectify this error. The second code sample typecasts the return value of the `malloc` function.

Listing 82.3 : using malloc for memory allocation

```
// Listing 82.3 : This program uses malloc to allocate memory

struct cust
{
    char *name;
    int accno;
    char acctype;
    float balance;
}*custptr;

custptr = (struct cust *)malloc(sizeof(struct cust));
```

The `malloc` function can also be used to allocate memory for the elements of an array of pointers. You can use the `malloc` function to allocate either a fixed number of bytes or a variable number of bytes for each pointer of an array of pointers. The first way of using the `malloc` function, which allocates a fixed number of bytes for each element of an array, is used when all data items have a fixed size.

An example that illustrates the first way of using the `malloc` function to allocate memory for an array of pointers is given below. The declaration of `arr`, an array of `15` pointers.

Listing 82.4 : memory allocation using malloc function

```
// Listing 82.4 : This program uses malloc to allocate memory

#include <stdlib.h>
#include <stdio.h>

main()
{
    int num;
    char *arr[15];

    for(num = 0; num < 15; num++)
    {
            arr[num] = malloc(50);
    }
}
```

In the example, a loop is used. In each pass of the loop, the function allocates `50` bytes of memory for a string and assigns the address of the first byte of each string to the respective pointer. The name of the pointer is followed by a subscript.

Each string may or may not require `50` bytes of memory. However, in the code sample given, each string is allocated `50` bytes of memory. The can result in additional memory being reserved if all strings do not require `50` bytes.

An example that illustrates the second way of using the `malloc` function is given below. The example accepts `50` student names by allocating the memory required for each student name. A string of `50` characters called `sname` is declared.

Listing 82.5 : using malloc to allocate memory

```
// Listing 82.5 : This program uses malloc to allocate memory

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

main()
{
    int num, length;
    char sname[50];
    char *nameptr[50};

    for(num = 0; num < 50; num++)
    {
            printf("Type name :";
            gets(sname);

            length=strlen(sname);

            nameptr[num]=malloc(length);
    }
}
```

Detailed Explanation

- The string variable `sname` is used in a loop to accept student names. After the student name is accepted, the actual length of each student name is calculated. The length is calculated using the `strlen` function, and the result is stored in the variable length.
- In line `20` of the example, the variable name length is passed to the `malloc` function. The second way of using the `malloc` function to allocate memory for an array of pointers is more effective because it allocates only the required memory for each name.

Self Review Exercise 82.1

1. *Choose the code sample that can be used to allocate memory for the elements of an array of* 50 *pointers to integers, which is called* numptr.

```
#include<stdlib.h>
#include<stdio.h>
main()
{
  int num, *number[50];
  for(num = 0; num < 50; num++)
        numptr[num] = malloc(int);
}
```
A

```
#include<stdlib.h>
#include<stdio.h>
main()
{
  int num, *number[50];
  for(num = 0; num < 50; num++)
        Numptr = malloc(sizeof(int));
}
```
B

```
#include<stdlib.h>
#include<stdio.h>
main()
{
  int num, *number[50];
  for(num = 0; num < 50; num++)
        numptr[num] = malloc(1);
}
```
C

```
#include<stdlib.h>
#include<stdio.h>
main()
{
  int num, *number[50];
  for(num = 0; num < 50; num++)
        numptr[num] = malloc(sizeof(int));
}
```
D

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

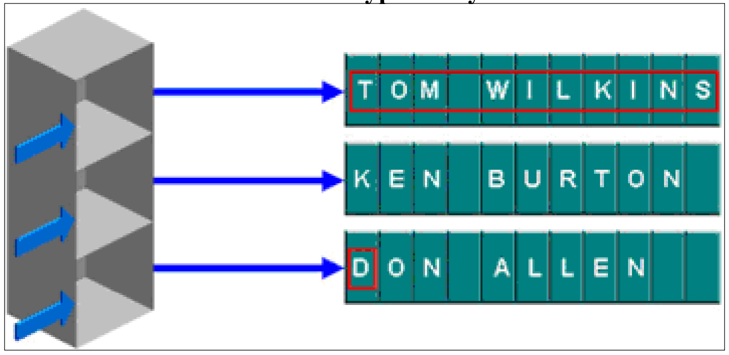
The elements of an array can be accessed and used in various ways. For example, programs can use an array of pointer to display all of specific values of its elements, accept values into its elements, or perform calculations.

To display and accept data into the elements of an array of pointers, standard C functions are used. These functions accept strings or the variables of fundamental data types as arguments. The standard C functions do not accept arrays of pointers as arguments.

To use the standard C functions to accept or display the data from an array of pointers, the elements of the array of pointer should be accessed separately.

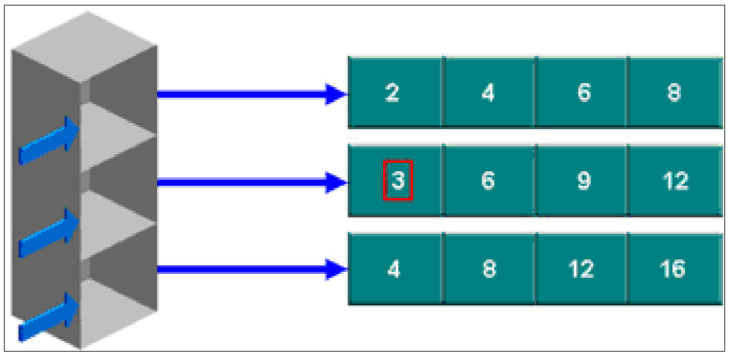
Consider the diagrammatic representation of an array given below. It is possible to access a complete name, such as `TOM WILKINS`, or a single character, such as `D`, from the array of pointers.

Character Type Array



Another example below shows the diagrammatic representation of an integer type array. The elements of this array of pointers can be used to perform calculations such as adding `1` to the value

Integer Type Array



LESSON 83 : ACCESSING ELEMENTS

It is possible to access a complete string or a single character from a character type array of pointers. In an integer type array of pointers, it is possible to access only a single element.

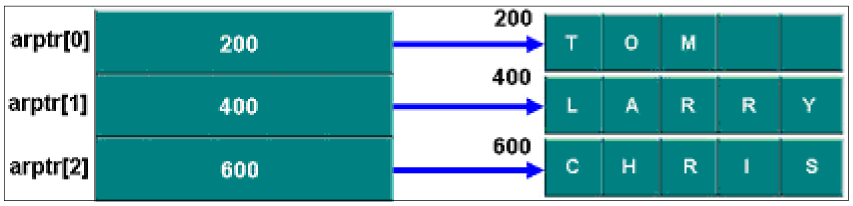
In the example given below, a character type array of pointers, `arptr`, is created. The `malloc` function is called in a loop to reserve memory for each string. The pointer name `arptr` is used with a single subscript to refer to each string in the array of pointers.

Listing 83.1 : Demonstrating array of pointers

```
// Listing 83.1 : This program demonstrate array of pointers

#inlcude <stdlib.h>
#include <stdio.h>
main()
{
    int num;
    char *arptr[5];
    for(num=0;num<5;num++)
    {
            arptr[num]=malloc(20);
    }
}
```

The addresses stored in the pointers after calling the `malloc` function are given below diagrammatically. Each pointer in the array of pointers stores the address of the first character of a string. In other words, `arptr` is a two-dimensional array, which can store three strings.

Array of Pointers



For example, the first pointer in the array of pointers, `arptr[0]`, stores the address of the first character of the first string. The pointer `arptr[1]` stores the address of the first character of the second string. Therefore, the second string can be accessed as `arptr[1]`.

The pointer `arptr[2]` stores the value 600, which is the address of the first character of the third string. Therefore, the third string can be accessed as `arptr[2]`.

Example that access individual string from an array of pointers is given below. The `gets` function is called inside a loop to accept data into individual strings by referring to the strings as `arptr[num]`.

Listing 83.2 : demonstrating array of pointers

```
// Listing 83.2 : This program  uses for memory allocation and accesses elements via a pointer

#include <stdlib.h>
#include <stdio.h>

main()
{
    int num;
    char *arptr[5];

    for(num=0;num<5;num++)
    {
            arptr[num=malloc(11);
            gets(arptr[num]);
    }
}
```

The pointer `arptr` stores the address of the first pointer in the array of pointers. Therefore, the first string in the array can also be accessed as `*arptr`. The expression `*arptr` is the same as `arptr[0]`. A single subscript is used to refer to a string because a string can be referred to using its starting address.

In addition to accessing strings, it is also possible to access each character of a string. A character of the first string is accessed as `arptr[0][n]` where n is replaced with the character number minus `1`. For example, the second character of the first string is accessed as `arptr[0][1]`.

The characters of the second string can be accessed as `arptr[1][n]` where n is replaced with the character number minus `1`. For example, the third character of the second string is accessed as `arptr[1][2]`.

The indirection operator (`*`) can also be used to access characters from a string. The expression `*(*arptr)` can be used to access the first character of the first string.

The first character of the second string can be accessed as `*(arptr[1])`. Similarly, the first character of any of the strings can be accessed as `*(arptr[n])` where n is the string number minus `1`.

The elements of an array of pointers to integers are accessed using a double subscript. The first subscript specifies the starting address of an array, and the second subscript specifies the location of an integer within the array.

Each integer is stored at a single location. Therefore, an integer is accessed using the expression `intptr[r][c]` where r is replaced with the row number and c is replaced with thee column number in the array. For example, `intptrp2][1]` is used to access the value `8`.

It is possible to access each string and each character from a character type array of pointers. In an integer enter array of pointers, it is possible to access only an element.

Self Review Exercise 83.1

1. *Partial code that declares and initializes an array of pointer is given. Choose the code sample that is used to display the second string from the array of pointers called* `arrp`.

```
#include <stdlib.h>
#include <stdio.h>
main()
{
        int num;
        char *arrp[5];
        for(num=0;num<5;num++)
        {
        arrp[num]=malloc(20);
        gets{arrp[num]);
        }
/*Remaining part of the code is here*/
}
```

A. `puts(arrp);`
B. `puts(*arrp);`
C. `puts(arrp[1]);`
D. `puts(arrp[2]);`

2. *You learned how to access individual characters in an array of pointers. Choose the option that can be used to access the character* `B` *from the array.*

```
#include <stdlib.h>
#include <stdio.h>
main()
{
        int num;
        char *str[5];
        for(num=0;num<5;num++)
str[num]=malloc(11);
        str[0]=”Tom Wilkins”;
        str[1]=”Ken Burton”;
        str[2]=”Larry Williams”;
/*Remaining part of the code is here*/
}
```

A. `str[1]`
B. `str[1][4]`
C. `str[2][5]`
D. `str[2]`

Self Review Exercise 83.2

3. *A code sample that declares and initializes a character array is displayed. Choose the option that can be used to access the first character from the second string.*

```
#include <stdlib.h>
#include <stdio.h>
main()
{
       int num;
       char *str[5];
       for(num=0;num<5;num++)
        str[num]=malloc(11);
       str[0]="Tom Wilkins";
       str[1]="Ken Burton";
       str[2]="Larry Williams";
/*Remaining part of the code is here*/
}
```

A. `*(*str)`

B. `str[1]`

C. `*(str[1])`

D. `*(*str[1])`

LESSON 84 : ARITHMETIC

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

POINTER ARITHMETIC

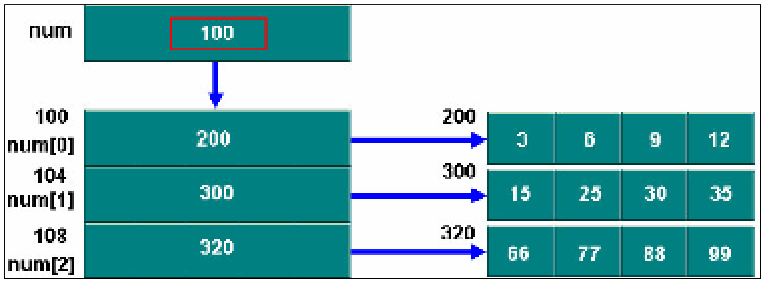
Arithmetic expressions can be used on arrays of pointers in various types of programs. These include programs that need to access the various elements in an array of pointers. Arithmetic operators are also used to assign an address to a pointer or increment or decrement the value stored in a pointer. Arithmetic operators, such as the addition operator (+) and the subtraction operator (-), can be combined with the * operator to access the elements in an array. This process of combining the two operators is called pointer arithmetic.

A part of the code that accesses the elements of an array of integers is given below. The statement in line declares an array of three pointers: `num[0]`, `num[1]`, and `num[2]`.

Listing 84.1 : Demonstrating pointer arithmetic

```
// Listing 84.1 : This program demonstrating pointer arithmetic

#include <stdio.h>
int a[3][4] = {
                     {3,6,9,12},
                     {15,25,30,35},
                     {66,77,88,99}
              };
main()
{
    int *num[3];
    num[0]=a[0];
    num[1]=a[1];
    num[2]=a[2];
/*Remaining part of the code is here*/
}
```

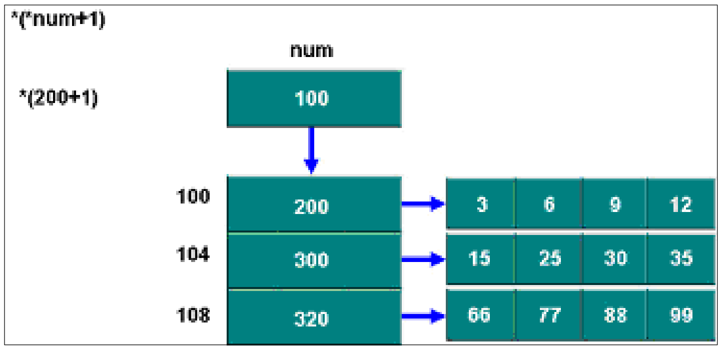
The name of the array `num` contains the address 100, which is the address of the first pointer in the array of pointers. The pointer `num` stores the address of `num[0]`. Therefore, the expression `*num` is equal to `num [0]` and refers to the address 200.



Each pointer in the array of pointers contains the address of the first element of the three single dimensional arrays.

In an integer array, each element can be accessed separately. To access the first element of the first array, the expression `*(*num)` can be used. The expression `*(*num)` is the same as `num[0][0]` and refers to the value 3 in the array as defined in figure above.

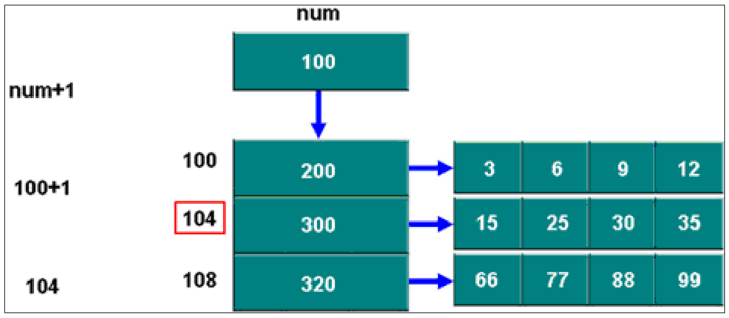
The second element of the first array, `num[0]` can be accessed using the expression `*(*num+1)`. The entire expression can be resolved in three steps. In the first step, the expression `*num` is resolved because the *operator has the highest precedence.



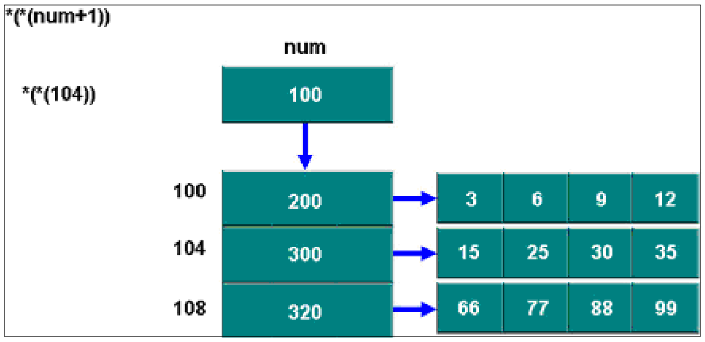
In the second step the expression `*num+1` increments the address 200 so that the address of the next element is generated. This generates the result 204.

The expression `*(204)` is finally resolved to refer to the value at the address 204, which is the value 6.

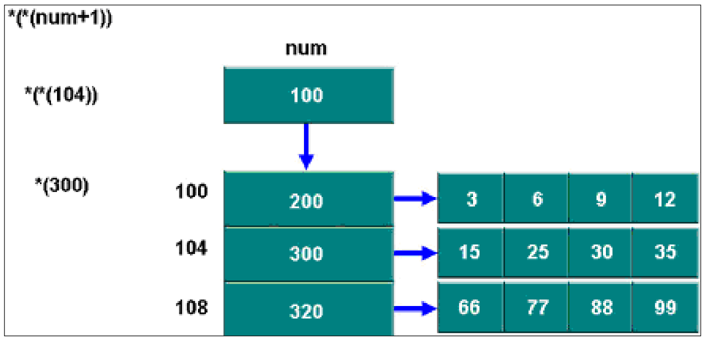
The element `num[1]` can also be written as `num+1`. This is because the address stored in `num` is 100. Therefore, the expression `num+1` will generate the address 104. this is the same as the address stored in `num [1]`.



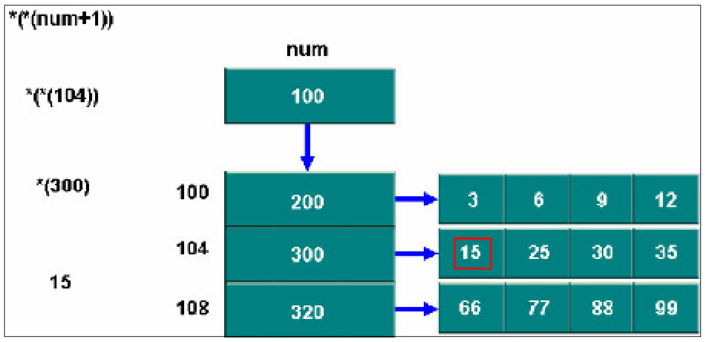
The first element of `num[1]` can be accessed using the expression `*(*(num+1))`. This expression is resolved in three steps. In the first step, the expression `(num+1)` is resolved as 104.



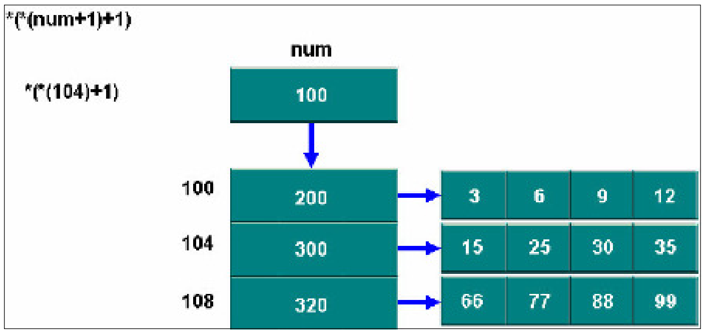
In the second step, the expression becomes `*(*(104))`. The value stored at the address 104, which is 300, is another address. Therefore, the expression `*(*104))` is resolved in the next step as `*(300)`.



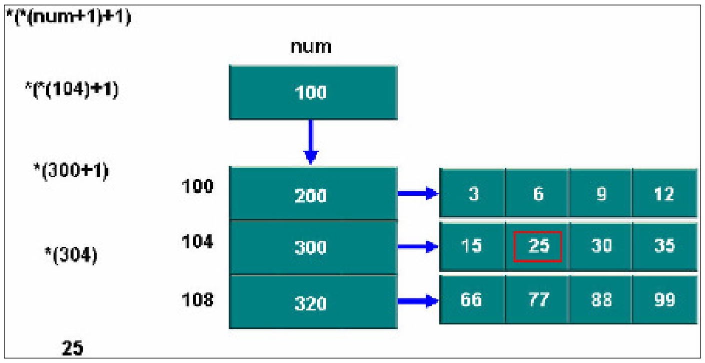
Finally, the expression `*(300)` is resolved and the value at the address 300, which is 15, is extracted. This is how the expression `*(*(num+1))` is used to access the first element of `num[1]`.



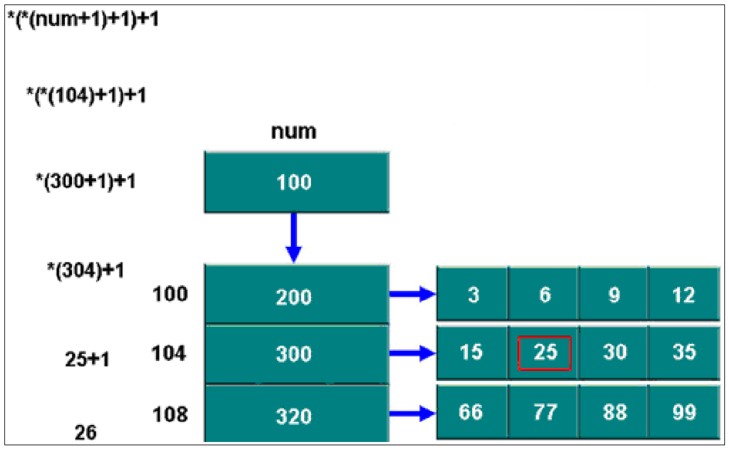
The second element in the second array can be referred to as `*(*(num+1)+1)`. In this expression, `num+1` is resolved first. The expression becomes `*(*(104)+1)` because the expression `num +1` generates the result 104.



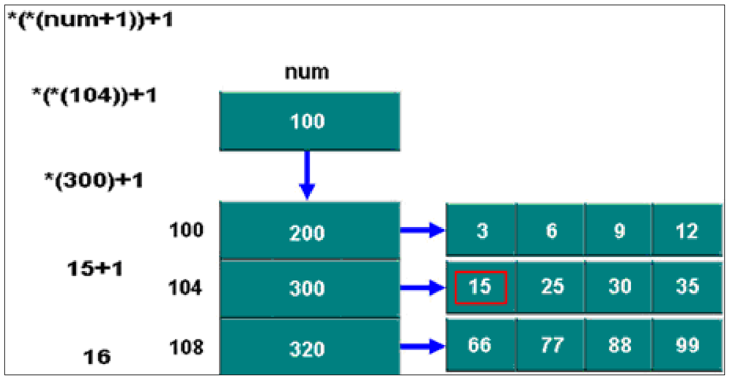
The expression `*(*104)+1)` is resolved to `*(300+1)`. This, in turn, becomes `*(304)`. This generates the result 25.



Arithmetic expressions can also be used with pointers to perform calculations. Calculations can be performed on the values stored in the elements. For example, the expression `*(*(num+1)+1)+1` generates the result 26.



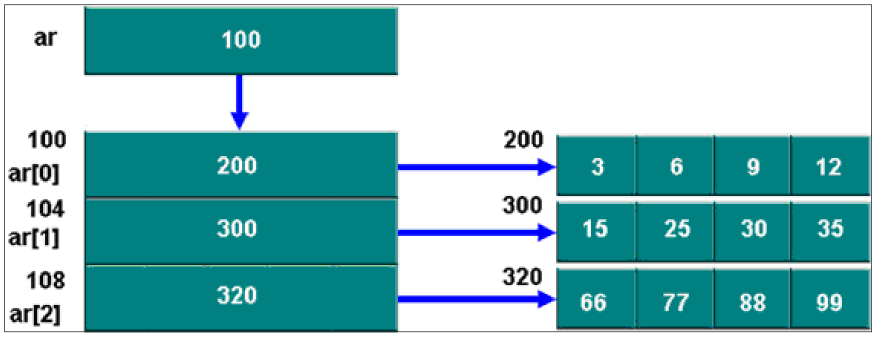
Consider another example. The expression `*(*(num+1))+1` is resolved to `*(*(104)+1`. in the next step, it becomes `*(300)+1`. the expression `*(300)` results in 15. therefore, `*(300)+1` results in 16.



It also possible to use pointer arithmetic to assign an address to a pointer. For example, the address stored in one pointer can be decremented and stored in one pointer can be decremented and stored in another pointer. In the partial code given below, the address in num2 is decremented by two and stored in num1.

Self Review Exercise 84.1

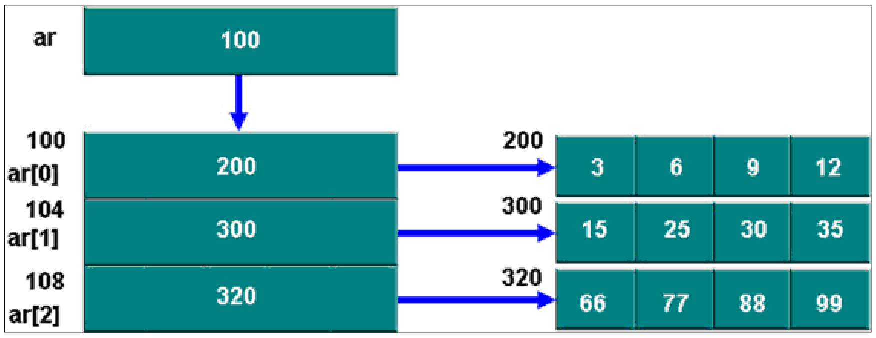
1. *An array of three pointers called ar is created and initialized. Choose the output of the expression ar+2.*



A. 108
B. 300
C. 200
D. 15

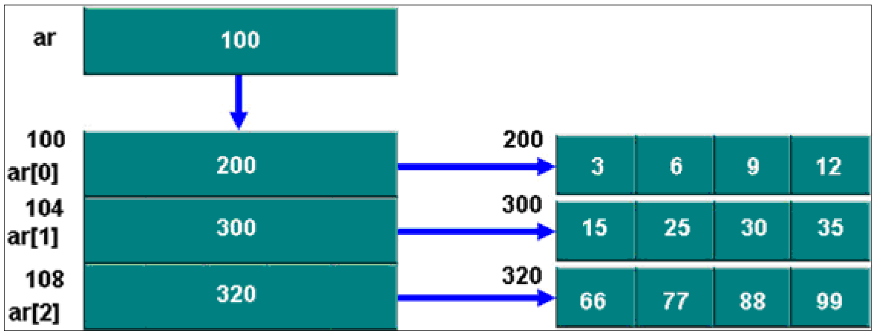
Self Review Exercise 84.2

2. *Choose the output of the expression *(ar+1).*



A. 108
B. 66
C. 300
D. 200

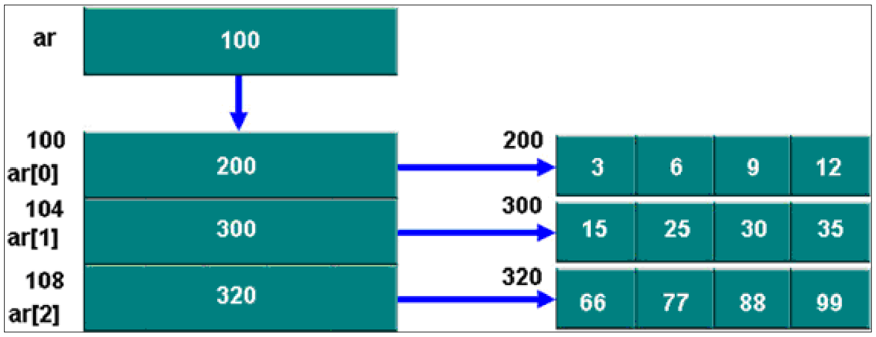
3. *Choose the output of the expression *(ar).*



A.108
B.300
C. 200
D. 66

Self Review Exercise 84.3

4. *Choose the output of the expression *(*(ar+2)*



A.66

B. 300

C. 200

D. 108

LESSON 85 : PASSING ARRAY OF POINTERS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

PASSING ARRAY OF POINTERS

A large function in a program can be divided into smaller functions. This improves the readability of the function. This also enables you to debug the function easily.

Although a function can call another function, the arrays of pointer that are declared within a function are accessible only within that function. If the array of pointers is to be accessed from another function, it can be passed as an argument to that function.

It is possible to pass a complete array of pointers or an elements of an array of pointers to a function when it is called.

Arrays of pointers are always passed by reference because the name of an array contains the address of the first element in the array.

To pass an array of pointers to a function, three statements are added to the code. These statements are used to call the function, declare the called function, and define the called function.

Passing an Array of Pointers

- Call a function.
- Declare the called function.
- Define the called function.

First, to call a function, the name of the function is specified. The name of the function is followed by the list of arguments. For each argument, the name of the argument that is passed is specified. An example that can be used to call a function named display by passing an array of pointers called array is given. In this example, the complete array is passed to the display function when it is called.

```
display(array);
```

You can declare a function before calling it from another function. In the second statement to pass an array of pointer to a function, the function is declared along with all the arguments that are passed to it.

The syntax to declare the called function is given below. The function name is preceded by the return type of the function. The name of the function is followed by the list of arguments, which is specified within parentheses.

A function can also be declared without specifying the variable names in the list of arguments. The data type, the * operator, and the brackets are retained in the syntax.

```
        return_type function_name(data_type *[array_size]);
```

An example to declare the display function without the variable name is given below.

```
        void display(char *[50]);
```

The number of elements specified within the brackets are also optional. Therefore, the `display` function can also be declared as below. The number of elements in the array of pointers is determined by the array that is passed to the display function when it is called.

```
        void display(char *[]);
```

When the called function is defined, the arguments of the function must be declared. The arguments are declared in the same way as they are declared during function declaration.

The various ways of defining the `display` function are given below. This function accepts an array of three pointers, which is called `arrptr,` as an argument.

```
        void display(char *arrptr[3])
        void display(char *arrptr[])
```

Instead of passing the complete array of pointers, it is also possible to pass an individual array element from an array of pointers. The function `disp` is called by passing the second string in the array of pointers named array.

```
        disp(array[1]);
```

To pass a string to the `disp` function, the function can be declared in three ways. The first declaration given below specifies the length of the string within brackets. The second and third declaration do not specify the length of the string within the brackets.

```
        void disp(char str[20]);
        void disp(char str[]);
        void disp(char[]);
```

The definition and declaration of the `disp` function are similar. The only difference is that a semicolon is not used while defining a function. The two ways of defining the `disp` function are given below.

```
        void disp(char str[20])
        void disp(char str[])
```

Self Review Exercise 85.1

1. *You learned the code to call a function by passing an array of pointers, which is called* `sales`*, is created. Choose the option that can be used to call the function named* `calctarget` *by passing the array sales.*

```
#include <stdio.h>
int a[3][4]={
             {3,6,9,12},
             {15,25,30,35},
             {66,77,88,99}
};
main()
{
int *sales[3];
/*Remaining part of the code is here*/
}
```

A. `calctarget(sales);`
B. `calctarget(sales[3]);`
C. `calctarget(int *sales[3]);`
D. `void calctarget{sales);`

2. *An array of* 15 *character pointers, which is called* `ar`*, is to be passed to the function called* `sortarray`*. The* `sortarray` *function returns an integer. Select the correct option to call the* `sortarray` *function.*

A. `x=sortarray(*ar[15]);`
B. `x=sortarray(char *ar[]);`
C. `x=sortarray9char ar[15]);`
D. `x=sortarray(ar);`

3. *An array of* 15 *character pointers, which is called* `ar`*, is to be passed to the function called* `sortarray`*. The* `sortarray` *function returns an integer. Select the correct option to define the* `sortarray` *function.*

A. `int sortarray(char *a[])`
B. `int sortarray(char *ar[]);`
C. `int sortarray(char [15][])`
D. `int sortarray(char ar[15);`

LESSON 86 : ALLOCATING MEMORY

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

ALLOCATING MEMORY

A structure pointer is a pointer that can store the address of a structure. A structure pointer is used for allocating memory dynamically and conserving memory. Memory can be allocated to a structure pointer according to the memory requirement of a structure.

When a structure pointer is created, it does not store a valid address. The pointer has to be initialized with a valid address after it is created.

Memory can be allocated for a structure pointer in three ways:

- by assigning the address of a structure variable to the structure pointer, using the `malloc` function
- by assigning the value of another structure pointer to it.

The method of assigning the address of a structure variable is used when a structure variable already exists in a program. In an example shown below, a structure called `employee` is created and a pointer to the structure `employee` is declared.

Listing 86.1 : using structure pointers

```
1   //Listing 86.1 : this program uses structure pointers
2   #include <stdio.h>
3
4   main()
5   {
6   struct employee
7   {
8       char name[25];
9       char addr[35];
10      char grade;
11  }emp;
12
13      struct employee *p;
14      p=&emp;
15
16  }
```

Detailed Explanation

- Notice the declaration of the structure `employee` in the program. It has three structure members: `name`, `addr`, and `grade`. The declaration of the variable `emp` results in contiguous memory being allocated for all three members of the structure.
- In the Line No `14` statement, the address of the variable `emp` is assigned to the pointer `p` by using the address.

➥ operator (&). After this assignment, the members of the structure can be accessed using either the variable emp or the pointer p.

Using malloc()

The other way to allocate memory for a structure pointer to use the malloc function. The malloc function can be used to allocate memory for a structure and store the address of the structure in a pointer.

Notice the code used to allocate memory for the structure employee by using the malloc function. The address of this structure is assigned to the pointer p.

Listing 86.2 : using malloc to allocate memory

```
//Listing 86.2 : allocating memory using malloc function
#include <stdlib.h>
#include <stdio.h>

main()
{

struct employee
{
    char name[25];
    char addr[35];
    char grade;
};

    struct employee *p;
    p=malloc(sizeof(struct employee));
}
```

Assign a value of another structure pointer

A structure pointer can also be initialized by assigning the address stored in another structure pointer to it. The example given assigns the address stored in the structure pointer named strucptr to the structure pointer named newstrucptr.

```
newstrucptr=strucptr;
```

An array of structure pointers can also be created. Each structure pointer in this array can store the address of a structure in memory. The code to allocate memory for all the pointers in an array of structure pointers is given. An array of 10 structure pointers called arrayptr is created in the code.

Listing 86.3 : working with an array of structure pointers

```
//Listing 86.3 : creating an array of structure pointers
#include <stdlib.h>
#include <stdio.h>
main()
{
    int count;
    struct employee
    {
            char name[25];
            char addr[35];
            char grade;
    };
    struct employee *arrayptr[10];
    for(count=0;count<10;count++)
            arrayptr[count]=malloc(sizeof(struct employee));
}
```

The Line 15 shows the use of a subscript with the pointer to refer to a pointer inside the loop. The value of count is incremented each time the loop is executed. Therefore, the addresses are stored in all the pointers in the array.

After the code is executed, all the pointers in the array of pointers are initialized as shown. Each structure can be accessed using the corresponding pointer.

Self Review Exercise 86.1

1. *You learned how to allocate memory to a structure pointer. Choose the correct option to assign the address of the variable* `sal` *to the pointer* `ptr` *now.*

```
#include <stdlib.h>
#include <stdio.h>
main()
{
       struct employee
       {
               char name[25];
               char addr[35];
char grade;
       }sal;
       struct employee *ptr;
/*Remaining part of the code is here*/
```

A. `&ptr=sal;`
B. `ptr=*sal;`
C. `ptr=&sal;`
D. `ptr=sal;`

2. *A pointer to a structure named voter is declared. Which is the correct option to allocate memory to the pointer name* `vtptr`*?*

A. `&vtptr=malloc(sizeof(struct voter));`
B. `*vtptr=malloc(sizeof(struct voter));`
C. `vtptr=malloc(struct voter);`
D. `vtptr=malloc(sizeof(struct voter));`

3. *An array of structure pointers,* `arrayvoter`*, of the struct voter type is declared. Which is the correct option to allocate memory to all the pointers in the array of pointers?*

A.
```
for(i=0;i<25;i++)
          arrayvoter[i=malloc(sizeof(struct voter));
```
B.
```
for(i=0;i<25;i++)
          arrayvoter[i]=malloc(struct voter);
```
C.
```
for(i=0;i<25;i++)
         *arrayvoter[i]=malloc(sizeof(struct voter));
```
D.
```
for(i=0;i<25;i++)
          arrayvoter=malloc(struct voter);
```

LESSON 87 : ACCESSING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The member variables of a structure can be accessed using a pointer. Member variables can be accessed separately in various situations in programs. An example of such a situation is a program that needs to accept or display data in each member variable separately.

To accept or display data in the member variables of a structure, standard input/output (I/O) functions are used. However, these functions do not accept a structure as an argument. Therefore, each member variable is accessed separately and used with the I/O functions.

There are two ways to access the member variables of a structure by using a pointer. Out of these, the more common technique is to use the member access operator (->) with the pointer name.

The -> operator is used with a pointer name to its left and the member variable name to its right. The -> operator belongs to the highest precedence group. Its associativity is from left to right.

An example that uses the -> operator for accessing a member variable is given below. The structure pointer is called sptr, and it is used to access the member variable called country.

```
sptr->country
```

The -> operator can be used in a statement that accesses a member variable. For example, the -> operator can be used for assigning values. The code that is shown assigns values to member variables by using a structure pointer.

Listing 87.1 : assigning values to member variables using structure pointer

```
//Listing 87.1 : this program assign values to member variables by using a structure pointer
#include <stdio.h>

main()
{

int count;

struct employee
{
char name[25];
int empid;
char grade;
}emp;

struct employee *arrayptr;
arrayptr=&emp;
arrayptr->empid=9128;
arrayptr->grade='A';

}
```

A structure can have a member that is an array of pointers. It is possible to access the elements of such an array by using a structure pointer. The pointer `ptr` is used to access the third element of the member variable called `name`.

```
putchar(ptr->name[2]);
```

The `->` operator can also be used with the member variables in mathematical expressions. An example is given below. The member variable is multiplied by `20`, and the result is assigned to a variable called `num`.

```
num=p->score*20;
```

The `->` operator can also be used to accept data into integer and character member variables. In the example given below, the pointer ptr is used to access the character member variable `ch` and the integer member variable num.

```
scanf("%c",&ptr->ch);
scanf("%d",&ptr->num);
```

The other method to access member variables by using a pointer is to use the `*` operator with the dot operator `(.)`. the . operator alone cannot be used with a pointer.

Parentheses are used in the expression that uses the . operator to set the precedence. If the parentheses are not used, the compiler reports an error because a pointer is not directly compatible with the . operator.

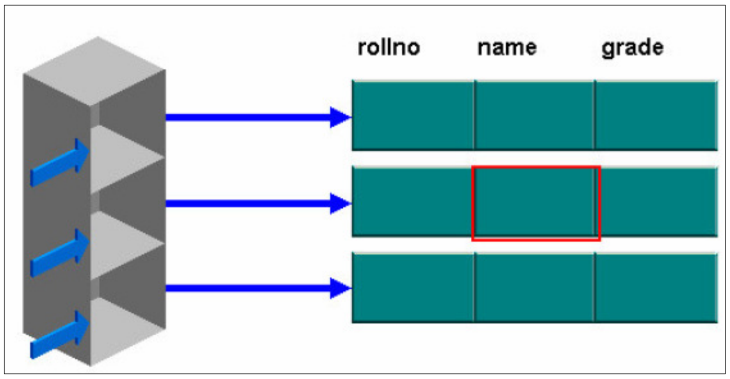
```
(*structure_pointer).member_variable
```

Notice the following examples that use the . operator and the `*` operator. The member variables name and address are accessed using the pointer `ptr`.

```
(*ptr).name
(*ptr).address
```

The `->` operator and the . operator can also be used to access member variables in an array of structure pointers..

A diagrammatic representation of an array of pointer to a structure is given below. The member variable of the second pointer in the array can be accessed separately.



Notice the syntax used to access a member variable by using a subscript with the pointer name. the value of `n` is replaces by the element number minus `1`.

```
ptr[n]->member_variable
```

An example to access the member variable grade of the second pointer in the array of pointer is displayed.

```
ptr[1]->grade
```

In the code below, data is to be accepted into the member variable `sname` of the third pointer in the array of structure pointers called `salesptr`.

Listing 87.1 : assigning values to member variables using structure pointers

```
//Listing 87.2 : assigning values to member variables using structure pointers
#include <stdlib.h>
#include <stdio.h>

main()
{

int num;

struct sales
{
char sname[25];
int scode;
int sgrade;
};

struct sales *salesptr[50];
for(num=0;num<50;num++)
salesptr[num]=malloc(sizeof(struct sales));
Gets(salesptr[2] -> sname);

```

Self Review Exercise 87.1

1. *You learned how to access structure members by using a pointer. The initials of each customer's last name are to be displayed. Which is the correct option to access the first character of the member variable* `lastname`*?*

   A. `sp->lastname`
   B. `*(sp->lastname)`
   C. `sp->lastname[1]`
   D. `&sp->lastname`

2. *A structure pointer named* `salesptr` *is created for the sales structure. Which is the correct option to access the character member called* `sname` *by using the structure pointer?*

   A. `(*salesptr).sname`
   B. `*salesptr->sname`
   C. `(&salesptr).sname`
   D. `&salesptr->sname`

LESSON 88 : PASSING STRUCTURE POINTERS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

In a program, it is advisable to have many small functions instead of one large function. This improves the readability of the program and makes the debugging process easier. Functions in a program can call each other to perform the required tasks. For example, a function can call another function to manipulate the values stored in a memory address.

A structure pointer created inside a function is local to the function that created the structure pointer. If another function is to access the structure pointer, the structure pointer is passed to the other function as an argument. When a structure pointer is passed to a called function, the complete structure can be accessed within the called function. Passing a structure pointer to a function always results in a call by reference.

There is a set of three statements that are added to the program that needs to pass a structure pointer to a function as an argument.

- The first is to declare the called function
- The second is to call the function, and the third is to define the called function.

The first step to pass a structure pointer is to declare the called function in the calling function. The syntax to declare the function is given below. The function name is preceded by the return type of the function.

```
return_type function_name(data_type structure_pointer);
```

The data type of the structure pointer that is to be passed to the function is supplied within the parentheses.

The called function can also be declared without the argument name as given below. This is because the function declaration statement requires the declaration of the argument type and not the argument. The argument is actually declared when the function is defined.

```
return_type function_name(data_type);
```

In the example, the pointer to the structure `employee` is to be passed to the function `accept`. The declaration of the function `accept` is given below. The `accept` function does not return any value.

```
void accept(struct employee *);
```

The function declaration uses the name of the structure. Therefore, the structure declaration must be global so that the function can refer to it.

The next statement to be added to the program to pass a structure pointer is the statement to call the function. This statement is added at the line in the program where control is to be shifted to the function to be called.

Notice the syntax for calling a function by passing a structure pointer. A structure pointer is passed to a function by specifying the structure pointer name as the argument.

```
function_name(structure_pointer_name);
```

Another way to pass a structure pointer is to specify the `&` operator with the structure variable when the function is called. The syntax for passing the address of a structure variable is given below.

```
function_name(&structure_variable_name);
```

The last step to pass a structure pointer to a function is to define the called function. The function definition includes the return type and the declarations for the arguments to the function.

In the example below, the `printptr` function is defined. The `printptr` function returns an integer and accepts the structure pointer `emp` as an argument.

```
int printptr(struct emp *)
```

An array of structure pointers can also be passed to a function while calling. An example that declares the function `readstruct` is given below. The function `readstruct` accepts an array of `15` structure pointers as an argument.

```
int readstruct(struct employee *[15]);
```

The `readstruct` function can also be declared as given below. In this declaration, the dimension of the array of structure pointers is not specified, the dimension of the array of structure pointers is determined when an array of structure pointers is passed to the function.

An example to call a function by passing an array of structure pointers is given. The function `readstruct` is called by specifying the function name and passing the structure pointer name as the argument.

```
readstruct(array);
```

An example that calls a function by passing an array of structure pointers is given. The function read`struct` is called by passing the array of structure pointers named `sptr`.

```
readstruct(sptr);
```

The definition of the function `readstruct` is given. The function `readstruct` returns a pointer to a character and declares the structure pointer `argptr` as an argument.

```
Char * readstruct(struct employee *argptr)
```

A partial code sample that finds a customer record corresponding to a specified account number is given. A pointer to the structure `cust` name `custptr` is declared.

LESSON 88 : PASSING STRUCTURE POINTERS

Listing 88.1 : passing structure pointers to functions

```
// Listing 88.1 : this program deals with passing structure pointers to functions
#include <stdio.h>
#include <stdlib.h>
struct cust {
    char *nem;
    int accno;
    char acctype;
    float balance;};
Struct cust *search(struct cust *table[3], int acctn);
main()
{
struct cust customer[3]={{“Debbie”,3333,’C’,33.33},
                    {“Ronald”,6666,’O’,66.66},
                    {“Chris”,9999,’D’,99.99}};
int acctn=3;
struct cust *pt, *custptr[3];
custptr[0]=&customer[0];
custptr[1]=&customer[1];
custptr[2]=&customer[2];
while(acctn!=0)
{   printf(“\nCustomer Account Locator”);
    printf(“\nTo end search, type 0 for account number \n Account No. :”);
    scanf(“%d”, &acctn);
    if(acctn==0)
            exit(0);
    pt=search(custptr,acctn);
struct cust *search(struct cust *table[3],int acctn)
{
    int count;
    for(count=0;count<3;count++)
    if(table[count]->accno==acctn)
    return table[count];
    return 0;
}
```

Detailed Explanation

- The search function is declared at the beginning of the main function along with the other declarations. The search function returns a pointer to the structure cust.
- The scanf function is called for accepting the account number of the employee whose data is to be displayed. The variable acctn is declared as an integer variable.
- The structure pointer called custptr and the account number to be searched are passed to the search function as arguments. The search function by passing the arguments is called search (custptr, acctn);.
- The search function returns a pointer to the structure cust if the account number is found. The address returned by the search function is stored in the pointer pt. If pt is not NULL, customer details are displayed.

Self Review Exercise 88.1

1. *A structure pointer of struct* `sale type`, `sales`, *is created. The* `calc` *function, which does not return any value, is to be called by passing the* `sales` *pointer to it. Which is the correct option for declaring the* `calc` *function?*

   A. `void calc(sales);`
   B. `void calc(struct sale*);`
   C. `calc(sales);`
   D. `struct sale *calc(void);`

2. *A structure pointer called* `sales` *is created. Which is the correct option for calling the function named* `calc` *by passing the structure pointer?*

   A. `calc(sales);`
   B. `calc(struct sales);`
   C. `calc(struct sale sales);`
   D. `void calc(sales);`

3. *An array of* 15 *structure pointers called* `ar` *is created to point to the structure* `dat`. *The array is to be passed to the* `srt` *function. The* `srt` *function returns an integer. Which is the correct option to declare the* `srt` *function?*

   A. `int srt(struct dat[][]);`
   B. `int srt(struct dat a[])`
   C. `int srt(struct dat *[15]);`
   D. `int srt(struct dat [15]);`

4. *An array of* 15 *structure pointers called* `ar` *is created to point to the structure* `dat`. *The array is to be passed to the* `srt` *function. The* `srt` *function returns an integer. Which is the correct option to call the* `srt` *function?*

   A. `i=srt(*ar[15]);`
   B. `i=srt(*ar);`
   C. `i=srt(struct data r[15]);`
   D. `i=srt(ar);`

5. *An array of* 15 *structure pointers called* `ar` *is created to point to the structure* `dat`. *The array is to be passed to the function called* `srt`. *The* `srt` *function returns an integer. Which is the correct option for defining the* `srt` *function?*

   A. `int srt(struct dat ar[][])`
   B. `int srt(struct dat *a[])`
   C. `int srt(struct dat *[]);`
   D. `int srt(struct dat ar[15])`

LESSON 89 : POINTER MEMBERS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A structure can have a member variable that is a pointer and can store the address of another variable. When memory is to be allocated dynamically for a member variable of a structure it is declared as a pointer.

When memory is allocated for the structure, the pointer is created in memory. However, the pointer does not store any address when it is created.

To store an address in the pointer, the `malloc` function can be called. The pointer member can be initialized with the address of the memory that is allocated using the `malloc` function.

Listing 89.1 : implementing pointer usage inside a structure

```
// Listing 89.1 : this program uses pointers inside a structure
#include <stdio.h>
main()
{
    struct emp
    {
            char  *Ename;
            int  Ecode;
            char Egrade;
    };
    struct emp empdat;
    empdat.Ename=malloc(50);
}
```

When a pointer member is used, memory can be allocated according to the requirement of the program. Using a pointer member instead of a static array helps in saving memory. A static array has a fixed size that cannot be altered according to the memory requirement.

A part of the code that uses a pointer member in a structure is given below. The code accepts data for customers. A structure `cust` is created, which contains a pointer member to store the names of customers.

Listing 89.2 : implementing pointer usage inside a structure

```
// Listing 89.2 : implementing pointer usage inside a structure
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
main()
{
    int count=0;
    struct cust
    {
            char  *fname;
            char addr[35];
            char grade;
    };
    struct cust *sptr[100];
    char resp='y';
    char fn[40];
    while(resp=='y')
    {
            Sptr[count]=malloc(sizeof(struct cust));
            Printf("Type name :");
            Scanf("%s",fn);
            sptr[count]->fname=malloc(strlen(fn));
            Strcpy(sptr[count]->fname,fn);
            /*Remaining part of the code is here*/
    }
}
```

Detailed Explanation

- The structure is to be accessed using the array of pointers called `sptr`. A loop is created to accept customer data. The loop is executed depending on the number of customers whose data is to be entered.
- The `malloc` function is called to reserve memory for the first element in the array of structure pointers. To allocate memory enter `malloc(sizeof(struct cust());`.
- You completed the code to allocate memory for an element of the array of pointers inside the loop.
- Memory is allocated according to the length of the name that is entered using the `strlen` function with the `malloc` function. To allocate the required number of bytes, enter `sptr[count]->fname=malloc(strlen(fn));`.
- You entered the statement to allocate memory for the pointer member `fname`.
- After you enter the statement to allocate memory for the pointer member `fname`, the data from the string fn is copied to the pointer member `fname`.

A pointer member enables you to save memory by allocating memory according to the requirement of the program.

LESSON 90 : POINTER TO A POINTER

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A pointer that stores the address of another pointer is called a pointer to a pointer. A pointer itself is a variable. Therefore, the address of a pointer can also be stored in a pointer. To use a pointer to a pointer in a program, three statements are added to a program. The first two statements are used to create and initialize a pointer to a pointer. The third statement is used to access a variable by using a pointer to a pointer.

The syntax to create a pointer to a pointer is given below. The pointer to a pointer can store the address of a pointer that has the same data type as the pointer to a pointer.

```
data_type **<pointer_to_pointer>;
```

The syntax for declaring a pointer to a pointer is given. The pointer to a pointer that is given can store the address of an integer type pointer.

```
int **ptr;
```

After a pointer to a pointer is created, it has to be initialized with the address of a pointer. Notice the use of the `&` operator to initialize a pointer to a pointer is as follows.

```
pptr = &ptr;
```

Consider an example. A pointer named `p` stores the address of the variable `i`. A pointer to a pointer named `ptr_to_ptr` stores the address of the pointer `p`. The value of the variable `i` can be accessed using the variable name `i`, the pointer name `p` or the pointer to a pointer.

```
printf("%d",i);
printf("%d",*p);
printf("%d",**ptr_to_ptr);
```

The `**` operator can also be used to access the value of a variable by using a pointer to a pointer. An example that accesses a variable by using a pointer to a pointer is given in the third line.

A program in which a pointer that stores the address of a pointer to an integer for a displaying the value of the variable called number is given. The program creates an integer type pointer to a pointer called `pptr` and initializes it with the address of the pointer `ptr`.

- The ** operator is used to display the value of the variable named number.

Consider another example. This example uses a character type pointer. A character type pointer, `charptr`, is declared to store the address of the array called `studname`. A pointer to a pointer, `ptr`, is created to point to the pointer `charptr`.

Listing 90.2 : A pointer taking care of another

```
// Listing 90.2 : Accessing a string of characters via another pointer pointing to a
// pointer
#include <stdio.h>
char studname[]=”string”;

main()
{
    char *charptr=studname;
    char **ptr;
    ptr=&charptr;
    printf(“%c”,**ptr);
    printf(“%s”,*ptr);
}
```

Detailed Explanation

- In the example , the ** operator refers to the first character of the array called `studname`. The ** operator can be used in the example because `*ptr` refers to the address of `studname`.
- To access a complete string by using the pointer to a pointer, the * operator is used. The expression `*ptr` refers to the address of the first character of the string `studname`. Therefore, the expression `*ptr` is used to display the string.

Self Review Exercise 90.1

1. *A pointer to a pointer named* `ptrtoptr` *is to used in a program. The address of an integer pointer named* `ptr` *is to be stored in* `ptrtoptr`*. Which is the correct option to create and initialize* `ptrtoptr`*?*

   A. `int **ptrtoptr; ptrtoptr=ptr;`
   B. `int *ptrtoptr; ptrtoptr=&ptr;`
   C. `int **ptrtoptr; ptrtoptr=*ptr;`
   D. `int **ptrtoptr; ptrtoptr=&ptr;`

LESSON 91 : POINTER TO A FUNCTION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Pointers to functions provide another method of calling functions in addition to using the function name. A pointer to a function provides a more flexible method of calling function than by using the function name.

Pointers to functions can also be used to make generalized functions. a generalized function can be used to handle various tasks of the same type. For example, a menu-handling function that can display any menu is a generalized function.

A pointer can store an address, which does not have to be the address of a variable. The address can be any address in memory, such as the address of a function. When a program runs, the code for each function is loaded in memory at the specific address.

A pointer that stores the starting address of a function in memory is called a pointer to a function. The starting address of a function can be referred to by using the name of the function.

To implement a pointer to a function, three statements are added to the program.

- The first is to declare the pointer to a function.
- The second is to initialize the pointer to a function.
- The third is to call a function by using the pointer to a function.

The syntax to declare a pointer to a function is as follows.

```
return_type(*function_pointer_name)(list_of_arguments);
```

The pointer name is preceded by the return type of the function. The pointer name is followed by the list of arguments specified in parentheses.

Certain examples of declarations of a pointer to a function are given. The first declaration declares a pointer to a function called `func1`. The `func1` pointer can store the address of a function that returns an integer and accepts an integer as an argument.

```
int (*func1)(int x);
```

In the `func1` declaration, parentheses are used with the function name. This is because without the parentheses, this statement declares a function `func1`, which returns an integer pointer and accepts an integer as an argument.

```
void(*func2)(double y, double z);
```

The second declaration a pointer called `func2`. This pointer can store the address of a function that does not return any value and accepts two double variables as the arguments.

```
void (*func3)();
```

The third declaration declares a pointer called `func3`. This pointer is used to store the address of a function that has no return value and does not accept any argument.

The pointer to a function does not contain any address at the time of declaration. It has to be initialized by assigning the address of a function to it.

The syntax to initialize a pointer to a function is given.

```
function_pointer_name=function_name;
```

The address of a function can be stored in the function pointer. The only requirement is that the return type and argument list of the function must match the return type and argument list of the pointer to a function.

Consider the example below. In the example, the `square` function and the pointer to a function named `p` are declared. The declarations are followed by the definition of the `square` function.

Listing 91.1 : Pointing to a function

```
1   // Listing 91.1 : A pointer pointing to a function
2   #include <stdio.h>
3   float square(float x);
4   main()
5   {
6       float x,answer;
7       float (*p)(float x);
8       p=square;
9       printf("Type a number :");
10      scanf("%f",&x);
11      fflush(stdin);
12      answer=p(x);
13  }
14
15  float square(float x)
16  {
17      return x*x;
18  }
```

Detailed Explanation

- The pointer to a function named `p` can be initialized using the statement in Line `7`. The initialization can be done because the square function and `p` have the same return type and argument list.
- After the pointer to a function is initialized, it can be used to call a function. The function that is called is the function that was used to initialize the pointer to a function.
- Notice the statement to call a function by using the pointer to a function. In the statement in Line `11`, the argument named `x` is passed to the `square` function that is called.

- The complete program to call the `square` function by using the pointer to a function p is given. The statements for declaring, initializing, and calling the `square` function by using the pointer to a function are Lines `6`, `7`, and `11`.

Another program that uses a pointer to a function is as follows. In the part of the program given below, a generalized menu-handling function is created. The generalized menu-handling function called `menu` is given.

Listing 91.2 : passing a pointer as an argument

```
// Listing 91.2 : This program passes a void pointer to a function
#include <stdio.h>
char menu(void (*fptr)(), char nc)
{
    char ch=' ';
    fptr();
    printf("\nType choice");
    do
    {
            ch=getchar();
    }while(ch<'0'||ch>nc);
    return ch;
}
```

- The `menu` function accepts two arguments. The first argument is a pointer to a function, and the second argument is the maximum number of options in the `menu`. The `menu` function is used to display a function that is passed as the first argument.

As an example, two `menu` functions are created. The `menumain` function displays a main menu, and the `menuadd` function displays a menu for adding a single character or multiple characters.

Listing 91.3 : working with functions

```
// Listing 91.3 : a simple program working with functions
void menumain()
{
printf("\nMain Menu\n");
printf("\n1. Display char \n2. Add char \n3. Exit");
}
void menuadd()
{
printf("\nAdd Menu\n");
printf("\n1. Single char \n2. Multiple chars");
printf("\n3. Exit");
}
```

In the next example, the main function integrates all three functions to generate the two menus. The `menu` function, which is the generalized menu-handling function, is called by passing the function name `menumain` as an argument.

Listing 91.3 : pointing to a function

```
// Listing 91.3 : A pointer pointing to a function
#include <stdio.h>
char menu(void (*fptr)(), char nc)
{
    char ch=' ';
    fptr();
    printf("\nType choice");
    do
    {
            ch=getchar();
    }while(ch<'0'||ch>nc);
    return ch;
}
main()
{
    char option, menu();
    void menumain(), menuadd();
    while((option=menu(menumain, '3'))!='3')
    {
           switch(option)
           {
               case '1' : putchar('*'); break;
               case '2' : while((option=menu(menuadd, '3'))!='3')
                              {
                              switch(option)
                              {
                                      case '1' : puts("Single char function");
                                                    break;
                                      case '2' : puts("Multiple char function");
                                                    break;
                              }
                          }break;
            }
    }
}
```

Detailed Explanation

- The menu function is supplied with the function name and the number of commands in the menu called `menumain`. To call the `menu` function by passing the function name `menumain`, enter `menu(menumain,'3'))`.
- You entered the statement to call the `menu` function.
- The `menu` function displays the various commands in the menu `menumain`. A valid option is returned by the `menu` function. Depending on the command selected by the user, the `main` function displays a character or calls the `menuadd` function.
- If the user selects the second command from `menumain`, the `menuadd` function is to be called. This is done by calling the `menu` function by passing the function name as `menuadd` and specifying the number of commands in the `menuadd` menu.
- The option selected by the user from the `menuadd` menu is stored in the variable option. Depending on the option selected by the user, the remaining part of the code is executed.

Self Review Exercise 91.1

1. *The pointer to a function called* `funptr` *is to be created to store the address of the* `trimspace` *function. The* `trimspace` *function does not return any value and accepts a string as an argument. Which is the correct option to declare* `funptr`*?*

   A. `void(*funptr)(char *str);`
   B. `void *(funptr)(char *str);`
   C. `void (funptr)(char *str);`
   D. `void (*funptr)();`

2. *The pointer to a function called* `funptr` *is to be created to store the address of the* `trimspace` *function. The* `trimspace` *function does not return any value and accepts a string as an argument. Which is the correct option to initialize* `funptr`*?*

   A. `funptr=trimspace(str);`
   B. `funptr=&trimspace;`
   C. `funptr=trimspace;`
   D. `*funptr=trimspace(char *str);`

REVIEW EXERCISE

1. Identify the code that can be used to allocate memory for the elements of an integer array of 25 *pointers called array by using the* malloc *function.*

```
A. for(num=0;num<25;num++) array[num]=malloc((int));
B. for(num=0;num<25;num++) array[num]=malloc(1);
C. for(num=0;num<25;num++) array=malloc(sizeof(int));
D. for(num=0;num<25;num++) array[num]=malloc(sizeof(int));
```

2. A segment of code is given. The array is to be passed to the function called disp()*. The* disp() *function does not return anything. Choose the option for declaring the* disp() *function, calling the* disp() *function, and defining the* disp() *function.*

```
#include<stdlib.h>
#include<stdio.h>
struct emp{
               char name[25];
               char addr[35];
               int age;
             }*arrptr[10];
main()
{
       int i;
       for(i = 0;i < 10; i++)
               arrptr[i] = malloc(sizeof(struct emp));
```

```
A.  void disp(struct emp [][]);          B.  void disp(struct emp *[]);
    disp(*emp[10]);                          disp(arrptr);
    void disp(struct emp ar[][])             void disp(struct emp *a[])

C.  void disp(struct emp *[10]);         D.  void disp(struct emp [10]);
    disp(struct emp arrptr[10]);             disp(arrptr);
    void disp(struct emp *[10])              void disp(struct emp ar[10])
```

3. Identify the correct option to access the members grade *for the sixth student in the structure. An array of structure pointers named* studentptr *is created.*

```
A. studentptr[5]->grade (*studentptr[5]).grade
B. &studentptr[5]->grade  (*studentptr[5]).grade
C. &studentptr[6]->grade (*studentptr[6]).grade
D. studentptr[6]->grade *(studentptr[6]).grade
```

4. Match the expressions to access the elements of an array of five pointers called arr *with their outputs. Match the expression label to the appropriate output label. Each item is used only once.*

```
#include<stdlib.h>
#include<stdio.h>
main()
{
       int num, col;
       char *arr[5];
       for(num = 0;num < 10; num++)
               arr[num] = malloc(6);
       arr[0] = “one”;
       arr[1] = “two”;
       arr[2] = “three”;
       arr[3] = “four”;
       arr[4] = “five”;
/*Remaining part of the code is here*/
}
```

A.	arr[2]	A.	‘o’
B.	*arr	B.	‘w’
C.	arr[1][1]	C.	“one”
D.	*(*arr)	D.	“three”

REVIEW EXERCISE

5. A segment of code is given. The `street` *is entered in a variable named* `temp`*. Memory is to be allocated for* `street` *depending on the length of* `temp` *and its terminating null. Write the code to allocate memory to the pointer member* `street` *by using the -> operator.*

```
#include<stdlib.h>
#include<stdio.h>
        struct account
        {
                int accno;
                char *name;
                char *street;
                int balance;
        };
main()
{
        int i = 0;
        char temp[100];
        struct account p[200];
        for(i = 0;i < 200; i++)
        {
                printf("\n Type street for vendor number %d :", i + 1);
                gets(temp);
                fflush(stdin);
                //
```

6. An array of 10 *character pointers, which is called* `ptr`*, is to be passed to the function called* `search()`*. The* `search()` *function returns an integer. Choose the option for calling , declaring and defining the* `search()` *function.*

A.
```
search(*ptr[10]);
int search(char [][]);
int search(char arr[][])
```

B.
```
search(ptr);
int search(char *[]);
int search(char *a[])
```

C.
```
search(ptr[10]);
int search(char *[10]);
int search(char *a[10])
```

D.
```
search(ptr);
int search(char [10]);
int search(char a[10])
```

7. Choose the code sample that can be used to allocate memory to the pointer `acptr`*.*

A.
```
struct account
    {
            int accno;
            char *name;
            char *street;
            int balance;
    }*acptr;
acptr = malloc(struct account);
```

B.
```
struct account
    {
            int accno;
            char *name;
            char *street;
            int balance;
    }*acptr;
*acptr = malloc(sizeof(struct account));
```

C.
```
struct account
    {
            int accno;
            char *name;
            char *street;
            int balance;
    }*acptr;
&acptr = malloc(struct account);
```

D.
```
struct account
    {
            int accno;
            char *name;
            char *street;
            int balance;
    }*acptr;
acptr = malloc(sizeof(struct account));
```

7. Choose the code sample that can be used to allocate memory to all the pointers in the array of pointers `arptr`.

A.
```
struct account
{
    int accno;
    char *name;
    char *street;
    int balance;
};
struct account *arptr[50];
for(i = 0;i < 50; i++)
*arptr = malloc(sizeof(struct account));
```

B.
```
struct account
    {
            int accno;
            char *name;
            char *street;
            int balance;
    };
struct account *arptr[50];
for(i = 0;i < 50; i++)
    arptr[i] = malloc(struct account);
```

C.
```
struct account
    {
            int accno;
            char *name;
            char *street;
            int balance;
    };
struct account *arptr[50];
for(i = 0;i < 50; i++)
arptr[i] = malloc(sizeof(struct account));
```

D.
```
struct account
    {
            int accno;
            char *name;
            char *street;
            int balance;
    };
struct account *arptr[50];
for(i = 0;i < 50; i++)
    &arptr[i] = malloc(struct account);
```

8. A pointer to a pointer `ptop` *is to be declared. The address of the pointer* `p` *is to be assigned to* `ptop`. *The contents of the variable* `var` *are to be displayed using the pointer to a pointer. Choose the correct option to implement the pointer to a pointer.*

```
#include<stdio.h>
char name[] = "string";
main()
{
        int *p;
int var = 100;
        p = &var;
/*Remaining part of the code is here*/
}
```

A.
```
int **ptop;
ptop = &p;
printf("%d", **ptop);
```

B.
```
int **ptop;
ptop = &p;
printf("%d", *ptop);
```

C.
```
int **ptop;
ptop = *p;
printf("%d", *ptop);
```

D.
```
int *ptop;
ptop = &p;
printf("%d", **ptop);
```

9. Match the arithmetic expressions involving an array of pointers with their outputs.

```
int num[3][4] = {{3, 6, 9, 12}, {15, 25, 30, 35}, {66, 77, 88, 99}};
int *arr[3];
arr[0] = num[0];
arr[1] = num[1];
arr[2] = num[2];
```

A.	`*(*arr + 1)`		*A.*	26
B.	`*(*(arr + 1)`		*B.*	26
C.	`*(*(arr + 1) + 1)`		*C.*	6
D.	`*(*(arr + 1) + 1) + 1`		*D.*	15

10. A pointer to a function named `fp` *is to be declared to call the* `dat1` *function. This function returns a* `float` *pointer and accepts one integer as an argument. Choose the correct option for declaring, initializing and calling the* `dat1` *function by using* `fp`.

A.
```
float *(fp) (int x);
fp = &dat1;
float *fp(int x);
```

B.
```
float *(*fp) (int);
fp = dat1;
(*fp)(x);
```

C.
```
float (*fp) (int x);
fp = &dat1(int x);
fp(int x);
```

D.
```
float (fp) (int x);
fp = &dat1(int x);
*fp(int x);
```

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The most common use of the C language is for writing the code for operating systems, such as UNIX and LINUX, compilers, editors, and database management systems. Compilers and operating system utilities require the direct manipulation of bits in memory. Since C is a middle-level language, it supports the direct manipulation of bits, bytes, and memory addresses.

BINARY FACTS

- A bit can be defined as the basic and smallest unit of computer data storage. A bit can have only two values, 0 or 1. In a binary number system, an array of bits can represent numeric values.
- The binary number system has the base 2. An example of a binary number is 1100 1010. This is an 8-bit binary number. A set of 8 bits forms a byte.
- Bytes are the smallest addressable units of memory. Computer memory is measured in bytes. For example, phrases such as 32 kilobytes (KB) of RAM or 1 gigabyte (GB) of hard disk space represent the number of bytes that can be stored on a system.
- The amount of memory that can be processed by the CPU in a single operation is known as a word. Depending on the hardware configuration of a system, a word can have 1 byte or 2, 4, 8, or more bytes.
- Besides the hardware configuration, the addressing system used also determines the word size. For example, the word length is 2 bytes on a 16-bit operating system and 4 bytes on a 32-bit operating system.
- To manipulate bits directly through a C program, you should be familiar with the method for converting a binary number to a decimal number. This will help you verify results when you run the program for tracking logical and syntactical errors.
- To convert a binary number to its decimal equivalent, you start multiplying the bits of the binary number by increasing powers of 2. You start from the bit on the extreme right and multiply it by 2 to the power of 0, which is equal to 1.
- To convert a binary number 1010 to a decimal number, you first multiply the rightmost bit by 2 to the power of 0. Next, you multiply the second bit from the right by 2 to the power of 1. This process continues until you reach the leftmost bit.
- Finally, you add the resultant values to obtain the decimal equivalent While converting a binary number to its decimal equivalent, the bits are multiplied by an increasing power of 2. The power of 2 is incremented by 1.

LESSON 92 : BINARY TO DECIMAL CONVERSION

```
Binary Number:   0000 1010

Conversion Method

First Strep:

0*2⁰

                                                                        1*2¹ + 0*2⁰

                                                                0*2² + 1 *2¹ + 0*2⁰

                                                                               …..

                                                                               …..

                                        0 *2⁷ + 0*2⁶+   0*2⁵ + 0 *2⁴ + 1*2³+   0*2² + 1 *2¹ + 0*2⁰

                          0    +    0   +     0    +    0    +   8   +     0     +    2    +      0

Decimal Equivalent : 10
```

Like in the decimal number system, in the binary number system, the zeroes added to the higher bits do not increase the value of the binary number system and can be safely ignored. For example, `0000 0101` both represent the decimal number `5`.

Self Review Exercise 92.1

1. *Identify the method for converting the binary number 1101 to its decimal equivalent.*

   A. $1*2^3+0*2^2+1*2^1+1*2^0$
   B. $1*2^4+0*2^3+1*2^2+1*2^1$
   C. $1*2^3+1*2^2+0*2^1+1*2^0$
   D. $1*2^4+1*2^3+0*2^2+1*2^1$

2. *Identify the decimal equivalent of the binary number 1101.*

   A. 11
   B. 13
   C. 22
   D. 26

3. *Identify the decimal equivalent of the binary number 11101.*

   A. 58
   B. 23
   C. 29
   D. 21

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

While developing applications in C, you declare variables to store the values entered by the user. For example, you may declare an integer to store only the values that range from `0` to `3`.

Declaring an integer variable to store the values that range from `0` through `3` leads to a waste of memory space. This is because an integer occupies a minimum of `2` bytes whereas a memory space of only `2` bits is sufficient to store values if bit fields are used.

A bit field is a variable that has a storage space of less than a byte. Bit fields are defined as the structure members that contain only the specified number of bits.

When bit fields are used to accept data, more than one data item can be packed in one word of memory. Each bit of the word can then be accessed as a structure member. A word is defined as the amount of memory that can be processed by the CPU in a single operation.

The general syntax for declaring a bit field is given. The type specifier specifies the bit field type, such as `unsigned` and `int`. Most compilers have bit fields of the type `unsigned` integer, but there are some compilers that support signed bit fields.

```
struct <struct_name>
{
unsigned <field_name> : n;
…
};
```

While using bit fields in an application, the use of unsigned bit fields should be preferred. This is because signed bit fields need more bits to store the same information. One extra bit, which is the signed bit, is required to indicate positive or negative numbers.

After specifying the bit field type, you specify the name of the bit field. Finally, you specify the number of bits for the bit field. The number of bits is always preceded by a colon. The number of bits occupied by a bit field is its width.

A bit field declaration structure stores the data of birth in the `mmddyy` format. The structure occupies one word instead of the three words that would be required if three integers were used to store the `month`, the `day`, and the `year`.

Listing 93.1 : Dealing with bit fields

```
//Listing 93.1 : A program dealing with bit fields

#include <stdio.h>

main()
{

int m,d,y,i=0;

typedef struct
{

unsigned month : 4;
unsigned day : 5;
unsigned year : 7;

}date;

struct
{

char name{15];
date bday;

}emp[2];

for(i=0;i<2;i++)
{

printf("Type the Employee Name: \n");
scanf("%s", emp[i].name);
printf("Type the date of birth in mm dd yy format:\n");
scanf("%d %d %d", &m, &d, &y);
emp[i].bday.month=m;
emp[i].bday.day=d;
emp[i].bday.year=y;

}

for(i=0;i<2;i++)
{

printf("\nEmployee Name: %s", emp[i].name);
printf("\nBirth day:%d/%d/19%d", emp[i].bday.month, emp[i].bday.day, emp[i].bday.year);

}

}
```

Detailed Explanation

- The width of a bit field depends on the expected range of values that are to be stored in it. For example, the values that can be stored in the `month` field range from `1` through `12`. Since the binary representation of `12` is `1100`, only `4` bits are reserved for the `month` field.
- The values in the `day` field like those in the `month` field range from `1` through `31`, only `5` bits are required. Therefore, the bit field named day is assigned `5` bits. Similarly, to represent the year, `7` bits are needed.
- The bit fields can be accessed just like you access any other member of a structure, such as an integer member or a character member. For example, to access the year from the variable `bday`, which is of the type `date`, you use the notation `bday.year`.

LESSON 93 : BIT FIELDS

You learned the syntax for declaring a bit field. Add a bit field declaration to the structure called `sample`. The `unsigned` field is called `headoffice` and is set to `1` if the employee is posted at the `headoffice`. Otherwise, the bit field is set to `0`.

Listing 93.2 : Adding bit fields to a structure

```c
// Listing 93.2 : A program to add a bit field to a structure

#include <stdio.h>

main()
{
struct
{
   unsigned a : 1;
}sample;
}
```

- The unsigned bit field `headoffice` will have a width of `1` because `headoffice` can have either a value `0` or `1`.

Listing 93.3 : Adding a bit field to a structure

```c
// Listing 93.3 : A program to add a bit field to a structure

#include <stdio.h>

main()
{
struct
{
   unsigned a : 1;
   Unsigned headoffice : 1;

}sample;
}
```

There are some guidelines for using bit fields in a structure. Bit fields that occupy `8` bits or `16` bits should not be used. In such cases, the space occupied by the bit field is equal to `1` or more bytes. Since memory space is not saved, it is better to use other data types.

Bit Fields that occupy Less than 8 Bits

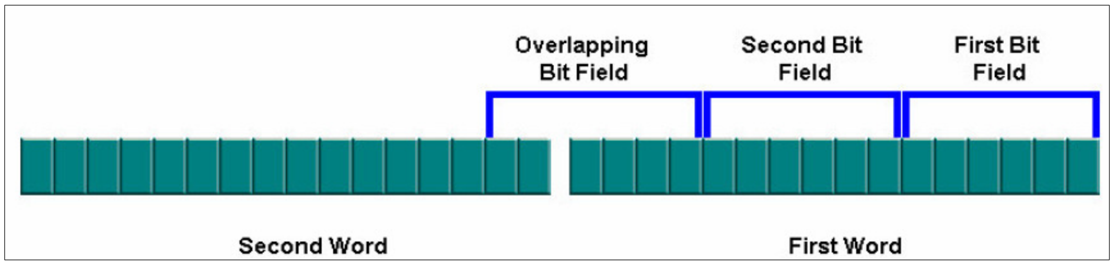
```c
#include <stdio.h>
main()
{
struct abc
{
  unsigned a : 5;
  unsigned b : 3;
  unsigned c : 3;
  unsigned d : 7;
}x;
}
```

Bit Fields that occupy 8 or 16 Bits

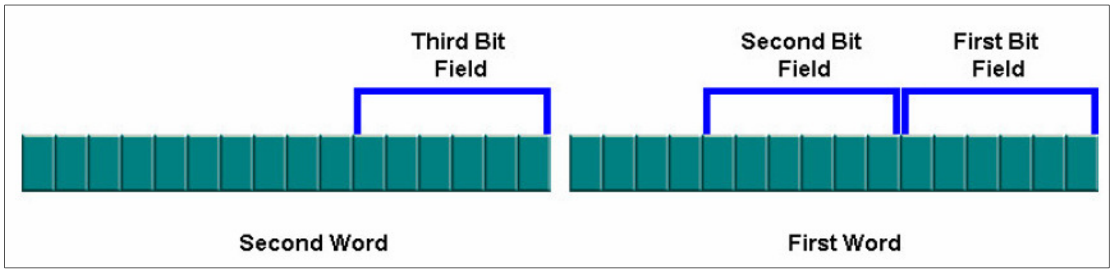
```c
#include <stdio.h>
main()
{
struct abc
{
  unsigned a : 5;
  unsigned b : 3;
  unsigned c : 3;
  unsigned d : 8;
}x;
}
```

```c
#include <stdio.h>
main()
{
struct abc
{
  unsigned a : 8;
  unsigned b : 7;
  unsigned c : 16;
  unsigned d : 5;
}x;
}
```

On a 16-bit machine, the total number of bits used in a structure is greater than 16. In this case, two words are required for the structure declaration. One of the bit fields starts from the first word and ends at the second word.

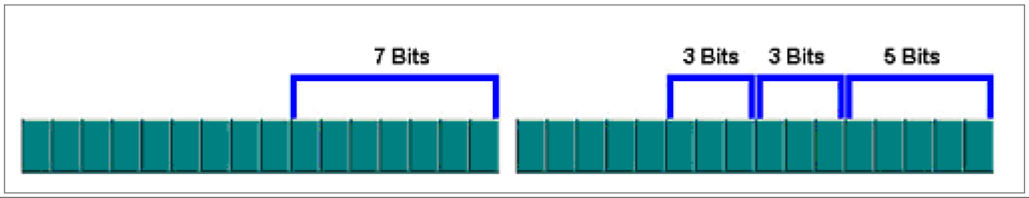


In cases where the bit field starts from one word and ends at the second word, the overlapping bit field is forced to start from the second word.



For example, the bit field d that has a width of 7 in the code sample displayed should start from the twelfth bit and end at the second bit of the second word. To avoid this overlap, the bit field d starts from the first bit of the second word instead of the twelfth bit of the first word.

```
#include <stdio.h>
main()
{
struct abc
{
  unsigned a : 5;
  unsigned b : 3;
  unsigned c : 3;
  unsigned d : 7;
}x;
}
```



You can also control the alignment of the bit fields within memory by using unnamed bit fields. The width of the unnamed bit field is used to fill the bits between two bit fields within a word with zeroes.

For example, if the first 2 bit fields in a bit field structure occupy 12 bits and you want the third bit field to start from the second word, then you declare an unnamed bit field that has a width of 4. As a result, the remaining 4 bits are filled with zeroes.

Listing 93.4 : working with bit fields

```
// Listing 93.4 : working with bit fields
#include <stdio.h>

main()
{

struct abc
{

  unsigned a : 5;
  unsigned b : 7;
  unsigned   : 4;
  unsigned d : 5;

}x;

}
```

A structure declaration with bit field members is given below. To provide a padding of 3 bits between the bit fields a and b, an unnamed bit field of a width of 3 is declared in the structure. Like other bit fields, unnamed bit fields cannot be referenced.

Listing 93.5 : working with bit fields

```
// Listing 93.5 : working with bit fields
#include <stdio.h>

main()
{

struct abc
{

  unsigned a : 5;
  unsigned   : 3;
  unsigned b : 6;

}x;

}
```

In addition to unnamed bit fields, unnamed bit fields with a width of 0 can also be used to ensure alignment to the next word. A code sample to create three 1-bit bit fields in three separate words is given below.

Listing 93.6 : working with bit fields

```
// Listing 93.6 : working with bit fields
#include <stdio.h>

main()
{

struct abc
{

  unsigned a : 1;
  unsigned   : 0;
  unsigned b : 1;
  unsigned   : 0;
  unsigned c : 1;

}x;

}
```

It is not always advantageous to use bit fields in an application. They are effective only when the data being handled is voluminous.

The first disadvantage of using bit fields is that pointers cannot be used to access the bit fields directly. This is because the address operator (&) cannot be applied to bit fields. The second disadvantage is that it is not possible to have arrays of bit fields.

Finally, the use of bit fields restricts programs to the machine type. For example, the code developed on a 32-bit machine, which supports signed bit fields, may not be portable to a 16-bit machine, which supports only unsigned bit fields.

The way in which bits are assigned to bit fields is also machine-dependent. Some compilers assign bits from left to right whereas others assign bits from right to left. As result, a program written for one type of computer may produce incorrect results in another machine.

Self Review Exercise 93.1

1. *You learned the rules for declaring bit fields. Choose the option that describes the structures displayed.*

```
#include <stdio.h>
main()
{
struct sample1
{
unsigned a : 5;
unsigned b : 5;
unsigned   : 6;
unsigned c : 5;
}v={1,2,3};
}
```

```
#include <stdio.h>
main()
{
struct sample2
{
unsigned a : 5;
unsigned b : 5;
unsigned c : 5;
}v={1,2,3};
}
```

A. Both structures need 2 bytes to store the three bit fields.
B. Both structures occupy one word on a machine with a 2-byte word.
C. On a 16-bit operating system, the first two bit fields of sample1 are stored in the first word and the third bit field is stored in the second word.
D. On a 16-bit operating system, the first two bit fields of sample2 are stored in the first word and the third bit field is stored in the second word.

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

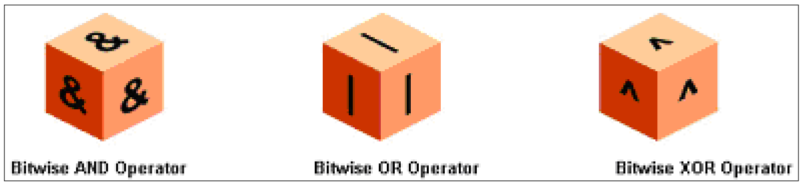
BITWISE LOGICAL OPERATOR

C provides some operators that can be used to directly manipulate bits. These are called bit-manipulation operators. These operators can broadly be classified as the bitwise logical operator, the bitwise shift operator, and the complement operator.

Bitwise logical operators are the binary operators that accept integer type operands. Within the operands, within the operands, the bits of the same order are compared.

There are three bitwise logical operators. These are the bitwise AND (`&`) operator, the bitwise OR (`|`) operator, and the bitwise eXclusive OR (`^`) operator, which is commonly known as the XOR operator.

Bitwise Logical Operator



To be able to use bitwise logical operators in your programs, you should know how these operators work, when to use these operators, and the order in which these operators are executed if all three occur together in an expression.

The bitwise AND operator compares `2` bits and returns `1` if both the bits being compared have the value `1`. Otherwise, the result bit is `0`. The AND operation between `0` and `1` generates the result `0`.

A matrix with the possible values of the two operands and the result of the AND operation is given below.

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Binary Representation				
7	0	1	1	1
8	1	0	0	0
7&8	0	0	0	0

For example, the expression `7&8` will evaluate to `0`. The binary representation of `7` and `8` are `0111` and `1000`, respectively. Performing a bitwise AND operation between the two numbers sets all the bits to `0`.

The AND operator is extensively used to extract the values of specified bits from a binary number. This process is called masking. To extract the specified bit, you use a binary constant, which has the bit to be extracted, set to `1`. This constant is called a mask.

For example, the fourth bit of the variable A contains the information on whether or not the disk drive is ready for data transfer. To extract the fourth bit and mask the rest of the bits, you select the mask as the binary number `0000 1000` and perform the bitwise AND operation. As a result of masking using the bitwise AND operator, all the bits except the fourth bit of the result are set to `0`.

A	1	1	0	0	1	1	1	1
Mask	0	0	0	0	1	0	0	0
All bits except the fourth bit are masked.	0	0	0	0	1	0	0	0

The second bitwise logical operator is bitwise OR. This operator compares `2` bits and returns `1` either bit or both the bits have the value `1`. A matrix with the possible values of the two operands and the result of the OR operation is given.

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

For example, the expression `9|7` evaluates to `15`. The binary equivalents of `9` and `7` are `1001` and `0111`, respectively. After the OR operation, the low order `4` bits are set to `1`.

Binary Representation								
9	0	0	0	0	1	0	0	1
7	0	0	0	0	0	1	1	1
9|7	0	0	0	0	1	1	1	1

The bitwise OR operator is used to set bits of a binary number to `1`. For example, you can set the `4` least significant bits of a binary number to `1` while the remaining bits retain their original values. To do this, select the mask value with the lower four bits set to `1`.

Next, perform the OR operation between the mask value and the original number. In the example given, the least significant bits have the original value, which is `1010`, and the remaining bits have the value `1110`. After the bitwise OR operation is performed, the values of the `4` least significant bits change to `1111` whereas the other bits retain the original value.

A	1	1	1	0	1	0	1	0
Mask	0	0	0	0	1	1	1	1
A|Mask	1	1	1	0	1	1	1	1

Finally, the bitwise XOR operator compares `2` bits and returns `1` if `1` bit has the value `1` and the other bit has the value `0`. The result bit is `0` if the corresponding bits in both the operands are `0` or `1`. A matrix with the possible values of the two operands and the result of the XOR operator is displayed.

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

For example, the expression `9^7` will evaluate to `14`. The binary equivalents of `9` and `7` are `1001` and `0111`, respectively. The result of the XOR operation is `1110`.

Binary Representation								
9	0	0	0	0	1	0	0	1
7	0	0	0	0	0	1	1	1
9^7	0	0	0	0	1	1	1	0

Notice that the XOR operator sets the result bit to `1` if the corresponding bits in the operands are different. In other words, the result bit is set to `1` if `1` bit is `1` and the other bit is `0`. Otherwise, the result bit is set to `0`.

The XOR operator is also used to encrypt data. For example, if you want to reverse the `4` least significant bits of a number and preserve the rest of the bits, you perform an XOR operation.

A	1	1	1	0	1	0	1	0
Mask	0	0	0	0	1	1	1	1
A^Mask	1	1	1	0	0	1	0	1

To encrypt the data in the desired format, you first select the mask as `0000 1111`, which is the binary equivalent of `155`. Next, perform the XOR operation between the original number and the mask. In the example given, the original value of the least significant bits is `1010`.

After the XOR operation is performed, the values of the `4` least significant bits change to `0101` whereas the values of the other bits are the same as the original number.

LESSON 94 : BITWISE LOGICAL OPERATORS

In a C program, the expression `a=a^` can also be written as `a^=b`. The expression `^=` is a type of bitwise assignment operator. The bitwise assignment operators combine the preceding bitwise operations with the assignment process.

Bitwise assignment operators accept two operands. The operand to the left of the operator should be of the integer type, and the expression on the right must be a bitwise expression.

A list of the available bitwise assignment operators and their equivalent expressions are given. Consider an example. The expression `a&=b` is equivalent to `a=a&b`.

Bitwise Assignment Operators	*Expression*	*Equivalent Expression*
&=	a&=b	a=a&b
|=	a|=b	a=a|b
^=	a^=b	a=a^b

Like the other operators in C, the bitwise logical operators also follow a precedence rule. The bitwise AND operator, which has the highest precedence, is followed by the XOR operator and finally by the bitwise OR operator.

Self Review Exercise 94.1

1. *Extract the fifth and sixth bits from a* 1*-byte word,* 1001 0001*. Identify the mask value and the output of the expression (*`1001 0001 & mask`*).*

   A. `Mask=0011 0000, Output=0001 0000`
   B. `Mask=1100 1111, Output=1101 0001`
   C. `Mask=0011 0000, Output=0011 0000`
   D. `Mask=0000 1100, Output=0000 0000`

2. *If the variables x and y store the values* 5 *and* 3*, respectively. The binary equivalents of* 3 *and* 5 *are* 0011 *and* 0101*, respectively. The decimal equivalent of the expression* `x|y` *is :*

3. *Choose the output of the specified code.*

```
#include <stdio.h>
Main()
{
        int One=5, Two=3;

        One^=Two;
        Two^=One;
        One^=Two;

        printf("One=%d Two=%d\n", One, Two);
}
```

   A. `One=5 Two=3`
   B. `One=3 Two=5`
   C. `One=6 Two=3`
   D. `One=5 Two=6`

Self Review Exercise 94.2

4. *Identify the mask and the bitwise operation that should be performed to extract the lower* 5 *bits of a binary number.*

   A. AND with 0001 1111
   B. AND with 1111 1000
   C. OR with 0001 1111
   D. OR with 1111 1000

5. *Identify the expression to extract the* 3 *least significant bits of the variable* B *and set the other bits to* 1*. The result of the bitwise operation should be stored in the variable* B*.*

   A. B&=0001 1111
   B. B&=1111 1000
   C. B|=0001 1111
   D. B | =1111 1000

LESSON 95 : BITWISE SHIFT OPERATORS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

BITWISE SHIFT OPERATORS

An important category of bitwise operators available in C is bitwise shift operators. There are two types of bitwise shift operators. These are left shift (<<) and right shift (>>) operators. Bitwise shift operators move the bits of an integer a specified number of position. The general syntax of these operators is displayed. Bitwise shift operators need two operands.

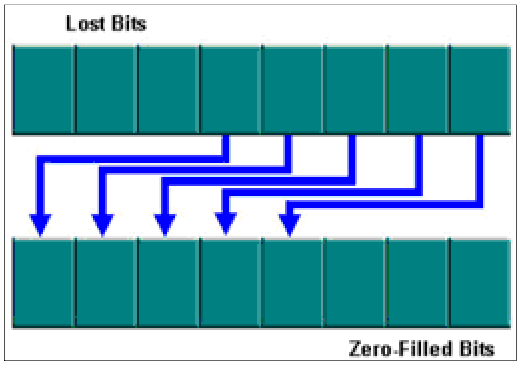
The first operand is an integer variable whose bit positions are to be shifted. The second operand is an unsigned integer that specifies the number of positions by which each bit has to be moved.

```
Operand 1 <Bitwise Shift Operator> Operand 2
Integer                            Unsigned Integer
```

For example, if the second operand has the value 1, each of the bits in the first operand move by a 1-bit position to the left or the right depending on the operator being used.

Left Shift Operation

The left shift (<<) operator shifts bits to the left. As a result, the higher order bits are lost and the lower order bits are filled with zeros. An example of the change in the bit positions after the left shift operation is performed is given.



Left shift operators provide a quick and efficient method of multiplying by the powers of 2. This is because only one operation of shifting bits is carried out with the left shift operator. Otherwise, the integers are first converted to their binary equivalents and the multiplied.

For example, the expression 5<<1 shifts the bits of the binary equivalent of 5 by one position to the left. The binary equivalent of 5 is 0101. Shifting bits by one position to the left results in the binary number 1010.

The decimal equivalent of 1010 is 10, which is equal to 5*2^1. Similarly, 5<<2 is equal to 20, which is equal to 5*2^2.

Binary Equivalent of 5	0000	0101	
Result After 5<<1	0000	1010	*Binary Equivalent of 10*
Result After 5<<2	0001	0100	*Binary Equivalent of 20*

The result of the left shift operation is different depending on the hardware configuration. For example, on an 8-bit machine, the output of the expression 255<<1 is 254. On a 16-bit machine, 255<<1 will give the output 510.

Left Shift Operation on an 8-bit Machine

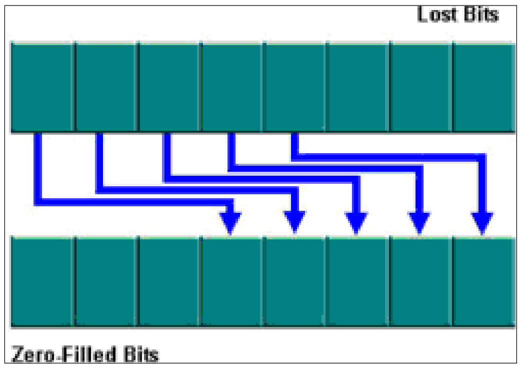
Binary Representation of 255	1 1 1 1 1 1 1 1
255<<1	1 1 1 1 1 1 1 0
Decimal Equivalent of the Output	254

Left Shift Operation on an 16-bit Machine

Binary Representation of 255	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
255<<1	0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0
Decimal Equivalent of the Output	510

Right Shift Operator

The second bitwise shift operator is the right shift (>>) operator. This operator shifts bits to the right. As a result, lower order bits are lost and higher order bits are filled with zeros.



For example, the expression `5>>1` shifts the bits in the binary equivalent of `5` by one position to the right. The binary equivalent of `5` is `0101`. After the right shift operation is performed, the binary number changes to `0010`. The expression `5>>2` shifts the bits in the binary equivalent of `5` by two positions to the right. As a result of this shift operation, all the bits expect the least significant bit are set to `0`.

Binary Equivalent of 5	0000	0101
Result After 5>>1	0000	0010
Result After 5>>2	0001	0001

The right shift operator provides a fast method of dividing an integer by `2`. For example, the result of the expression `6>>1` is `3`. The binary representation of `6` is `0110`. The binary number obtained after the right shift operation is `0011`. The decimal equivalent of `0011` is `3`, which is equal to `6` divided by `2^1`. Similarly, the output of the expression `6>>2` is `1`, which is equal to `6` divided by `2^2`.

Binary Equivalent of 6	0000	0110
Result After 6>>1	0000	0011
Result After 6>>2	0001	0001

An example of the code that uses bitwise shift operators to convert a decimal number to its binary equivalent is given below. The code uses the bitwise right shift operator to mask all bits expect the one that needs to be given.

The initial value of the mask is set depending on the hardware configuration of the machine used to execute the code. The bitwise left shift operator is used to set the initial value of the mask.

Listing 95.1 : Display the Binary Equivalent of a decimal number

```
//Listing 95.1 : The program to display the binary equivalent of a decimal number

#include <stdio.h>

main()
{
    int a,b,m,count,nbits;
    unsigned mask;
    nbits=8*sizeof(int);                    //Set the initial value of the mask
    m=1<<(nbits-1);
    do
    {
            printf("\nType an integer value:");
            scanf("%d",&a);
            mask=m;
            for(count=1;count<=nbits;count++)
                    {
                    b=(a&mask)?1:0;
                    printf("%x",b);

                    if(count%4==0)
                            printf(" ");
                    mask>>=1;               //Change the mask value to display each bit
}
}while(a!=0);
}
```

If the first operand is a signed integer, the behavior of the right shift operator will differ from compiler to compiler. In the binary representation of a signed integer. The highest order bit is used to indicate if the integer is positive or negative.

Listing 95.2 : working with bitwise shift operators

```
//Listing 95.2 : working with bitwise shift operators

#include <stdio.h>

main()
{

    unsigned int a=5;
    int b=-5;

    printf("\n%d \t%d", a,b);
    printf("\n%d", a>>2);
    printf("\n%d", b>>2);
}
```

If the highest order bit is set to 1, it indicates a negative number. After the right shift operation, some compilers fill the vacant bit positions with the contents of the highest order bits.

The difference in the outputs of a right shift operation on a signed bit and an unsigned bit is given. The right shift operation on b, which is a signed integer, generates the output -2, which is different from the output for an unsigned integer, a.

Besides providing a faster method of multiplying and dividing a number by 2, bitwise shift operators are also used for data compression. A code sample that uses bitwise shift operators for data compression is given below.

Listing 95.3 : working with shift operators

```
//Listing 95.3 : Code to pack four characters in one integer by using shift operators

#include <stdio.h>

int packchar(char a,char b,char c,char d);

void printbits(int a);

main()
{

    printf("abcd=");
    printbits(packchar('A','B','C','D'));
    putchar('\n');
}

//Function to pack four character values in an integer.

int packchar(char a,char b,char c,char d)
{

    int x=a;
    x=(x<<8)|b;
    x=(x<<8)|c;
    x=(x<<8)|d;
    return x;
}

//Function to display the binary equivalent of an integer value.

void printbits(int a)
{

    int i;
    int n=8*sizeof(int);
    int mask=1<<(n-1);

    for(i=1;i<=n;i++)
            {
            putchar(((a & mask) == 0) ? '0' : '1');
            a<<=1;
            if(i%8==0&&i<n)
                    putchar(' ');
            }
}
```

Detailed Explanation

- In this example, four characters are stored in one integer value itself. After the first character is stored, the left shift operation is performed eight times. Performing a bitwise OR operation between the integer and the character saves the second character in the next 8 bits.
- This process of shifting the bits to the left eight times and then performing the OR operation continues until all four characters are saved. The first character is stored in the 8 bits to the extreme left. Then, the second character is stored.

Bitwise shift operators are also used to retrieve original characters from an integer value. Using this data compression technique of packing multiple characters in one integer saves time spent in processing data.

In addition to the bitwise left and right shift operators, C also supports two bitwise assignment operators that are used to assign the results of shift operations. These are the left shift assignment `(<<=)` operator and the right shift assignment `(>>=)` operator.

Using a bitwise assignment operator, the expression `a=a<<2` can be written as `a<<=2`. Similarly, the expression `a=a>>1` is equivalent to `a>>=1`. Both these expressions shift the bit position of a by `1` to the right, and the result is stored in the variable `a`.

Bitwise Shift Assignment Operators	*Expression*	*Equivalent Expression*
`<<=`	`a<<=b`	`a = a<<=b`
`>>=`	`a>>=b`	`a = a>>=b`

Self Review Exercise 95.1

1. *Identify the output of the expression* a<<3. *The value of the variable a is* 5. *The binary representation is* 5 *is* 0000 0101.

   A. 0100 0000
   B. 0001 0100
   C. 0010 1000
   D. 0000 0010

2. *What does* 1001 0010 << 4 *evaluate to on an* 8*-bit machine? Enter the decimal equivalent of the result after the bitwise left shift operation is performed.*

3. *What does* 1001 0010 >> 4 *evaluate to?*

   A. 146
   B. 73
   C. 36
   D. 72
   E. 9

4. *What does* 5>>2 *evaluate to?*

   A. 10
   B. 1
   C. 2
   D. 64

5. *The variables* a *and* b *store the values* 4 *and* 2*, respectively. The binary equivalent of* 4 *is* 0000 0100*, and the binary equivalent of* 2 *is* 0000 0010. *Identify the values stored in* a *and* b *after the expression* a<<=b *is executed.*

   A. a=0000 0100, b=0001 0000
   B. a=0001 0000, b=0000 0010
   C. a=0001 0000, b=0010 0000
   D. a=0010 0000, b=0000 0010

LESSON 96 : ONE'S COMPLEMENT AND TWO'S COMPLEMENT

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

C provides a special set of operators called bitwise operators that support the direct manipulation of the bits within a word stored in memory. A commonly used bitwise operator is the complement `(~)` operator.

The complement operator is also referred to as the bitwise complement operator or the one's complement operator. The bitwise complement operator is a unary operator. It accepts only one operand of the type integer.

The bitwise complement operator precedes the operand. The operand can be any integer data type, such as `long`, `short`, `unsigned`, or `char`. The general syntax for using the unary operator is `~a`, where `a` is a variable.

The bitwise complement operator reverses the bits of the binary number. As a result, all the ones in the binary number change to zeros and all the zeros are replaced by ones.

8-Bit Binary Number A	1101 0011
Bitwise complement ~A	0010 1100

For example, the expression `~255` generates the result `0`. The binary equivalent of `255` has all its bits set to `1`. The bitwise complement operator reverses all the bits and sets them to `0`. Similarly, the output of the expression `~(1010 0010)` is `0101 1101`.

Binary Equivalent of 255	1111 1111 1111 1111
~255	0000 0000 0000 0000

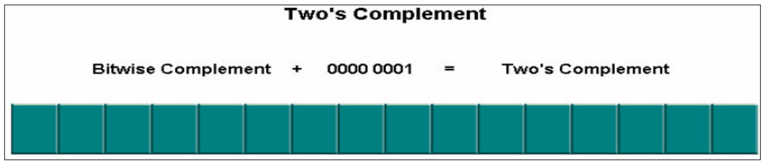
The bitwise complement operator is used to set and reset the bits of a binary number. To do this, the bitwise complement operator is used along with the bitwise OR and AND operators.

In the code given below, the expression `flag=flag&~mask` is used to reset the fourth bit of the flag to `0`. The value of the mask is set to `8` in the initial part of the code. Using a bitwise complement operator also reduces machine dependency.

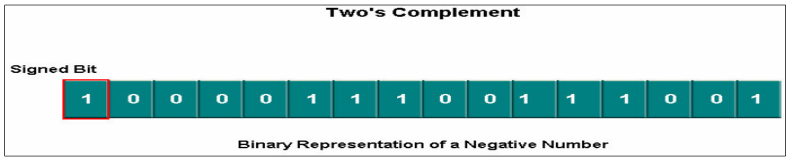
Listing 96.1 : Using one’s complement and two’s complement

```
// Listing 96.1 : this program deals with implementation of one’s complement and two’s
// complement
//Code to reset the fourth bit
#include <stdio.h>
void decimaltobinary(int);
main()
{
int flag;
unsigned mask;
mask=1<<3;
printf(“\nType an integer value for the flag:”);
scanf(“%d”,&flag);
decimaltobinary(flag);                        //Display binary value of the input
printf(“\n”);
flag=flag&~mask;                              //Code to reset the fourth bit
decimaltobinary(flag);                        //Display binary value of the input
}
void decimaltobinary(int a)                   //Function to display binary equivalent
{
int b,m,count,nbits;
unsigned mask;
nbits=8*sizeof(int);
m=1<<(nbits-1);
mask=m;
for(count=1;count<=nbits;count++)
{
b=(a&mask)?1:0;
printf(“%x”,b);
if(count%4==0)
printf(“ “);
mask>>=1;
}
}
```

Besides the bitwise complement, C also supports the concept of two’s complement. Some machines use the two’s complement to store negative numbers.



To obtain the two’s complement of a binary number, add 1 to the one’s complement of the binary number. In two’s complement representation, the highest order bit is the signed bit. If the signed bit is set to 0, the number is positive. Otherwise, it is a negative number.



For example, ~7 is stored in memory as the two's complement representation of 7. The two's complement representation of ~7 is given below. The range of values that can be represented depends on whether signed or unsigned representation is used.

```
Binary Representation of 7              0000 0111

Bitwise Complement                      1111 1000

Two's  Complement                        1111 1001              Binary Representation of -7

Decimal Equivalent     (-) 1*2^7  + 1*2^6  + 1*2^5  + 1*2^4  + 1*2^3  + 0*2^2 + 1*2^1 + 1*2^0

                       -128 + 64 + 32 + 16 + 8 + 0 + 0 + 1 -128 + 121 -7
```

For example, on a 8-bit operating system, numbers from 0 through 255 can be represented using unsigned representation. On the other hand, numbers from 128 to -128 can be represented using signed representation.

Range of Unsigned Numbers	0000 0000 to 1111 1111
Range of Signed Numbers	1111 1111 to 0111 1111

Self Review Exercise 96.1

1. *Identify the option that lists the one's complement of* 0010 1010.

   A. 42
   B. 84
   C. 213
   D. 171

2. *The binary equivalent of* 13 *is* 0000 1101. *Enter the binary equivalent of the bitwise complement of* 13.

3. *Identify the option that lists the one's and two's complements of* 6. *the binary equivalent of* 6 *is* 0000 0110.

   A. 0000 0110, 1111 1001
   B. 1111 1001, 1111 1010
   C. 1111 1001, 1111 1000
   D. 1111 1001, 0111 1001

REVIEW EXERCISE

1. Identify the option that lists the one's and two's complements of the binary number 1001.

```
A. 9, 10
B. 6, 7
C. 9, 7
D. 6, 10
```

2. Identify the outputs of the expressions `x & y ^ z`, `x & y \ z`, *and* `y ^ z` *in which* x, y, *and* z *have the values* 5, 7, *and* 3, *respectively. The binary equations of* 3,5, *and* 7 *are* 0011,0101, *and* 0111, *respectively.*

```
A. 7, 6, 7
B. 7, 6, 4
C. 6, 7, 7
D. 6, 7, 4
```

3. The binary representation of 5 *is* 0000 0101*. Identify the output of the expression* a<<2. *(A* `signed` 8*-bit integer* a *stores the value* 5*. The binary equivalent of* 5 *is specified. Tick the output of the expression that uses binary shift operators.)*

```
A. 20
B. 1
C. 64
D. 0
```

4. The binary representation is 6 *is* 0000 0110*. Identify the output after the specified code is executed.*

```
{….
   a>>=2;
   a<<=2;
….}
```

(An unsigned 8*-bit integer* a *stores that value* 6*. The binary equivalent of* 6 *is specified. Tick the output of the expression that uses binary shift operators.)*

```
A. 24
B. 128
C. 4
D. 1
```

5. Complete the code to declare a bit field in the `employee` *structure. The* `unsigned` *bit field* `office` *is used to indicate the* `office` *where and* `employee` *is posted. Six regional offices of the company are indicated by the number* 0 *to* 5 *in the database. Write the code.*

```
#include<stdio.h>
main()
{
        struct
        {
                unsigned ID: 5;
                //Enter here
        }employee;
}
```

6. What is the decimal equivalent of the binary number 0000 10*10?*

```
A. 5
B. 10
C. 20
D. 8
```

REVIEW EXERCISE

7. Complete the code to declare a bit field in a structure called `customer`*. An* `unsigned` *bit field is used to indicate the credit card used to make a payment. The card can be Master Card, Visa, or American Express. The bit field is called* `payment`.

```
#include<stdio.h>
main()
{
        struct
        {
                unsigned ID: 4;
                //Enter here
        }customer;
}
```

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The most common function used for accepting input and displaying data to and from standard input/output (I/O) devices are `scanf()` and `printf()`. Depending on the conversion characters, these functions accept various data types, such as characters, integers, and strings.

The `scanf` and `printf` functions are also used to perform formatted input and output operations. Formatted output is useful because it increases readability. You can use formatted input to accept only the required data.

You can format the output by using modifiers. Modifiers can be an integer value, a space, or any other symbol that is placed between a percent`(%)` sign and the conversion character.

Minimum Field Width Specifier

The most commonly used modifier is the minimum field width specifier. This is an integer placed between the `%` sign and the conversion character. When this modifier is used, the output is padded with white spaces to ensure a minimum width.

The modifier will not have any effect if the number of digits in the variable to be displayed is more than the number of digits in the minimum field width specifier. The values stored in a variable are never truncated because of the format width specifier.

For example, in the code given below, the value of the variable `a`, which is `99`, will be preceded by two blank spaces because the minimum width specified is `4`. If the value stored in the variable a is changed to `9999`, then the modifier has no effect and the output is `9999`.

Listing 97.1 : formatted output

```
// Listing 97.1 : A program illustrating formatted output

#include <stdio.h>

main()
{
   int a=9;
   printf(“%4d”,a);
}
```

Adding a zero between the `%` sign and the minimum field width specifier changes the output. Instead of white spaces, the output is padded with zeros. In the example given, the output, `0099`, is padded with zeros instead of blank spaces.

The minimum field width specifier is commonly used to display tables in which columns should be aligned. An example of the code that uses the minimum field width specifier to increase the readability of the output is given below.

Listing 97.2 : formatted output

```
// Listing 97.2 : A program illustrating formatted output

#include <stdio.h>

main()
{

   int i;
   for(i=1;i<10;i++)
   printf("\n%5d %5d",i,i*i);

}
```

Precision Specifier

Another modifier is the precision specifier. A precision specifier consists of a period followed by an unsigned integer. Usually, the precision specifier follows the minimum field with specifier but it can also be used independently.

The effect of the precision specifier on the output depends on the conversion character used in `printf()`. When used with floats, the precision specifier specifies the number of decimal places to be displayed. In the example, the output has three decimal places.

Listing 97.3 : formatted output

```
// Listing 97.3 : A program illustrating formatted output

#include <stdio.h>

main()
{

float i=123.45;
printf("%.3f",i);

}
```

An example in which the precision specifier follows the minimum field width specifier is given below. In this case, the output will be at least eight characters wide of which three characters are reserved for three decimal places and one for the decimal point.

Listing 97.4 : formatted output

```
// Listing 97.4 : A program illustrating formatted output

#include <stdio.h>

main()
{

float i=123.45;
printf("%8.3f",i);

}
```

When used with strings, the precision specifier defines the maximum number of characters acceptable. As per the `printf` function given in the example, a maximum of `12` characters can be displayed. Therefore, the output has only the first `12` characters of the string stored in `a`.

Listing 97.5 : formatted output

```
// Listing 97.5 : A program illustrating formatted output

#include <stdio.h>

main()
{

Char a[]="This is a course on C.";
printf("%.12s",a);

}
```

When a precision specifier is used along with the integer data type, it specifies the minimum number of digits for that number.

Listing 97.6 : formatted output

```
// Listing 97.6 : A program illustrating formatted output

#include <stdio.h>

main()
{

int i=4;
printf("%33d"i);

}
```

At times, in `printf()`, both the field width and precision specifiers are indicated using an asterisk `(*)` symbol instead of constants. The `*` symbol is used when the values of the width and precision specifiers are passed as arguments to `printf()` during program execution.

```
                  printf("%*.*f", a,b,c);

Equivalent to:    printf("%a.bf",c);
```

For example, in the code given below, the first `*` symbol will be replaced by the value of variable `a`, which is `9`, and the second `*` symbol will be replaced by the value of variable `b`, which is `5`. As a result, the value of variable `c` when displayed will have `5` decimal places.

Listing 97.7 : formatted output

```
// Listing 97.7 : A program illustrating formatted output

#include<stdio.h>

main()
{

   int a=9,b=5;
   Float c=123.45;
   printf("%*.*",a,b,c);

}
```

Left Aligned and Right Aligned Specification

Besides the minimum field width specifier and precision specifiers, there are other modifiers that are used to display the output either as left-aligned or as right-aligned. By default, the output is always right-aligned. Placing a minus `(-)` sign between the `%` sign and the conversion character indicates that the output should be left-aligned.

Left Aligned

```
#include<stdio.h>
main()
{
int i=4;
printf("%-3d",i);
}
```

Right Aligned

```
#include<stdio.h>
main()
{
int i=4;
printf("%3d",i);
}
```

The next modifier is the plus `(+)` sign. A `+` sign between the `%` sign and the conversion character indicates that if the output is a nonnegative number, the output should be preceded by a `+` sign.

```
#include<stdio.h>
main()
{
unsigned i=4;
printf("%3.3d",i);
}
```

```
#include<stdio.h>
main()
{
unsigned i=4;
printf("%+3.3d",i);
}
```

If a space is placed between the `%` sign and the conversion character, the positive integer is preceded by a blank space. If both the space and the `+` space is ignored.

Listing 97.8 : formatted output

```
// Listing 97.8 : A program illustrating formatted output

#include <stdio.h>

main()
{

unsigned i=4;
printf("% +3.3d",a);

}
```

The hash`(#)` Specifier

Another modifier is the hash `(#)` symbol. A `#` symbol between the `%` sign and the conversion character is used to place a decimal point even if there are no decimal digits with `g`, `f`, and `e` conversion characters. It is not used with other conversion characters.

You learned to modify the output of the `printf` function by using format modifiers. Similarly, you can also format the data that is read using the `scanf` function.

An integer placed between the `%` sign and the conversion character specifies the maximum width of the variable. This can be used when you want to limit the size of input.

For example, in the code given below, the variable `str` will store only the first nine characters, regardless of the length of the input string. This modifier can be used in the programs in which the name entered by a user should not exceed nine characters.

Listing 97.9 : formatted output

```
// Listing 97.9 : A program illustrating formatted output

#include <stdio.h>

main()
{

char str[15];
scanf("%9s",str);

}
```

Similarly, you can also have an `*` symbol between the `%` sign and the conversion character. This specifies that the character is read and discarded without being assigned to any variable. This is mainly used to accept only the required data from the input string.

In the example below, the minus string will be read but will not be assigned to any variable.

Listing 97.10 : formatted output

```
// Listing 97.10 : A program illustrating formatted output

#include<stdio.h>

main()
{

   int a,b;
   char str[10];
   scanf("%d%*5s%d", &a,&b);
   printf("\n%d",a,b);

}
```

Finally, you can format the string to read only a specified set of characters. This can be done using a scanset. A scanset can be a set of characters, a space, or a newline character that is specified within brackets.

When a scanset is used, the string `str` will only have the characters that are specified in the scanset. Depending on the format of the `printf` function, the remaining characters that have been entered can either be stored in another variable or are lost.

For example, the string stored in the variable `str` can only have the characters `a`, `e`, `i`, `o`, and `u`. The reading process stops whenever any other character is encountered. Therefore, only character `a` will be read in the variable `str`. The characters starting from `b` will be stored in `str1`.

Listing 97.11 : formatted output

```
// Listing 97.11 : A program illustrating formatted output

#include<stdio.h>

main()
{

   char str[10], str1[12];
   scanf("%[aeiou]%s", str,str1);
   printf("%s %s", str, str1);

}
```

If the input string does not start with `a`, `e`, `i`, `o`, or `u`, then the `str` variable will not store any value. In the example, the first letter of the input string, `C`, is not specified in the scanset. Therefore, the input will not be read into any of the variables. Both `str` and `str1` will not store any data.

The characters specified in the scanset are case-sensitive. In the example given, only the lowercase a of the input `aEIOU` is stored in the variable `str`. The remaining input string is stored in `str1`.

If a caret `(^)` symbol is used within the scanset, then all the other characters besides the ones included in the scanset are read into the string. For example, if the input is `Clanguage`, the string `str` will store `Cl`. The remaining string that starts from `a` is stored in `str1`.

Listing 97.12 : formatted output

```
// Listing 97.12 : A program illustrating formatted output

#include<stdio.h>

main()
{

   char str[10], str1[12];
   scanf("%[^aeiou]%s", str,str1);
   printf("%s %s", str, str1);

}
```

Self Review Exercise 97.1

1. *You learned about the minimum field width specifier. A code sample that uses the minimum field width specifier in the printf function is given. Choose the output of the specified code now.*

```
#include<stdio.h>

main()
{

   int a=12345;
   char b[]="Print";
   printf("[%4d]", a);
   printf("\n[%07d]", b);
   printf("\n[%4s]", b);
   printf("\n[%6s]", b);

}
```

```
[1234]
[0012345]
[Print]
[ Print]
```

```
[12345]
[0012345]
[Print]
[ Print]
```

```
[12345]
[  12345]
[Print]
[ Print]
```

```
[12345]
[0012345]
[Print]
[0Print]
```

2. *You learned to format the output of the printf function by using modifiers. A code sample that uses modifiers in the printf function is given. Identify the output of the code. Choose the output of the specified code given.*

```
#include<stdio.h>

main()
{
   Char a[]= "This is a code in C
               language.";
   float x=246.86;
   printf("[%9.4f]",x);
   printf("\n[%.9s]",a);

}
```

```
[246.8600]
[This is a]
```

```
[246.8600]
[This is a co]
```

```
[  246.8600]
[This is a]
```

```
[ 246.8600]
[This is a]
```

Self Review Exercise 97.2

3. *You learned to format input and output by using format modifiers. Choose the output that will be displayed when the input string is aeiou now.*

```
#include<stdio.h>

main()
{
  int a,b;
  char str[10], str1[12];
  scanf("%[aAc \ti]%s", str, str1);
  printf("[%5s],[%s]",str, str1);

}
```

```
[a], [eiou]
```

```
[    a], [eiou]
```

```
[  ai], [eou]
```

```
[    A], [eiou]
```

Lesson Objectives

In this Lesson you will learn :

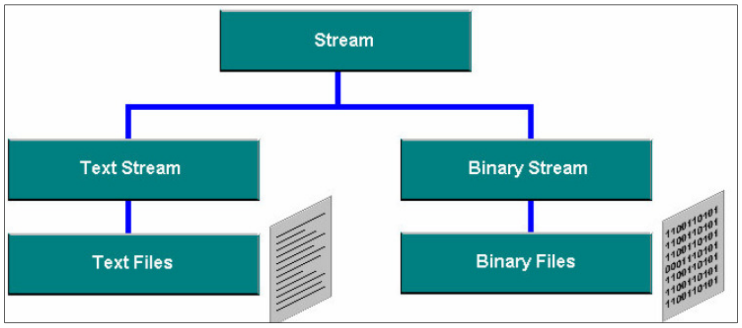
- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

In C, the various auxiliary storage devices, such as a disk file, a terminal, a printer, and a tape drive, are treated as files. Data files can be classified as system-oriented data files and stream-oriented data files.

System-oriented data files perform direct I/O. This means that files are handled at the operating system level and no separate memory area is maintained for the file.

In stream-oriented data files, the exchange of data is through streams. A stream is a sequence of either characters or bytes. A stream of characters is referred to as a text stream, and a stream of bytes is referred to as a binary stream.

Stream-oriented files are more common because it is easy to use these files. Stream-oriented data files are of two types, text and binary. When a text stream is used, the data file is called a text file. When a binary stream is used, the data file is called a binary file.



In text files, data is stored in the form of characters. The data in a text file can be organized in the form of lines where each line is terminated using a newline character. Characters such as newline and null have special meaning when read in text mode.

Binary files store data in the form of blocks. These blocks contain information in the form of bytes. Binary files are used to store complex data structures, such as arrays and structures.

For example, a diskfile containing employee records will be treated as a binary file. This is because the records are stored in the form of structures and the complete structure is treated as a block of data.

Binary files can be compared to an array of structures, which is stored on the disk instead of in memory. The structures in a binary file can be randomly accessed like an element of an array.

The read and write operations are faster on binary files that text files. This is because the binary form of record is transferred directly from memory to the disk when binary files are used.

In the case of text files, each character has to be converted from the binary format to the text format and vice versa. This conversion is required because all instructions on a computer are internally converted to the binary format before execution.

Stream-oriented data files, which include text and binary files, use buffers to perform input and output operations. A buffer is a memory area used for temporarily storing data before it is actually sent from the memory of the computer to the storage device and vice versa.

Using buffers increases the efficiency of I/O operations because the number of times the I/O device is accessed is greatly reduced. When you open a file, its contents are stored in the buffer and the additions and deletions are executed in the buffer itself.

Another advantage of using stream-oriented data files is that they provide a consistent interface to C programmers, regardless of the storage device used. This is because the functions used to perform buffered I/O operations are identical.

For example, the function used to open a file stored in a tape drive will be identical to the function used to open a file stored in magnetic drums.

LESSON 99 : FILE ACCESS MODES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

To access data from a file, you first open the file. Depending on whether you want to read or write data from a file and how you want to read or write the data, you can open the file in different modes

There are six different modes in which a file can be opened. In C, the notations used to represent the mode in which a file has to be opened depend on whether the file being opened is a text file or a binary file.

The First Mode

The first mode in which a file can be opened is the read-only mode. A file opened in this mode can be used only for reading the data. An error is generated if the file to be opened does not exist or you do not have the required permission to open the file.

The letter r is used to represent the read-only mode when a text file is to be opened for reading. For opening a binary file, the letter b is suffixed to the notation used for opening a text file. Therefore, if a binary file is to be opened for reading, the representation used is rb.

The Second Mode

The second mode in which a file can be opened is the write-only mode. It is not possible to read data from the file opened in the write-only mode. You can only write to the file.

If the file to be opened for writing does not exist, a new blank file is created. If the file already exists, the contents of the file are deleted without warning and a new blank file with the same name is created.

If you open an already existing file in the write mode and then immediately close the file without making any changes to it, the original contents of the file will be lost. The write-only mode is represented by the letter w for text files and the letters wb for binary files.

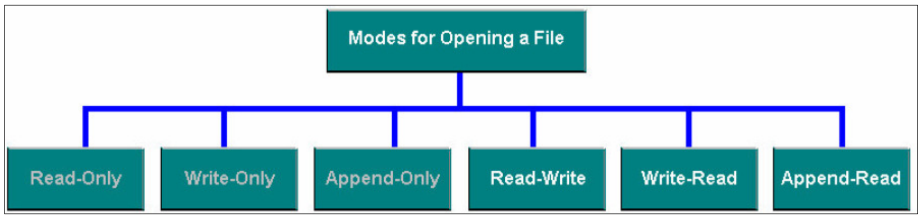
The Third Mode

The third mode is the append-only mode. If a text file is opened in the append-only mode, the new data entered into the file is added to the end of the file.

If the specified file does not exist, a new blank file is created and data is added at the beginning of the file. For example, assume that the file sample.c does not exist. A command to open sample.c in the append-only mode will create a new blank file named sample.c and open it. The append-only mode is represented by the letter a for text files and the letters ab for binary files.

If two separate processes open the same file for appending, each process may write to the file without the risk of destroying the output being written by the other. The outputs from the two processes will be written to the file in a haphazard manner.

Besides the read-only, write-only, and append-only modes, a file can also be opened in the read-write, write-read, and append-read modes.



An advantage of opening files in multiple modes is that the read and write operations can be performed without closing and reopening a file.

The Fourth Mode

A file opened in the read-write mode allows you to read the data in the file and write the data to the file. To represent that a text file is opened in the read-write mode, the notation r+ is used. For binary files, the notation used is rb+ or r+b. Similar to the read-only mode, only an existing file can be opened in the r+ mode. The read-write mode is extensively used when you open a file for updating its contents. At times, this mode is also referred to as the update mode.

The read-write mode is also called the update mode because you can change only a part of the file while the rest of the contents are not modified. For example, you can change the character written at the thirty-first byte without modifying the existing contents for the rest of the byte positions.

When a file is opened in the update mode, both input and output operations can be performed on the same stream associated with the open file. However, to reduce the probability of errors, the output must not be directly followed by input and vice versa.

Between an output and an input operation, there should preferably be a function call to either flush the buffer or reposition the file pointer. Similarly, input must not be directly followed by output, unless the input operation encounters the end of the file.

The Fifth Mode

The next mode is the write-read mode. This mode is represented as w+ and wb+ or w+b for text files and binary files, respectively. If you try to open a nonexistent file in the write-read mode, a newfile is created. If the file already exists, its contents are deleted without warning and a new empty file is created.

If a file is opened in the write-read mode, you first enter the data and then perform the read operation. This is because the write operation always starts on a blank file. It is important to remember that the write-read mode is different from the read-write mode. In the write-read mode, the file is opened for writing but you can also perform the read operation on the data that has been written.

The write-read mode is similar to the write-only mode because if the file does not exist, a new file will be created. If an existing file is opened, its contents are overwritten. In the read-write mode, the file is opened for reading but you can also write to the file. This mode is similar to the read-only mode because an error will be generated if the file does not exist.

The Sixth Mode

Finally, you can also open the file in the append-read mode. This mode is represented as a+ for text files and as ab+ or a+b for binary files. In this mode, the file is opened for appending and reading. If the file does not exist, a new file is created. If the file already exists, the new content is added at the end of the file.

Modes for Opening a File	*Notation Used for a Text File*	*Notation Used for a Binary File*
Read-Only	`r`	`rb`
Write-Only	`w`	`rb`
Append-Only	`a`	`ab`
Read-Write	`r +`	`rb+or r+b`
Write-Read	`w+`	`wb+ or w+b`
Append-Read	`a+`	`ab+ or a+b`

Self Review Exercise 99.1

1. *Identify the notation used if a binary file has to be opened only for reading data.*

   A. `r`
   B. `w`
   C. `rb`
   D. `wb`

3. *The binary file named students stores the records of all the students in a class. Identify the mode in which the file should be opened if you want to add a new record to the existing file. Choose the correct option now.*

   A. `r`
   B. `a`
   C. `w`
   D. `rb`
   E. `ab`
   F. `wb`

Self Review Exercise 99.2

3. *Identify the mode in which files should be opened if you want to copy the data in File1 to a new file, File2.*

   A. File1 in the append-only mode and File2 in the write-only mode
   B. File1 in the read-only mode and File2 in the write-only mode
   C. File1 in the write-only mode and File2 in the read-only mode
   D. Both File1 and File2 in the write-only mode
   E. Both File1 and File2 in the append-only mode

4. *Identify the mode in which a file can be opened to support both reading and writing. The new contents should be added at the end of the file.*

   A. Read-write
   B. Append-read
   C. Write-read
   D. Write-only

5. *All occurrences of the word would are to be replaced with the word should. Identify the mode in which the file should be opened.*

   A. Read-write
   B. Read-only
   C. Write-read
   D. Write-only
   E. Append-read

LESSON 100 : OPENING A FILE

Lesson Objectives
In this Lesson you will learn :
☛ *the Genesis of C Language*
☛ *the Stages in the Evolution of C Language*

To be able to perform any input or output operations on a disk file, you first open a file. To open a file, you use the `fopen` function. The syntax for this function is given below. This function accepts two parameters. The first parameter is the name of the file that is to be opened. In addition to the file name, the complete path of the file is also acceptable.

```
Fopen()
FILE *fopen(const char *filename, const char *mode);

                          Read-Only
                          Write-Only
                          Append-Only
                          Read-Write
                          Write-Read
                          Append-Read
```

When a file is opened using `fopen()`, certain information about the file is automatically stored in different predefined variables. This information includes whether the file is being read or written to, the file name, the buffer location, and the current character position in the buffer.

Listing 100.1 : opening a file

```
//Listing  100.1 : this program deals with opening a file
#define SEEK_SET     0
#define SEEK_CUR     1
#define SEEK_END     2
__DJ_Va_list
#udef __DJ_va_list
#define __DJ_va_list
__Dj_size_t
#define __DJ_size_t
#define __DJ_size_t

/*Note that the definitions of these fields are NOT guaranteed! They may change with any
release without notice! The fact that they are here at all is to comply with ANSI
specifications.*/

Typedef struct {
int _cnt;
char *_ptr;
char  *_base;
int bufsiz;
int _flag;
int _file;
char *_name_to_remove;
int _filesize;
}FILE;
typedef unsigned long                 fops_t;

extern FILE __dj_stdin, __dj_stdout, __dj_stderr;
#define stdin        (&__dj_stdin)
#define stdout       (&__dj_stdout)
#define stderr       (&__dj_stderr)
```

LESSON 100 : OPENING A FILE

The variables that contain information about an open file are collectively defined as the `FILE` data type. `FILE` is a `typedef` structure that is predefined in the header file `stdio.h`.

The `fopen` function returns a pointer called the file pointer of the `FILE` data type. The file pointer is used to direct the results of buffered I/O operations to the associated file. Each file that is opened by the `fopen` function should be associated with a unique file pointer.

```
fopen()
FILE *fopen(const char *filename, const char *mode);
```

To store this file pointer, you declare a pointer of the type `FILE` before the `fopen` function is called.

The syntax for declaring a pointer of the type `FILE` is given below. In an application, if there are two or more different files open at a specific time, then each of the files should have a unique pointer associated with it. After performing the first step of declaring a pointer of the type `FILE`, call the `fopen` function to open the file in the required mode. For example, to open a text file called `Csample`, you can specify the file name as `Csample` or `Csample.txt`.

The file names specified in the `fopen` function are case sensitive or insensitive depending on the operating system being used. For example, file names are case sensitive in a UNIX environment.

Declaring a File Pointer

```
FILE *<pointername>;
FILE *fopen(const char *filename,const char *mode);
```

Both the file name and the mode in which the file has to be opened should be enclosed within double quotation marks.

The next example shows the various ways in which the file name can be passed as a parameter to the `fopen` function. These lines of code perform the same task of opening the file `trial.c` in the read-only mode

Code Sample 1:

```
#include<stdio.h>
main()
{
FILE *fp;
fp=fopen("c:\\Sample\\trial.c","r");
}
```

Code Sample 2:

```
#include<stdio.h>
main()
{
FILE *fp;
fp=fopen("Sample\\trial.c","r");
}
```

Code Sample 3:

```
#include<stdio.h>
main()
{
FILE *fp;
fp=fopen("trial","r");
}
```

The code sample 1 uses the complete path. The file name and its extension are used in the code sample 2. The code sample 3 uses only the file name. When the code sample 2 and 3 are executed, files are searched in the current directory only.

Self Review Exercise 100.1

1. *You learned the steps to open a file. Choose the code sample to open the file sample1.txt in the read-only mode and the file sample2.txt in the append-only mode,*

```
#include<stdio.h>
main()
{
FILE *ptr, *fp;
ptr=fopen(“sample1.txt”,”r+”);
fp=fopen(“sample2.txt”,”a”);
}
```
A

```
#include<stdio.h>
main()
{
FILE *ptr, *fp;
ptr=fopen(sample1.txt,”r”);
fp=fopen(sample2.txt,”a+”);
}
```
B

```
#include<stdio.h>
main()
{
FILE *ptr, *fp;
ptr=fopen(“sample1.txt”,”r”);
fp=fopen(“sample2.txt”,”a”);
}
```
C

```
#include<stdio.h>
main()
{
FILE *ptr, *fp;
ptr=fopen(“sample1.txt”,”r+”);
fp=fopen(“sample2.txt”,”a+”);
}
```
D

LESSON 101 : READING FROM A FILE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The C language provides a large set of functions that can be used to perform input/output operations on a file. These functions are prototyped in the header file `stdio.h`. One of the most common operations performed on a file is reading a file.

The functions prototyped in `stdio.h` for reading data from a file are the `fgetc`, `fgets`, `fscanf`, and `fread` functions. The first three functions, `fgetc`, `fgets`, and `fscanf`, are used to read data from text files. The `fread` function is used to read data from binary files.

Functions Available in C for Reading Data
fgetc()
fgets()
fscanf()
fread()

`fgetc()`

The `fgetc` function is used for reading one character at a time from a file. The prototype of the function is given. The argument passed to the function is the file pointer returned by the `fopen` function.

```
int fgetc(FILE *fp);
```

After the data is read using the `fgetc` function, the file pointer is moved `1` byte ahead of the position. For example, if the file pointer is at byte position `3`, the file pointer moves to byte position `4` after the `fgetc` function is executed once.

In the Listing `101.1`, data is being read from the `sample.c` file by using the `fgetc` function. When this code is executed, the letter `T` is displayed on the screen. To read all characters, you can include the `fgetc` and `putc` functions in a loop as displayed in the Listing `101.2`.

Listing 101.1 : testing the functionality of the C programs

```
//Listing 101.1 : Code to Test the Functionality of the C Programs

#include <stdio.h>
main()
{
    FILE *fp;
    char d;
    fp = fopen("sample.c","r+");
    d = fgetc(fp);
    putc(d, stdout);
}
```

Listing 101.1 : testing the functionality of the C programs

```
//Listing 101.2 : Code to Test the Functionality of the C Programs

#include <stdio.h>

main()
{

    FILE *fp;
    char d;

    fp = fopen("sample.c","r+");

    do
            {
                    d = fgetc(fp);
                    putc(d, stdout);

            }while(d != EOF);
}
```

fgets()

The next function available for reading data is the `fgets` function. This function reads a string of a specified length.

```
char *fgets(char *line, int n, FILE *fp);
```

This function accepts three parameters.

- The first parameter is the pointer to the array used for storing the data that is read using the `fgets` function.
- The second parameter is an integer value that indicates the number of characters to be read. The function will read `n-1` characters. The last character is reserved for null. The third parameter is a pointer to the file that is used for reading data.
- The `fgets` function reads characters into the array until `n-1` characters are read, a newline character is read, or an `EOF` condition is encountered. When a newline character is encountered, the character is also added to the array.
- The newline character is added only if an entire line has been read without reaching the limit of characters for input. After the data is read into the array, the string is terminated with a null character. As a result, the last element of the array is a null character.

For example, the statement `fgets(c,100,fp);` will read a maximum of `99` bytes from the file pointed to by the stream or the file pointer `fp`. The last byte will be reserved for the null character.

A code sample that uses the `fgets` function to read data from a file named `data.txt` and the contents of the `data.txt` file are given. When this code is executed, the first nine characters including the blanks will be read. Therefore, the output will be This is `a`.

Listing 101.3 : writing a value to a file

```
//Listing 101.3 : Code to open a file in write mode and to write a value to that file

#include <stdio.h>

main()
{

    FILE *fp;
    char d[10];

    fp = fopen("datd.txt", "r+");

    fgets(d, 10, fp);

    puts(d);
}
```

fscanf()

To read formatted data from disk files, you use the `fscanf` function. The first parameter passed to the `fscanf` function is the file pointer associated with the file which data is to be read.

The second parameter is the format specifier, and the third parameter is the variable name. The syntax for reading a string, an integer, and a character is given. When the format specifier is `%s`, the reading stops after a space is encountered.

`int fscanf(FILE *stream, const char *format, variable, …);`

Reading a String	Reading an Integer	Reading a Character

`{`
`...`
`    char c[5];`
`...`
`    fscanf(fp, "%s", c);`
`...`
`}`
	`{`
	`...`
	`    int c;`
	`...`
	`    fscanf(fp, "%d", c);`
	`...`
	`}`
		`{`
		`...`
		`    char c;`
		`...`
		`    fscanf(fp, "%c", c);`
		`...`
		`}`

Finally, you use the `fread` function to read unformatted data. This function is also used to read blocks of data from a binary file. The `fread` function accepts four parameters.

```
int fread(void *ptr, int size, int nitems, FILE *stream);
```

- The first parameter is the address of the structure variable. In other words, it is a pointer to the memory location for storing the data.
- The second parameter defines the size of each block in the data file. The third parameter defines the number of blocks to be read. The fourth parameter is a file pointer associated with the file from which data is to be read.
- The `fread` function reads data into an array pointed to by `&data1`. This function reads only the number of data items specified by the third parameter. Each data item is a sequence of bytes of the length specified by the second parameter.

Listing 101.4 : Reading data from a file using fread function

```
//Listing 101.4 : Code to read data from a file using fread
#include <stdio.h>
struct data
    {
            char c[50];
    };

main()
{
    FILE *fp;
    struct data data1;
    fp = fopen("R1.c", "rb");
    fread(&data1, sizeof(struct data), 1, fp);
    printf("%s", data1.c);
}
```

The reading process stops when an `EOF` is encountered while reading data from the file or when the data items specified in the third parameter have been read.

The position of the file pointer changes once the `fread` function is executed. This function increments the position of the file pointer by the number of bytes read. Therefore, the file pointer will point to the byte following the last byte read. The main difference between the functions available in `stdio.h` for reading data from a disk file is the way in which the functions read data.

The `fgetc` function reads data character by character whereas the `fgets` function continues reading until either the number of characters mentioned or the newline character `\n` or the `EOF` are encountered.

Depending on the format specifier, the `fscanf` function can be used to read both characters and strings. Finally, the `fread` function is useful for reading structures or blocks of data from binary files.

`fgetc()`	Reads characters
`fgets()`	Reads a string of the specified length or terminated by `\n` or the `EOF`
`fscanf()`	Reads an integer, a character, and a string terminated by a space
`fread()`	Reads a block of binary data

In this lesson, you learned about the functions defined in `stdio.h` that are used for reading data from a disk file. Using these functions, you can read data from a file in the preferred manner. Reading data will enable you to access and retrieve information from files.

Self Review Exercise 101.1

1. *Identify the output of the code sample that uses the* `fgetc` *function to read the contents of the* `data.txt` *file. Choose the output.*

```
#include <stdio.h>
main()
{
        FILE *ptr;
        char c;
        int i;
        ptr = fopen("datd.txt", "r+");
        for(i=0; i<=8; i++)
                 C = fgetc(ptr);
        putc(c, stdout);
}
```

Contents of data.txt

This is a sample code.

A. T
B. White Space
C. a
D. h

2. *The partial code for reading data from* `data.dat` *and the contents of* `data.dat` *are given. Choose the option that lists contents read in the buffer when the partial code is executed.*

```
#include <stdio.h>
main()
{
        FILE *fp;
        char d[10];
        fp = fopen("datd.dat", "r+");
        fgets(d, 5, fp);
}
```

Contentsof data.dat

Execute this code.
Use GCC compiler.

A. Exec
B. E
C. Execute this code.
D. Execute this code.
   Use GCC compiler.
E. Execu

3. *Identify the data that is read in the file buffer when the code sample is executed. Choose the correct option.*

```
#include <stdio.h>
main()
{
        char d[10];
        FILE *fp;
        fp = fopen("R1.c", "r");
        fscanf(fp, "%c", c);
}
```

Contentsof data.dat

Hello.
This program reads a file.

A. H
B. Hello
C. e
D. Hello.
   This program reads a file.

Lesson Objectives
In this Lesson you will learn : - *the Genesis of C Language* - *the Stages in the Evolution of C Language*

Disk files are commonly used for making backups of the data stored on a computer. To make a backup, you should be able to write data to a disk file. In C, all the functions related to writing data on a file are prototyped in the predefined header file `stdio.h`.

The functions prototyped in `stdio.h` that are used for writing data to a file are the `fputc`, `fputs`, `fprintf`, and `fwrite` functions. Of these, the first three functions are used for writing to a text file. The last function, which is the `fwrite` function, is used to write to a binary file.

Functions Available in C for Writing Data
fputc()
fputs()
fprintf()
fwrite()

fputc()

To write data to a file, you first open a file and then call a function for writing to the file. To write one character at a time, you use the `fputc` function.

```
int fputc(int c, FILE *fp);
```

This function accepts two parameters.

- The first parameter is a character that has to be written to the file.
- The second parameter is the file pointer associated with the stream of the file opened for writing.
- When `fputc()` is used to write a character, the file pointer advances by `1` byte. For example, in the code displayed, when the file is opened for writing, the file pointer is at the beginning of the file. After the first `fputc()` is executed, the file pointer is positioned at the second byte.

LESSON 102 : WRITING TO A FILE

Listing 102.1 : writing data to a file

```
//Listing 102.1 : Code to write data to a file

#include <stdio.h>

main()
{

    FILE *fp, *ptr;
    char d[10];

    Fp = fopen("sample.c", "r+);

    Ptr = fopen(sample1.c","w");

    fgets(d, 5, fp);
    fputc(d[0], ptr);
    fputc(d[1], ptr);

}
```

Contents of sample.c

Execute This code.

Detailed explanation

- After the second `fputc` function is executed, the file pointer moves by `1` byte. The contents of the `sample.c` file are given. The `fgets` function will read a string of four characters, which is `Exec`, into the `d` array.
- After the first call to the `fputc` function, the first character stored in the array will be written to `sample1.c`. Therefore, the contents of the file will be the letter `E` written at the beginning of the file.
- When the second `fputc` function is executed, `x` is written to `sample1.c`. Therefore, after the complete code is executed, the contents of the file `sample1.c` will be `Ex`.

`fputs()`

Another function used for writing data is the `fputs` function.

```
int fputs(const char *s, FILE *fp);
```

- This function is used to write a string to the file. The `fputs` function accepts two parameters, a pointer to an array and the file pointer. The first parameter, which is a pointer to an array of characters, points to the array that contains the characters to be written to the file associated with the file pointer.
- After the `fputs` function is executed, the file pointer advances by the number of bytes written to the file. An example of the code that uses the `fputs` function to write to the file `out.c` is given below.

Listing 102.4 : writing data to a file

```
//Listing 102.4 : Code to write data to a file in a specified format using format specifiers

#include <stdio.h>
main()
{
    FILE *fp, *ptr;
    char d[10];
    fp = fopen("sample.c","r+");
    ptr = fopen("out.c","w");
    fgets(d,5,fp);
    fprintf(ptr,"%s",d);
}
```

Contents of sample.c
Execute this code

Contents of out.c
Exec

Listing 102.5 : writing data to a file

```
//Listing 102.5 : Code to write data to a file in a specified format using format specifiers

#include <stdio.h>
main()
{
    FILE *fp, *ptr;
    char d[10];
    fp = fopen("sample.c","r+");
    ptr = fopen("out.c","w");
    fgets(d,5,fp);
    fpritnf(ptr,"%s",d[0]);
}
```

Contents of sample.c
Execute this code

Contents of out.c
E

Detailed Explanation

- In the first code sample, `fprintf()` is used to write a string to the file associated with the file pointer `fp`. The `fgets` function in the code will read a string of four characters from the file `sample.c` into array `d`. When `fprintf()` is executed, the string stored in d will be written to `out.c`.
- In the second code sample, the `fprintf` function is used to write a character to the file. When the `fprintf` function is executed, the character stored in the first element of the array, `d[0]`, will be written to the `out.c` file. Therefore, only one character, `E`, is written to the file.

`fwrite()`

Finally, you use the `fwrite` function to write unformatted data to a disk file.

```
size_t fwrite(const void *ptr, size_t, size_t nitems, FILE *stream);
```

This function is used to write structures or blocks of data to a binary file. This function accepts four parameters.

- The first parameter is a pointer to the array that contains the data to be written to the file.
- The second parameter specifies the length of each block of data. The length of a block is equal to the size of the array or the structure pointed to by the first parameter.
- The third parameter is the number of blocks of data to be written, and the last parameter is the file pointer associated with the file on which data has to be written.

Listing 102.2 : writing data to a file

```
//Listing 102.2 : Code to write a string to a file

#include <stdio.h>

main()
{

    FILE *ptr;
    char a[] = ”Hello”;
    char b[] = ”How are you!”;

    ptr = fopen(“out.c”,”w”);

    fputs(a, ptr);
}
```

Contents of out.c

Hello

- After the first `fputs` function is executed, the string stored in array `a`, `Hello`, is written to the file. When the second `fputs` function is executed, the string `b` that reads `How are you!` Is written immediately after `Hello`.

Listing 102.3 : writing data to a file

```
//Listing 102.3 : Code to write a string to a file

#include <stdio.h>

main()
{
    FILE *ptr;
    char a[] = ”Hello”;
    char b[] = ”How are you!”;

    ptr = fopen(“out.c”,”w”);

    fputs(a, ptr);
    fputs(b, ptr);

}
```

Contents of out.c

HelloHow are you!

`fprintf()`

Another important function used to write data is the `fprintf` function.

```
int fprintf(FILE *fp, const char * conversion charcter, variable….);
```

This function is used for writing formatted data to a file. The `fprintf` function accepts three parameters.

- The first parameter is the file pointer associated with the file to which data is to be written. The value of the file pointer is returned by the `fopen` function that is used to open the file for writing.
- The second parameter specifies the data type that is to be written to the file. To write an integer, you use `%d`. To write a character and a string, the conversion characters used are `c` and `s`, respectively. The third parameter, `variable`, stores the data to be written to the file.

The code samples that use the `fprintf` function are given. The data that is written to the `out.c` file is also given.

An example of the `fwrite` function is given below. This function will write the data contained in the structure `student1` to be file pointed to by `fp`.

```
fwrite(&student1, sizeof(struct student),2,fp);
```

The data will be in the form of a block or a sequence of bytes of length specified by the return value of the `sizeof` function. The two data items with a size equal to `struct student` will be written to the file.

The `fwrite` function stops writing when the number of items specified by the third parameter has been written to the file. This function returns the number of items written to the file. The number of bytes written to the file advances the position of the file pointer.

Self Review Exercise 102.1

1. *Identify the contents that will be stored in the file* `out.c` *when the specified code is executed. Choose the correct option.*

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

        char c[100];
        FILE *fp, *ptr;

        ptr=fopen("out.c", "w");
        scanf("%s",c);
        fputc(c[0], ptr);
        fputc(c[0], ptr);
}
```

User Input :
Programming in C.

A. P
B. Programming
C. PP
D. Pr

2. *Identify the contents that will be stored in the file* `out.c` *when the specified code is executed. Choose the correct option.*

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

        char c[100];
        FILE *fp, *ptr;

        ptr=fopen("out.c", "w");
        scanf("%s",c);
        fputc(c[0],ptr);
        fputs(c,ptr);
}
```

User Input :
Programming in C.

A. P
B. Programming
C. PP
D. PProgramming

LESSON 103 : CLOSING A FILE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

After you have performed various file operations, such as reading a file or writing to file, you close the data file. Although most of the C compilers close an open file automatically at the end of program execution, it is a good programming practice to explicitly close a file. To close an open file, you use the `fclose` function.

```
int fclose(FILE *fp);
```

- This function accepts only parameter. Which is the file pointer associated with the open file.
- The stream that was opened by the `fopen` function is closed by `fclose()`. When `fclose()` is executed, the file buffer is flushed. The data in the buffer is transferred to the file, and the block of memory or the buffer associated with the stream is made available by other programs.
- The use of `fclose()` is recommended because most of the operating systems have a limit to the maximum number of files that can be opened at a time. Closing the file when no I/O operations are being performed on it ensures that there is option utilization of memory.

A code sample for reading data from an existing file, sample, and displaying it on the screen is given below. At the end of the program, a call to the `fclose` function closes the sample file.

Listing 103.1 : closing a file

```
//Listing 103.1 : Code to demonstrate closing a file

#include <stdio.h>
main()
{
    FILE *fp;
    char c;
    fp = fopen(“sample”,”r”);
    do
    {
            c=fgetc(fp);
            putc(c,stdout);
    }while(c!=EOF);
fclose(fp);
}
```

Detailed Explanation

- Notice that the value of the parameter passed to the `fclose` function is identical to the value returned by the `fopen` function at the beginning of he program.

You learned about the fclose function used to close a file. A sample code for reading data from `R1.c` and writing it to `W1.c` is given below.

Listing 103.2 : Dealing with file read and write operations

```
//Listing 103.2 : Code to read and write data to a file

#include <stdio.h>

main()
{

    char c;
    FILE *fp, *ptr;

    fp=fopen("R1.c","r");
    ptr=fopen("W1.c","w");

    do
    {
            c=fgetc(fp);
            fputc(c,ptr);
    }while(c!=EOF);

    fclose(fp);
    fclose(ptr);

}
```

Detailed Explanation

- To close the file, you pass the file pointer as a parameter to the `fclose` function. Type the code to close the file to which data is being written.
- The data is being written to the file `W1.c`. To close `W1.c`, the value passed to the `fclose` function is `ptr`. The complete code with `fclose` function statement is given in the example.

In addition to `fclose()`, there is another function available that can be used for closing files. This is the `fcloseall` function. The syntax of the function is:

```
int fcloseall(void);
```

- The `fcloseall` function does not accept any parameter. It closes all open file and flushes the memory area. In other words, all the streams associated with disk files are closed and any data that was stored in the file buffer is transferred to the corresponding files.
- The `fcloseall` function returns the number of files that were closed. An example of the code that uses the `fcloseall` function is given below.

Listing 103.3 : reading and writing data to a file

```
//Listing 103.3 : Code to read and write data to a file

#include <stdio.h>
main()
{
    char c;
    FILE *fp, *ptr;
    fp = fopen(“sample”,”r”);
    ptr= fopen(“newsam”,”w”);

    do
    {
            c=fgetc(fp);
            fputc(c,ptr);
    }while(c!=EOF);

    fcloseall();
}
```

Detailed Explanation

- In the earlier versions of ANSI C, `fcloseall()` was not supported. The compilers that do not support `fcloseall()` generate the error message as displayed. Compilers such as the `MS-VC version 1.5` and later and the Borland C compiler version `3.0` and later support `fclose()`.
- In the listing `103.2`, the first call to the `fclose` function is to close the file from which data is being read and the second call is to close the file to which data is being written. You can replace these two function calls with one call to `fcloseall` as in the Listing `103.2`.
- In addition to the `fclose` and `fcloseall` functions, there are other functions available that flush data while various file operations are being performed.
- Flushing is the process of transferring the contents of the file buffer to the disk file. You can use the function `fflush` and `fflushall` the buffer associated with a file during program execution without disrupting the I/O operation.
- To flush a file buffer during program execution, you use the `fflush` function. This function accepts one parameter, which is the pointer associated with the open file. The buffer of the file associated with the file pointer is flushed.

```
int fflush(FILE *fp);
```

- After the `fflush` function is called, the buffer does not have any data. If there is more than one file open, you can call the `fflushall` function. This function flushes the buffer of all the files open at that point in time.

```
int fflushall(void);
```

- In addition to the `fclose` and `fcloseall` functions, there are other functions available that flush data while various file operations are being performed.

The code to close all files in the `103.4`. To close all open files, the `fcloseall` function is used.

Listing 103.4 : closing all files at once

```
//Listing 103.4 : Code to close all the files.

#include <stdio.h>
main()
{
    char c;
    FILE *fp, *ptr;

    fp=fopen("R1.c","r");
    ptr=fopen("W1.c","w");

    do
    {
            c=fgetc(fp);
            fputc(c,ptr);
    }while(c!=EOF);

    fcloseall();
}
```

LESSON 104 : RANDOMLY ACCESSING A FILE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

For all file-based operations, such as reading, writing, and closing files, file pointers are used. A file pointer is a unique pointer associated with a file that has been opened using the `fopen` function.

After every read operation and write operation, the position of the file pointer changes. For example, When you open a file, the file pointer is at the beginning of the file. After a call to the `fgetc` function or the `fputc` function, the file pointer advances by `1` byte.

Every read operation and write operation is performed at the current position of the file pointer. To access the contents of the file randomly, it should be possible to change the position of the file pointer randomly. To do this, you use the `fseek`, `rewind`, and `ftell` functions.

Random File Access	
`fseek()`	used to place the file pointer at the desired location.
`rewind()`	used to place the file pointer at the beginning of the file.
`ftell()`	used to indicate the current position of the pointer.

The `fseek` function is used to place the file pointer at the desired location. To place the file pointer at the beginning of the file, the `rewind` function is used. The `ftell` function is used to indicate the current position of the pointer.

The `ftell` function is reliable only when used with binary files. This is because a file pointer only keeps track of the number of bytes and does not point to any character in the stream. Binary equivalents of tabs, white spaces, and newline character differ on different machines.

The* `fseek` *function

The syntax for the `fseek` function is given below.

```
int fseek(FILE *stream, long offset, int origin)
```

This function accepts three parameters: a file pointer, an offset, and the origin from where the offset is to be calculated.

- The first parameter, which is a file pointer, is the stream associated with the open file.
- The second parameter is the offset value. This value indicates the number of bytes by which the file pointer should move. This parameter can have both positive and negative values.
- The third parameter indicates the position from where the offset value is to be calculated. The offset value can be calculated from the current position of the file pointer or the beginning or the end of the file.

- The third parameter, origin, can have three values: `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`. These values are defined in the header file `stdio.h` by using `#define` directives. The corresponding integer values are `0`, `1`, and `2`, respectively.

	Integer equivalent
SEEK_SET	0
SEEK_CUR	1
SEEK_END	2

- In the `fseek` function, you can use either the `#define` values or their integer equivalents to indicate the position from where the offset values should be calculated. If the value is `SEEK_SET`, the offset will be calculated from the beginning of the file.
- When the value of the third parameter is `SEEK_END`, the offset is to be calculated from the end of the file. The final position of the file pointer is equal to the `EOF` plus the offset value. This allows you to write beyond the `EOF` if the offset is positive.
- Finally, if the value of the third parameter is `SEEK_CUR`, the offset value is to be calculated from the current position of the file pointer. The file pointer is set at a position obtained by adding the offset value to the current location of the file pointer.

A code sample that uses the `fseek` functions to place the pointer 8 bytes from the beginning of the file is given below. When the file is opened, the file pointer is placed at the beginning of the file, which is the first byte. The first call to the `fgetc` function will read the first character of the file.

Listing 104.1 : Demonstrating fseek function

```
//Listing 104.1 : program to demonstrate fseek function.

#include <stdio.h>
main()
{
    FILE *fp;
    int k;
    char a;

    fp=fopen("sample2.c","r");
    fgetc(fp);

    fseek(fp,8,SEEK_SET);

    a=fgetc(fp);

    fclose(fp);
}
```

Contents of sample2.c
This is a sample script.
This is to test the functionality of the C programs.

Detailed Explanation

- The `fseek` function will place the file pointer `8` bytes from the beginning of the file. Therefore, the `fgetc` function, which is called after the `fseek` function, will read the ninth character, `a`, and assign this value to variable `a`. this function call will also advance the file pointer by `1` byte.
- In the second code sample given below, variable `c` will again store the value `a`. The first call to the `fgetc` function will read the first character of the file and place the file pointer on the second byte.
- The `fseek` function will place the file pointer `7` bytes ahead of its current location. The second call to the `fgetc` function will read the ninth character, `a`, and assign it to variable `c`. Therefore, the actual position of the file pointer depends on both the origin and the offset value.
- The second parameter, which is the offset value, can have both negative and positive values. Negative values are used to move the file pointer in the reverse direction.

A code sample to read a string from the `data.txt` file and then reposition the file pointer at the beginning of that string is given below.

Listing 104.2 : working with fseek

```
//Listing 104.2 : program to read a string from a file and repositioning the file pointer
// at the beginning of the string

#include <stdio.h>
main()
{
    FILE *fp;
    int i;
    char c[50];
    char a;

    fp=fopen("data.txt","r+");
    fscanf(fp,"%s",c);
    printf("%s\n",c);

    i=strlen(c);

    fseek(fp,-i,SEEK_CUR);

    a=fgetc(fp);
    putc(a,stdout);

    fclose(fp);
}
```

Contents of data.txt
This is a sample code.

Detailed Explanation

- Using a negative offset value with the origin as `0` or `SEEK_SET` does not place the file pointer before the beginning of file `(BOF)`. When the displayed code sample is executed, the contents of the `Position.c` file do not change because the write operation is not performed.

Listing 104.3 : working with fseek

```
//Listing 104.3 : program to work with fseek

#include <stdio.h>
main()
{
    FILE *fp;
    char k[]="Hello";

    fp=fopen("Position.c","r+");

    fseek(fp,-10,SEEK_SET);

    fprintf(fp,"%s",k);
    fclose(fp);
}
```

- The write operation is not performed because the file pointer cannot be placed `10` bytes before `BOF`. In other words, `BOF` cannot be redefined.

In the listing `104.4`, the `fseek` function is used to place the file pointer at a specific position and then write the string `Hello`. In the first code sample, the string will overwrite the text that was initially `10` bytes before the `EOF`.

Listing 104.4 : working with fseek

```
//Listing 104.4 : program to work with fseek

#include <stdio.h>
main()
{
    FILE *fp;
    char k[]="Hello";

    fp=fopen("Position.c","r+");

    fseek(fp,-10,SEEK_END);

    fprintf(fp,"%s",k);
    fclose(fp);
}
```

The content before the specified code is executed.

The contents of this file would change.

The content after the specified code is executed.

*The contents of this file wou**Hello**ange.*

In the listing `104.5`, the specified string, `Hello`, will be written `10` bytes after the `EOF`. This will change the `EOF`. At times, the output may display junk character in these `10` bytes.

Listing 104.5 : working with fseek

```
//Listing 104.5 : program to work with fseek

#include <stdio.h>
main()
{
    FILE *fp;
    char k[]="Hello";

    fp=fopen("Position.c","r+");

    fseek(fp,10,SEEK_END);

    fprintf(fp,"%s",k);
    fclose(fp);
}
```

The content before the specified code is executed.

*The contents of this file wou**ld ch**ange.*

The content after the specified code is executed.

The contents of this file would change.   Hello

The second function that can be used to change the position of the file pointer is the rewind function.

```
void rewind(FILE *stream);
```

- This function accepts only one parameter, which is the file pointer to be repositioned.
- The rewind function place the file pointer at the beginning of the file, regardless of its current position.
- The rewind function is equivalent to the seek function in which the offset is to be calculated from the beginning of the file and the offset value is `0`.

```
Rewind(fp);            Equivalent To            fseek(fp,0,SEEK_SET);
```

In the code sample given below, the contents of a file are read and all the occurrences of the word would should. To position the file pointer at the required position, `fseek()` is used. To place the file pointer at the beginning of the file, `rewind()` is used.

Listing 104.6 : working with fseek and rewind

```
//Listing 104.6 : program working with fseek and rewind

#include <stdio.h>
main()                                          Contents of sample.c
{
    FILE *fp;                                   The contents of this file would change.
    char word[20];
    char k[]="should";
    char c;
    fp=fopen("sample.c","r+");
    printf("\n Original contents of the file:\n");

    do
    {
            c=fgetc(fp);     //Code to display the original contents of the file
            putc(c,stdout);
    }while(c!=EOF);

    rewind(fp);

    do
            fscanf(fp,"%s", word);    //Code to replace would
    while(strcmp(word,"would"));

    fseek(fp,-5,1);

    fflush(fp);
    fprintf(fp,"%s",k);

    rewind(fp);

    printf("\nNew contents of the file"\n"); //Code to display the new contents of the file

    do
    {
            c=getc(fp);
            putc(c,stdout);
    }while(c!=EOF);

    fclose(fp);
}
```

In the output of the program, notice that "`would`" in the original file has been replaced by "`should`".

Finally, the `ftell` function is used to obtain the current value of the file pointer.

```
long ftell(FILE *fp);
```

- This function accepts only one parameter, which is the file pointer.
- The `ftell` function can be used reliably only with binary files. Therefore, the use of the `fseek` function is recommended to calculate offsets. Arithmetical operations on an offset value returned by the `ftell` function may not be meaningful.

For example, a code sample that uses the offset returned by the `ftell` function to find relative positions may provide reliable results on a UNIX system but may not provide equally reliable results on a non -UNIX system where the offset is not measured in bytes.

Listing 104.7 : working with ftell function

```
//Listing 104.7 : program to work with ftell function

#include <stdio.h>
main()
{
    FILE *fp;
    int i,j;
    char c;
    fp=fopen("data.txt","r+");

    do
    {
            c=fgetc(fp);
            putc(c,stdout);
            j=ftell(fp);
    }while(c!=EOF);

fclose(fp);
}
```

The code sample given below uses the `fseek` and rewind functions to randomly access a file. This code sample reads the data at the position specified by a user, displays the data, and then places the file pointer the file pointer at `BOF`. To end the program, you type the number `0`.

Listing 104.8 : working with fseek and rewind

```
//Listing 104.8 : program to work with fseek, rewind

#include <stdio.h>
main()
{
    FILE *fp;
    int i,j;
    char c;
    fp=fopen("data.txt","r+");

    do
    {
            printf("\nType the position from where the data is to be read:\n");
            scanf("%d",&i);
            fseek(fp,i-1,SEEK_SET);
            c=fgetc(fp);
            putc(c,stdout);
            rewind(fp);
    }while(i!=0);
    fclose(fp);
}
```

For example, if the value entered by the user is `7`, the seventh element in the file will be read and displayed. The `rewind` function places the file pointer at the beginning of the file.

Self Review Exercise 104.1

1. *Identify the* `fseek` *function in which the offset value is calculated from the beginning of the file*

   A. `fseek(fp,10,SEEK_CUR)`
   B. `fseek(fp,10,1)`
   C. `fseek(fp,10,SEEK_END)`
   D. `fseek(fp,10,SEEK_SET)`

2. *You learned to change the position of the file pointer by using the* `fseek` *function. Identify the value stored in variable a after the specified piece of code is executed. Choose the option that lists the correct answer.*

```
#include <stdio.h>
main()
{
        FILE *fp;
        int k;
        char a;
        fp=fopen("sample3.c","r");
        for(k=0;k<5;k++)
        fgetc(fp);
        fseek(fp,5,1);
        a=fgetc(fp);
        printf("%c",a);
        fclose(fp);
}
```

Contents of sample3.c
You are testing sample C program.

   A. s
   B. r
   C. i
   D. o

3. *Identify the set of function calls that will place the file pointer* 4 *bytes from the beginning of the file. Choose the correct option.*

`fp = fopen("sample.c", "r+b");`
`a = fgetc(fp);`
`fseek(fp, 8, SEEK_CUR);`
`a = fgetc(fp);`
`fseek(fp, -5, SEEK_CUR);`
A
	`fp = fopen("sample.c", "r+b");`
	`fseek(fp, 8, SEEK_SET);`
	`a = fgetc(fp);`
	`rewind(fp);`
	`fseek(fp, 4, SEEK_CUR);`
	B

`fp = fopen("sample.c", "r+b");`
`fseek(fp, 8, SEEK_SET);`
`a = fgetc(fp);`
`fseek(fp, -6, SEEK_CUR);`
`rewind(fp);`
C
	`fp = fopen("sample.c", "r+b");`
	`a = fgetc(fp);`
	`fseek(fp, 8, SEEK_SET);`
	`rewind(fp);`
	`fseek(fp, 2, SEEK_CUR);`
	D

LESSON 105 : MOVING A FILE

Lesson Objectives
In this Lesson you will learn : - *the Genesis of C Language* - *the Stages in the Evolution of C Language*

In addition to reading and writing to and from files, you can also perform certain file management functions, such as moving, copying, deleting, and renaming files. There is no special function available in c for moving a file. To move a file, you first copy the file. Next, delete the original file. Finally, rename the new file, if required.

Steps for Moving a File
Copy the file.
Delete the original file.
Rename the copied file.

The first for moving a file is to copy it to the directory or the folder. In C, there is no special function for copying files. A combination of the functions used for reading and writing data to and from files can be used to copy a file.

A code sample for copying a file is given below. First, you open the file to be copied and the destination file in the read and write modes, respectively. Opening the files in binary modes ensures that the code works for both text and binary files.

Listing 105.1 : Copying a file

```
//Listing 105.1 : program to copy a file.

#include <stdio.h>
main()
{
    FILE *fp, *ptr;
    char c;
    fp=fopen(“C:\\Cprograms\\sample.c”,”rb”);
    ptr=fopen(“D:\\sample2.c”,”wb”);
    do
    {
            c=fgetc(fp);
            fputc(c,ptr);
    }while(c!=EOF);
    fclose(fp);
    fclose(ptr);
}
```

Detailed Explanation

- Instead of specifying only file names, the completer paths of the original and destination file are specified in the `fopen` function. This is because files are usually moved from one directory or one drive to another.
- In the `do-while` loop, read the contents of the file character by character and copy them to the destination file until the end of file is reached.

- Finally, you call the `fclose` function to close both the files. Executing this piece of code will copy the contents of `sample.c` to `sample.2.c`.

The next task is to delete the original file. To do this, you use the remove function. This function deletes the file specified by the file name that is passed as a parameter to the function.

```
int remove(const char *filename);
```

The file that is deleted using the `remove` function cannot be recovered. Instead of specifying only the file name, you can pas the complete path along with the file name as a parameter to the `remove` function.

If you specify only the file name instead of the complete path as the parameter to the `remove` function, then the file is searched in the current directory. If the file exists, it is deleted. Otherwise, an error is generated depending on the operating system used.

```
remove(“sample.c”);
```

The piece of code that can be used to delete the original file, `sample.c`, stored in the C programs folder on drive C is as follows :

Listing 105.2 : deleting a file

```
//Listing 105.2 : program to delete a file.

#include <stdio.h>
main()
{
    FILE *fp, *ptr;
    char c;

    fp=fopen(“C:\\Cprograms\\sample.c”,”rb”);

    ptr=fopen(“D:\\sample2.c”,”wb”);

    do
    {
            c=fgetc(fp);
            fputc(c,ptr);
    }while(c!=EOF);

    fclose(fp);
    fclose(ptr);

    remove(“C:\\Cprograms\\sample.c”);
}
```

Finally, you rename the new file. The `rename` function, which is available in the header file `stdio.h`, is used for renaming files. This function accepts two parameters. The first parameter is the existing name of the file, and the second parameter is the new name of the life. Similar to the `remove` function, the `rename` function can also accept the complete paths of files as parameters.

```
int rename(const char *from, const char *to);

     int rename(path from, path to);
```

For example, to change the name of the file `sample2.c` to `sample.c`, follow the given code in listing `105.3`. The statement to rename the file is given. If you specify only the file name instead of the complete path as a parameter to the `rename` function, then the file is searched in the current directory.

Listing 105.3 : renaming a file

```
//Listing 105.3 : program to rename a file

#include <stdio.h>
main()
{

  FILE *fp, *ptr;
  char c;

  fp=fopen("C:\\Cprograms\\sample.c","rb");

  ptr=fopen("D:\\sample2.c","wb");

  do
  {
          c=fgetc(fp);
          fputc(c,ptr);
  }while(c!=EOF);

  fclose(fp);
  fclose(ptr);

  remove("C:\\Cprograms\\sample.c");
  rename("D:\\sample2.c","D:\\sample.c");


}
```

In certain, systems, such as UNIX, if a file with the new name already exists, it is overwritten.

```
int rename(const char *from, const char *to);
```

Overwrites the Existing File

LESSON 106 : STANDARD I/O --> USING FILE I/O FUNCTIONS

Lesson Objectives
In this Lesson you will learn :   - *the Genesis of C Language*   - *the Stages in the Evolution of C Language*

In C, all the input and output devices are treated as files. Therefore, the function used to read and write data to and from disk files can also be used to read and write data to and from the standard I/O devices, such as the keyboard and the monitor.

To perform file I/O functions, file pointer are used. The file pointers that are associated with standard I/ O device are predefined in the header file `stdio.h`. The file pointer associated with the standard input device, which is the keyboard, is `stdin`.

Similarly, `stdout` is associated with the standard output file which is the monitor. In addition to `stdin` and `stdout`, there is another file pointer, `stderr`, which associated with the standard error file. By default, the standard error file is the monitor of the computer.

File Pointer Associated with Standard Input Devices	`stdin`
File Pointer Associated with Standard Output Devices	`stdout`
File Pointer Associated with Standard Error File	`stderr`

You can use file I/O functions to accept data from the keyboard and write data to the monitor. Instead of a file pointer, `stdin` or `stdout` is passed as a parameter to these functions.

For example, the use of `fputs` function is given below. The second parameter passed to the `fputs` function is `stdout`, which is the pointer associated with the standard output device.

Listing 106.1 : using file i/o functions

```
//Listing 106.1 : program using file i/o functions

#include <stdio.h>

main()
{
    FILE *ptr;

    char a[]="Hello.";
    char b[]="How are you!";

    fputs(a,stdout);

    fputs(b,stdout);
}
```

In this situation, the `fputs` function works like the `puts` function. Similarly, the `fputc` function can be made to work like the `putc` function, the output of the `fputc` function is the same as that of the `putc` function.

Listing 106.2 : using file i/o functions

```
//Listing 106.2 : program using file i/o functions

#include <stdio.h>
main()
{
    FILE *ptr;

    char a[]=”Hello.”;
    char b[]=”How are you!”;

    fputc(a,stdout);

    putc(b,stdout);
}
```

You can accept data from the keyboard by using function such as `fgetc`, `fgets`, and `fscanf`. In these functions, `stdin` should be passed as a parameter instead of a file pointer.

Standard I/O Functions	*Equivalent File I/O Functions*
getc();	fgetc(stdin);
gets(string);	fgets(string, length of the string, stdin);
putc(char, stdout);	fputc(char, stdout);
scanf(data type, variable);	fscanf(stdin, data type, variable);

A code sample in listing `106.3` that uses the `fscanf` function to accept data from the keyboard is given below. The second code sample uses the `scanf` function to perform the same task.

Listing 106.3 : using file i/o functions

```
//Listing 106.3 : program using file i/o functions

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

    fscanf(stdin, “%d”, &x);

    fprintf(stdout, “%d”, x);
}
```

Listing 106.4 : using file i/o functions

```
//Listing 106.4 : program using file i/o functions

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

    scanf(“%d”, &x);

    printf(“%d”, x);
}
```

Self Review Exercise 106.1

1. *Identify the code to perform I/O operation from standard devices by using disc file I/O functions.*

A. `fscanf(stdout,”%s”,c);`
   `printf(“%s”,c);`

B. `scanf(“%s”,c);`
   `fprintf(stdin,”%s”,c);`

C. `fscanf(stdin,”%s”,c);`
   `fprintf(stdout,”%s”,c);`

D. `scanf(“%s”,c);`
   `printf(“%s”,c);`

LESSON 107 : INPUTS FROM THE COMMAND LINE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Function calls in C allow you to pass parameters to a function depending on the function declaration. Like all other functions, you can also pass parameters to the `main` function, which indicates the beginning of a C program.

Passing parameters to the `main` function is a very common method of passing parameters directly from the operating system to a program.

Most C compilers allow two parameters to be passed to the `main` function. Conventionally, the names of the parameters passed to the `main` function are `argc` and `argv`. The values of the parameters `argc` and `argv` are passed during program execution from the command line. Therefore, these parameters are as command line parameters.

The code sample below on the screen accepts the parameters passed to the `main` function. The first parameter, `argc`, is an integer variable. The second parameter, `argv`, is an array of strings.

Listing 107.1 : command line inputting

```
//Listing 107.1 : program to take inputs from command line

#include <stdio.h>

main(int argc, char *argv[])
{

    FILE *fp, *ptr;
    char c;
    fp=fopen(argv[1],"r");
    ptr=fopen(argv[2],"w");
    do
    {
            c=fgetc(fp);
            fputc(c,ptr);
    }while(c!=EOF);

    fclose(fp);

    fclose(ptr);
}
```

Detailed Explanation

- The value of the first parameter, `argc`, indicates the number of words in the command line. This value is not passed from the command line. Instead, it is assigned internally depending on the total number of words in the command line.
- The integer value stored in the variable `argc` is one more than the total number of parameter passed to the `main` function. This is because the name of the execution file is also counted.

- The second parameter, which is an array of stings, stores the value passed from the command line. For example, `argv[0]` stores the first parameter, which is the command name or the name of the executablefile. The value of the second parameter is stored in `argv[1]`. The command line syntax is used for executing a program in which parameters are passed to the `main` function. Every other string on the command line is a parameter.

An example to read the contents of a file and display them on the screen is given below. In this example, the name of the file to be read is passed as a parameter to the `main` function. Notice that instead of the file name, `arg[1]` is passed as the parameter to `fopen()`.

If the executable file name is `fileread` and the name of the file to be read is `sample.c`, then the command line for executing the code will be `fileread sample.c`. In this situation, the value stored in `argc` is `2` because `argv` stores two strings.

- The first string in the command line, which is the name of the executable file `fileread`, is stored in `argv[0]`. The value of the second string, which is the name of the file to be read, is stored in `argv[1]`.
- Even when you pas more than one string to the `main` function, the declaration for the `main` function remains the same. All parameters are stored in the array `argv[]`. Notice that spaces are used to separate command line parameters.

Consider the example that is used to display the contents of a specified file. If more than one file name is specified in the command line, no error is generated. However, only the contents of the file specified by the first file name are displayed in the output.

Listing 107.2 : command line inputting

```
//Listing 107.2 : program to take inputs from command line

#include <stdio.h>

main(int argc, char *argv[])
{
    FILE *fp;
    char a;

    fp=fopen(argv[1],"r");
    do
    {
            a=fgetc(fp);
            putc(a,stdout);
    }while(a!=EOF);

    fclose(fp);
}
```

Self Review Exercise 107.1

1. *What does* `argv[0]` *point to?*

   A. Words in the command line
   B. Name of the C source file
   C. Name of the executable file
   D. Number of parameters

LESSON 108 : ENVIRONMENT VARIABLE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

An environment variable can be defined as a string that consists of environment information, such as the current drive and the path of the Temp folder. An important feature of these variables is that their values remain the same throughout a session.

Environment variables are important because some of these affect program execution. Consider an example in which a C program, which is to be executed in the MS-DOS environment, requires file to be extracted in the Temp folder on drive C.

The program is designed s that it extracts a file into a temporary directory. If you hard code the destination directory as C:\Temp, it may create problems because the temporary directory may exist on a different drive depending on the installation of he operating system.

To avoid errors during program execution, you use an environment variable called TEMP or TMP that contains the path of a temporary directory. The value stored in this variable remains the same throughout the session.

```
main(int argc, char *argv[], char *env[]);
```

You can display environmental variables by using a parameter of the `main` function. By convention, this parameter is called `env` to indicate environment variables. The syntax of the `main` function declaration with the third parameter is given below. The parameter `env` is an array of strings. This array stores the current values of all environment variables.

The following example can be used to display the current values of environment variables. In this example, the contents of the array `env` are displayed.

Listing 108.1 : working with environment variables

```
//Listing 108.1 : this program deals with command line and environment variables

#include <stdio.h>

main(int argc, char *argv[], char *env[])
{

    int i;

    for(i=0;env[i]!=NULL;i++)
            {
                    printf("%s\n", env[i]);
            }
}
```

The result of the execution of the piece of code in a windows system is given. The value to the left of the equal to sign is the name of an environment variable, and the string to the right of the equal to sign is the value of the environmental variable.

Some of the environment variables displayed are the path of the TEMP directory, the current PROMPT being used, the PATH use for executing an exe file, and the command line value, represented by the variable CMDLINE, last executed.

On a UNIX system, the values of environment variables, such as USER, SHELL, HOME, and TERM will be displayed.

```
ENV=/Capri_1/john/.env
MOZILA_HOME=/usr/local/netscape/
USER=john
SHELL=/sbin/sh
HOSTTYPE=sparc
OSTYPE=solaris2.6
HOME=/Capri_1/john
TERM=vt100
```

```
char *getenv(const char *_name);
```

In addition to using the third argument passed to the main function, you can also access environment variables by using the getenv function of C library. The getenv function is prototyped in the header file stdib.h. This function accepts the name of an environment variable as a parameter and returns its value.

By convention, the names of environment variables are capitalized. Therefore, when you pass these names as parameters to the getenv function, they should be capitalized. Otherwise, the output will be NULL.

```
getenv("SHELL");
```

The example below displays the value of the environment variable PROMPT.

Listing 108.2 : working with environment variables

```
//Listing 108.2 : this program deals with command line and environment variables

#include <stdio.h>
#include <stdlib.h>

main()
{

    printf("%s\n", getenv("PROMPT"));

}
```

If the string passed to the getenv function is not an environment variable, the value returned by the function is NULL.

Listing 108.3 : working with environment variables

```
//Listing 108.3 : this program deals with command line and environment variables

#include <stdio.h>
#include <stdlib.h>

main()
{

    printf(“%s\n”, getenv(“Prompt”));

}
```

Listing 108.4 : working with environment variables

```
//Listing 108.4 : this program deals with command line and environment variables

#include <stdio.h>
#include <stdlib.h>

main()
{

    printf(“%s\n”, getenv(“PROMPT”));

}
```

Listing 108.5 : working with environment variables

```
//Listing 108.5 : this program deals with command line and environment variables

#include <stdio.h>
#include <stdlib.h>

main()
{

    printf(“%s\n”, getenv(“CORRESP”));

}
```

The first and second code samples generate the `output(null)` because the names of environment variables are not capitalized. The output of the third code sample is `(null)` because `CORRESP` is not an environment variable.

You can also display environment variables by using the `printenv` command on a `UNIX` system in the C shell. In MS-DOS and the `Bourne shell` on a `UNIX` system, the set command is used.

printenv()	*Used in UNIX*
set Command	*Used in MS-DOS*

Self Review Exercise 108.1

1. *Identify the correct prototype of the main function to be used for displaying environment variables.*

   A. `main(int argc, char *argv[], char *env[])`
   B. `main(int argc, char *argv[], char *env)`
   C. `main(int argc, char *argv[], char enb[])`
   D. `main(int argc, char *argv[], int *env[])`

2. *Identify the correct code to display the value of the environment variable* `PATH`.

   A. `printf(“%s”, getenv(“PATH”);`
   B. `printf(“%s”, getenv(“All”);`
   C. `printf(“%s\n”, getenv(env[i]));`
   D. `printf(“%s\n”, env[1]);`

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

While programming in C, you can use functions that directly interact with the operating system. Such functions are called system calls. An important advantage of using system calls in a program is that the operations related to the operating system can be performed from within the program. For example, you can invoke processes or commands within a C program.

For example, on a UNIX system, you want to stop the processing of a program for 10 seconds after processing a module. You can do this by using the sleep system call within the program. Similarly, you can use the read system call to read data. A system call is dependent on the operating system being used. A UNIX system call will not work in the MS-DOS environment.

Operating system commands can be executed from within a C program by using the system function. This is a library function that accepts the name of the command as a parameter.

The prototype of this function is included in the header file stdlib.h.

```
system(“command”);
```

For example, you can use the date command from a C program by passing the date as a parameter to the system function. Similarly, the name of an executable file that is coded in a language other than C can also be passed as a parameter to the system function.

```
system(“date”);
```

Using the system function in your program allows you to perform function such as formatting a disk or invoking an editor without quitting the program. An example for invoking the UNIX editor is given.

```
}
.....
      Char call[MAXSTRING]

      sprintf(command, “vi%s”, argv[1];
      printf(“vi on the file %s is coming u0...\n”, argv[1]);
      system(command);
.....
}
```

A call to the system function requires more memory. When these calls are executed, the program calling these functions remains loaded in RAM and a new copy of the operating system command processor is loaded. An error message is generated if memory is insufficient.

The second advantage of using system calls is that the system calls are language-independent. For example, if the system call to display the current date is date, then any language on that operating system can use the date system call and will obtain the same output.

For example, all read operations on a UNIX system use the read system call. Therefore, if the read function is used in any code, it will perform the read operation, regardless of the language being used.

Using system calls has a disadvantage. The program that uses system calls is not portable to other operating systems. This is because system calls are highly dependent on the operating system.

Self Review Exercise 109.1

1. *Identify the function available in C that is used to execute the operating system command from a C program.*

   A. `read()`
   B. `sleep()`
   C. `system()`
   D. `date()`

2. *Identify an advantage of using system calls in a C program.*

   A. Invokes an executable file coded in a programming language other than C
   B. Invokes a DOS system call from a UNIX server
   C. Increases code portability to other operating systems
   D. Decreases the response time of the code

LESSON 110 : INTERRUPTS

Lesson Objectives
In this Lesson you will learn : • *the Genesis of C Language* • *the Stages in the Evolution of C Language*

On modern computers when program execution starts, instructions are executed until either all instructions are executed or an interrupt occurs. An interrupt is a signal from a device attached to a computer or from a program within the computer. This signal stops the execution of the main program.

When the processor receives an interrupt, it suspend its current operations, saves the status of its work, and transfers control to a special routine known as an interrupt handler. An interrupt handler is a set of instructions that are executed when a particular type of interrupt occurs.

There are mainly two types of interrupts, hardware interrupts and software interrupts. Hardware interrupt are generated from hardware devices, such as a keyboard, a mouse, a printer, and a hard disk, interrupts are generated by hardware devices to request service or report problems.

Sources of Hardware Interrupts
Keyboard
Mouse
Printer
Hard Disk

For example, an interrupt from the auxiliary storage device to indicate that the date transfer is complete will be a hardware interrupt. Similarly, interrupts from a printer will also be hardware interrupts. Software interrupts are generated due to software factors, such as memory errors and illegal instructions, which include division by the number 0. Software interrupts can also be generated by the processor itself responsible to application errors or requests for operating system services.

Sources of Software Interrupts
Memory Errors
Illegal Instructions
Application Errors
Requests for Operating System Services

Consider an example in which an interrupt is generated when a page fault occurs while swapping pages in the memory, interrupts of this type are software interrupts. Software interrupts are extensively used for multitasking. A processor can execute only one computer instruction at a time. For multitasking, processors are interrupted after fixed time spans. As a result, various programs are executed serially.

The first parameter is the number that is called when the `int86` function is executed. The second and third parameters are union `REGS`, which are defined in the header file, `dos.h`. The second parameter specifies the registers to be used for passing information to the interrupts handler. Finally, the third parameter specified the registers used for storing the values returned by the interrupt handler.

The partial code that uses the `int86` function to clear the screen is displayed, in addition to the `int86` function, there are certain other functions, such as `intdos`, `intdosx`, and `int86x`, which are used to execute MS-DOS interrupts.

Listing 110.1 : illustrating interrupts

```
// Listing 110.1 : This program illustrates the concept of interrupts

#include <dos.h>

void cls(void)
{
    union REGS r;

    r.h.ah=6;
    r.h.al=0;
    r.h.ch=0;
    r.h.cl=0;
    r.h.dh=24;
    r.h.dl=79;
    r.h.bh=7;
    int86(0x19, &r, &r);
}

main()
{
    cls();
}
```

Self Review Exercise 110.1

1. *Identify the situation in which hardware interrupts will be generated.*

   A. Changing screen modes during program execution
   B. Clearing the screen by using the cls command within a program
   C. Dividing an integer by the number 0
   D. Halting program execution by using a combination of the Ctrl and C keys

2. *Identify the situation that will generate a software interrupt.*

   A. An interrupt signal from the floppy drive to indicate that the floppy is full
   B. Using the Escape key to stop a program
   C. Division by the number 0
   D. Signal from the storage device to indicate start of data transfer

REVIEW EXERCISE

1. Complete the code to move a file. The code for copying the contents of the file `data.txt` *to the file* `data2.txt` *is given. Write the code to delete the original file.*

```
#include<stdio.h>
main()
{
        FILE *fp, *ptr;
        char c;
        fp = fopen("data.txt", "rb");
        ptr = fopen("data2.txt","wb");
        do
        {
                c = fget(fp);
                fputc(c, ptr);
        }while(c != EOF)
        fclose(fp);
        fclose(ptr);
        //Enter here
        //Enter here
}
```

2. The code for copying the contents of the file `data.txt` *to the file* `data2.txt` *and deleting the original file is given. Write the code to rename the new file with the original file name.*

3. Chose the code sample to open the file `sample.txt` *in the read only mode.*

A.
```
#include<stdio.h>
main()
{
    FILE *ptr;
    ptr = fopen("sample.txt", "r+");
    printf("\n File is Open");
}
```

B.
```
#include<stdio.h>
main()
{
    ptr = fopen("sample.txt", "r+");
    printf("\n File is Open");
}
```

C.
```
#include<stdio.h>
main()
{
    FILE *ptr;
    ptr = fopen("sample.txt", "r");
    printf("\n File is Open");
```

D.
```
#include<stdio.h>
main()
{
    FILE *ptr;
    ptr = fopen("sample.txt", "r+b");
    printf("\n File is Open");
```

4. Select the advantages of using a system call in a C program.

A. Increases compiler dependencies.
B. Increases portability across different OSs
C. Allows direct access to operating system commands
D. Can be used in any programming language
E. Optimizes memory utilization

5. Four options are given here. Identify the option in which a software interrupt is generated. Tick the correct option.

A. A back-end process of copying the data from the memory to the disk drive is being carried out. An interrupt is generated when the disk runs out of space.
B. While executing a module, an interrupt is generated to indicate memory error.
C. A back-end process of printing is being carried out. An interrupt is generated when the printer completes the task.
D. A user presses the Esc key to quit a program.

6. A code sample for reading and writing data to and from a file is given. The file has 100 *characters. Choose the option that lists the offset position of the file pointer, which is measured in bytes, after the code to read data is executed.*

```
#include<stdio.h>
struct data {char c[6];};
main()
{
       FILE *fp;
       struct data data1;
       char c;
       int k;
       fp = fopen("sample.c", "r");
       fread(&data1, sizeof(struct data), 1, fp);
}
```

A. Byte at the position zero
B. Sixth byte
C. Seventh byte
D. Fifth byte

7. Identify the offset position of the file pointer, which is measured in bytes, after the code to read data is executed. Choose the option that lists the correct offset position.

```
#include<stdio.h>
main()
{
       FILE *fp;
       int k;
       char c[10], a;
       fp = fopen("sample.c", "r");
       fgets(c, 10, fp);
}
```

A. Byte at the position zero
B. Eleventh byte
C. Tenth byte
D. Ninth byte

8. A code sample to read the contents of the file `sample2.c` *is given. Identify the offset position of the file pointer after all statements for reading data and repositioning the file pointer are executed. Tick the option that lists the correct offset position.*

```
#include<stdio.h>
main()
{
       FILE *fp;
       int k;
       char c;
       fp = fopen("sample2.c", "r");
       fgetc(fp);
       fseek(fp, -5, SEEK_END);
       c = fgetc(fp);
       printf("%c", c);
}
```

A. Fifth byte
B. Fifth byte from the EOF
C. Fourth byte from the EOF
D. Sixth byte from the EOF

REVIEW EXERCISE

9. Identify the offset position of the file pointer, which is measured in bytes, after all statements for reading data and repositioning the file pointer are executed. Tick the option that lists the correct offset position.

```
#include<stdio.h>
main()
{
        FILE *fp;
        int k;
        fp = fopen("sample.c", "r");
        fgetc(fp);
        rewind(fp);
        fseek(fp, 5, SEEk_CUR);
}
```

A. Fifth byte
B. Sixth byte
C. First byte
D. Seventh byte

10. Choose the option I which a hardware interrupts is generated.

A. While executing a module, an interrupt is generated to indicate an error in the power supply.
B. While executing a module, an interrupt is generated to indicate a memory error.
C. A back-end process of calculating averages is being executed. An interrupt is generated when the program executes a statement that involves division by the number 0.
D. While swapping pages in the memory, page faults occur, which generate interrupts.

11. The program `emp` *accepts employee names and displays the names and the value stored in* `argc`*. Using the* `emp` *program, display the names* `Tom`, `Ken`, `Larry`, *and* `Don` *on the screen in the same order. Write the command line statement.*

```
#include<stdio.h>
main(int argc, char *argv[])
{
        int cpunt;
        printf("\nNo of employees: \t%d\n", argc-1);
        for(count = 1;count < argc;count++)
                printf("argv[%d] = %s\n", count, argv[count]);
}
```

12. Identify the code to perform I/O operations from standard devices by using file I/O functions.

A. fprintf(stdin,"%",c); fscanf(stdout,"%",c);
B. fprintf(stdout,"%",c); fscanf(stdin,"%",&c);
C. printf(stdin,"%".c); scanf(stdout,ptr,c);
D. printf(stdin,"%",c); scanf("%",&c);

13. The contents of the file `R1.c` *and the code used for reading the file are showed. Choose the option that shows the contents of the file buffer after the code to read the file is executed.*

```
#include<stdio.h>
main()
{
        char c;
        FILE *fp;
        fp = fopen("R1.c", "r");
        c = fgetc(fp);
        printf("%c", c);
}
```

```
Contents of R1.c
she sells
seashells on
the seashore
```

A. `S`
B. `she`
C. `she sells`
D. `she sells seashells on the seashore.`

14. Choose the option that shows the output of the file read operation.

```
#include<stdio.h>
main()
{
        char c[100];
        FILE *fp;
        fp = fopen("R1.c", "r");
        fgets(c, 100, fp);
        printf("%s", c);
}
```

```
A.  She      B.  s        C.  she sells    D.  she sells
                                               seashells
                                               on the seashore.
```

15. Choose the option that displays the output of the file read operation.

```
#include<stdio.h>
main()
{
        char c[100];
        FILE *fp;
        fp = fopen("R1.c", "r");
        fscanf(fp, "%s", c);
        printf("%s", c);
}
```

```
A.  She sells  B.  she     C.  s          D.  she sells
                                              seashells on
                                              the seashore.
```

16. Choose the option that displays the output of the file read operation.

```
#include<stdio.h>
struct data{ char c[50];};
main()
{
        FILE *fp;
        struct data data1;
        fp = fopen("R1.c", "r");
        fread(&data1, sizeof(struct data), 1, fp);
        printf("%s", data1.c);
}
```

```
A.  She sells  B.  she     C.  s          D.  she sells
                                              seashells on
                                              the seashore.
```

17. Choose the code sample used to display the value of the value of the environment variable `COMSPEC`.

```
A.  #include<stdio.h>                    B.  #include<stdio.h>
    #include<stdlib.h>                       #include<stdlib.h>
    main()                                   main()
    {                                        {
    printf("%s\n", getenv());                printf("%s\n", getenv("compec"));
    }                                        }

C.  #include<stdio.h>                    D.  #include<stdio.h>
    #include<stdlib.h>                       #include<stdlib.h>
    main()                                   main()
    {                                        {
    printf("%s\n", getenv("con*"));          printf("%s\n", getenv("COMSPEC"));
    }                                        }
```

REVIEW EXERCISE

18. Four code samples are given. Choose the code sample that can be used to display all environment variables.

A.
```
#include<stdio.h>
main(int argc, char *argv[], char *env[])
{
    int i;
    for(i = 0; env[i] != NULL; i++)
    {
            printf("%s\n", env[i]);
    }
}
```

B.
```
#include<stdio.h>
main()
{
    int i;
    for(i = 0; env[i] != NULL; i++)
    {
            printf("%s\n", getenv(i));
    }
}
```

C.
```
#include<stdio.h>
main(int argc, char *argv[])
{
    int i;
    for(i = 0; env[i] != NULL; i++)
    {
            printf("%s\n", env[i]);
    }
    printf("%s\n", getenv("C"));
}
```

D.
```
#include<stdio.h>
main(int argc, char *argv[], char *env[])
{
    int i;
    for(i = 0; env[i] != NULL; i++)
    {
            printf("%s\n", getenv(i));
    }
}
```

19. Identify the advantages of using stream-oriented files.

A. Provides a consistent interface
B. Requires minimum access to storage devices
C. Stores data in the binary format
D. Performs unbuffered I/O

20. The partial code to read data from the `sample1.txt` *file and copy it to the* `sample2.txt` *file is given. Write the code to close the file from which data is being read.*

```
#include<stdio.h>
main()
{
       FILE *fp, *ptr;
       char a;
       fp = fopen("sample1.txt", "r");
       ptr = fopen("sample2.txt","w");
       do
       {
               a = fgetc(fp);
               fputc(a, ptr);
       }while(a != EOF)
//Enter Here
}
```

21. In the code sample given above. Several files are open. The code sample is to be compiled on a VC *compiler. Write the code to close all open files.*

22. *The contents of the file* `R1.c` *and the code sample used to read the data from* `R1.c` *and write it to* `W1.c` *are given. Choose the option that lists the contents stored in the file* `W1.c` *after the specified piece of code is executed.*

```
#include<stdio.h>
main()
{
        FILE *fp, *ptr;
        char a;
        fp = fopen("R1.c", "r");
        ptr = fopen("W1.c","w");
        a = fgetc(fp);
        printf("%c", a);
        a = fgetc((fp);
        fputc(a, ptr);
}
```

A. s
B. seashells on the seashore
C. h
D. sh
E. she sells
seashells on
the seashore
F. seashells on

23. *Chose the option that lists the contents stored in the file* `W1.c` *when the specified piece of code is executed.*

```
#include<stdio.h>
main()
{
        FILE *fp, *ptr;
        char c[100];
        fp = fopen("R1.c", "r");
        ptr = fopen("W1.c","w");
        fgets(c, 20, fp);
        printf("%s", c);
        fgets((c, 20, fp);
        fputc(c, ptr);
}
```

A. h
B. she sells seashell on the seashore
C. she sells
D. sh
E. she
F. seashells on

24. *Tick the option that lists the contents stored in the file* `W1.c` *when the specified piece of code is executed.*

```
#include<stdio.h>
main()
{
        FILE *fp, *ptr;
        char c[100];
        fp = fopen("R1.c", "r");
        ptr = fopen("W1.c","w");
        fscanf(fp, "%s", c);
        fprintf(ptr, "%s", c);
}
```

A. s
B. she
C. sh
D. h
E. she sells seashells on the seashore
F. seashells on

REVIEW EXERCISE

25. Choose the potion that lists the contents stored in the file `W1.c` *when the specified piece of code is executed.*

```
#include<stdio.h>
struct data{ char c[50];};
main()
{
        FILE *fp;
        FILE *ptr;
        struct data data1;
        fp = fopen("R1.c", "r");
        ptr = fopen("W1.c","w");
        fread(&data1, sizeof(struct data), 1, fp);
        fwrite(&data1, sizeof(struct data), 1, ptr);
}
```

A. s
B. seashells on the seashore
C. she sells
D. sh
E. she sells seashells on the seashore
F. seashells on

26. The file opened should support both reading and writing. The new contents should be added at the end of the file.

A. r *B. r+* *C. w* *D. w+* *E. a* *F. a+*

27. The file opened should support both reading and writing. If the file name specified does not exists, then the function should not create a new file but return a NULL pointer.

A. r *B. a* *C. w* *D. r+* *E. a+* *F. w+*

28. The file should support both reading writing. If an already existing file is opened, its previous contents should be completely deleted.

A. r *B. w* *C. r+* *D. a* *E. w+* *F. a+*

29. The file should allow the user to write data but not read data. A new file should be created if the file does not exist already. If the file exists, then none of the contents should be overwritten. New content should be added at the end.

A. r *B. w+* *C. a* *D. a+* *E. w* *F. r+*

30. The file should be opened so that a new file is created if the file does not exist already. If the file exists, then it should be overwritten with a new file. The open file should not support the reading process.

A. w *B. a+* *C. r* *D. w+* *E. a* *F. r+*

31. The file should be opened in a mode that does not allow any changes to be made to the contents of the file.

A. r *B. w* *C. a* *D. a+* *E. w+* *F. r+*

32. A code sample and four sets of output are given. In the code sample, the `scanf` *function is used to accept formatted input and the* `printf` *function is used to display formatted output. Choose the correct option.*

```
#include<stdio.h>
main()
{
        char str[10], str1[12];
        scanf("%[^aeiou]%s", str, str1);
        printf("[%s][%s]", str, str1);
}
```

```
Input:   trythiscode
```

A. [tryth][iscode] *B. [][]* *C. [ioe][trythscd]* *D. [][trythiscode]*

33. The code sample given below uses the `printf` *function to display formatted output. Tick the output that will be obtained when the specified code sample is executed.*

```
#include<stdio.h>
main()
{
       char s[] = "C Language";
       printf("%.6s,",s);
       printf("\n%-3.8s,",s);
       printf("\n%-15s,",s);
       printf("\n%10s,",s);
}
```

```
A. C Lang,             B. C Lang,         C. C,              D. C Lang,
   C Langua,              C Langua,          C Langua,          C Langua,
         C Language,      C Language ,       C Language ,       C Language ,
   C Language,            C Language,        C Language,             C Language,
```

34. Choose the output that will be obtained when the specified code sample is executed.

```
#include<stdio.h>
main()
{
       float a = 246.897;
       printf("\n%-2.3f",a);
       printf("\n%+4.3f",a);
       printf("\n%09.2f",a);
       printf("\n%3.1f",a);
}
```

```
A. -246.897            B. 246.897         C. 246.897         D. 246.897
   +246.897               0246.897           +246.897           +246.897
   000246.90              000246.90          000246.90          0000246.90
   246.9                  246.9              246.9              246.9
```

35. Tick the correct format of the output that will be obtained when the specified code sample is executed.

```
#include<stdio.h>
main()
{
       printf("%*,*f", 5, 2, 123, 456);
}
```

```
A. 5
B. 2
C. 123.46
D. 123.456
```

36. The code given here uses the `scanf` *function to accept formatted input. Tick the output that will be obtained when the specified code sample is executed.*

```
#include<stdio.h>
main()
{
       int i = 123;
       float a = 45.68;
       printf("[%d],[%f]", i, a);
       printf("\n[%2d],[%3.0f]", i, a);
       printf("\n[%7d],[%5.2f]", i, a);
       printf("\n[%04d],[%07.1f]", i, a);
}
```

```
A. [123],[45.680000]   B. [123],[45.680000]   C. [123],[45.680000]    D. [123],[45.680000]
   [123],[  46]           [12],[  46]            [123],[  46]            [123],[46.0]
   [     123],[45.68]     [     123],[45.68]     [     123],[45.68]      [     123],[45.68]
   [0123],[00045.7]       [0123],[00045.7]       [0123],[0000045.7]      [0123],[00045.7]
```

LESSON 111 : PREPROCESSOR COMMANDS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

When an error occurs in a program, some functions, such as the main function, return an integer value indicating that an error has occurred. However, it is not possible to determine the exact reason for the error only with this return value.

To determine the exact reason for the error, the external variable `errno` is used along with the `perror` function. The external variable `errno` is maintained by the operating system. This variable is assigned an integer value by the operating system to indicate the cause of the error.

The `perror` function is used in a program to display the system error message. The error message that is displayed by the `perror` function corresponding to the error number stored in the variable `errno`.

Consider an example. When a program attempts to open a file in the read mode and the file does not exist, the `fopen` function returns an error. The variable `errno` is set to an integer value indicating the reason for the error.

The value stored the variable `errno` is used by the `perror` function to display the exact description of the error that hated the program. This error description indicates the exact problem to the user, and the user can rectify the error before executing the program again.The variable `errno` is defined in the header file called `errno.h`. Each error number is associated with an error message indicating the error description in this header file.

In the header file `errno.h`, the list of the list of the descriptions of the errors is stored in a two -dimensional character type array called `sys_errlist`. The `perror` function uses the value in the vari-able `errno` as an index to the character array `sys_errlist`.

The perror function accepts a character pointer as an argument. You can use this argument to specify the name of the file or the function in which the error occurred. This argument helps the user who executes the program to identify the location of the error.

```
perror(char *);
```

When an error occurs, the argument that is specified with the perror function is defined along with the error message. The message displayed by the perror function is in a specific format.

The specific format in which the message is displayed by the perror function is given below. The string that is specified as the argument is displayed followed by a colon `(;)`. The colon is followed by a description of the error as defined in the header file errno.h.

```
Perror(str);

Str:<Description of the Error>
```

A code sample that uses the perror function is given below. Notice the use of the perror function to determine the exact reason for the failure of the fopen function. After identifying the reason, the perror function displays the system error message.

Listing 111.1 : using perror function

```
//Listing 111.1 : this program uses the perror function
#include <stdio.h>
#include <errno.h>

main()
{

int code, name]30];
FILE *fp;
fp=fopen(“rec.dat”,”r”);
if(errno!=0)
perror(“rec.dat”);

}
```

Detailed Explanation

- When using the perror function, care has to be taken because the variable errno is reset only when an error occurs. The value of errno is not affected by a successful function call.
- When a function is executed successfully. The old value corresponding to a previous error can still exist in the variable errno. Therefore, the perror function must be called immediately after the function that can cause an error is called.
- You should also check the return value of the function that can cause an error before calling the perror function. If you do not check the return value of the function that could cause the error, an error message will be displayed even if the function is executed successfully.
- The error message that will be displayed will be irrelevant for the user because the error message will not be related to the function that was called before the display of the error message.
- A code sample that shows an incorrect error message is given below. In this code, an error occurred at line 7. You display the error message by using the perror function at line 13, which did not have any error.

Listing 111.2 : displaying incorrect error message

```
1   //Listing 111.2 : this program displays an incorrect error message
2   #include <stdio.h>
3   #include <errno.h>
4
5   main()
6   {
7
8   int code, name[30];
9   FILE *fp;
10  fp=fopen("rec.dat","r");
11  printf("Type code");
12  scanf("%d", &code);
13  printf("Type name");
14  scanf("%d", name);
15
16  if(errno!=0)
17  perror("rec.dat");
18
19  }
```

- The error message that is displayed using the `perror` function will not be related to line `11` because the variable `errno` will have the `errno` number corresponding to the `errno` at line `7` and not line `11`.

Consider another example that shows the correct error message. In this code, the `perror` function is called immediately after line `7` where an error can occur.

Listing 111.3 : displaying correct error message

```
1   //Listing 111.3 : displaying correct error messages
2   #include <stdio.h>
3   #include <errno.h>
4
5   main()
6   {
7
8   int code, name[30];
9   FILE *fp;
10  fp=fopen("rec.dat","r");
11
12  if(errno!=0)
13      perror("rec.dat");
14
15  printf("Type code");
16  scanf("%d", &code);
17  printf("Type name");
18  scanf("%d", name);
19  }
```

- The variable `errno` is assigned a value by the operating system to indicate the error that halted the program. The `perror` function is used to display an error message corresponding to the value of the variable `errno`.

Self Review Exercise 111.1

1. *A program that accepts a student's name and street name is to e created. If the name cannot be accepted because of an error, then an error message is to be displayed. Choose code sample that uses the* `perror` *function to display the system error message now.*

```
#include<stdio.h>
#include<errno.h>
main()
{
char name[21],street[21];
printf("Type name");
gets(name);
if(errno()!=0)
perror(0);
printf("Type street");
gets(street);
}
```
A

```
#include<stdio.h>
#include<errno.h>
main()
{
char name[21],street[21];
printf("Type name");
gets(name);
if(errno()!=0)
perror("Data not accepted");
printf("Type street");
gets(street);
}
```
B

```
#include<stdio.h>
#include<errno.h>
main()
{
char name[21],street[21];
printf("Type name");
gets(name);
printf("Type street");
gets(street);
if(errno()!=0)
perror("Data not accepted");
}
```
C

```
#include<stdio.h>
#include<errno.h>
main()
{
char name[21],street[21];
printf("Type name");
gets(name);
printf("Type street");
gets(street);
if(errno!=0)
perror("Data not accepted");
}
```
D

LESSON 112 : ASSERTIONS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Often, a program contains a mistake that causes the program to halt during execution. These mistakes are usually very difficulty to locate.

Mistakes can occur in programs when you make some assumptions about the value of variables while creating program. However, during program execution, some erroneous values may be assigned to variables because of certain incorrect statements in the program.

The assert macro is used to check the validity of the values of variables used in a program. The C language provides the assert macro in the standard header file assert.h. You include this header file in the programs that use the assert macro during program testing.

Notice that the assert macro accepts a single expression as its argument. The expression is a logical expression and returns TRUE or FALSE. The statement containing the assert macro is terminated by a semicolon.

```
assert(<logical expression>);
```

If the expression returns FALSE, then it indicates that the assertion was not successful. If an assertion fails, then an error message is displayed with the line number where the assertion failed and the program execution stops.

A code sample that uses the assert macro to check the value of a variable is given below. In this code sample, the find function returns a positive integer.

Listing 112.1 : using assert macro

```
//Listing 112.1 : this program uses the assert macro
#include <assert.h>
#include <stdio.h>
main()
{
int a,b,c;
int find(int, int);
scanf("%d%d",&a,&b);
c=find(a,b);
assert(c>0);
}

int find(int x, int y)
{
return 5;
}
```

Consider another example. In the example below, it may be essential for the arguments of the find function to satisfy certain conditions. For example, the argument called x has the value 1 or the value -1and the argument called y has a value ranging from 7 through 11.

Listing 112.2 : using assert macro

```
//Listing 112.2 : using the assert macro
#include <assert.h>
#include <stdio.h>
main()
{
int a,b,c;
int find(int, int);
scanf("%d%d",&a,&b);
c=find(a,b);
assert(c>0);
}
int find(int x, int y)
{
assert(x==1||x==-1);
assert(y>=7&&y<=11);
}
```

Detailed Explanation

- You can use the assert macro to ensure that the program proceeds only when the arguments called x and y contain valid values. The first assertion displayed on the screen checks the validity of the argument called x.
- The second assertion checks the validity of the argument called y. If the value of x or y is found to be invalid, program execution is terminated.
- You can add assertions to a program during the testing phase of program development. After testing is complete, you can remove assert statement from the program manually or by using the NDEBUG macro.

NDEBUG Macro

If the NDEBUG macro is defined, all the assertions in the program are ignored. This allows you to use assertions freely during program development and then discard them later by defining the NDEBUG macro.

Self Review Exercise 112.1

1. *A program works only if the value returned by the trap function in the pointer named p is not NULL. An assert statement is to be used to check the value of the find function.*

```
#include<stdio.h>
#include<assert.h>
main()
{
char *p, *trap;
p=trap;
assert(p!=NULL);
}
```
A

```
#include<stdio.h>
main()
{
char *p, *trap;
p=trap;
assert(p!=NULL);
}
```
B

```
#include<stdio.h>
#include<assert.h>
main()
{
char *p, *trap;
p=trap;
assert(NULL);
}
```
C

```
#include<stdio.h>
#include<assert.h>
main()
{
char *p, *trap;
p=trap;
assert(p==NULL);
}
```
D

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The preprocessor can play an important role in debugging a C program. The preprocessor provides the #error preprocessor directive, which you can use when an abnormal condition is detected during preprocessing.

The #error preprocessor directive can be used inside a conditional statement that detects a combination of the parameters that are not supported by your program.

For example, if your program does not compile properly on a `vax` machine, you can use a conditional statement to detect the error. After you detect that the program is being compiled on a `vax` machine, you use the `#error` preprocessor directive to display a compilation error. This helps the user to identify the error and take action by compiling the program on another machine.

The syntax for writing the #error preprocessor directive is given below. The statement that uses the `#error` preprocessor directive does not end with a semicolon. The `#error` preprocessor directive is followed by a message.

```
#error <Error Message>
```

The message in the syntax of the `#error` preprocessor directive is not enclosed in quotation marks. If you use quotation marks in the message, they will be displayed with the massage

If the program reaches the `#error` preprocessor directive during compilation, then the message following the `#error` preprocessor directive is displayed and the program compilation is terminated.

A example that uses the `#error` preprocessor directive to display an error message is given. This example cannot be compiled if the value of the symbolic constant `DIFF` exceeds `500`. The first line in the code checks for the value of the symbolic constant `DIFF`.

Listing 113.1 : implementing routines

```
//Listing 113.1 : this program implements routines
#include <stdio.h>
#define DIFF 200
#if DIFF>500
#error Invalid Value
#endif
main()
{
/*Remaining part of the code is here*/
}
```

If the condition that checks for the value of the symbolic constant `DIFF` returns `TRUE`, then the statement of the code containing the `#error` preprocessor directive is processed. The `#error` preprocessor directive the compilation of the code.

This directive is used to display an error message and about the compilation of the program if a specified condition becomes `TRUE`. The message used with the `#error` preprocessor directive is written without quotation marks.

Self Review Exercise 113.1

1. *A program gives the correct output only if the symbolic constant* ORDER *is less than or equals to* 50000. *Choose the code sample to show an error message if* ORDER *exceeds* 50000.

```
#if ORDER > 50000
#error Invalid ORDER;
#endif
```
A

```
#if ORDER <= 50000
#error Invalid ORDER;
#endif
```
B

```
#if ORDER > 50000
error Invalid ORDER;
#endif
```
C

```
#if ORDER > 50000
#error Invalid ORDER
#endif
```
D

LESSON 114 : DEBUGGING STRATEGIES

Lesson Objectives
In this Lesson you will learn : • *the Genesis of C Language* • *the Stages in the Evolution of C Language*

To identify errors in code, you can use various debugging strategies. For example, you may use one debugging strategy to locate the line that is causing an error and another debugging strategy to view the effect of the values of an expression on the execution of code.

Debugging strategies help you in checking the values of variables when an error occurs. You can view the code line by line while it executes to determine when the value of an expression changes.

To use debugging strategies, you need debugging tools. A debugging tool is a programming that provides options to select a debugging strategy. You use a debugging strategy. You use a debugging tool to stop the program at a specified statement and to locate the cause of the error.

Some of the debugging strategies that you can use with a debugging tool are: setting and disabling breakpoints, setting break conditions, and setting watch points.

Debugging Strategies
Setting Breakpoints
Disabling Breakpoints
Setting Break Conditions
Setting Watchpoints

Setting Breakpoints

A breakpoint halts a program when a certain point in the program is reached. The point where a program halts can be a line number or a function address that you specify.

Consider an example in which a program works without any error until reaches line 14. After line 14, the program reports an error and stops. In this example, you can set a breakpoint at line 14 and examine the cause of the error.

You can add a breakpoint to a program by using a debugging tool. Adding a breakpoint is called setting a breakpoint. The program halts at the point at which a breakpoint is set.

Disabling Breakpoints

After locating the error by using a breakpoint, you can delete the breakpoint that is deleted cannot be used again. You can disable a breakpoint if you feel that it will be required again in the program. You can reuse a disabled breakpoint by enabling it.

Setting Breakpoints

You can also use a breakpoint with a condition to find an error. Setting a breakpoint with a condition is called setting a break condition. Program execution stops only if the break condition evaluates to `TRUE`.

Consider a code example where you set a breakpoint at line `20`. However, the execution of this code should stop only when the value of the variable `num` becomes `0`.

To stop the execution of the program when the value of the variable num becomes `0`, you can add a condition to check for the value of `num` to the breakpoint that is set at line `20`.

Setting Watchpoints

Another debugging strategy that you can use is to set a watchpoint. A watchpoint is a special breakpoint halts the program when the value of an expression changes.

You can use a watchpoint when a variable receives an unpredictable or invalid value. A watchpoint is helpful when the exact point at which the value of the variable change and becomes invalid is not known.

A sample expression that can be used with a watch point is given. In this expression, the program execution stops when the value of the variable total becomes less than `1000`.

```
total<1000
```

A watchpoint is managed in the same way as a breakpoint you can set, delete, disable, and enable a watchpoint. It is also possible to add a condition for a watchpoint.

Self Review Exercise 114.1

1. *Identify a function of a breakpoint.*

   A. Stops program execution when the value of an expression changes
   B. Stops program execution when a breakpoint is encountered for the first time
   C. Stops program execution when a break condition evaluates to FALSE
   D. Stops program execution at a specified line where a breakpoint is encountered

2. *Select the debugging strategy that can be used to stop the program at line* 14.

   A. Setting a breakpoint
   B. Setting a watchpoint
   C. Adding a condition to a breakpoint
   D. Enabling a deleted breakpoint

Lesson Objectives
In this Lesson you will learn :
• *the Genesis of C Language*
• *the Stages in the Evolution of C Language*

Various debugging tools that provide an interactive way to find logical errors in a program are available with C compilers. The debugging tools covered in this lesson are lint, cflow, cxrefs, adb, and symbolic debuggers.

The Lint Tool

The tool lint used to display warning messages about unused variables and incorrect numbers of arguments in a program. The warning displayed by lint are similar to the warning displayed by C compilers.

A program containing a function that is called with an incorrect number of arguments is given. You can use the tool lint to view a report of these types of errors.

Listing 115.1 : targeting the debugging tool Lint

```
//Listing 115.1 : this program deals with the debugging tool Lint

#include <stdio.h>
int a[3][4]={
{3,6,9,12},
{15,25,30,35},
{66,77,88,99}
};

main()
{
int *sales[3];
int num,col;
sales[0]=a[0];
sales[1]=a[1];
sales[2]=a[2];
target(sales);
}

void target(int *arr[], int i)
{
for(i=0;i<4;i++)
printf("%d\n",arr[i]);
}
```

The cflow Tool

The tool cflow shows the relations the relationship between various functions that are used in a program. This tool indicates the called function and the calling function in a program.

The tool cflow produces an indented output, watch indicates that the function calls are nested in a program.

A program that uses multiple functions is given below. You can use the cflow tool to find which function called the menu function in the program.

Listing 115.2 : using cflow tool

```
//Listing 115.2 : this program uses cflow tool

#include <stdio.h>
Char menu(void (*fptr)(), char nc)
{

Char ch=' ';
Fptr();
Printf("\nType choice");

Do
{
Ch=getchar();
}while(ch<'0'||ch>nc);
Return ch;
}

Main()
{
Char option,menu();
Void menumain(),menuadd();
While((option=menu(menumain, '3'))!='3')
{

Switch(option)
{
Case '1' : putchar('*'); break;
Case '2' : while((option=menu(menuadd, '3'))!='3')
{

Switch(option)
{
Case '1' : puts("Single char function"); break;
Case '2' : puts("Multiple char function"); break;
}

}break
}
}
}

Void menumain()
{
Printf("\nMain Menu\n);
Printf("\n1. Display char \n2. Add char \n3. Exit ");
}
```

The cxrefs Tool

The tool cxrefs indicates the locations where each variable and function is defined in a program. This is extremely helpful in large applications having multiple source code files.

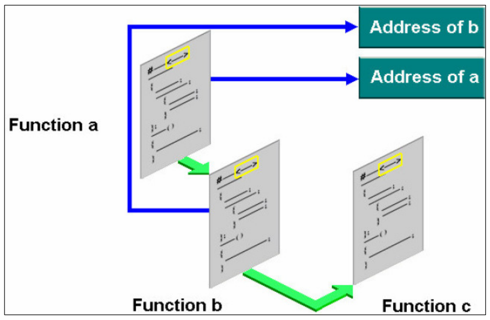
Consider an example when you create an application consisting of two program files, `main.c`. The `cxrefs` tool indicates whether the variables and functions are declared in the program file `main.c` or `sub.c`.

The adb tool

The tool `adb` shows the function address on the function stack when a program crashes.

When a function calls another function in a program, the address of the calling function is stored in the stack before the control is transferred to the called function.

In the example displayed in the diagram, if an error occurs in a line of function c, then the tool adb can be used to find out the function addresses of all the functions existing in the function stack. The tool adb also displays the arguments supplied to each function whose address is found in the stack.



The Symbolic Debugging

Symbolic debugger tools allow the program to be executed in the trace mode. The trace mode allows the use of breakpoints and watchpoints for debugging. The program can also be browser or edited using these tools.

The symbolic debuggers commonly available are bdx, xdb, and gdb. To use a symbolic debugger, the program is compiled with the –g option. This command creates an object file from the program.

Consider an example. The commands to create an object an object file by using the `gcc -g -o obfile filename.c`. This command is used to compile the program file named command is used to compile the program file named `filename.c`. the command creates an object file called `obfile`.

```
gcc -g -o obfile filename.c
```

After the program is compiled with the –g option, the program can be debugged using one of the symbolic debuggers. You can issue the command `dbx <objectfile>, xdb <objectfile>, or gdb<objectfile>` depending on the symbolic debugger to be used.

Commands To Invoke Symbolic Debuggers	
dbx	dbx<objectfile>
xdb	xdb<objectfile>
gdb	gdb<objectfile>

When you invoke a symbolic debugger, the command prompt that corresponds to symbolic debugger is displayed. For example, the symbolic debugger gdb displays the percent (%) prompt. You can issue commands to debug programs at the prompt.

Self Review Exercise 115.1

1. *Which is a function of the cflow debugging tool?*

   A. Locating the incorrect arguments in a program
   B. Listing the unused variables in a program
   C. Displaying the function call nesting in a program
   D. Locating variable declarations in a program

2. *Identify a function of a symbolic debugger.*

   A. Displays the function addresses on the function stack when a program crashes
   B. Runs the program in the trace mode and sets breakpoints and watchpoints
   C. Displays an indented output of function call nesting
   D. Determines where functions are declared in a multi-source code file application

LESSON 116 : COMMON MISTAKES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Sometimes a program can produce an incorrect output. This happens if the program contains logical errors. These errors are very difficult trace.

You can reduce the logical errors in a program if you know how to debug the mistakes in programs. These mistakes can be classified into three categories: declaration mismatch mistakes, pointer initialization mistakes, and miscellaneous mistakes.

COMMON MISTAKES

Declaration Mismatch

A declaration mismatch happens when you use variable of different data types in a statement. Using variable of different data types in a statement can result in incorrect output. For example, assigning a `float` value to an integer variable in a program result in data loss.

An incorrect output is generated because C performs an automatic conversion of the data types of variables with different data types. The automatic conversion of data types refers to the conversion of one data type to another by the compiler and can result in critical data being lost.

A code that contains a declaration mismatch is shown. The first line of the code declares a function `mean`, which accepts two integers as arguments. The function `mean` calculates the mean of two integers and returns it as a `float` value.

Listing 1116.1 : declaration mismatch

```
1   //Listing 116.1 : this program contains a declaration mismatch
2
3   #include <stdio.h>
4   main()
5   {
6   float mean(int, int);
7   float f1=12.3333;
8   float f2=10.888, fmean;
9   fmean=mean(f1,f2);
10  printf("%f", fmean);
11  }
12
13  float mean(int i, int j)
14  {
15  return (i+j)/2;
16  }
```

Detailed Explanation

- Lines `5`, `6` declare and initialize the variables `f1` and `f2`. A variable called `fmean` is also declared. This variable is declared to store the value that is returned by the `mean` function.

- The next statement calls the `mean` function by passing the values of the `float` variables `f1` and `f2`. The `mean` function calculates the mean of two integers as an integer and returns an integer value, which is stored in the float variable `fmean`.
- The `printf` function, in Line `8` given the mean. This mean value is incorrect because the arguments of the `mean` function are declared as integers. When two `float` values are passed to the `mean` function as arguments, they are converted to integers.
- To obtain the correct mean, you can rewrite the code as follows. In this code, the arguments of the `mean` function are declared to be of the `float` type. A declaration mismatch should be avoided in a program because the program can start producing incorrect outputs.

Pointer Initialization

Another common mistake in programs is incorrect pointer initialization. Most of the errors in C programs are related to the memory allocation of pointers.

A code sample that does not initialize a pointer correctly is given. In the first line of the code sample, a character type pointer, `ans`, is declared. The pointer is created to store a number corresponding to the option selected by the user.

Listing 116.2 : pointer intitialization

```
//Listing 116.2 : this program demonstrates pointer initialization
#include <stdio.h>
main()
{
char *ans;
printf(“Type the answer :”);
gets(ans);
}
```

Detailed Explanation

- In the next part of the code sample, the user is prompted to type a choice. The `gets` function is called to accept a string in the pointer `ans`. The pointer `ans` does not point to any location in memory. Therefore, there is no memory available to store a value in the pointer `ans`.
- You can correct this code sample to accept a choice from the user by initializing the pointer ans. A statement is added to the code sample to get the correct output. All the pointers used in a program should be initialized before they are used.

Listing 116.3 : pointer initialization

```
//Listing 116.3 : this program demonstrates the concept of pointer initialization
#include <stdio.h>
main()
{
char *ans, str[40];
ans=str;
printf(“Type the answer :”);
gets(ans);
}
```

Miscellaneous Mistakes

Miscellaneous Mistakes
Use of the = Operator
Use of Nested Comments
Use of a Semicolon with Looping Constructs
Use of Braces
Operator precedence
Use of & operator

The third category of errors called miscellaneous mistakes contains a group of commonly made mistakes.

- One of the miscellaneous mistakes is using the assignment (=) operator instead of the equal to (==) operator while comparing values.

A code sample that incorrectly uses the = operator is given. The use of the = operator results in the assignment of the value 3930 to the variable i.

An assignment statement always returns the value that was assigned in the statement. Therefore, the if statement evaluates to TRUE and displays the message OK even though the variable i did not have the value 3930 before the if statement was called.

Another miscellaneous mistake is the use of nested comments in programs. A nested comment within another comment.

A code sample containing a nested comment is given. The line that is given in the code indicates the beginning of the first comment.

Listing 116.4 : miscellaneous mistakes

```
//Listing 116.4 : this program deals with miscellaneous mistakes
#include <stdio.h>
main()
{
  int i;
  printf("Type code");
  scanf("%d",&i);
  /*Check values
  /*Check TRUE value*/
  if(i==3930)
  printf("OK");
  /*Check FALSE value*/
  else
  printf("Not OK");
  */
}
```

The beginning of the nested comment is taken as a comment statement as it is enclosed within the first comment, which is continued. However, the end of the second comment is interpreted by the compiler as the end of the first comment. This causes the compiler to report an error in the line where the first comment ends.

LESSON 116 : COMMON MISTAKES

An erroneous piece of code is displayed. Choose the option that can be used for debugging the error in the displayed piece of code now.

Another mistake in the miscellaneous category is the use of a semicolon with looping constructs, such as the `while` constructs, or conditional constructs, such as the `if` construct.

A code sample that uses a semicolon with the `while` construct is given below. The while loop executes infinitely because the semicolon terminates the `while` loop. Therefore, the condition in the `while` constructs always remains `TRUE` and the loop continues infinitely.

Listing 116.5 : miscellaneous mistakes

```
//Listing 116.5 : this program deals with miscellaneous mistakes

#include <stdlib.h>
#include <stdio.h>
main()
{
   int num=0, *numptr[50];
   while(num<50);
   {
   numptr[num]=malloc(sizeof(int));
   num++;
   }
}
```

Another mistake in the miscellaneous category is the incorrect use of braces. Braces are used to indicate a block of statement in a program.

A code sample that uses braces incorrectly is given below. This code sample has two `if` statement occurring consecutively. The condition is `TRUE`.

Listing 116.6 : miscellaneous mistakes

```
//Listing 116.6 : this program deals with miscellaneous mistakes
#include <stdio.h>
main()
{
  int i=50;
  if(i<20)
  if(i>10)
  puts("Within the range");
  else
  puts("Greater than 20");
}
```

A message, greater than `20`, is displayed when the variable `i` contains a value than `10`. this is logically incorrect because the message does not match with the value of he variable `i`. The error occurs because the `else` statement is matched with the closest `if` statement that occurs before the `else` statement in the program.

You can correct the error by using braces correctly as in the of code. In the code, the second message is defined appropriately.

Another miscellaneous mistake in a program is related to operator precedence. A piece of code is given. It illustrates an error caused due to operator precedence. In this piece of code, a file is opened and then the resulting value in the variable `fp` is compared with `NULL` to see whether the `fopen` function failed.

Listing 116.7 : miscellaneous mistakes

```
//Listing 116.7 : this program deals with miscellaneous mistakes
#include <stdio.h>
main()
{
  FILE *fp;
  if(fp=fopen("student.dat","r")==NULL)
  return;
}
```

However, in this piece of code, the condition `fopen("student.dat"."I") == NULL` is tested first. This returns a nonzero value if the condition is `TRUE`. In the second step, the result of the condition is assigned to `fp`, instead of the file pointer.

You can correct the code by adding an additional set of parentheses as follows.

Another mistake in the miscellaneous category is a missing `&` operator with the scanf function. The `&` operator is used with the `scanf` function for accepting data into all variable except string or character arrays.

A code sample that incorrectly uses the `=` operator is given. The use of the `=` operator results in the assignment of the value `3930` to the variable `i`.

Listing 116.8 : miscellaneous mistakes

```
Listing 116.8 : this program miscellaneous mistakes

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

Self Review Exercise 116.1

1. *An erroneous piece of code is given. Choose the option that describes the error in the displayed piece of the code now.*

```
#include <stdio.h>
main()
{
   double BillAmount;
   int PaidAmount;
   printf("Type bill amount");
   scanf("%f",&BillAmount);
   PaidAmount=BillAmount*.02;
   printf("Amount to be paid is %d", PaidAmount);
}
```

A. When the double variable BillAmount is multiplied with a value, data is lost.
B. When the value of PaidAmount is displayed using the printf function, data is lost.
C. When the result of the expression involving BillAmount is assigned to PaidAmount data is lost.
D. When the result of the expression involving BillAmount is assigned to PaidAmount, the compiler reports an error.

Self Review Exercise 116.2

1. *An erroneous piece of code is given below. Choose the option that can be used for debugging the error in the displayed piece of code.*

```
#include <stdio.h>
main()
{
int *number;
float price;
scanf("%d",number);
printf("%d",*number);
price=(*number)=.01;
if(price>80)
printf("Exceeds value");
```

A. The assignment of the expression involving the variable number to the variable price causes data loss.
B. The pointer is not initialized.
C. The = operator should not be used with the if statement.
D. The assignment of the expression involving the variable called number of the variable price causes a compiler error.

2. *An erroneous piece of code to fine the maximum number from the numbers entered by the user is displayed. Choose the option for debugging the error in the displayed piece of code now.*

```
int number;
float max;  /*Loop to find maximum number*/
while(number!=0)
{/*Accept number*/
  scanf(":%d",&number);
  if(number==0)
  printf("Want to see max number?");
  scanf("%c", &rep);
  if(rep=='y")
  printf("max is %d",max);
  exit(0);
  else
  {
  printf("Number is non zero");
  if(number>max)
  max=number;
  }
}
```

A. Remover nested comments from the code
B. Use an = operator with the first if statement.
C. Use braces with the first if statement
D. Remove the declaration mismatch from the statement that assigns the value of number to max.

REVIEW EXERCISE

1. The `calc()` *function is called to calculate a value. The calculated value should range from* 40 *to* 60. *Tick the code sample to terminate the program if the* `calc()` *function returns a value that is outside the specific range.*

A.
```
#include<assert.h>
#include<stdio.h>
main()
{
   int calc();
   int perc = calc();
   assert(perc < 40 || perc > 60);
}
```

B.
```
#include<assert.h>
#include<stdio.h>
main()
{
   int calc();
   int perc = calc();
   assert(perc > 40 && perc > 60);
}
```

C.
```
#include<assert.h>
#include<stdio.h>
main()
{
   int calc();
   int perc = calc();
   assert(perc < 40 || perc > 60);
}
```

D.
```
#include<assert.h>
#include<stdio.h>
main()
{
   int calc();
   int perc = calc();
   assert(perc > 40 && perc < 60);
}
```

2. Match the debugging tools with their functions.

A.	*adb*	A.	*Indicates where a function is declared.*
B.	*cxrefs*	B.	*Allows the use of the trace mode*
C.	*lint*	C.	*Specifies function call nesting*
D.	*Symbolic debugger*	D.	*Shows warning messages about unused variables*
E.	*cflow*	E.	*Reports return addresses of functions*

3. A series of erroneous code blocks are given. Tick the option to debug the mistake in the given code block.

```
float fcast, sales1, sales2;
/*Accept the sales figure*/
printf("Enter sales1");
scanf("%f", &sales1);
printf("Enter sales2");
scanf("%f", &sales2);
/*Calculate forecast*/
fcast = fcal(sales1, sales2);
printf("The forecast value is %f", fcast);
}
float fcal(int i, int j)
{
```

A. Do not use the `printf()` *function to display numeric values.*

B. Do not nest the comment lines.

C. Use the == operator instead of the = operator.

D. Use the `&` *operator with* `fcast` *in the* `printf()` *function.*

E. Declare `i` *and* `j` *as* `float` *variables.*

F. Use proper grouping.

STRUCTURED PROGRAM DEVELOPMENT USING C PROGRAMMING LANGUAGE

REVIEW EXERCISE

4. Choose the option to debug the mistake in the given code block.

```
int *ptr, score;
char *nptr, name[40];
ptr = &score;
nptr = name;
/*Accept name*/
scanf("%s", nptr);
/*Accept score*/
scanf("%d", score);
/*Display name and score*/
printf("Name is %s, Score is %d", nptr, score);
}
```

A. Do not nest the comment lines.
B. Use correct arguments with the `printf()` *function.*
C. Use the & *operator with score in the* `scanf()` *function.*
D. Initialize the pointer correctly.
E. Use the & *operator with* `nptr` *in the* `printf()`
F. Use proper grouping.

5. Choose the option to debug the mistake in the given code block.

```
char *charac, ch;
charac = &ch;
scanf("%c", charac);
/*Check start*/
if(*charac > 'A' && *charac < 'Z')
       puts("Upper case");
else
if(*charac = '0')
       puts("Zero");
else
       puts("None");
/*Check complete*/
```

A. Use the == operator instead of the = operator.
B. Initialize the pointer correctly.
C. Use a semicolon with the `if` *statements.*
D. Do not nest the comment lines.
E. Use the & *operator with the* `if` *statements.*
F. Use proper grouping.

6. Choose the option to debug the mistake in the given code block.

```
char *code, str[41];
code = str;
/*Code for accepting
/*Accept code*/
scanf("%s", code);
/*Display name*/
printf(code);
*/
```

A. Use the & *operator with code in the* `scanf()` *function.*
B. Use the & *operator with code in the* `printf()` *function.*
C. Do not nest the comment lines.
D. Use == operator instead of = operator.
E. Use proper grouping.
F. Initialize the pointer correctly.

7. Choose the option to debug the mistake in the given code block.

```
/*Accept score*/
int *score, m;
score = &m;
printf("Type score :");
scanf("%d", score);
/*Display score*/
if(*score > 90)
       if(*score < 95)
               puts("Between 90 and 95");
else
       puts("Below 90");
```

A. Initialize the pointer correctly.

B. Do not nest the comment lines.

C. Use the & *operator with the* `scanf()` *function.*

D. Do not use numbers with the `puts()` *function.*

E. Use proper grouping.

F. Use a semicolon with the `if` *statement.*

8. Choose the option to debug the mistake in the given code block.

```
char *ptr1, *ptr2, ch[5], ch2[5];
int len = 0;
ptr1 = ch;
printf("Type the first string");
scanf("%s", ptr1);
printf("Type the first string");
ptr2 = ch2;
scanf("%s", ptr2);
/*Check the difference*/
if(len = strlen(ptr1) > 0)
/*Show the length*/
printf("Length is %d", len);
/*Continue the program*/
```

A. Add a set of parentheses to the `strlen()` *function.*

B. Do not nest the comment lines.

C. Use the == operator with the `strlen()` *function.*

D. Use the & *operator with the* `scanf()` *function.*

E. Use the & *operator with the* `printf()` *function.*

F. Initialize the pointer correctly.

9. Match the debugging strategies with their functions.

A.	*Setting a break condition*		*A.*	*Used to stop the program when the value of a variable changes*
B.	*Setting a watch point*		*B.*	*Used to stop the program when a specified line is reached*
C.	*Disabling a breakpoint*		*C.*	*Used to remove a breakpoint temporarily*
D.	*Setting a breakpoint*		*D.*	*Used to stop program when the value of an expression becomes...*

REVIEW EXERCISE

10. Choose the option to debug the mistake in the given code block.

```
char *one, *two, ch;
printf("Type the first character");
/*Accept first character*/
one = &ch;
scanf("%c", one);
fflush(stdin);
printf("Type the second character");
/*Accept second character*/
two = &ch;
scanf("%c", two);
if(*one > *two)
puts("One is bigger");
```

A. Initialize the pointer correctly.

B. Add set of parentheses to the `scanf()` *function.*

C. Use the & *operator with the* `scanf()` *function.*

D. Use the & *operator with* `ptr` *in the* `printf()` *function.*

E. Use proper grouping.

F. Do not nest the comment lines.

11. Choose the correct code sample to display the system error message by using the `perror` *function.*

A.
```
#include<errno.h>
#include<stdio.h>
main(int argc, char *argv[1])
{
   FILE *ptr;
   ptr = fopen(argv[1], "r");
   if(errno !=0)
       perror(argv[1]);
}
```

B.
```
#include<errno.h>
#include<stdio.h>
main(int argc, char *argv[1])
{
   FILE *ptr;
   ptr = fopen(argv[1], "r");
   if(errno !=0)
       perror();
}
```

C.
```
#include<errno.h>
#include<stdio.h>
main(int argc, char *argv[1])
{
   FILE *ptr;
   ptr = fopen(argv[1], "r");
   if(errno !=0)
       perror(0);
```

D.
```
#include<stdio.h>
main(int argc, char *argv[1])
{
   FILE *ptr;
   ptr = fopen(argv[1], "r");
   if(errno !=0)
       perror();
}
```

12. A program function correctly only if the symbolic constant `SIZE` *is divisible by the number* 2 *and the number* 3*. Choose the code sample to display and error message is* `SIZE does not have correct values`.

A.
```
#if SIZE % 2 || SIZE % 3
#error Ivalid SIZE
#endif
```

B.
```
#if SIZE % 2 && SIZE % 3
#error "Ivalid SIZE"
#endif
```

C.
```
#ifdef SIZE % 2 && SIZE % 3
#error Ivalid SIZE
#endif
```

D.
```
#ifdef SIZE % 2 || SIZE % 3
#error Ivalid SIZE
#endif
```

LESSON 117 : SELF REFERENTIAL STRUCTURES —> CREATION

Lesson Objectives

In this Lesson you will learn :

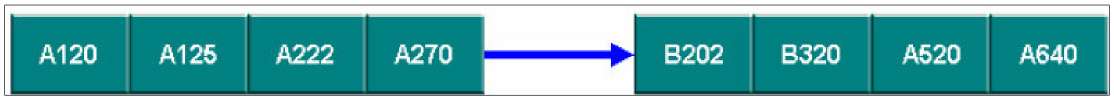
- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

In the C language, applications can be created using structures that help to organize complex data. A structure is a collection of variables that is referenced by a single name. The variable in a structure are treated as a single unit and not as separate entities.

You can link various structures to simplify the coding of applications developed using the C language.

In the C language, several structures of the same type can be linked together by creating self-referential structure, a pointer referencing create structure is declared within the structure definition. The declared pointer is assigned the address of the memory location containing another structure of the same type.

A diagram to represent a self-referential structure is given below. The diagram displays two data structures that represent two lists of item numbers. The first list of item number is to be linked to the second list. To link the two lists of item numbers, the first list contains a pointer that points to the second list. The pointer stores the address of the second list. This links the two structures.



A code sample representing a self-referential structure is given below. The code sample first declares a structure named `list`.

Listing 117.1 : dealing with self - referential structure

```
// Listing 117.1 : This program declares a self - referential structure.

struct list
{
    int data;
    struct list *next;
};
```

The integer variable `data` and a pointer named `next` are declared within the structure. The pointer `next` to the memory location of another structure of the type `struct list`.

When a pointer to a structure of the type struct `list` is declared within the `list` structure, it becomes a self-referential structure.

This enables you to link various data structures of the same type. The use of self-referential structures simplifies the algorithm required for manipulating several similar structures.

Self Review Exercise 117.1

1. *You learned about self - referential structures. Four code samples to declare a self - referential structure are given below. Choose the code sample that represents the declaration of a self - referential structure below.*

```
#include<stdlib.h>
struct prod
{
char ProdName[40];
int ProdID;
struct prod *next;
};
```
A

```
#include<stdlib.h>
struct
{
char ProdName[40];
int ProdID;
struct prod *next;
};
```
B

```
#include<stdlib.h>
struct prod
{
char ProdName[40];
int ProdID;
};
```
C

```
#include<stdlib.h>
struct prod
{
char ProdName[40];
int ProdID;
struct item *next;
};
```
D

LESSON 118 : NESTED STRUCTURES —> CREATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Developing business applications by using the C language may require large and complex program code. To simplify the manipulation of data and have a greater control over such program code, nested structures may be used.

Declaring a nested structure helps in calling the same structure in various programs. A code sample of a nested structure is given below. Two structures are declared in the code sample. The first structure is named `invoice`.

Listing 118.1 : nesting structure

```
//Listing 118.1 : this program deals with nested structures
// This program declares a nested structure.

Struct invoice
{
     char CustName[25];
     char CustAddress[80];
};

struct cust
{
     int AccNo;
     short AccType;
     struct invoice payments;
};
```

Detailed Explanation

- To reference the `invoice` structure from another structure, you prefix the keyword `struct` to the name of the `invoice` structure. In the example given below, the `invoice` structure is referenced in the structure `cust`.
- A variable called payments of the data type `struct invoice` is declared in the `cust` structure. To refer to the `invoice` structure, the `payments` variable can be used.
- When the nested structure is created, the communications between the structures `invoice` and `cust` becomes easy. This is because the `invoice` structure is referenced from the `cust` structure.

Self Review Exercise 118.1

1. *You learned about the creation about the creation of nested structures. A code sample is given below. Choose the code statement that has to be included in the structure called* `info` *to represent a nested structure now.*

```
struct product
{
     char ProdName[40];
     int ProdID;
};
struct  info
{
     int InvoNo;
     long InvDate;
     long invAnt;

};
```

A. `struct  info  ProdDetails;`

B. `product  ProdDetails;`

C. `info  ProdDetails;`

D. `struct  product  ProdDetails;`

LESSON 119 : DYNAMIC MEMORY ALLOCATION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The C language provides certain memory management functions that are used for the dynamic allocation of memory. The dynamic allocation of memory is the process by which a program can allocate memory during its execution. It is important to enable a program to obtain memory at run time because it is difficult to determine and allocate the exact amount of memory required by the program in advance.

Allowing a program to allocate memory dynamically helps in the optimum utilization of memory. This is because the amount of memory required by a program may vary substantially each time it is executed depending on the size of the data involved.

The `malloc` function is used for the dynamic allocation of `uninitialized` memory. The starting address of the allocated memory is returned by this function. This address should be assigned to a pointer of the type for which memory is being allocated.

The prototype of the `malloc` function is displayed. In the prototype, the `malloc` function returns a pointer of the type void. The `size_t` parameter passed to the `malloc` function specifies the number of bytes of memory to be allocated. It is defined as an unsigned int in the standard header file `malloc.h`.

```
void *malloc(size_t size);
```

In the example below, the `malloc` function obtains 50 bytes of memory. The starting address of the allocated memory block is assigned to the pointer variable named `ptr`.

Listing 119.1 : dynamic memory allocation

```
//Listing 119.1 : this program uses malloc to demonstrate dynamic memory allocation
//This program allocates 50 bytes of memory..

#include <stdlib.h>
#include <stdio.h>

main( )
{
     char *ptr;
     ptr=(char *)malloc(50);
}
```

If the required amount of memory is not available, the `malloc` function returns a NULL value.

Another example to allocate memory by using the `malloc` function is given. In this example, the `malloc` function allocates memory for 20 integers.

Listing 119.2 : allocating memory dynamically

```
//Listing 119.2 : this program uses malloc to implement dynamic memory allocation
//This program allocates memory for 20 integers.

#include <stdlib.h>
#include <stdio.h>

main ( )
{
    int *ptr;
    ptr=malloc(20*sizeof(int));
}
```

Another function that is used to allocate memory dynamically in the C language is the `calloc` function. Unlike the `malloc` function, the `calloc` function initializes the allocated memory to the value zero. It returns the pointer to the first byte.

The `calloc` function accepts two parameters of the `size_t` data type.

The prototype of the `calloc` function is as follows. Two arguments, `n` and `size`, are accepted by this `calloc` function.

```
void *calloc(size_t n,size_t size);
```

The first parameter, `n`, represents the number of elements for which memory has to be allocated. The second parameter, `size`, represents the size of each element in bytes.

Consider the example , the first parameter to `the calloc` function is `5`. This indicates that memory has to be allocated for five elements.

```
int *arr;
arr=(int *)calloc(5,20);
```

In the prototype, the second parameter passed to the `calloc` function is `20`. This indicates that the size of each of the five elements is `20` bytes. After allocating `20` bytes of memory for each element, the `calloc` function initializes the allocated memory to the value `0`. The function returns a pointer to the first of the allocated memory. If the `calloc` function fails to allocate memory or the number of bytes is set to `0`, the `calloc` function returns a `NULL` value. You should always check the value returned by the `calloc` function. If a `NULL` value is returned, an error message can be displayed to the user.

Self Review Exercise 119.1

1. *Identify the prototype of the* `malloc` *function.*

   A. `void *malloc(size_t size);`
   B. `*malloc(size_t);`
   C. `void *malloc();`
   D. `void malloc(size_t);`

2. *Identify the piece of code that will make a program a program portable and allocate uninitialized memory.*

   A.
```
int *ar;
      ar=malloc(15*sizeof(int));
```
   B.
```
char*ch
      ch=malloc()
```
   C.
```
char*ch
      calloc(15,20);
```
   D.
```
int*ar;
      ar=calloc(15*sizeof(int));
```

LESSON 120 : OPTIMIZING MEMORY USE

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

When developing large applications, it is important to ensure that the memory allocated to the program is used optimally. The optimal use of memory enhances the processing of applications.

The C library provides several memory management. Functions to enable the optimum use of memory. These functions are `malloc`, `calloc`, `realloc`, and `free`. The optimal use of memory helps to improve performance. The `realloc` function is used to change the size of a memory block that was allocated using `malloc` or `calloc`.

The prototype of the `realloc` function is given below. In this prototype, the `ptr` parameter is a pointer to the original block of memory allocated using the `malloc` function or the `calloc` function.

```
void*realloc(void*ptr,size_t size);
```

The new size of the memory block is specified by the `size` variable. This variable is of the `size_t` data type.

A code sample to resize a memory block by using the `realloc` function is displayed. First, the code sample reserves a memory block of `17` bytes by using the `malloc` function. Next, a string of `16` characters is copied to the memory block.

Listing 120.1 : using realloc function

```
// Listing 120.1 : This program resizes a memory block by using the realloc function.

#include <stdlib.h>
#include <stdio.h>

main( )
{

     char *ptr;
     ptr=malloc(17);

               if(ptr= = NULL)
          {
               printf("\n This is an error");
               exit(1);
          }

     strcpy(ptr,"This is 16 chars");
     ptr=realloc(ptr,18);

               if(ptr= = NULL)
          {
               printf("\n This is an error");
               exit(1);
          }

          strcat(ptr, ".");
          printf(ptr);
          free(ptr);
}
```

Detailed Explanation

- After copying `16` characters to the memory block, another byte is added to this block so that a period can be placed at the end of the string of characters. To add one byte to an existing memory block, the block is resized using the `realloc` function.

Another memory management function that is used to optimize the use of memory is the `free` function. This function is used to deallocate the memory that is allocated using the `malloc` and `calloc` functions.

Deallocation of memory to make it available for future use. The prototype of the function is given below. In the prototype, the `free` function releases the memory of the node pointed by the pointer `ptr`.

```
void free(void *ptr)
```

If the `ptr` pointer points to `NULL`, the `free` function does not deallocate memory.

This lesson covered the memory management functions that can be used to manipulate and optimize the use of memory. These functions can be used to change the size of a memory block and deallocate memory.

Self Review Exercise 120.1

1. *Identify the prototype of the* `realloc` *function.*

   A. `void *realloc(void,size_t size);`
   B. `void *realloc(void *ptr,size_t size);`
   C. `void *realloc`
   D. `void *realloc(void *ptr);`

2. *Identify the use of the* `realloc` *function.*

   A. Resizes the memory allocated using the malloc function or the calloc function
   B. Deallocates memory allocated using the malloc function or the calloc function
   C. Allocates uninitialized memory
   D. Allocates initialized memory

REVIEW EXERCISE

1. Choose the code sample that represents that `Employee` *self-referential structure.*

A.
```
#include<stdlib.h>
struct Employee
{
    char EmployeeName[40];
    int EmployeeID;
    struct Employee *next;
};
```

B.
```
#include<stdlib.h>
struct
{
    char EmployeeName[40];
    int EmployeeID;
};
```

C.
```
#include<stdlib.h>
struct Employee
{
    char EmployeeName[40];
    int EmployeeID;
};
```

D.
```
#include<stdlib.h>
struct Employee
{
    char EmployeeName[40];
    int EmployeeID;
    Employee *next;
};
```

2. Partial code is given here. Identify the code that has to be included in the structure called `info` *to represent the code for a nested structure. Choose the appropriate code sample.*

```
struct item
{
       char ItemName[40];
       int ItemID;
};

struct info
{
       int InvNo;
       long InvDate;
       long InvAmt;
       long InvPaid;
};
```

A. struct info details;
B. item details;
C. info details;
D. struct item details;

3. Identify the code that can be used to allocate 75 *bytes of memory for each object in a group of* 20 *objects.*

*A. int*arr;callo(75,20);*
*B. int*arr; malloc(5,20);*
*C. int*arr; arr=malloc(5,20);*
D. arr=(int) callo(20,75);*

4. Select the option that specifies that use of the `realloc()` *function.*

A. Deallocates memory
B. Allocates uninitialized memory to programs.
C. Allocates initialized memory to variables
D. Modifies the number of memory blocks

LESSON 121 : LINKED LIST --> ADVANTAGES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Business applications usually involve storing and processing data. For example, a data entry program needs to store and process data. The list that contains data can be stored in memory by using arrays.

ARRAY DISADVANTAGES

There are a few disadvantages of using arrays to store data in memory. First, it is an inefficient method of storing data. This is because the array size is prespecified and memory is allocated for array elements when the array is created.

The prespecified size results in the waste of memory when the number of elements to be stored is less than the array size. It results in an overflow when the number of elements to be stored is greater than the array size.

Another disadvantage of using an array is that the process of inserting and deleting elements in an array is difficult because it involves the restructuring of the array.

You can overcome the disadvantage of using arrays by using a linked list to store data in memory. This lesson covers the advantage of using a linked list to store data in the memory of a computer.

Linked List Advantages

A linked list is a chain of structures in which each structure contains data and the pointers that store the addresses of the other logical structures in the list. The structures in a linked list are also called nodes.

An advantage of using a linked list to store data in memory is that the size of a linked list flexible. When you create a linked list , it is not necessary to know the number of nodes in a linked list and allocate memory for all the nodes.

In a linked list, a new node can be created dynamically at run time. Memory can be allocated to a node dynamically at run time, and the new node can be added to an existing list. When a node in a linked list is deleted, the memory occupied by the node can be released. Therefore, when you use linked lists, memory is not wasted.

Another advantage of using linked lists to store data in memory is that the processes of inserting and deleting nodes are efficient. You can insert or delete a node from a linked list by rearranging the pointers of the nodes that precede and succeed the node. When you add or delete a node in a linked list, the linked list need not be restructured.

Self Review Exercise 121.1

1. *Identify the advantage of using a linked list to store data in memory.*

   A. Memory is allocated for all nodes before the linked list is created.
   B. The process of inserting and deleting nodes do not require restructuring the list.
   C. Nodes can be added to a linked list by moving the existing nodes.
   D. The total number of nodes in the linked list is declared when the linked list is created.

LESSON 122 : SINGLE LINKED LIST FIRST NODE --> CREATING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

LINKED LIST

Linked list enables you to efficiently store and process data in the memory of a computer. This is because memory is not allocated for all the nodes of a linked list at the time of creating the list. Memory is dynamically allocated to nodes when they are added to list.

There are various types of linked lists. One of the most commonly used linked lists is a single linked list. In a single linked list, each node contains data and a single link that points to the succeeding node in the linked list.

The Two Parts of Single Linked Lists

Each node in a single linked list consists of two parts. The first part contains the data to be stored in the single linked list node. The second part contains a pointer that stores the address of the succeeding node in the single linked list. The last node does not point to any other node. The pointer of the last node is set to `NULL`. This indicates the end of the list. The address of the first node is stored in a pointer that is used to keep track of the beginning of the list.

Creating a Single Linked Lists

To create a single linked list, create the first node of the list and assign the address of the first node to the pointer that indicates the beginning of the list. The other nodes can be added to the list when required. The first step in the process of creating the first node of a single linked list is to declare a structure that represents a node of the single linked list.

Consider an example of a single linked list that is used to store information about certain products. The information to be stored is the codes and names of items. The item code is a string of 4 bytes, and the item name is a string of 12 bytes.

The code to declare the structure `ListTag` that represents a single linked list node is given.

Listing 122.1 : single linked list

```
// Listing 122.1 : This program enables users to create a single linked list.

#include <stdlib.h>
#include <stdio.h>

struct ListTag
{
     char ItermCode[5];
     char ItemName[13];
     struct ListTag *next;
};
```

It contains two variables, `ItemCode` and `ItemName`, to store the code and name of an item, respectively. The extra bytes in these variables is for the `NULL` character.

Another element of the structure `ListTag` is a pointer named `next`. This pointer is declared to store the address of the subsequent node in the single linked list.

After declaring the structure to represent a linked list node, a pointer named `start` that contains the address of the first node is declared.

Listing 122.2 : using single linked list

```
// Listing 122.2 : This program enables users to create a single linked list.

#include <stdlib.h>
#include <stdio.h>

struct ListTag
{

char ItermCode[5];
char ItemName[13];
struct ListTag *next;

};

struct ListTag *start;
struct ListTag *newnode;

//Creates a single linked list node by allocating memory and assigning data

GetNode( )
{

     newnode = (struct ListTag *)malloc(sizeof(struct ListTag));
     printf("Type the item code: ");
     scanf("%",newnode-> Item code);
     printf("Type the item name: ");
     scanf("%", newnode-> ItemName);
     newnode->next=NULL;

}

//Adds the first node to a single linked list

CreateList ( )
{

if(start= =NULL)
{

     GetNode( );
       Start = newnode;

    }

}
```

Detailed Explanation

- The node is created by invoking the user-defined function `GetNode`. This function starts the process of creating a linked list node by allocating memory for the node by using the `malloc` function. The size of the structure `ListTag` is passed as an argument to the `malloc` function.
- After allocating memory to the node, the `GetNode` function prompts the user to type the item code and item name for the first node.

- The item code and item name values typed by the user are assigned to the variables `ItemCode` and `ItemName`, respectively, in the linked list node by using the `scanf` function.
- The pointer `next` in the new node is assigned the `NULL` value. When this node is added to the linked list, the address of the subsequent node is assigned to the pointer `next`.
- Another user-defined function, `CreateList`, is declared to create the first node of the linked list. First, this function verifies whether the linked list exists. This is done by checking if the value of the pointer `start` is equal to `NULL`.
- If the pointer `start` is equal to `NULL`, a new node is created by invoking the `GetNode` function.
- Function creating the node by invoking the `GetNode` function, the address of the first node is assigned to the pointer `start`. This pointer indicates the beginning of the linked list.

Self Review Exercise 122.1

1. *You learned to declare the structure to represent a single linked list node. Four code sample are given below. Choose the code sample that represents a node in the linked list that stores the identification numbers, names, and address of employees now.*

A
```
struct Employee
{
char EmployeeID[5];
char Name[15];
char Address[25];
struct Employee *next;
}
```

B
```
struct Employee
{
char EmployeeID[5];
char Name[15];
char Address[25];
}
```

C
```
struct Employee
{
char EmployeeID[5];
char Name[15];
char Address[25];
char *next;
}
```

D
```
struct Employee
{
char EmployeeID[5];
char Name[15];
char Address[25];
struct ListTag *next;
}
```

Self Review Exercise 122.2

1. *You learned to create the first node of a single linked list. Identify the code sample used to create the first node of the single linked list in which nodes are represented by the structure Prod. Choose the correct code sample.*

```
GetNode()
{
  nd=(struct Prod *)malloc(sizeof(struct
  Prod));
  printf("Type the product ID : ");
  scanf("%s", nd->ProductID);
  nd->next=NULL;
}

CreateList()
{
  if(start==NULL)
  {
    GetNode();
    start=nd;
  }
}
```

A

```
GetNode()
{
  nd=(struct Prod *)malloc(sizeof(struct
  Prod));
  printf("Type the product ID : ");
  scanf("%s", nd->ProductID);
  nd->next=NULL;
}

CreateList()
{
  if(start==NULL)
  {
    GetNode();
    start=nd->next;
  }
}
```

B

```
GetNode()
{
  printf("Type the product ID : ");
  scanf("%s", nd->ProductID);
  nd->next=NULL;
}

CreateList()
{
  if(start==NULL)
  {
    GetNode();
    start=nd;
  }
}
```

C

```
GetNode()
{
  nd=(struct Prod *)malloc(sizeof(struct
  Prod));
  printf("Type the product ID : ");
  scanf("%s", nd->ProductID);
  nd->next=NULL;
}

CreateList()
{
  if(start==NULL)
  {
    GetNode();
  }
}
```

D

LESSON 123 : SINGLE LINKED LIST --> TRAVERSING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

To perform various tasks on a single linked list, such as counting, searching, or deleting nodes, it is necessary to access the linked list nodes. To access linked list nodes, the single linked list needs to be traversed. You can traverse a single linked list by using a pointer. To begin the process of traversing a single linked list, declare a pointer that points to the beginning of the list.

Next, access each node in the linked list by advantage the pointer until you reach the last node.

Consider an example of a single linked list that stores the codes and names of products. A structure named ListTag that is used to represent a node in the single linked list is given below. The address of the node is stored in a pointer name start.

Listing 123.1 : traversing single linked list

```
Listing 123.1 : this program deals with traversing single linked list

//This program enables users to traverse a single linked list.

#include <stdlib.h>
#include <stdio.h>
struct ListTag
{
        char ItemCode[5];
        char ItemName[13];
        struct ListTag *next;
};
struct ListTag *start;
struct ListTag *ptr;

//Traverses the nodes of the single linked list
DisplayList( )
{
            ptr=start;
            while(ptr!=NULL)
           {
                  printf("\nThe item code is %s",ptr-> ItemCode);
                  printf("\nThe item name is%s\n",ptr-> ItemName);
                  ptr=ptr->next;
       }
}
```

Detailed Explanation

- The data stored in the nodes of the single linked list is to be displayed. To display the data in each node, the single linked list needs to be traversed. A user-defined function, DisplayList, is declared to traverse the single linked list.
- The first step in the process of traversing a single linked list is to declare a pointer to store the address of the current single linked list node. In the example displayed on the screen, the pointer ptr is declared to store the address of the current node.

- To traverse the nodes of the single linked list, the second step is to set the pointer ptr to point to the first node of the single linked list. This is done by assigning the address in the pointer start to the pointer ptr.
- The third step is to display the item code and item name of each node in the linked list. This can be done using the printf function.
- Finally, advance the pointer ptr to point to the subsequent node. The pointer next contains the address of the subsequent node. Therefore, the pointer ptr can be advanced to the subsequent node by assigning the value in the pointer next to the pointer ptr.
- The third and fourth steps are repeated until the last node is reached. The last node is indicated by the NULL value in the pointer next. Therefore, the while loop continues until the value in the pointer ptr is NULL.

Self Review Exercise 123.1

1. *You learned to traverse a single linked list. Partial code used to traverse the nodes of a single linked list is given below. Tour code sample are also displayed. Choose the code sample required the to complete the partial code.*

```
#include<stdio.h>
#include<stdlib.h>
struct Employee
{
char Name[13];
struct Employee *newx;
};
struct Employee *start;
```

```
struct Employee *ptr;
traverselist()
{
ptr=start;
while(ptr!=NULL)
{
printf("%s", ptr-> Name);
}
}
```
A

```
char *ptr;
traverselist()
{
ptr=start;
while(ptr!=NULL)
{
printf("%s", ptr-> Name);
ptr=ptr->next;
}
}
```
B

```
struct Employee *ptr;
traverselist()
{
ptr=start;
while(ptr!=NULL)
{
printf("%s", ptr-> Name);
ptr=ptr->next;
}
}
```
C

```
struct Employee *ptr;
traverselist()
{
ptr=NULL;
while(ptr!=NULL)
{
printf("%s", ptr-> Name);
ptr=ptr->next;
}
}
```
D

LESSON 124 : UNSORTED SINGLE LINKED LIST NODE --> ADDING

Lesson Objectives
In this Lesson you will learn :   - *the Genesis of C Language*   - *the Stages in the Evolution of C Language*

Single linked lists are useful for storing and processing data. The addition of nodes is an important step in the process of creating a single linked list. Single linked lists can be sorted or unsorted. In a sorted single linked list, nodes are arranged in a sorted order according to the data stored in nodes. In addition, new nodes are inserted at the sorted in a sorted single linked list.

In an unsorted single linked list, nodes are not sorted in any order. New nodes are either added in the beginning or at the end of an unsorted single linked list. In this lesson, you learn to add nodes in the beginning of an unsorted single linked list.

Steps for Adding Nodes in the Beginning of an Unsorted Single Linked Lists
Declare a pointer that will contain the address of the new node.
Allocate memory and assign data values to the new node.
Link the new node to the current first node.
Update the pointer that indicates the beginning of the linked list.

There are four steps in the process of adding nodes in the beginning of an unsorted single linked list.

- The first step is to declare a pointer that will store the address of the new node.
- The second step is to create the new node by allocating memory for the new node and assigning data values to the data part of the node.
- The third step is to link the new node to the current first node.
- Finally, the pointer that indicates the beginning of the single linked list is updated with the address of the new node.

Consider an example of a single linked list that is used to store product data in memory. The nodes of this single linked list are represented by the structure `ListTag`. The variables `ItemCode` and `ItemName` contain the codes and names of products, respectively.

Listing 124.1 : Adding nodes in an unsorted order

```
//Listing 124.1 : This program enables users to add nodes in an unsorted order.

#include <stdlib.h>
#inlcude <stdio.h>

struct ListTag
{
        char ItemCode[5];
        char ItemName[13];
        struct ListTag *next;
};

Struct  ListTag  *start;
struct ListTag *newnode;

//Creates a single linked list node by allowing memory and assigning data

GetNode( )
{
         newnode=(struct ListTag *)malloc(sizeof(struct ListTag));
         printf("Type the item code: ");
         scanf("%s",newnode->ItemCode);
         printf("Type the item name: ");
         scanf("%s",newnode-> ItemName);
         newnode->next=NULL;
}

//Adds nodes to a single linked list in an unsorted order
UnsortedInsertList( )
{
         GetNode( );
        newnode->next=NULL;
              start=newnode;
    }
```

Detailed Explanation

- The pointer next contains the address of the succeeding node in the single linked list. A pointer named `start` is used to indicate the beginning of the single linked list. The pointer start contains the address of the first node of the single linked list.
- To begin the process of adding nodes in the beginning of the single linked list, declare a pointer named `newnode`. This pointer stores the address of the node to be added to the single linked list.
- The second step for adding a node is to create the new node. To do this, you declare a user-defined function named `GetNode`. This function allocates memory for the new node by using the `malloc` function available in the standard ANSI C library.
- After allocating memory for the new node, the `GetNode` function prompts the user to type the code and name of the product. The item code and item name values typed by the user are assigned to the variables `ItemCode` and `ItemName`, respectively.
- Next, the pointer `next` of the new node is assigned the `NULL` value.
- After the new node is created, it is should be added in the beginning of the single linked list. To do this, another user-defined function named `UnsortedInsertList` is declared. This function invokes the . `GetNode` function to create the node.

- After invoking the `GetNode` function, the new node is to be linked to the current first node in the list. The pointer `start` contains the address of the first node. The new node is linked to the first node by assigning the address in the pointer `start` to the pointer `next` of the new node.
- After linking the new node to the first node of the linked list, the pointer `start` is updated with the address of the new node.

Self Review Exercise 124.1

1. *You learned the steps for adding nodes in the beginning of an unsorted single linked list . Four sequences are given below. Choose the correct sequence of steps for adding a node in the beginning of an unsorted single linked list now.*

   A. Declare a pointer to store the address of the new node.
   Allocate memory and assign data values to the new node.
   Link the new node to the current first node.
   Update the pointer that indicates the beginning of the linked list.

   B. Declare a pointer to store the address of the new node.
   Link the new node to the current first node.
   Allocate memory and assign data values to the new node.
   Update the pointer that indicates the beginning of the linked list.

   C. Allocate memory and assign data values to the new node.
   Declare a pointer to store the address of the new node.
   Link the new node to the current first node.
   Update the pointer that indicates the beginning of the linked list.

   D. Declare a pointer to store the address of the new node.
   Update the pointer that indicates the beginning of the linked list.
   Allocate memory and assign data values to the new node.
   Link the new node to the current first node.

Self Review Exercise 124.2

1. *You learned to add a node in the beginning of an unsorted single linked list. Partial code used to add a node in the beginning of an unsorted single linked list is given. Choose the code sample required to complete the partial code now.*

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

struct Employee
{
         char EmployeeID[5];
         struct Employee *next;
};
struct Employee *start,*newnode;

Getnode( )
{
        newnode=(struct Employee *)malloc(sizeof(struct Employee));
        printf("Type the employee ID: ");
        scanf("%s",newnode->EmployeeID);
        newnode->next=NULL;
}
```

```
InsertedUnsortedList( )
{
         GetNode( );
          newnode->next=start;
}
```
A

```
InsertUnsortedList( )
{
         GetNode( );
         newnode->next=start;
         start=newnode;
}
```
B

```
InsertUnsortedList( )
{
         newnode->next=start;
}
```
C

```
InsertUnsortedList( )
{
          GetNode( );
          newnode->next=NULL;
          start=newnode;
}
```
D

LESSON 125 : SORTED SINGLE LINKED LIST NODE --> ADDING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

In a sorted single linked list, nodes are sorted according to the data stored in the nodes. In a sorted single linked list, new nodes should be added in the sorted order to maintain the sorted order of the single linked list. To add a node to a single linked list that is sorted in ascending order, the sorted position of the node the single linked list should be found.

The sorted position of a node is found by traversing the single linked list until a node that contains the data value greater than the data value of the node to be added is located. If a single linked list is sorted in descending order, the sorted position of a new node is located by traversing the single linked list until a node that contains the data value less than the data value of the node to be added is located.

Next, rearrange the links to add the node is the sorted position. When you add a new node to a sorted linked list, three situations can occur. The new node can be added in the beginning, middle, or end of the list.

The C instruction to add a node in the middle and end of the list are identical to each other. However, the C instructions to add a node in the beginning of the linked list are different

Consider an example of a single linked list that stores product details. It is sorted in ascending order according to the item code. The nodes of this single linked list are represented by the structure `ListTag` given below.

LESSON 125 : SORTED SINGLE LINKED LIST NODE --> ADDING

Listing 125.1 : adding nodes in a sorted order

```
//Listing 125.1 : This program enables users to add nodes in a sorted order.

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct ListTag
{
     char ItemCode[5];
     char ItemName[13];
     struct ListTag *next;
};
struct ListTag *start;
struct ListTag *newnode;
struct ListTag *prev;
struct ListTag *ptr;

//Creates a single linked list node by allocating memory and assigning data

    GetNode( )
    {
     newnode=(struct ListTag *)malloc(sizeof(struct ListTag));
     printf("Type the item code: ");
     scanf("%s",newnode->ItemCode);
     printf("Type the item name:  ");
     scanf("%s",newnode->ItemName);
     newnode->next=NULL;
    }

//Adds nodes to a single linked list in the sorted order

SortedInsertList( )
    {
     GetNode( );
     for(ptr= =start;(ptr!=NULL)&&strcmp(newnode->ItemCode,ptr->
     if(ptr= =start)
   {
          newnode->next=start;
          start=newnode;
   }
    else
    {
           pre->next=newnode;
           newnode->next=ptr;
    }
}
```

Detailed Explanation

- The addresses of the successive nodes in the single linked list are stored in the pointer `next`. The pointer start contains the address of the first node of the single linked list.
- A pointer named `newnode` is declared to store the address of the new node to be added to the single linked list.
- To find the sorted position of the new node in the single linked list, find the address of the adjacent nodes. The pointers `prev` and `ptr` point to the nodes before and after the position where the new node is to be added, respectively.
- A user-defined function, `GetNode`, is declared to create a new node. This function creates a new node by allocating memory to the node by using the `malloc` function and then prompting the user to type the item code and the item name.

- Next, the values typed by the user are assigned to the node elements `ItemCode` and `ItemName`. The pointer `next` of the new node is assigned the `NULL` value.
- To add the new node created by the user-defined function `GetNode`, another user-defined function named `SortedInsertList` is declared. The `SortedInsertList` function invokes the `GetNode` function to create a new node.
- After the new node is created, the linked list is traversed to find the sorted position of the new node. This is done using the `for` loop. The pointer `ptr` is set to point to the first node by assigning the address in the pointer `start` to the pointer `ptr`.
- The pointer ptr is advanced until you reach a node that contains an item code value greater than the item code value in the new node or you reach the end of the list.
- Before advancing the pointer `ptr`, its value is assigned the pointer `prev`. This is done to ensure that the pointer `prev` points to the node that precedes the point where the new node is to be added.
- To add the new node at the beginning of the single linked list, assign the address the address of the current first node stored in the pointer `start` to the pointer `next` of the new node.
- Next, update the pointer `start` with the address of the new node.
- If the pointer `ptr` is not equal to `start`, the new node should be added in the middle or at the end of the single linked list.
- The node pointed to by the pointer `prev` the point where the new node is to be added. The pointer next of this node is assigned the value in the pointer value in the pointer `newnode`.
- The node pointed to by the pointer `ptr` succeeds the point where the new node is to be added. The address in the pointer `ptr` is assigned to the pointer `next` of the new node. If the new node is to be added at the end of the single linked list, the value in the pointer `ptr` is `NULL`.

Self Review Exercise 125.1

1. *In a sorted linked list, a new node is always added:*

   A. at the position sorted according to the data value in the node.
   B. at the position sorted according to the addresses of nodes.
   C. in the beginning of the list.
   D. at the end of the list.

Self Review Exercise 125.2

2. *You learned to find the sorted position of a new node in a sorted single linked list. Partial code used to find the sorted position of a new node is given below. Choose the correct code sample used to complete this partial code now.*

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
struct ProductList
{
          char ProductID[5];
          struct ProductList *next;
};
struct ProductList *start,*newnode,*ptr,*prev;
GetNode( )
{
         newnode=(struct ProductList *)malloc(sizeof(struct ProductList));
         printf("Type the product ID: ");
         scanf("%s",newnode->ProductID);
         newnode->next=NULL;
}
sortedinsertlist( )
{
                      GetNode( );
```

A. `for(ptr=start;(ptr!=start)&&strcmp(newnode->ProductID,ptr->ProductID) = = 0;ptr=ptr->next);`

B. `for(ptr=NULL;(ptr!=NULL)&&strcmp(newnode->ProductID,ptr->   ProductID) >0;prev=ptr,ptr=ptr->next);`

C. `for(ptr=start;(ptr!=NULL)&&strcmp(newnode->ProductID,ptr->ProductID)<=0;prev=ptr);`

Self Review Exercise 125.3

3. *You learned to add a node to a sorted single linked list. Partial code used to add a node to a sorted single linked list is given below. Choose the code sample required to complete this partial code now.*

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
struct ProductList
{
        char ProductID[5];
        struct ProductList *next;
};
struct ProductList *start,*newnode,*ptr,*prev;
GetNode( )
{
        newnode=(struct ProductList *)malloc(sizeof(struct ProductList));
        printf("Type the product ID: ");
        scanf("%s", newnode->ProductID);
        newnode->next=NULL;
}
sortedinsertlist( )
{
  GetNode ( )
  for(ptr=start;(ptr!=NULL)&&strcmp(newnode->ProductID,ptr-
>ProductID);prev=ptr,ptr=ptr->next);
}
```

```
 if(ptr= = start)
{
newnode->next=start;
start=newnode;
}
else
{
prev->next=newnode;
newnode->next=ptr;
}
```
A

```
if(ptr= =NULL)
{
newnode->next=start;
start=newnode;
}
else
{
prev->next=newnode;
newnode->next=ptr;
}
```
B

```
if(ptr= =start)
{
newnode->next=start;
start=newnode;
}
else
{
prev->next=newnode;
newnode->next=prev;
}
```
C

```
if(ptr= = start)
{
newnode->next=start;
}
else
{
prev->next=newnode;
newnode->next=ptr;
}
```
D

LESSON 126 : SINGLE LINKED LIST NODE --> SEARCHING

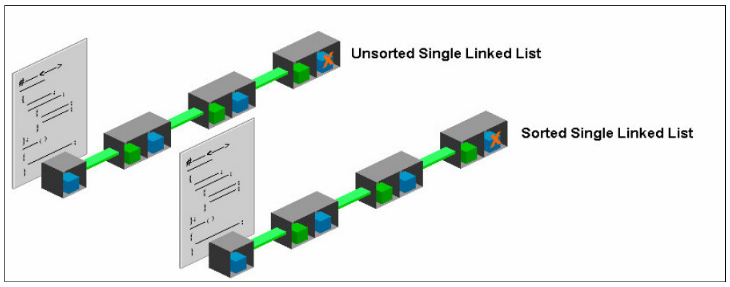
Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

You may want to search for a node in a single linked list that contains specific data. Consider an example of a single linked list that contains product data. To display the name of the product for which the item code is `20`, search for the node in which the item code value is `20`.

In this lesson, you learn to search for a single linked list node that contains specific data. The code to search for a node in an unsorted single linked list is different from searching a node in a sorted single linked list.



Consider an example of an unsorted single linked list in which nodes are represented by the structure `ListTag`. Each node in the unsorted single linked list contains the item code of a product and a pointer named `next` that stores the address of the subsequent node.

LESSON 126 : SINGLE LINKED LIST NODE --> SEARCHING

Listing 126.1 : searching nodes in an unsorted single linked list

```
// Listing 126.1 :This program enables users to search nodes in an unsorted single linked
// list.

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct ListTag
{

     char ItemCode[5];
     char ItemName[13];
     struct ListTag *next;

};

struct ListTag *start;
struct ListTag *ptr;

//Searches for a node in an unsorted single linked list

searchUnsortedList( )
{

     char num[5];
     printf("Type the code of the item to be searched:  ");
     scanf("%s",num);
     for(ptr=start;(ptr!=NULL)&&strcmp(num,ptr->ItemCode)!=0;ptr=ptr->next);

         if(ptr= =NULL)
        {

                 printf("\nEnd of list");

         }

         else
         {

              pintf("\nThe item code in the node is %s",ptr->ItemCode);

         }

}
```

Detailed Explanation

- The address of the first node in the single linked list is stored in a pointer named `start`.
- To search for a node in the single linked list, the single linked list needs to be traversed. A pointer named `ptr` is declared to store the address of the current node while traversing the single linked list.
- A user-defined function, `SearchUnsortedList`, is declared to search for a node in an unsorted single linked list. First, the user is prompted to type the item code of the node that is to be located. The item code value typed by the user is stored in a string variable named `num`.
- Next, the pointer `ptr` is set to point to the first node. This is done by assigning the address in the pointer start to the pointer `ptr`.

- Next, the single linked list is traversed and the item code value in each node is compared with the `num` value typed by the user by using the `for` loop. This is done until the node that is being searched for is found or the end of the list is reached.
- In the `for` loop, the pointer `ptr` is advanced until the node that contains the `num` value typed by the user is found or the end of the single linked list is reached.
- After you quit the `for` loop, verify if the value of the pointer `ptr` is `NULL` The `NULL` value in the pointer `ptr` indicates that the last node is reached. In this situation, display the message that the end of the list is reached.
- However, if the value of the pointer `ptr` is not `NULL`, it indicates that the node that contains the item code value equal to the `num` value typed by the user is located. Display the item code of the node pointed to by the pointer `ptr`.

The code used to search for a node in a sorted single linked list is different. Consider an example of a single linked list that is sorted in ascending order. The nodes of this single linked list are represented by the structure given below.

LESSON 126 : SINGLE LINKED LIST NODE --> SEARCHING

Listing 126.2 : searching nodes in a sorted single linked list

```
// Listing 126.2 : This program enables users to search nodes in a sorted single linked
// list.

#include <string.h>
#include  <stdlib.h>
#include <stdio.h>

struct ListTag
{

      char ItemCode[5];
      char ItemName[13];
      struct ListTag *next;

 };

 struct ListTag *start;
 struct ListTag *ptr;

//Searches for a node in a sorted single linked list

SearchSortedList( )
{

       char val[5];
       printf("Type the code of the item to be searched:  ");
       scanf("%s",val);
       for(ptr=start;(ptr!=NULL)&&strcmp(val,ptr->ItemCode)>0;ptr=ptr->next);

       if(ptr= =NULL)
       {
          printf("\nEnd of list");
       }

       else
       {
          if(strcmp(val,ptr->ItemCode)= =0)
          {
              printf("\nThe record searched for is %s",ptr->ItemCode);
          }

          else
          {
              printf("\nNode does not exist.");
          }

        }

}
```

Detailed Explanation

- To search for a node in the single linked list represented by the structure `ListTag`, a pointer named `ptr` is declared. This pointer is used to point to the current node for traversing the single linked list.
- To find a node in a sorted single linked list, a user-defined function, `SearchSortedList`, is declared. This function prompts the user to type the item code of the node to be searched for by using the `printf` function.
- The item code value typed by the user is stored in the variable `val` by using the `scanf` function.

- To being the process of searching for a single linked list node, assign the address of the first node to the pointer `ptr`.
- Next, compare the item code value in the node to be searched for with the item code values in the single linked list nodes by using the `for` loop. This is done until the required node is found or a node that contains a greater item code value is found.
- If the single linked list is sorted in descending order, the item code value in the node to be searched for is compared with the item code values in the single linked list nodes. This is done until the required node is found or a node that contains a smaller item code value is found.
- In the `for` loop, advance the pointer `ptr` until you reach the end of the list or you find a node that contains an item code value greater than or equal to the item code value of the node to be searched for.
- After quitting the `for` loop, verify if the value of the pointer `ptr` is `NULL` if the pointer `ptr` is `NULL`, display the message that the end of the list is reached.
- If the value of the pointer `ptr` is not `NULL`, verify if the item code value of the node pointed to by the pointer `ptr` is equal to the item code value of the item to be searched for. It the two values are equal, display that the required node is located.
- However, if the item code value of the node pointed to by the pointer `ptr` and the node to be searched for are not equal, display a message that states that states that the node does not exist.
- Notice that codes used to search for a node in a sorted single linked list and an unsorted single linked list are slightly different. In an unsorted single linked list, if the node to be searching for does not exist, the complete list is searched before quitting the `for` loop.
- However, in a sorted single linked list that is sorted in ascending order, if the node being searched for does not exist, the for loop terminates when you locate a node that contains a data value a data value greater than the item code value being searched for.

Self Review Exercise 126.1

1. *You learned to search for a node is an unsorted single linked list. Partial code used to search for a node in an unsorted single linked list is given below. Choose the appropriate code sample used to complete this partial code.*

```
// Listing 126.1 :This program enables users to search nodes in an unsorted
//single linked list.

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct ListTag
{

     char ItemCode[5];
     char ItemName[13];
     struct ListTag *next;

};

struct ListTag *start;
struct ListTag *ptr;

//Searches for a node in an unsorted single linked list

searchUnsortedList( )
{

     char num[5];
     printf("Type the code of the item to be searched:  ");
     scanf("%s",num);
         if(ptr= =NULL)
        {

                 printf("\nEnd of list");

         }

         else
         {

              pintf("\nThe item code in the node is %s",ptr->ItemCode);

         }

}
```

A. `for(ptr=start;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)==0;ptr=ptr->next);`

B. `for(ptr=NULL;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)!=0;ptr=ptr->next);`

C. `for(ptr=start;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)!=0;ptr=ptr->next);`

D. `for(ptr=start;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)==0;ptr=ptr->start);`

Self Review Exercise 126.2

2. *You learned to search for a node in a sorted single linked list. Partial code used to search for a node in a single linked list sorted in ascending order is given below. Choose the appropriate code sample used to complete this partial code.*

```
// Listing 126.2 : This program enables users to search nodes in a sorted
//single linked list.

#include <string.h>
#include  <stdlib.h>
#include <stdio.h>

struct ListTag
{

      char ItemCode[5];
      char ItemName[13];
      struct ListTag *next;

 };

 struct ListTag *start;
 struct ListTag *ptr;

//Searches for a node in a sorted single linked list

SearchSortedList( )
{

       char val[5];
       printf("Type the code of the item to be searched:  ");
       scanf("%s",val);

       if(ptr= =NULL)
       {
          printf("\nEnd of list");
       }

       else
       {
          if(strcmp(val,ptr->ItemCode)= =0)
          {
              printf("\nThe record searched for is %s",ptr->ItemCode);
          }

          else
          {
              printf("\nNode does not exist.");
          }

        }

}
```

A. `for(ptr=start;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)>0;ptr=ptr->next);`
B. `for(ptr=NULL;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)>0;ptr=ptr->next);`
C. `for(ptr=start;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)<0;ptr=ptr->next);`
D. `for(ptr=start;(ptr!=NULL)&&strcmp(num, ptr->EmployeeID)==0;ptr=ptr->start);`

LESSON 127 : SINGLE LINKED LIST NODE --> DELETING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A single linked list can be used to store data in memory. When the data stored in a node of the single linked list is not required, the node should be deleted and the memory occupied by the node should be released.

The process of deleting a node from a single linked list involves two activities, the logical and physical deletion of the node.

The logical deletion of a node involves removing the links that attach node to the single linked list. This is done by assigning the address of the succeeding node to the pointer `next` of the previous node. The pointer `next` stores the address of the succeeding node.

The physical deletion of a node involves releasing the memory occupied by the node. This can be done by using the `free` function provided by the standard ANSI C library. To delete a specific node in a linked list, the linked list is traversed to find the node to be deleted. The C instructions to delete a node from the beginning of a linked list are different from the C instructions to delete a node from the middle and at the end of the list.

Consider an example of an unsorted single linked list in which nodes are represented by the structure `ListTag`. The single linked list contains the code and names of the items. The pointer `next` points to the succeeding node in the single linked list.

Listing 127.1 : Deleting nodes from a single linked list

```
//Listing 127.1 : This program enables users to delete nodes from a single linked list.

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct ListTag
{
         char ItemCode[5];
         char ItemName[13];
         struct ListTag *next;
};
struct ListTag *start;
struct ListTag *ptr,*prev;

//Deletes a node from a single linked list

DeleteList( )
{
   char item[5];
   printfs("Type the item code of the node to be deleted:  ");
   scanf("%s",item);
   for(ptr=start;(ptr!=NULL)&&strcmp(item,ptr- >ItemCode)!=0;prev=ptr,ptr=ptr->next);

   if(ptr!=NULL)
   {
      if(ptr= =start)
      {
        start=ptr->next;
        free(ptr);
      }
      else
        {
           prev->next=ptr->next;
           free(ptr);
        }
      }

      else
      {
        printf("\nNode to be deleted does not exist.");
      }
     }
```

- The single linked list should be traversed to find the node to be deleted.
- First, declare a pointer name `ptr` to store the address of the node that is to be deleted. In addition, declare another pointer, `prev`, that stores the address of the node preceding the node that is to be deleted.
- A user-defined function, `DeleteList`, is declared to delete a single linked list node that contains specific data. This function prompts the user to type the code of the item to be deleted. The item value typed by the user is assigned to a variable named item.
- Next, the single linked list should be traversed to find the node to be deleted. To begin this process, set the pointer `ptr` to point to the first node of the single linked list.

- Next, the `for` loop is executed until the end of the list is reached or a node in which the item code value is equal to the item code value of the node to be deleted is located.
- Assign the value of the pointer `ptr` to the pointer `prev` before advancing the pointer `ptr`. This is done to ensure that the pointer `prev` points to the node that precedes the node to be deleted.
- After quitting the `for` loop, verify whether the value in the pointer `ptr` is `NULL`. If the value in the pointer `ptr` is not `NULL`, verify if the value in the pointer `ptr` is equal to `start`.
- If the value in the pointer `ptr` is equal to `start`, the node to be deleted is the first node of the linked list.
- To delete the first node of the single linked list, assign the address of the node that is currently the second node to the pointer start. This removes the links the links attach the first node to the single linked list.
- Next, the memory occupied by the node to be deleted should be released. This is done using the `free` function. To release the memory occupied by the node pointed to by the pointer `ptr`, the value in the pointer `ptr` is passed as an argument to the `free` function.
- You learned about the code used to delete the first node in a single linked list. However, if the pointer `ptr` is not equal to start after quitting the `for` loop, the node to be deleted is either in the middle of the single linked list or at the end of the single linked list.
- The node to be deleted is pointed to by the pointer `ptr`, and the preceding node is pointed to by the pointer `prev`.
- To logically delete the node pointed to by the pointer `ptr`, assign the value in the pointer next of the node to be deleted to the pointer `next` of the preceding node. This removes the link that attaches the node pointed to by the pointer `ptr` to the single linked list.
- After deleting the node logically, delete the node pointed to by the pointer `ptr` physically by using the `free` function.
- Another situation is also possible. The pointer `ptr` can be equal to `NULL` after quitting the `for` loop used to find the node to be deleted. This indicates that the end of the list is reached. In this situation, display a message that the node to be deleted does not exist in the list.

Self Review Exercise 127.1

1. *The process for deleting a node in a linked list involves:*

   A. Releasing the memory occupied by the node.
   B. Deleting the link address stored in the pointer of the node.
   C. Assigning the address of the succeeding node to the pointer of the preceding node and releasing the memory occupied by the node.
   D. Deleting the data stored in the node and releasing the memory occupied by the node.

2. *You learned to delete the first node in a single linked list. Partial code used to delete the node that contains specific data is displayed on the screen. Identify the code sample used to delete the first node in the single linked list. Choose the correct code sample now.*

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
struct Employee
{
            char EmployeeID[5];
            struct Employee *next;
};
struct Employee *start,*ptr,*prev;
DeleteList( )
{
            char eid[5];
            printf("Type the employee ID value of the node to be deleted:  ");
            scanf("%s",eid);
            for(ptr=start;(ptr!=NULL)&&strcmp(eid,ptr->EmployeeID)!=0;prev=ptr,ptr=ptr
-       >next);
           if(ptr!=NULL)
          {
                  if(ptr= =start)
          {


          }
```

A. `start=ptr;`
       `free(ptr);`
B. `start=ptr->next;`
       `free(ptr);`
C. `start=ptr->next;`
D. `start=ptr->next;`
       `free(start);`

LESSON 128 : INSERTING ELEMENTS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Data processing in a data structure involves the insertions and deleting of data. In a data structure, such as a linked list, insertions and deletions take place in a random order.

However, there are situation when data can be inserted and deleted from the top or bottom of a list.

Stacks

Stacks are an ordered collection of items into which new items are inserted and deleted from the top of the stack list. Stacks are also called Last In First Out(LIFO) lists because the last item inserted in the stack is the first item to be deleted.

The example details a stack of item numbers, if you add a new item number to the existing item numbers, the new item number must be inserted above the first item number. Insertions and deletions in stack are popularly known as push and pop, respectively.

A code sample for inserting an item in a stack is given below. In this code, the push function allocates memory to an item that has to be inserted in the stack. The address of the item is assigned to the pointer `ptr`.

Listing 128.1 : inserting items in a stack

```
//Listing 128.1 : This program inserts an item in a stack.
#include <stdio.h>
#include <stdlib.h>
struct tag
{
         int rum;
         struct tag *next;
}*top,*ptr;
push( )
{
  prt=(struct tag *)malloc(sizeof(struct tag));
  printf (“\nPlease input the EmpID:  “);
  scanf(“%d”,&ptr->num);
   if(top= =NULL)
   {
      top=ptr;
   }
   else
   {
      ptr->next=top;
      top=ptr;
   }
 }
```

Detailed Explanation

- After allocating memory, the `push` function requests the user to type the employee ID. The new employee ID is inserted at the top of the stack.
- If you remove a particular item from the stack, the order for deleting items will be in the reverse order of inserting items. Consider the code sample given below. In this code sample, the order for deleting items will start from the top of the stack.

Listing 128.2 : deleting an item from a stack

```
//Listing 128.2 : This program deletes an item from a stack.
#include <stdio.h>
#include <stdlib.h>
struct tag
{
        int no;
        struct tag *next;
}*top,*ptr;
pop( )
{
          printf("\nThe EmpID: %d",top->no);
          ptr=top;
          top=top->next;
          free(ptr);
}
```

Detailed Explanation

- The `top` pointer stores the address of the first item in the stack. The `next` pointer stores the address of the second item in the stack. Before deleting an item, the `top` pointer is moved to the next item in the stack.
- After moving the pointer to the next item in the stack, you can deallocate memory for the first item in the stack by using the `free` function.

Self Review Exercise 128.1

1. *You learned about implementation of stacks. Choose the code sample to insert an item into a stack.*

```
push()
{
ptr=(struct Stag*)malloc(sizeof(struct Stag));
printf("\nPlease input the employee name: ");
scanf("%s",ptr->EmpName);

if(ptr!=NULL)
        ptr->next=top;
        top=ptr;
else
        break;
}
```
A

```
push()
{
ptr=(struct Stag*)malloc(sizeof(struct Stag));
printf("\nPlease input the employee name: ");
scanf("%s",ptr->EmpName);

if(top==NULL)
        top=ptr;
else
        ptr->next=top;
        top=ptr;
}
```
B

```
push()
{
ptr=(struct Stag*)malloc(sizeof(struct Stag));
printf("\nPlease input the employee name: ");
scanf("%s",ptr->EmpName);

if(ptr==NULL)
        top=ptr;
else
        ptr->next=top;
        top=ptr;
}
```
C

```
push()
{
ptr=(struct Stag*)malloc(sizeof(struct Stag));
printf("\nPlease input the employee name: ");
scanf("%s",ptr->EmpName);

if(ptr!=NULL)
        top=ptr;
else
        top=NULL;
}
```
D

Lesson Objectives
In this Lesson you will learn : ☛ *the Genesis of C Language* ☛ *the Stages in the Evolution of C Language*

Infix Notation

The most common method of writing arithmetic expressions is placing an operator symbol between two operands. For example, to add the operands x and y, the addition operator is placed between the two operands. This is called the infix notation.

X+Y

Postfix Notation

There is another notation called the postfix notation in which the operator symbol is placed after the two operands. For example, when adding the operands x and y, the addition operator will be placed after the two operands.

XY+

The computer usually evaluates an arithmetic expression written in the infix notation in two steps. First, it converts the infix expression to a postfix expression and then it evaluates the postfix expression.

A fundamental property of the postfix notation is that parentheses are not required when writing expression in the postfix notation. The step-by-step logic to evaluate a postfix expression is given below. The first step is to assign the postfix expression as a string and add a parenthesis at the end of the string.

Evaluating a Postfix Expression	
Step 1	Assign the postfix expression as a string and add a parenthesis at the end of the string
Step 2	Scan the postfix expression from left to right until the parenthesis is found.(Repeat steps 3 and 4 until the parenthesis is found.)
Step 3	If an operand is encountered, convert it into a number and push it into the stack.
Step 4	If an operator is encountered, remove the top two operands from the stack, evaluate the operands depending upon the operator encountered, and place the result in the stack.
Step 5	Print the top element in the stack.

Listing 129.1 : evaluating postfix expression

```
// Listing 129.1 : This program evaluates a postfix expression.
#include <stdio.h>
#include <mallod.h>
#define SIZE_OF_PFEXPR 30
struct StackTag
{
        int item;
        struct StakTag *next;
}      *ptr,*top;

char pfexpr[SIZE_OF_PFEXPR],*ch;
int valueA,valueB,chint;

main( )
{
        top=NULL;
        printf("\nPlease input the postfix expression: ");
        scanf("%s",pfexpr);
        stract(pfexpr,");
        for(ch=pfexpr;*ch!=')';ch++)
       {
        switch(*ch)
            {
                  case '+':
                          valueA=pop( );
                          valueB=pop( );
                          push(valueB+valueA);
                          break;

                   case '-':
                           valueA=pop( );
                           valueB=pop( );
                           push(valueB-valueA);
                           break;
                   case '*':
                           valueA=pop( );
                           valueB=pop( );
                           push(valueB*valueA);
                           break;

                   case '/':
                           valueA=pop( );
                           valueB=pop( );
                           push(valueB/valueA);
                           break;

                   default:
                           chint=*ch-'0';
                           puch(chint);
               }
         }
          printf("\n\nResult of postfix expression is: %d",pop( ));
}
  push(SomeInt)
int SomeInt;
{
            ptr=(struct StrackTag*)malloc(sizeof(struct StrackTag));
            ptr->item=SomeInt;
            ptr->next=top;
            top=ptr;
}
pop( )
{
          int PoppedItem;
          PoppedItem=top->item;
          ptr=top;
          top=top->next;
          free(ptr);
          return PoppedItem;
}
```

Next, the code scans the postfix expression from left to right until the end of the string. The end of the string will be indicated when a parenthesis is encountered.

If the addition operator is encountered, the top two operands, `valueA` and `valueB`, are removed from the stack. After the operands `valueA` and `valueB` are removed, they are added and the result is pushed back into the stack. If an operand is encountered, it is converted into an integer and pushed into the stack.

Finally, the `pop` function is used to retrieve the last element from the stack. The `printf` statement is used to display the element that is the result of the postfix evaluation.

Self Review Exercise 129.1

1. *You learned about the steps to evaluate a postfix expression. A sequence of steps to evaluate the postfix expression is given below on the left side. Choose the missing step below on the right side.*

Assign the postfix expression as a string, and add a parenthesis at the end of the string.	*If an operator is encountered, insert and result of the evaluation into the stack.* A
Scan the string from left to right until an operand is found.	*If an operator is encountered, remove the top two elements from the stack.* B
▨	*If an operand is encountered, remove all the contents of the stack.* C
Print the top element in the stack.	*If an operand is encountered, move the pointer to the second item in the stack.* D

2. *You learned about the method to evaluate a postfix notation. The push and pop functions in the code sample insert and delete items, respectively. Choose the code sample used to evaluate the operands and place the result in the stack.*

A
```
case ‘-’ :
   ValueA=push();
   ValueB=push();
   Push(ValueB-ValueA);
break;
```

B
```
case ‘-’ :
   ValueA=push();
   ValueB=pop();
   Push(ValueB-ValueA);
break;
```

C
```
case ‘-’ :
   ValueA=pop();
   ValueB=pop();
   Pop(ValueB-ValueA);
break;
```

D
```
case ‘-’ :
   ValueA=pop();
   ValueB=pop();
   Push(ValueB-ValueA);
break;
```

LESSON 130 : INSERTING ELEMENTS

Lesson Objectives

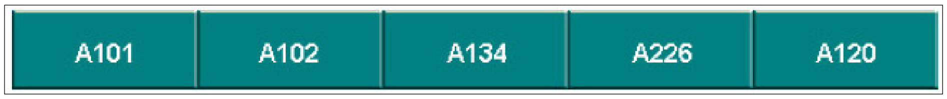
In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A queue is a data structure in which items can be inserted in one end and deleted from the other end. A queue is also called first in first out(FIFO) method because the first item inserted in a queue is also the first item to be deleted.

A common use of queues is for printing documents. A printer stores the documents that have to be printed in a queue. The first document that is sent to the printer queue is the first document to be printed.

A diagrammatic representation of a queue is given below. This queue displays a list of item numbers.



The item number `A118` is to be inserted in the queue. This item number will be added after the item number `A120`. This is because insertions can take place only from the rear end of the queue. Queues are representing by a similar structure that is used to represent a stack and a single linked list.

A code sample representations in a queue is given below. This code sample accepts an item number and then inserts it at the end of the queue.

Listing 130.1 : working with queue data structure

```
//Listing 130.1 : This program declares a queue structure.

#include <stdio.h>
#inlcude <stdlib.h>
#define SIZE_OF_ITEMNO5
struct QueueTag
{
          char ItemNo[SIZE_OF_ITEMNO];
          struct  QueueTag *next;
}*rear;

insert( )
{
       struct QueueTag *ptr;
       ptr=(struct QueueTag *)malloc(sizeof(struct QueueTag));
       printf("%s",ptr->ItemNo);
       if(rear!=NULL)
      {
             rear->next=ptr;
      }
       rear=ptr;
}
```

Detailed Explanation

- There are two pointer variables declared in the code. Memory is allocated to the pointer `ptr` that stores the address of the queue element to be inserted in the queue.

- The other pointer variable declared is `rear`. this variable is declared to store the address location of the last item inserted in the queue.
- The program obtains an item number from the user. After accepting the input, the program reads the item number to the new node to be added.
- The variable `ptr` that contains the address location of the new node is inserted at the end of the queue. The pointer variable `rear` is then updated so that it points to the new item that is inserted.

Self Review Exercise 130.1

1. *You learned about inserting items in a queue. Four code samples are given below. Identify the code sample used it insert an item in a queue. Choose the appropriate code sample that will insert the item in the queue.*

```
insert()
{
struct Stag *ptr;
ptr=(struct Stag *)malloc(sizeof(struct
      Stag));
printf("\nPlease input the item number:");
Scanf("%s", ptr->ItemNo);

if(rear!=NULL)
{
rear -> next=ptr;
}
rear=ptr;
}
```

A

```
insert()
{
struct Stag *ptr;
ptr=(struct Stag *)malloc(sizeof(struct
      Stag));
printf("\nPlease input the item number:");
Scanf("%s", ptr->ItemNo);
if(rear!=NULL)
rear=ptr;
}
```

B

```
insert()
{
struct Stag *ptr;
ptr=(struct Stag *)malloc(sizeof(struct
      Stag));
printf("\nPlease input the item number:");
Ptr->next=rear;
Scanf("%s", ptr->ItemNo);

if(rear!=NULL)
top=ptr;
}
```

C

```
insert()
{
struct Stag *ptr;
ptr=(struct Stag *)malloc(sizeof(struct
      Stag));
printf("\nPlease input the item number:");
Scanf("%s", ptr->ItemNo);
if(rear!=NULL)
Ptr->next=rear;
}
```

D

LESSON 131 : DELETING ELEMENTS

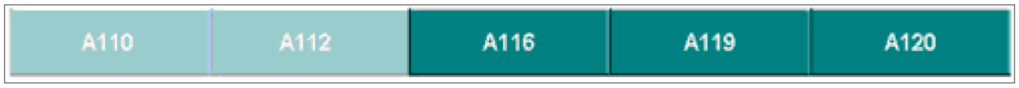
Lesson Objectives

In this Lesson you will learn :

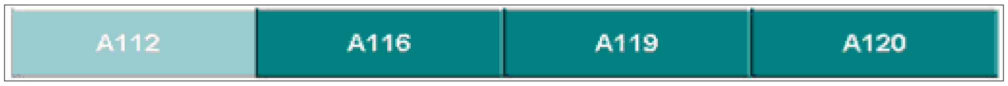
- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

You may want to delete unwanted elements in a queue periodically. In this lesson, you learn the method to delete elements from a queue. In a queue, the order of deletion depends on the order of insertion. The first elements to be deleted.

A queue structure representing a list of items is given below. The item `A112` has to be deleted. To delete this item, the item `A110` has to be deleted before deleting `A112`. This is because the item `A110` is the first item in the queue.



After deleting the item `A110`, the item `A12` can be deleted



The code sample to delete an element from a queue is given below. In the code sample, the first element in the queue is to be deleted. Two pointers, `ptr` and `front`, are declared. The `front` pointer stores the address of the first element in the queue.

<u>`Listing 131.1 : deleting an item from a queue`</u>

```
//Listing 131.1 : This program deletes an item from a queue.

#include <stdio.h>
#include <stdlib.h>
#define SIZE_OF_ITEMNO 5

struct QueueTag
{
        char ItemNo[SIZE_OF_ITEMNO];
        struct QueueTag *next;
};
struct QueueTag *ptr,*front;

delete ( )
{
         printf("\nItem number deleted is: %s",front->ItemNo);
         ptr=front;
         front=front->next;
         free(ptr);
}
```

Detailed Explanation

- The `delete` function contains a set of C instructions to delete the element. The instruction first prints the item number to be deleted using the `printf` function.
- The first item number is assigned to `ptr` for deallocating memory later.
- Next, the `front` pointer is moved to the succeeding item in the queue structure.
- After moving the pointer to the next item number, you deallocate the memory assigned to the first item number. You can deallocate memory by using the `free` function.

Self Review Exercise 131.1

1. *You learned about deleting items from a queue. Four code samples are given below. Choose the correct code sample required to delete* `EmpID` *from the queue.*

```
struct queueTag *ptr, *front;
delete()
{
printf("\EmpID deleted: %s", front->EMPID);
front=front->next;
free(Ptr);
}
```
A

```
struct queueTag *ptr, *front;
delete()
{
printf("\EmpID deleted: %s", front->EMPID);
ptr=front;
front=front->next;
free(Ptr);
}
```
B

```
struct queueTag *ptr, *front;
delete()
{
printf("\EmpID deleted: %s", front->EMPID);
ptr=front;
front=front->next;
}
```
C

```
struct queueTag *ptr, *front;
delete()
{
ptr=front;

printf("\EmpID deleted: %s", front->EMPID);
free(Ptr);
}
```
D

LESSON 132 : FIRST NODE --> CREATING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Single linked lists are not suitable for the database or word-processing applications that are required to scan and process data in the forward and backward directions. This is because nodes in a single linked list can be traversed only in the forward direction.

To access a node before the current node in a single linked list, the traversal procedure must start from the first node. To overcome this limitation, you use a double linked list. Double linked lists are useful for applications such as word processors and databases.

A double linked list is a linked list in which nodes are represented by structures that contain data and two links. One link stores the address of the succeeding node, and the other link contains the address of the preceding node. A double linked list contains pointers to both the preceding and succeeding nodes. Therefore, you can traverse a double linked list in the forward and backward directions.

An example of a structure that is used to represent a node in a double linked list is given below. This structure contains the variable `ItemCode` that stores data, and it also contains two pointers, `previous` and `next`.

```
struct DoubleList
{
  char ItemCode[5];
  struct DoubleList *previous;
  struct DoubleList *next;
};
```

The pointer `previous` contains the address of the preceding node in the double linked list while the pointer `next` contains the address of the succeeding node in the double linked list. The first node in a double linked list points to only the succeeding node, and the last node in a double linked list points to only the preceding node. Therefore, the pointer `previous` of the first node and the pointer of the last node are assigned the `NULL` value.

The address of the first node in a double linked list is stored in a pointer that indicates the beginning of the double linked list. The address of the last node in a double linked list is stored in another pointer that indicates the end of the double linked list.

To create a double linked list, the first step is to declare a structure that represents the nodes in a double linked list. After declaring the structure to represent a node, create the first node of the linked list. This involves allocating memory to the node, assigning the data value, and assigning the address of the node to the pointers that store the addresses of the first and last nodes. The other nodes can be added to the double linked list subsequently.

Consider an example of the structure `DoubleList` given below. This structure is declared to represent a node of a double linked list that stores the item code of products. The item code value is store in a character variable named `ItemCode`. The pointers `previous` and `next` are used to store the addressing of the preceding and succeeding nodes, respectively.

Listing 132.1 : creating nodes of a double linked list

```
// Listing 132.1 : This program creates the first node of a double linked list.

#include <stdlib.h>
#include <stdio.h>

struct DoubleList
{
        char ItemCode[5];
        struct DoubleList *previous;
        struct DoubleList *next;
};

struct DoubleList *start;
struct DoubleList *last;
struct DoubleList *newnode;

//Creates a node by allocating memory and assigning data

GetNode( )
{
       newnode=(struct DoubleList *)malloc(sizeof(struct DoubleList));
       printf("Type the item code:  ");
       scanf("%s", newnode->ItemCode);
       newnode->next=NULL;
       newnode->previous=NULL;
}
//Creates the first node of the double linked list

DlistCreate( )
{
         if(start= =NULL)
        {
                GetNode( );
                start=newnode;
                last=newnode;
        }
}
```

Detailed Explanation

- The pointers `start` and `last` are declared to store the addresses of the first and last nodes of the double linked list, respectively.
- Next, a pointer named `newnode` is declared. This pointer contains the address of the new node that is to be added to the double linked list.
- To allocate memory and assign data to the node, a user-defined function, `GetNode`, is declared. This function allocates memory to the node by using the `malloc` function. The size of the linked list node is passed as an argument to the `malloc` function.
- After allocating memory to the node, the user is prompted to type an item code value. The item code value typed by the user is assigned to the variable `ItemCode.Next`, the function `Get-Node` assigns the `NULL` value to the pointers `next` and `previous`.

- Another user-defined function, `DlistCreate`, is declared to create the first node of the double linked list. This function tests the value in the pointer `start` to verify whether a double linked list exist.
- If the value in the pointer `start` is `NULL`, this indicates that the list does not exist. In this situation, the function `DlistCreate` invokes another user-defined function, `GetNode`, to create the first node.
- Next, the pointer `start` is assigned the address of the new node. The first node in the double linked list is also the last node because this is the only node in the linked list. Therefore, the pointer last is assigned the address of the new node.

Self Review Exercise 132.1

1. *The structure that represents a double linked list node contains:*

   A. Data and a pointer to the succeeding node.
   B. Data, a pointer to the succeeding node, and a pointer to the first node.
   C. Data and a pointer to the previous node.
   D. Data and pointers to the previous node and succeeding node.

Self Review Exercise 132.2

2. *You learned to declare a structure to represent a double linked list node, allocate memory to the new node, and assign data to the new node. Choose the code used to declare the structure* Emp*, allocate memory to the new node, and assign data to the new node.*

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

Struct Emp  {

Char EmpName[25];
Struct Emp *previous;

};

Struct Emp *start, *last, *node;
GetNode()  {

Node=(struct Emp *)malloc(sizeof(struct Emp));
Printf("Type the name of the employee" ");
Scanf("%s", node->EmpName);
Node->next=NULL;
Node->previous=NULL;

}
```
A

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

Struct Emp  {

Char EmpName[25];
Struct Emp *previous;
Struct Emp *next;

};

Struct Emp *start, *last, *node;
GetNode()  {

Printf("Type the name of the employee" ");
Scanf("%s", node->EmpName);
Node->next=NULL;
Node->previous=NULL;

}
```
B

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

Struct Emp  {

Char EmpName[25];
Struct Emp *previous;
Struct Emp *next;

};

Struct Emp *start, *last;
GetNode()  {

Node=(struct Emp *)malloc(sizeof(struct Emp));
Printf("Type the name of the employee" ");
Scanf("%s", node->EmpName);
Node->next=NULL;
Node->previous=NULL;
}
```
C

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

Struct Emp  {

Char EmpName[25];
Struct Emp *previous;
Struct Emp *next;

};

Struct Emp *start, *last, *node;
GetNode()  {

Node=(struct Emp *)malloc(sizeof(struct Emp));
Printf("Type the name of the employee" ");
Scanf("%s", node->EmpName);
Node->next=NULL;
Node->previous=NULL;
}
```
D

STRUCTURED PROGRAM DEVELOPMENT USING C PROGRAMMING LANGUAGE

LESSON 132 : FIRST NODE --> CREATING

Self Review Exercise 132.3

3. *You learned to create the first node of a double linked list. Partial code used to create the first node of the double linked list that is based on the structure* `Employees` *is given below. Choose code sample required to complete the partial code.*

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

struct DoubleList
{
        char ItemCode[5];
        struct DoubleList *previous;
        struct DoubleList *next;
};

struct DoubleList *start;
struct DoubleList *last;
struct DoubleList *newnode;

//Creates a node by allocating memory and assigning data

GetNode( )
{
       newnode=(struct DoubleList *)malloc(sizeof(struct DoubleList));
       printf("Type the item code:  ");
       scanf("%s", newnode->ItemCode);
       newnode->next=NULL;
       newnode->previous=NULL;
}
```

`DlistCreate()`
`{`
`  if(start==NULL)`
`  {`
`   GetNode();`
`   last=newnode;`
`  }`
`}`
A
	`DlistCreate()`
	`{`
	`  if(start==NULL)`
	`  {`
	`   GetNode();`
	`   start=newnode;`
	`   last=newnode;`
	`  }`
	`}`
	B

`DlistCreate()`
`{`
`  if(start==NULL)`
`  {`
`   GetNode();`
`   Start=NULL;`
`  }`
`}`
C
	`DlistCreate()`
	`{`
	`  if(start==NULL)`
	`  {`
	`   start=newnode;`
	`   last=newnode;`
	`  }`
	`}`
	D

LESSON 133 : DOUBLE LINKED LIST : TRAVERSING

Lesson Objectives
In this Lesson you will learn :
☛ *the Genesis of C Language*
☛ *the Stages in the Evolution of C Language*

In database and word-processing applications, it is preferable to use double linked lists to store data in memory. This is because double linked list enable you to access and process data in both the forward and backward directions.

To access and process the data stored in the nodes of a double linked list, the double linked list needs to be traversed. In this lesson, you learn to traverse a double linked list in the forward and backward directions.

The process of traversing a double linked list in the forward direction involves four steps. The first step is to declare a pointer to store the address of the current node in the double linked list while traversing the nodes of the double linked list.

The second step is to assign the address of the first node to the pointer declared in the first step. The third step is to process the information in the node that is currently pointed to by the pointer. The last step is to advance the pointer to point to the succeeding node in the double linked list.

Steps for Traversing a Double Linked List in the Forward Direction
Declare a pointer to store the address of the current node in the double linked list.
Assign the address of the first node to the pointer.
Process the information in the node that is currently pointed to by the pointer.
Advance the pointer to point to the succeeding node in the double linked list.

The third and fourth steps in the process of traversing a double linked list, processing information and advancing the pointer, are repeated until the information in the last node of the list is displayed.

Consider an example of a double linked list in which nodes are represented by the structure `DoubleList` given below. The pointers `previous` and `next` are used to store the addresses of the preceding and succeeding nodes, respectively.

LESSON 133 : DOUBLE LINKED LIST : TRAVERSING

Listing 133.1 : Traversing a double linked list forward

```
// Listing 133.1 : This program traverses a double linked list in the forward
// direction.

#include <stdlib.h>
#include <stdio.h>

struct DoubleList
{
        char ItemCode[5];
        struct DoubleList *previous;
        struct DoubleList *next;
};

struct DoubleList *start;
struct DoubleList *last;
struct DoubleList *ptr;

ForwardTraverse( )
{
      ptr=start;
      while(ptr!=NULL)
     {
          printf("\nThe item code is %s",ptr->ItemCode);
          Ptr=ptr->next;
     }
}
```

Detailed Explanation

- The address of the first and last nodes of the double linked list are stored in the pointers `start` and `last`, respectively. These pointers are used to indicate the beginning and end of the list.
- The first step for traversing the double linked list is to declare a pointer to store the addresses of the nodes while traversing the list. The pointer `ptr` is declared to store the address of the current node in the double linked list.
- A user-defined function, `ForwardTraverse`, is declared to traverse the double linked list in the forward direction. In this function, the process of traversing the double linked list is initiated by setting the pointer `ptr` to point to the first node of the double linked list.
- Next, the information stored in the node pointed to by the pointer `ptr` is displayed using the `printf` function.
- After displaying the information stored in the node pointed to by the pointer `ptr`, the pointer `ptr` is advanced to point to the succeeding node.
- The C instructions to display the information in the node pointed to by the pointer `ptr` and advance the pointer `ptr` are enclosed in a `while` loop. The `while` loop executes these statements until the value in the pointer `ptr` is `NULL`, which indicates that the last node is reached.

You learned to traverse double linked list nodes in the forward direction.

Steps for Traversing a Double Linked List in the Backward Direction
Declare a pointer to store the address of the current node in the double linked list.
Assign the address of the last node to the pointer.
Process the information in the node that is currently pointed to by the pointer.
Advance the pointer to point to the preceding node in the double linked list.

The process of traversing double linked list nodes in the backward direction also involves four steps.

- The first step is to declare a pointer to store the address of the current node while traversing the list.
- The second step is to assign the address of the last node of the linked list to the pointer declared in the first step.
- The third step involves processing the information in the node that is currently pointed to by the pointer.
- Finally, advance the pointer in the backward direction. This is done by setting the pointer to point to the previous node. To traverse the complete linked list, repeat the third and fourth steps until you reach the first node.

Notice that in the process of traversing a double linked list in the forward direction, the pointer is assigned the address of the first node initially and the pointer is advanced by setting it to point to the succeeding node.

However, in the process of traversing a double linked list in the backward direction, the pointer is assigned the address of the last node initially and the pointer is advanced by setting it to point to the previous node.

An example of the structure that represents a double linked list is displayed on the screen. The variable `ItemCode` stores the item code of the products. The pointers `previous` and `next` point to the preceding and succeeding nodes in the linked list, respectively.

Listing 133.2 : traversing double linked list backward

```
// Listing 133.2 : This program traverses a double linked list in the backward direction.

#include <stdlib.h>
#include <stdio.h>

struct DoubleList
{
        char ItemCode[5];
        struct DoubleList *previous;
        struct DoubleList *next;
};

struct DoubleList *start;
struct DoubleList *last;
struct DoubleList *ptr;

ReverseTraverse( )
{
       ptr=last;
       while(ptr!=NULL)
       {
              printf("\nThe current item code is %s",ptr->ItemCode);
              ptr=ptr->previous;
       }
 }
```

Detailed Explanation

- The beginning and end of the double linked list are indicated by the pointers `start` and `last`, respectively. The pointer `ptr` is declared to store the address of the current node in the double linked list for traversing the nodes of the list.
- To start the process of traversing the linked list in the backward direction, declare a user-defined function, `ReverseTraverse`. First, this function assigns the address of the last node to the pointer `ptr`. Next, the `printf` function is used to display the information in the node pointed to by the pointer `ptr`.
- Finally, the pointer `ptr` is set to point to the node preceding the node pointed to by the pointer `ptr`.
- The information in the node pointed to by the pointer `ptr` is displayed and the pointer `ptr` is set to point to the preceding node until the value in the pointer `ptr` is `NULL`. The `NULL` value indicates that the first node of the list is reached.

Self Review Exercise 133.1

1. *You learned about the steps that are involved in the process of traversing a double linked list in the forward direction. Four sequences are given below. Choose the correct sequence of steps to traverse a double linked list in the forward direction.*

Set the pointer to point to the first node.
Declare a pointer to store the address of the current node.
Process the data contained in the node that is pointed to by the pointer.
Advance the pointer to point to the succeeding node.
A

Declare a pointer to store the address of the current node.
Advance the pointer to point to the succeeding node.
Set the pointer to point to the first node.
Process the data contained in the node that is pointed to by the pointer.
B

Declare a pointer to store the address of the current node.
Set the pointer to point to the first node.
Process the data contained in the node that is pointed to by the pointer.
Advanced a pointer to point to the succeeding node.
C

Declare a pointer to store the address of the current node.
Process the data contained in the node that is pointed to by the pointer.
Set the pointer to point to the first node.
Advance the pointer to point to the succeeding node.
D

Self Review Exercise 133.2

2. *You learned to traverse to traverse a double linked list in the forward direction. Four code samples are given below. Choose the code sample used to traverse the double linked list, in which nodes are represented by the structure Employees, in the forward direction.*

```
#include <stdlib.h>
#include <stdio.h>
#struct Employees
{
   char EmpName[25];
   struct Employees *previous;
   struct Employees *next;
};
struct Employees *start, *last;
```

A

```
ForwardTraverse( )
{
 ptr=start;
 while(ptr!=NULL)
 {
  printf("\nItem code: %s",ptr->
                      EmpName);
  Ptr=ptr->next;
 }
}
```

B

```
struct Employees *ptr;
ForwardTraverse( )
{
 ptr=start;
 while(ptr!=NULL)
 {
  printf("\nItem code: %s",ptr->
                      EmpName);
  Ptr=ptr->previous;
 }
}
```

C

```
struct Employees *ptr;
ForwardTraverse( )
{
 ptr=NULL;
 while(ptr!=NULL)
 {
  printf("\nItem code: %s",ptr->
                      EmpName);
  Ptr=ptr->next;
 }
}
```

D

```
struct Employees *ptr;
ForwardTraverse( )
{
 ptr=start;
 while(ptr!=NULL)
 {
  printf("\nItem code: %s",ptr->
                      EmpName);
  Ptr=ptr->next;
 }
}
```

Self Review Exercise 133.3

3. *You learned about the steps that are involved in the process of traversing a double linked list in the backward direction. Four sequences are given below. Choose the correct sequence of steps to traverse a double linked list in the backward direction.*

Declare a pointer to store the address of the current node.
Process the information stored in the node pointed to by the pointer.
Assign the address of the last node to the pointer.
Set the pointer to point to the previous node.

A

Declare a pointer to store the address of the current node.
Assign the address of the last node to the pointer.
Process the information stored in the node pointed to by the pointer.
Set the pointer to point to the previous node.

B

Assign the address of the last node to the pointer.
Declare a pointer to store the address of the current node.
Process the information stored in the node pointed to by the pointer.
Set the pointer to point to the previous node.

C

Process the information stored in the node pointed to by the pointer.
Declare a pointer to store the address of the current node.
Assign the address of the last node to the pointer.
Set the pointer to point to the previous node.

D

Self Review Exercise 133.4

4. *You learned to traverse to traverse a double linked list in the reverse direction. Four code samples are given below. Choose the code sample used to traverse the double linked list in the reverse direction.*

```
#include <stdlib.h>
#include <stdio.h>
#struct Employees
{
   char EmpName[25];
   struct Employees *prior;
   struct Employees *next;
};
struct Employees *start, *last;
```

A

```
Struct Employees *ptr;
ReverseTraverse( )
{
 ptr=last;
 while(ptr!=NULL)
 {
  printf("\nEmp Name : %s",ptr->
                     EmpName);
  Ptr=ptr->prior;
 }
}
```

B

```
struct Employees *ptr;
ReverseTraverse( )
{
 ptr=NULL;
 while(ptr!=NULL)
 {
  printf("\nEmp Name : %s",ptr->
                     EmpName);
  Ptr=ptr->prior;
 }
}
```

C

```
struct Employees *ptr;
ReverseTraverse( )
{
 ptr=last;
 while(ptr!=NULL)
 {
  printf("\nEmp Name : %s",ptr->
                     EmpName);
  Ptr=ptr->next;
 }
}
```

D

```
ReverseTraverse( )
{
 ptr=last;
 while(ptr!=NULL)
 {
  printf("\nEmp Name : %s",ptr->
                     EmpName);
  Ptr=ptr->prior;
 }
}
```

LESSON 134 : LIST NODE --> INSERTING

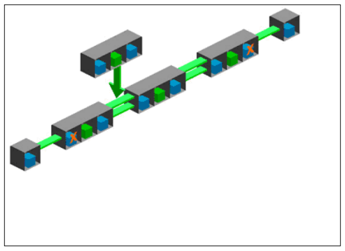
Lesson Objectives
In this Lesson you will learn :
the Genesis of C Language
the Stages in the Evolution of C Language

You inserted nodes in a double linked list when you create a double linked list or update information in an existing double linked list. This lesson covers the C code used to insert a node in a double linked list.

When you are creating a double linked list, it is preferable to insert nodes in the sorted position. This is because it is easier to maintain the sorted order of a linked list at the time of creating the list rather than sorting the list after it is created.

In addition, if you are adding nodes to an existing double linked list that is sorted by the information stored in the nodes, insert the nodes in the sorted position to maintain the sorted order of the double linked list.

Steps for Inserting a Node in a Double Linked List
Declare a pointer what will store the address of the new node.
Create a new node by allocating memory and assigning data.
Locate the sorted position of the new node.
Insert the new node in the sorted position.



The process of inserting nodes in a double linked list involves four steps.

- The first step is to declare a pointer that will store the address of the new node.
- The second step is to create the new node by allocating memory for the node and assigning a data value to the node.
- The third step in the process of adding a new node is to locate the sorted position of the new node in the double linked list.
- The last step is to insert the new node in the sorted position in the double linked list by rearranging the pointers.

When you insert a node in a sorted double linked list, there are three possibilities. The new node can be inserted in the beginning, middle, or end of the double linked list. The C code to insert the node is different in these three situations.

Consider an example of a double linked list in which nodes are represented by the structure `DoubleList`. The structure `DoubleList` contains the item code value and two pointers, `previous` and `next`. The double linked list is sorted in ascending order according to item code values.

LESSON 134 : LIST NODE --> INSERTING

Listing 134.1 : inserting nodes in a double linked list

```
//Listing 134.1 : This program inserts nodes in a double linked list.

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct DoubleList
{
         char ItemCode[5];
         struct DoubleList *previous;
         struct DoubleList *next;
};

struct DoubleList *start;
struct DoubleList *last;
struct DoubleList *newnode;
struct DoubleList *ptr;
struct DoubleList *prev;

//Creates a new node by allocating memory and assigning data
GetNode( )
{
       newnode=(struct DoubleList *)malloc(sizeof(struct DoubleList));
       printf(“Type the item code:  “);
       scanf(“%s”,newnode->ItemCode);
       newnode->next=NULL;
       newnode->previous=NULL;
}

//Inserts the new node at the sorted position in a double linked list

DlistInsert( )
{
GetNode( );
for(ptr=start;(ptr!=NULL)&&strcmp(newnode->ItemCode,ptr->ItemCode)>0;prev=ptr,ptr=ptr-
                                                                              >next);
if(ptr= =NULL)
{
   newnode->previous=prev;
   prev->next=newnode;
   last=newnode;
}
else
{
   if(ptr= =start)
   {
      newnode->next=start;
      start->previous=newnode;
      start=newnode;
   }
   else
   {
      newnode->next=ptr;
      newnode->previous=prev;
      prev->next=newnode;
      ptr->previous=newnode;
   }
  }
}
```

Detailed Explanation

- In the structure example, the pointers `previous` and `next` are used to store the address of the preceding and succeeding nodes, respectively.
- Two pointers `start` and `last`, are declare to indicate the beginning and end of the double linked list, respectively. The pointer `start` points to the first node, and the pointer `last` points to the node.
- A pointer to store the address of the new node should be declared before creating the new node. The pointer `newnode` is declared to point the node to be inserted in the double linked list.
- To create a new node to be inserted in the double linked list, a user-defined function, `GetNode`, is declared. This function allocates memory for the new node by using the `malloc` function.
- Next, the `GetNode` function prompts the user to type an item code value. The item code value typed by the user is assigned to the `ItemCode` variable in the new node.
- Finally, the pointers `next` and `previous` in the node are assigned the `NULL` value.
- Another user-defined function, `DlistInsert`, is declared to insert nodes in the double linked list in the sorted order. This function begins the process of inserting a new node by invoking the `GetNode` function to create a new node.
- The second step in the process of inserting a new node is to locate the sorted position of the node. To locate the sorted position, two pointers are required to point to the nodes after and before the point on the linked list where the new node is to be inserted.
- The pointer `ptr` is declared to store the address of the node that is after the point where the new node is to be inserted. Another pointer named `prev` is declared to store the address of the node located before the sorted position of the new node.
- In the for loop , the pointer `ptr` is set to point to the first node in the beginning.
- Next, the `ItemCode` value of the new node is compared with the `ItemCode` value of the node pointed to by the pointer `ptr` and pointer `ptr` is advanced. This is done until you find a node that contains the `ItemCode` value greater than the new node or the last node is reached.
- If the double linked list is sorted in descending order, the `for` loop continues until last node is reached or a node that contains a data value less than the data value of the new node is located.
- The pointer `prev` is assigned the value in the pointer `ptr` before the pointer `ptr` is advanced. This is done to ensure that the pointer `prev` points to the node before the point at which the new node is to be inserted.

- After quitting the `for` loop, verify whether the value of the pointer `ptr` is `NULL`. If the pointer `ptr` is equals to `NULL`, the new node should be inserted at the end of the list.
- The C instructions to insert a node at the end of the double linked list are given. The pointer previous of the new node should store the address of the current last node that is pointed to by the pointer `prev`.
- The pointer `next` of the current last node should be assigned the address of the new node. Finally, the pointer `last` should be updated with the address of the new node.
- You learned the C code to insert a node at the end of a double linked list node. If the pointer `ptr` is not equal to `NULL` after quitting the `for` loop, verify if the pointer `ptr` is equal to `start`. In this situation, the new node should be inserted in the beginning of the double linked list.
- To insert the new node in the beginning, the pointer `next` of the new node should point to the current first node and the pointer `previous` of the current first node should point to the new node.
- Next, the pointer `start` must be updated with the address of the new node to indicate the current beginning of the linked list.
- If the pointer `ptr` is not equal to `start` after quitting the `for` loop, the new node should be inserted in the middle of the list. The sorted position of the new node is between the pointers `prev` and `ptr`.
- To insert the node between the nodes pointed to by the pointers `ptr` and `prev`, the pointer `previous` and `next` of the new node should store the addresses of the nodes before and after the new node, respectively.
- In addition, the address of the new node should be assigned to the pointer `next` of the node preceding the new node and the pointer `previous` of the node succeeding the new node.

Self Review Exercise 134.1

1. *You learned the sequence of steps that in the process of inserting a node in a sorted double linked list four sequences are given below. Choose the correct sequence of steps used to insert a node in the sorted position in a double linked list now.*

A. Declare a pointer that will store the address of the new node.
Create a new node by allocating memory for the node and assigning the data value to the node elements.
Rearrange the pointers in the linked list to attach the new node to the linked list.
Locate the sorted position of the node in the double linked list.

B. Declare a pointer that will store the address of the new node.
Create a new node by allocating memory for the node and assigning the data value to the node elements.
Locate the sorted position of the node in the double linked list.
Rearrange the pointers in the linked list to attach the new node to the linked list.

C. Declare a pointer that will store the address of the new node.
Locate the sorted position of the node in the double linked list.
Create a new node by allocating memory for the node and assigning the data value to the node elements.
Rearrange the pointers in the linked list to attach the new node to the linked list.

D. Declare a pointer that will store the address of the new node.
Rearrange the pointers in the linked list to attach the new node to the linked list.
Create a new node by allocating memory for the node and assigning the data value to the node elements.
Locate the sorted position of the node in the double linked list.

Self Review Exercise 134.2

➥ *You learned to create a new node by allocating memory and assigned data t the node. Four code sample are given below. Choose the correct code sample used to create a new node to be inserted in the double linked list represented by the structure* `Emp` *now.*

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
struct Emp
{
char EmpName[25];
struct Emp *previous;
struct Emp *next;
};
struct Emp *start, *last;
```

```
GetNode()
{
node=(struct Emp *)malloc(sizeof(stuct Emp));
printf("Type the employee name: ");
scanf("%s",node->EmpName);
node->next=NULL;
node->previous=NULL;
}
```
A

```
struct Emp *node;
GetNode()
{
node=(struct Emp *)malloc(sizeof(stuct Emp));
printf("Type the employee name: ");
node->next=NULL;
node->previous=NULL;
}
```
B

```
struct Emp *node;
GetNode()
{
node=(struct Emp *)malloc(sizeof(stuct Emp));
scanf("%s",node->EmpName);
node->next=NULL;
node->previous=NULL;
}
```
C

```
struct Emp *node;
GetNode()
{
node=(struct Emp *)malloc(sizeof(stuct Emp));
printf("Type the employee name: ");
scanf("%s",node->EmpName);
node->next=NULL;
node->previous=NULL;
}
```
D

Self Review Exercise 134.3

3. *You learned to insert a new node at the end of a double linked list. Partial code used to insert a node in a double linked list is given below. Four code samples are also shown. Choose the correct code sample used to insert a node at the end.*

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
struct Emp
{
char EmpName[25];
struct Emp *previous;
struct Emp *next;
};
struct Emp *start, *last, *ptr, *prev, *node;
Getdata()
{
  node=(struct Emp *)malloc(sizeof(stuct Emp));
  printf("Type the employee name: ");
  node->next=NULL;
  node->previous=NULL;
 }
DlistInsert( )
{
GetNode( );
for(ptr=start;(ptr!=NULL)&&strcmp(node->EmpName,ptr->EmpName)>0;prev=ptr,ptr=ptr->

        next);
if(ptr= =NULL)
{

}
```

```
node ->previous=prev;
prev->next=node;
```
A

```
node ->previous=NULL;
prev->next=node;
last=node;
```
B

```
node ->previous=prev;
prev->next=node;
last=node;
```
C

```
node ->previous=prev;
prev->next=NULL;
last=node
```
D

Self Review Exercise 134.4

4. *You learned to insert a node in a double linked list, Partial code used to insert a node in a double linked list given below. Identify the code sample used to insert a node in the beginning of the double linked list. Tick on the correct code* sample.t

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
struct Emp
{
char EmpName[25];
struct Emp *previous;
struct Emp *next;
};
struct Emp *start, *last, *ptr, *prev, *node;
Getdata()
{
node=(struct Emp *)malloc(sizeof(stuct Emp));
printf("Type the employee name: ");
node->next=NULL;
node->previous=NULL;
}
DlistInsert( )
{
GetNode( );
for(ptr=start;(ptr!=NULL)&&strcmp(node->EmpName,ptr->EmpName)>0;prev=ptr,ptr=ptr->

       next);
if(ptr= =NULL)
{
   node->previous=prev;
   prev->next=node;
   last=node;
}
else
{
   if(ptr= =start)
   {

       }
```

```
node->next=start;
start->previous=node;
start=node;
```
A

```
node->next=start;
start->previous=node;
```
B

```
node->next=start;
start->next=new;
start=node;
```
C

```
start->previous=node;
start=node;
```
D

Self Review Exercise 134.5

5. *Partial code used to insert a node in a double linked list is displayed here. Identify the code sample used to insert a node in the middle of the double linked list. Tick the correct code sample.*

```
;
struct Emp *next;
};
struct Emp *start, *last, *ptr, *prev, *node;
Getdata()
{
 node=(struct Emp *)malloc(sizeof(stuct Emp));
 printf("Type the employee name: ");
 node->next=NULL;
 node->previous=NULL;
 }
  DlistInsert( )
   {
    GetNode( );
    for(ptr=start;(ptr!=NULL)&&strcmp(node->EmpName,ptr->EmpName)>0;prev=ptr,ptr=ptr-
>next);
if(ptr= =NULL)
{
node->previous=prev;
prev->next=node;
last=node;
 }
else
{

}
}
}
```

```
node ->next=ptr;
node ->previous=NULL;
prev->next=node;
ptr->previous=node;
```
A

```
node ->next=ptr;
node ->previous=prev;
prev->next=node;
ptr->previous=node;
```
B

```
node ->next=ptr;
node ->previous=prev;
prev->next=node;
ptr->next=node;
```
C

```
node ->next=NULL;
node ->previous=prev;
prev->next=node;
ptr->previous=node;
```
D

LESSON 135 : LIST NODE --> DELETING

Lesson Objectives
In this Lesson you will learn : *the Genesis of C Language* *the Stages in the Evolution of C Language*

Database and word processing applications use double linked lists to store data in memory. When the data stored in a node of a double linked list is not required, the node in the double linked list should also be deleted to release the memory occupied by the node.

The process of deleting a node from a double linked list includes two activities, logically and physically deleting a node.

- To delete a double linked list node, physically, release the memory allocated to the node. This can be done using the `free` function.
- To delete a double linked list node logically, the links that attach the node to the double linked list should be removed. This can be done by rearranging the pointers in the nodes that store the address of the node to be deleted.

Deleting a node
Before deleting a node from a double linked list, the double linked list needs to be traversed to find the position of the node in the double linked list. The C instructions to delete the first node, the last node, and a node in the middle of a double linked list are different.
To delete the first node of a double linked list, update the pointer that indicates the beginning of the list with the address of the second node and delete the link between the first and second nodes. This deletes the first node logically.
Next, delete the node physically by releasing the memory occupied by the node.
To delete the last node of a double linked list, update the pointer that indicates eh end of the list with the address of the second last node and remove the link between the last and second last node. This deletes the last node logically.
After deleting the last node logically, release the memory occupied by the last node to delete the last node physically.
However, if the node to be deleted is in the middle of the list, remove the links that attach the node to the previous and succeeding nodes to delete the node logically. Next, release the memory occupied by the node to delete the node physically.
Consider an example of a double linked list that stores the item codes of the products. The nodes of this double linked list are represented by the structure Double List is given below. The variable item Code stores the item code.

LESSON 135 : LIST NODE --> DELETING

Listing 135.1 : deleting nodes from a double linked list

```
//Listing 135.1 : This program deleted nodes from a double linked list.
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
struct DoubleList
{
         char ItemCode[5];
         struct DoubleList *previous;
         struct DoubleList *next;
};

struct DoubleList *start,*last;
struct DoubleList *ptr,*prev;
//Deletes a node from a double linked list
DistDelete( )
{
      char item[5];
      printf("Type the code of the item to be deleted:  ");
      scanf("%s",item);
for(ptr=start;(ptr!=NULL)&&strcmp(item,ptr->ItemCode)!=0;prev=ptr,ptr=tr->next);
      if(ptr!=NULL)
      {
         if(ptr= =start)
         {
           start=ptr->next;
           start->previous=NULL;
           free(ptr);
         }
           else
           {
             if(ptr = =last)
             {
                last=prev;
                prev->next=NULL;
                free(ptr);
             }
             last=prev;
             prev->next=NULL;
             free(ptr);
             else
           {
              prev->next=ptr->next;
              ptr->next->previous=prev;
              free(ptr);
           }
         }
       }
       else
       {
           Printf("The item code does not exist. ");
       }
  }
```

Detailed Explanation

- The pointers `previous` and `next` are used to store the addresses of the preceding and succeeding nodes, respectively.
- To indicate the beginning and end of the double linked list, two pointers named `start` and `last` are declared. The pointer `start` indicates the beginning of the list, and the pointer `last` indicates the end of the list.

- To delete a node in a double linked list that contains specific data, the node needs to be located. To search for the required node, two pointers named `ptr` and `prev` are declared to store the addresses of the node to be deleted and the node preceding it, respectively.
- A user-defined function, `DlistDelete`, is declared to delete a node from the double linked list. First, this function prompts the user to type the item code value of the node to be deleted and assigns the item code value typed by the user to the variable item.
- Next, the node to be deleted needs to be searched for. This can be done using the `for` loop given. To start searching for the required node from the beginning, the pointer `ptr` is set to point to the first node.
- Next, the item code value typed by the user is compared with the item code values in the node pointed to by the pointer `ptr` and the pointer `ptr` is advanced. This is done until a match is found or the last node is reached.
- Before advancing the pointer `ptr`, its value is assigned to the pointer `prev`. The pointer `ptr` to store the address of the node that precedes the node that is to be deleted.
- After quitting the `for` loop, check if the value of the pointer `ptr` is `NULL` if the value of the pointer `ptr` is not `NULL`, it indicates that the node to be deleted is either in the beginning, end, or middle of the double linked list.
- Next, verify if the node to be deleted is the first node. This is done by verifying if the value of the pointer `ptr` is equal to `start`. In this situation, remove the link between the first and second nodes by setting the pointer `previous` of the second node to `NULL`.
- After removing the link between the first and second nodes, update the pointer `start` with the address of the current second node. Next, release the memory occupied by the node pointed to by the pointer `ptr` by using the `free` function.
- You learned to delete the first node from a double linked list. However, if the pointer `ptr` is not equal to `start` after quitting the `for` loop, the node to be deleted is in the middle or end of the double linked list.
- To verify if the node to be deleted is the last node, check if the pointer `ptr` is equal to the pointer `last`. If the pointer `ptr` is equal to last, the node to be deleted is the last node of the list.
- To delete the last node logically, the links that attach the node to the list should be removed. First, assign the address of the second last node to the pointer `last`.
- Next, assign the `NULL` value to the pointer `next` of the node pointed to by the pointer `prev`.
- Finally, delete the node physically by releasing the memory occupied by the last node.
- If the pointer `ptr` is not equal to either `start` or `last` after quitting the `for` loop, the node to be deleted is in the middle of the linked list. The C instructions to delete a node in the middle of the double linked list are given.

- To logically delete the node in the middle, the pointer `next` of the previous node is assigned the address of the succeeding node.
- In addition, the pointer `previous` of the node that succeeds the node pointed to by the pointer `ptr` is assigned the address of the preceding node.
- After deleting the node logically, the memory occupied by the node is released using the `free` function.
- If the value of the pointer `ptr` is `NULL` after quitting the `for` loop, the item code type by the user does not exist in the linked list. In this situation, display the message that the item code to be deleted does not exist.

Self Review Exercise 135.1

1. *To delete the first node in a double linked list, the links between the first an second nodes should be removed. In addition, the pointer:*

   A. start should be assigned the address of the current second node and the memory occupied by the first node should be released.
   B. Start should be assigned the address of the current second node and the memory occupied by the second node should be released.
   C. Next of the first node should be set to NULL and the memory occupied by the pointer start should be released.
   D. Start should be set to NULL and the memory occupied by the first node should be released.

2. *You learned to delete the first node in a double linked list. Partial code used to delete a node in a double linked list is given below. Four code sample are also displayed. Tick the correct code sample used to delete the first node.*

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct Employee
{
       char EmployeeName[5];
       struct Employee *previous;
       struct Employee *next;
};
struct Employee *start, *last, *ptr, *prev;
DlistDelete()
{
char ename[5];
printf("Type the employee name of the node to be deleted: ");
scanf("%s", ename);
for(ptr=start;(ptr!=NULL)&&strcmp(ename,ptr->EmployeeName)!=0;prev=ptr,   ptr=ptr-
>next);
if(ptr!=NULL)
{
if(ptr==start)
{

}
```

```
start=NULL;
start->previous=NULL;
free(ptr);
```
A

```
start->previous=NULL;
free(ptr);
```
B

```
start=ptr->next;
start->previous=NULL;
```
C

```
start=ptr->next;
start->previous=NULL;
free(ptr);
```
D

Self Review Exercise 135.2

3. *You learned to delete a node from double linked list. Partial code used to delete a node from a double linked list is given below. Four code sample are also given. Tick the correct code sample used to delete the first node.*

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct Employee
{
       char EmployeeName[5];
       struct Employee *previous;
       struct Employee *next;
};
struct Employee *start, *last, *ptr, *prev;
DlistDelete()
{
char ename[5];
printf("Type the employee name of the node to be deleted: ");
scanf("%s", ename);
for(ptr=start;(ptr!=NULL)&&strcmp(ename,ptr->EmployeeName)!=0;prev=ptr,   ptr=ptr-
>next);

if(ptr!=NULL)
{
if(ptr==start)
{
start=ptr->next;
start->previous=NULL;
free(ptr);
}
else
{
if(ptr==last)
{

}
```

```
last=prev;
prev->next=NULL;
free(ptr);
                    A
```

```
last=prev;
prev->next=NULL;
free(prev);
                    B
```

```
prev->next=NULL;
free(ptr);

                    C
```

```
last=ptr->next;
free(ptr);

                    D
```

Self Review Exercise 135.3

4. *Partial code used to delete a node in a double linked list is given below. Four code samples are also displayed. Tick the correct code sample used to delete a node in the middle of the list.*

```
};
struct Employee *start, *last, *ptr, *prev;
DlistDelete()
{
char ename[5];
printf("Type the employee name of the node to be deleted: ");
scanf("%s", ename);
for(ptr=start;(ptr!=NULL)&&strcmp(ename,ptr->EmployeeName)!=0;prev=ptr,   ptr=ptr-
>next);

if(ptr!=NULL)
{
if(ptr==start)
{
start=ptr->next;
start->previous=NULL;
free(ptr);
}
else
{
if(ptr==last)
{
last=prev;
prev->next=NULL;
free(ptr);
}
else
{

}
}
}
else
{
printf("The employee name does not exist. ");
}
}
```

```
prev->next=ptr->next;
ptr->next->previous=prev;
free(prev);
```
A

```
prev->next=ptr->next;
ptr->next->previous=prev;
free(ptr);
```
B

```
prev->next=NULL;
ptr->next->previous=prev;
free(ptr);
```
C

```
prev->next=ptr->next;
ptr->next->previous=NULL;
free(ptr);
```
D

1. Partial codes used to delete a node in the double linked list that contains the names of customers is given here. Four code samples are also given. Choose the correct code used to delete the first node of the list.

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
struct Customer
{
       char CustomerName[25];
       struct Customer *previous;
       struct Customer *next;
};

struct Customer *start, *last, *ptr, *prev;
DlistDelete()
{
 char name[25];
 printf("Type the customer name of the node to be deleted: ");
 scanf("%s", name);
 for(ptr=start; (ptr!=NULL)&&strcmp(name,ptr->CustomerName)!=0; prev=ptr, ptr=ptr->next);
       if(ptr!=NULL)
       {
                if(ptr==start)
                {
                //
                }
```

A. `start=NULL;` `start->previous=NULL;` `free(ptr);`	*B.* `start=ptr->next;` `start->previous=NULL;`
C. `start=ptr->next;` `start->previous=NULL;`	*D.* `start=ptr->next;` `free(ptr);`

2. Partial code to delete a node in the double linked list that contains the names of customers is given. Four sample codes are also given. Choose the correct code used to delete that last node of the list.

```
       if(ptr==start)
                {
                        start=ptr->next;
                        start->previous=NULL;
                        free(ptr);
                }
                else
                {
                        if(ptr==last)
                        {
                        //
                        }
```

A. `last=ptr->next;` `prev->next=NULL;` `free(ptr);`	*B.* `last=prev;` `prev->next=NULL;` `free(ptr);`
C. `last=prev;` `prev->next=NULL;`	*D.* `last=prev;` `free(ptr);`

REVIEW EXERCISE

3. Partial code to delete a node in the double linked list that contains the names of customers is given. Four code samples are also given. Choose the correct code used to delete a node in the middle of the list.

```
if(ptr==start)
                {
                        start=ptr->next;
                        start->previous=NULL;
                        free(ptr);
                }
                else
                {
                        if(ptr==last)
                        {
                                last=prev;
                                prev->next=NULL;
                                free(ptr);
                        }
                        else
                        {
                        //
                        }
                }
        }
        else
        {
                printf("The record does not exist.");
        }
}
```

A.
```
prev->next=ptr->next;
ptr->next->previous=prev;
free(ptr);
```

B.
```
prev->next=ptr->previous;
ptr->next->previous=NULL;
free(ptr);
```

C.
```
prev->next=ptr;
ptr->next->previous=prev;
```

D.
```
prev->next=ptr->next;
ptr->next->previous=prev;
```

4. Identify the advantages of using linked lists to store data in memory.

A. Memory need not be allocated to nodes when creating a linked list.
B. A linked list need not be restructured for inserting a deleting nodes.
C. New nodes can be easily inserted by moving the existing nodes.
D. The nodes in a linked list occupy less memory that an array.

5. Partial code used to insert nodes in a double linked list that is sorted in ascending order is given here. Four pieces of code are also given. Choose the correct code used to insert a node at the end of the list.

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
struct Customer
{
        char CustomerName[25];
        struct Customer *previous;
        struct Customer *next;
};
struct Customer *start, *last, *ptr, *prev, *newnode;
GetNode()
{
        newnode=(struct Customer *)malloc(sizeof(struct Customer));
        printf("Type the customer name: ");
        scanf("%s", newnode->CustomerName);
        newnode->next=NULL;
        newnode->previous=NULL;
}
dlistinsert()
{
        GetNode();
        for(ptr=start; (ptr!=NULL)&&strcmp(newnode->CustomerName,ptr->CustomerName)>0;
            prev=ptr, ptr=ptr->next);
        if(ptr==NULL)
        {
        //
        }
```

A.
```
newnode->previous=NULL;
prev->next=newnode;
last=newnode;
```

B.
```
newnode->previous=prev;
prev->next=NULL;
last=newnode;
```

C.
```
newnode->previous=prev;
prev->next=newnode;
last=newnode;
```

D.
```
newnode->previous=prev;
prev->next=newnode;
```

6. Partial code used to insert nodes in a double linked list that is sorted in ascending order is given below. Four pieces of code are also given. Choose the correct code required to insert a node in the beginning of the list.

```
dlistinsert()
{
        GetNode();
        for(ptr=start; (ptr!=NULL)&&strcmp(newnode->CustomerName,ptr->CustomerName)>0;
            prev=ptr, ptr=ptr->next);
        if(ptr==NULL)
        {
                newnode->previous=prev;
                prev->next=newnode;
                last=newnode;
        }
        else
        {
                if(ptr==start)
                {
                //
```

A.
```
newnode->next=start;
start-> previous=newnode;
```

B.
```
newnode->next=start;
start-> previous=newnode;
start=newnode;
```

C.
```
newnode->next=start;
start-> previous=NULL;
```

D.
```
newnode->next=NULL;
start-> previous=newnode;
```

REVIEW EXERCISE

7. Partial code used to insert nodes in a double linked list that is sorted in ascending order is given below. Four pieces of code are also given. Choose the correct code used to insert a node in the middle of the list.

```
dlistinsert()
{
        GetNode();
        for(ptr=start; (ptr!=NULL)&&strcmp(newnode->CustomerName,ptr->CustomerName)>0;
            prev=ptr, ptr=ptr->next);
        if(ptr==NULL)
        {
                 newnode->previous=prev;
                 prev->next=newnode;
                 last=newnode;
        }
        else
        {
                 if(ptr==start)
                 {
                 newnode->next=start;
                 start-> previous=newnode;
                 start=newnode;
                 }
                 else
                 {
                 //
                 }
        }
```

A.
```
newnode->next=ptr;
newnode->previous=NULL;
prev->previous=newnode;
```

B.
```
newnode->next=ptr;
newnode->previous=prev;
ptr->previous=newnode;
```

C.
```
newnode->next=ptr;
newnode->previous=prev;
prev->next=newnode;
```

D.
```
newnode->next=ptr;
newnode->previous=prev;
prev->next=newnode;
ptr->previous=newnode;
```

8. Identify the code to crate the first node of the single linked list in which nodes are represented by the structure named `Cust`*. This structure contains data and a pointer to the succeeding node. Choose the appropriate code.*

A.
```
struct Cust *start, *nd;
GetNode(){
   nd=(struct Cust *)malloc(sizeof(struct Cust));
   printf("Type the Customer name: ");
   scanf("%s".nd->CustomerName);
   nd->next=NULL;
}
  CreateList() {
    if(start==NULL) {
       GetNode();
       start=nd;
    }
}
```

B.
```
struct Cust *start, *nd;
GetNode(){
   nd=(struct Cust *)malloc(sizeof(struct Cust));
   printf("Type the Customer name: ");
   scanf("%s".nd->CustomerName);
   nd->next=NULL;
}
  CreateList() {
    if(start==NULL) {
       GetNode();
    }
}
```

C.
```
struct Cust *start, *nd;
GetNode(){
   printf("Type the Customer name: ");
   scanf("%s".nd->CustomerName);
   nd->next=NULL;
}
  CreateList() {
    if(start==NULL) {
       GetNode();
       start=nd;
    }
}
```

D.
```
struct Cust *start, *nd;
GetNode(){
nd=(struct Cust *)malloc(sizeof(struct Cust));
printf("Type the Customer name: ");
scanf("%s".nd->CustomerName);
nd->next=NULL;
}
  CreateList() {
    if(start==NULL) {
       GetNode();
       start=nd->next;
    }
}
```

REVIEW EXERCISE

9. Choose the appropriate code sample that can be used to insert the invoice number in the queue.

A.
```
insert()
{
   ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nplease input the employee name:");
   scanf("%s".ptr->EName);
   if(rear==NULL)
   {
     rear->next=ptr;
   }
   rear=ptr;
```

B.
```
insert()
{
   ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nplease input the employee name:");
   scanf("%s".ptr->EName);
   if(rear!=NULL)
     rear=ptr;
}
```

C.
```
insert()
{
   ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nplease input the employee name:");
   scanf("%s".ptr->EName);
   if(rear!=NULL)
   {
     rear->next=ptr;
   }
   rear=ptr;
```

D.
```
insert()
{
   ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nplease input the employee name:");
   scanf("%s".ptr->EName);
   if(rear!=NULL)
   {
     rear->next=ptr;
   }
   free(ptr);

}
```

10. Partial code used to delete a node from a single linked list in which nodes are represented by the structure Customer is given. Four pieces of code are also given. Choose the correct code used to delete the first node.

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
struct Customer
{
        char CustomerName[25];
        struct Customer *next;
};

struct Customer *start, *ptr, *prev;

DeleteList()
{
        char name[25];
        printf("Type the customer name of the node to be deleted");
        scanf("%s", name);
}

if(ptr==NULL)
        {
        if(ptr==start)

        {
        //
        }
```

A.
```
start=ptr;
free(ptr);
```

B.
```
start=ptr->next;
```

C.
```
start=ptr->next;
free(start);
```

D.
```
start=ptr->next;
free(ptr);
```

11. Partial code used to delete a node from a single linked list that stores the names of customers is given here. Four pieces of code are also given. Choose the correct code used to delete a node in the middle or at the end of the single linked list.

```
if(ptr==NULL)
        {
        if(ptr==start)
                {
                start=ptr->next;
                free(ptr);
                }
                else
                {
                //
                }
        }
        else
        {
                printf("\nThe node does not exist.");
        }
}
```

A.
```
prev->next=ptr->next;
```

B.
```
prev->next=ptr->next;
free(prev);
```

C.
```
prev->next=ptr->next;
free(ptr);
```

D.
```
prev->next=ptr;
free(ptr);
```

REVIEW EXERCISE

12. Four code samples are given here. Identify the code used to insert nodes in the beginning of a single linked list in which nodes are represented by the structure `Cust`*. Choose the correct code sample.*

A.
```
struct Cust *start, *nd;
GetNode()
{
nd=(struct Cust *)malloc(sizeof(struct Cust));
printf("\nType the name of the customer: ");
scanf("%s".nd->CustomerName);
nd->next==NULL;
}
insertUnsortedList()
{
    nd->next=start;
    start=nd;
}
```

B.
```
struct Cust *start, *nd;
GetNode()
{
nd=(struct Cust *)malloc(sizeof(struct Cust));
   printf("\nType the name of the customer: ");
   scanf("%s".nd->CustomerName);
   nd->next==NULL;
}
insertUnsortedList()
{
    GetNode();
    nd->next=start;
}
```

C.
```
struct Cust *start, *nd;
GetNode()
{
   nd=(struct Cust *)malloc(sizeof(struct Cust));
   printf("\nType the name of the customer: ");
   scanf("%s".nd->CustomerName);
   nd->next==NULL;
}
insertUnsortedList()
{
    GetNode();
    nd->next=start;
    start=nd;
}
```

D.
```
struct Cust *start;
GetNode()
{
   nd=(struct Cust *)malloc(sizeof(struct Cust));
   printf("\nType the name of the customer: ");
   scanf("%s".nd->CustomerName);
   nd->next==NULL;
}
insertUnsortedList()
{
    GetNode();
    nd->next=start;
    start=nd;
}
```

13. The nodes of an unsorted single linked list are represented by the structure Product List Partial code used to search for anode in this single linked list is given below. Choose appropriate lines of code required to complete the partial code.

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
struct ProductList
{
       char ProductID[5];
       struct ProductList *next;
};

struct ProductList *start;
struct ProductList *ptr;
UnsortedSearch()
{
       char prod[5];
       printf("Type the product ID:");
       scanf("%s",prod);
       //
       if(ptr==NULL)
       {
               printf("\nEnd of list");
       }
       else
       {
               printf("\nProduct ID: %s",ptr->ProdID);
       }
}
```

A. `for(ptr=NULL; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)!=0; ptr=ptr->next);`

B. `for(ptr=start; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)==0; ptr=ptr->next);`

C. `for(ptr=start; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)<0; ptr=ptr->next);`

D. `for(ptr=start; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)!=0; ptr=ptr->next);`

REVIEW EXERCISE

14. The nodes of a single linked list sorted in ascending order are represented by the structure Product List Partial code used to search for a node in this single linked list is displayed on the screen. Tick the lines of code required to complete the partial code.

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
struct ProductList
{
        char PI[5];
        struct ProductList *next;
};

struct ProductList *start;
struct ProductList *ptr;
SortedSearch()
{
        char prod[5];
        printf("Type the product ID:");
        scanf("%s",prod);

        if(ptr==NULL)
        {
                printf("\nEnd of list");
        }
        else
        {
                if(strcmp(prod,ptr->ProdID)==0)
                {
                        printf("\nProduct ID: %s",ptr->ProdID);
                }
                else
                {
                        printf("The Product ID not found");
                }
        }
}
```

A. `for(ptr=start; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)==0; ptr=ptr->next);`

B. `for(ptr=start; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)>0; ptr=ptr->next);`

C. `for(ptr=start; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)<0; ptr=ptr->next);`

D. `for(ptr=NULL; (ptr!=NULL)&&strcmp(prod,ptr->ProdID)>0; ptr=ptr->next);`

15. Partial code used to add a new node in a single linked list sorted in ascending order is displayed here. Identify the correct code required to find the stored position of the new node. Tick the correct code.

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
struct Emp
{
       char Ed[5];
       struct Emp *next;
};

struct Emp *start, *pt, *nd, *prev;

GetNode()
{
       nd=(struct Emp *)malloc(sizeof(struct Emp));
       printf("Type the employee ID: ");
       scanf("%s", nd->Ed);
       //
       newnode->next=NULL;
       nd->next=NULL;
}
Sortedinsertlist()
{
       GetNode();
}
```

```
    for(pt=start; (pt!=NULL)&&strcmp(nd->Ed,pt->Ed)>0; prev=pt,pt=pt->next);

A.
B.  for(pt=NULL; (pt!=NULL)&&strcmp(nd->Ed,pt->Ed)>0; prev=pt,pt=pt->next);

C.  for(pt=start; (pt!=NULL)&&strcmp(nd->Ed,pt->next)>0; prev=pt,pt=pt->next);

D.  for(pt=start; (pt!=NULL)&&strcmp(nd->Ed,pt->Ed)>0; pt=pt->next);
```

16. Partial code used to add a new node in a single linked list sorted in ascending order is given below. Identify the correct lines of code required to add the new node at the sorted position. Choose the correct code.

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
struct Emp
{
       char Ed[5];
       struct Emp *next;
};

struct Emp *start, *pt, *nd, *prev;

GetNode()
{
       nd=(struct Emp *)malloc(sizeof(struct Emp));
       printf("Type the employee ID: ");
       scanf("%s", nd->Ed);
       newnode->next=NULL;
       nd->next=NULL;
}
Sortedinsertlist()
{
       GetNode();
       for(pt=start; (pt!=NULL)&&strcmp(nd->Ed,pt->Ed)>0; prev=pt,pt=pt->next);
}
```

A.
```
if(pt==start)
{
   nd->next=start;
}
else
{
   prev->next=nd;
   nd->next=pt;
```

B.
```
if(pt==start)
{
   nd->next=start;
   start=nd;
}
else
{
   prev->next=nd;
```

C.
```
if(pt==start)
{
   nd->next=start;
   start=nd;
}
else
{
   nd->next=pt;
}
```

D.
```
if(pt==start)
{
   nd->next=start;
   start=nd;
}
else
{
   prev->next=nd;
   nd->next=pt;
}
```

17. Choose the code sample that is used to insert an employee ID in a stack.

A.
```
push()
{
   ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nPlease input the EmpID:");
   scanf("%s".ptr->EmpID);
   ptr=top;
   top=top->next;
   free(ptr);
```

B.
```
push()
{
   ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nPlease input the EmpID:");
   scanf("%s".ptr->EmpID);
   if(last!=NULL)
      ptr->next=last;
   else
      ptr=last;
}
```

C.
```
push()
{
   ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nPlease input the EmpID:");
   scanf("%s".ptr->EmpID);
   top=ptr;
   if(ptr!=NULL)
      top=ptr;
}
```

D.
```
push()
{
ptr=(struct tag *)malloc(sizeof(struct tag));
   printf("\nPlease input the EmpID:");
   scanf("%s".ptr->EmpID);
   if(ptr!=NULL)
      top=ptr;
   else
      {
       ptr->next=top;
       top=ptr;
      }
```

18. A double linked list is represented by the structure Pd*. The pointers start, last, and node are declared to point to the first, last, and new nodes, respectively. Tick the correct code to create the first node of the double linked list.*

A.
```
GetNode(){
   nd=(struct Pd *)malloc(sizeof(struct Pd));
   printf("Type product code: ");
   nd->next=NULL;
   nd->previous=NULL;
}
DlistCreate() {
    if(start==NULL) {
       GetNode();
       start=nd;
       last=nd;
    }
```

B.
```
GetNode(){
   nd=(struct Pd *)malloc(sizeof(struct Pd));
   printf("Type product code: ");
   scanf("%s",nd->ProductCode);
   nd->next=NULL;
   nd->previous=NULL;
}
DlistCreate() {
    if(start==NULL) {
       GetNode();
    }
}
```

C.
```
GetNode(){
   nd=(struct Pd *)malloc(sizeof(struct Pd));
   printf("Type product code: ");
   scanf("%s",nd->ProductCode);
   nd->next=NULL;
   nd->previous=NULL;
}
DlistCreate() {
    if(start==NULL) {
       GetNode();
       start=nd;
       last=nd;
    }
}
```

D.
```
GetNode(){
   nd=(struct Pd *)malloc(sizeof(struct Pd));
   printf("Type product code: ");
   scanf("%s",nd->ProductCode);
   nd->next=NULL;
   nd->previous=NULL;
}
DlistCreate() {
    if(start==NULL) {
       start=nd;
       last=nd;
    }
}
```

19. *Identify the code to delete an item number from a queue. A partial code is displayed here. Tick the appropriate code sample.*

A.
```
delete()
{
   printf("\nProduct deleted: %s",front->PNo);
   if(rear!=NULL)
   rear=ptr;
   free(ptr);
```

B.
```
delete()
{
   printf("\nProduct deleted: %s",front->PNo);
   front=front->next;
   free(ptr);
```

C.
```
delete()
{
   printf("\nProduct deleted: %s",front->PNo);
   ptr=front;
   front=front->next;
```

D.
```
delete()
{
   printf("\nProduct deleted: %s",front->PNo);
   ptr=front;
   front=front->next;
   free(ptr);
}
```

20. *Sequence the steps to evaluate an arithmetic expression that is represented as a postfix notation.*

A. *If an operator is found, then remove the top elements of the stack.*
B. *Add a parenthesis at the end of the postfix expression.*
C. *Evaluate the expression and place the result into the stack.*
D. *Assign a postfix expression as a string.*
E. *Scan the postfix expression left to right until a parenthesis is found.*

21. The nodes of a double linked list are represented by the structure `Product` *given below. Four pieces of C code are also given. Choose the code used to traverse the double linked list in the forward direction.*

```
#include<stdlib.h>
#include<stdio.h>
struct Product
{
        char ProdCode[5];
        struct Product *previous;
        struct Product *next;
};

struct Product *start, *last;
```

A.
```
struct Product *ptr;
TraverseForward()
{
   ptr=start;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->next;
   }
}
```

B.
```
struct Product *ptr;
TraverseForward()
{
   ptr=NULL;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->next;
   }
```

C.
```
TraverseForward()
{
   ptr=start;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->next;
   }
}
```

D.
```
struct Product *ptr;
TraverseForward()
{
   ptr=start;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->previous;
   }
}
```

REVIEW EXERCISE

22. The nodes of a double linked list are represented by the structure `Product` *displayed here. Four pieces of C code are also displayed. Tick the code used to traverse the double linked list in the backward action.*

```
#include<stdlib.h>
#include<stdio.h>
struct Product
{
       char ProdCode[5];
       struct Product *previous;
       struct Product *next;
};

struct Product *start, *last;
```

A.
```
struct Product *ptr;
TraverseReverse()
{
   ptr=last;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->next;
   }
```

B.
```
struct Product *ptr;
TraverseReverse()
{
   ptr=NULL;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->previous;
   }
```

C.
```
TraverseReverse()
{
   ptr=last;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->previous;
   }
}
```

D.
```
struct Product *ptr;
TraverseReverse()
{
   ptr=last;
   while(ptr!=NULL)
   {
      printf("Product Code: %s",ptr->ProdCode);
      ptr=ptr->previous;
   }
}
```

REVIEW EXERCISE

23. Select the appropriate code to traverse a single linked list. Nodes of the single linked list are represented by the structure `Customer` *displayed here. Tick the partial code that enables you to traverse a single linked list.*

A.
```
char *ptr;
traverse()
{
   ptr=start;
   while(ptr!=NULL)
   {
      printf("CustomerID: %s",ptr->CustomerID);
      ptr=ptr->next;
   }
}
```

B.
```
struct Customer *start,*ptr;
traverse()
{
   ptr=start;
   while(ptr!=NULL)
   {
      printf("CustomerID: %s",ptr->CustomerID);
      ptr=ptr->next;
   }
```

C.
```
struct Customer *start,*ptr;
traverse()
{
   ptr=NULL;
   while(ptr!=NULL)
   {
      printf("CustomerID: %s",ptr->CustomerID);
      ptr=ptr->next;
   }
```

D.
```
struct Customer *start,*ptr;
traverse()
{
   ptr=start;
   while(start)
   {
      printf("CustomerID: %s",ptr->CustomerID);
      ptr=ptr->next;
   }
}
```

LESSON 136 : DEFINITION

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Binary trees are data structures that represent elements containing data in a hierarchical relationship. In a hierarchical relation, data elements do not follow a linear sequence as followed by the data elements stored in a linked list.

Although the representation of a binary tree is similar to that of a double linked list, the way in which data is organized in the two data structures is not the same. The way in which data is arranged in a binary tree leads to more efficient searches as compared to other data structures, such as linked lists and arrays.

The hierarchical structure of a binary tree contains the left and right subtrees. The left subtree can contain data that is lower in value than the data in the parent node. A parent node is a node that contains a child node. The right subtree contains values larger than the parent node. Each element in the binary tree is called a node. Each node stores information and pointers pointing to child nodes. The first node in a binary tree is called the root node. The entire tree used a node is called a subtree. A node may or may not contain a child node. A node that does not contain a child node is called a leaf node.

The binary tree is a special from of linked list. Each node contains pointers that store the addresses of the left and right nodes.

If a node does not contain a child node, its pointers to the left and right nodes contain `NULL` values. A structure declaration of a binary tree is given. In this code, two pointers, `left` and `right`, are defined.

Listing 136.1 : declaring a binary tree structure

```
//Listing 136.1 : This program declares a binary tree structure.

#include<stdio.h>
#include<stdlib.h>
#define SIZE_OF_ITENND 5
#define SIZE_OF_ITENNDESC  25
struct TreeTag
{
    char ItemNo[SIZE_OF_ITEMNO];
    char ItemDesc[SIZE_OF_ITENDSC];
    struct TreeTag *left;
    struct TreeTag *right;
};
struct TreeTag *root;
```

Detailed Explanation

- The two pointers in the binary tree structure are defined so that the addresses of the `left` and `right` child nodes can be stored in the parent node.

Self Review Exercise 136.1

1. *The leaf node is the:*

   A. node that has no child nodes.
   B. First node in a binary tree.
   C. Node that contains the left and right pointers that stores the addresses of the child nodes
   D. Node that contains a subtree.

2. *You learned about the code to declare a binary tree structures. Tick the code sample that declares a binary tree structure now.*

```
struct TreeTag
{
char ItemNo[SIZE_OF_ITEMNO];
struct TreeTag 8left;
};
Struct TreeTag *root;
```
A

```
struct TreeTag
{
char ItemNo[SIZE_OF_ITEMNO];
struct TreeTag 8left;
Struct TreeTag *right;
};
Struct TreeTag *root;
```
B

```
struct TreeTag
{
char ItemNo[SIZE_OF_ITEMNO];
Struct TreeTag *right;
};
Struct TreeTag *root;
```
C

```
struct TreeTag
{
char ItemNo[SIZE_OF_ITEMNO];
};
Struct TreeTag *root;
```
D

LESSON 137 : INSERTING A NODE

Lesson Objectives

In this Lesson you will learn :

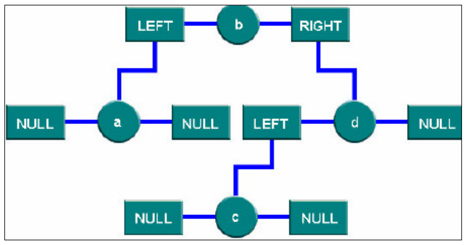
- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

You are writing an employee managed package in the C language by using binary trees. Every time a new employee is hired, the information regarding the new employee has to be inserted in the binary tree.

The process of inserting nodes in a binary tree involves the comparison of the new information with the existing information. Comparing the new information with the existing information ensures that information is sorted during insertion.

Insertion also involves the allocation of memory for the new node that contains the information. It also involves a change in the left pointer or the right pointer of the node under which the new node has to be inserted.

Consider the sample diagram of a binary tree displayed over here. A new item called c has to be inserted in the binary tree. To insert the new item, a pointer is made to point to the root node. The item in the new node is compared with the root node.



If the item does not match the existing item pointed to by the pointer, the pointer to the new item is advanced. The direction in which the pointer to the new item is advanced is based on the value of the new item. If the new item is grater than the existing item, the pointer should be advanced to the right subtree.

If the new item is less than the existing item, the pointer should be advanced to the left subtree.

The value c is first compared with the root node b. The pointer to the root node is advanced to the right node because the value c is greater than the value b.

Next, the value c is compared wit the value d. The insertion of the value c will take place to the left of the value d because the value c is less than value d.

The pointer to the root node is advanced to the lift of the node d. If the node that is to the left of the node d contains the value NULL, the value c is inserted in this node. The value NULL implies that the node is free for insertion

LESSON 137 : INSERTING A NODE

Consider a code sample of a binary tree structure given below. The structure contains the variable `itemNo` and the two pointers `left` and `right`.

Listing 137.1 : creates a binary tree structure and insert nodes

```
//Listing 137.1 : This  program creates a binary tree structure and inserts nodes in the
//structure.

#include<stdio.h>
#include<stdlib.h>
#define SIZE_OF_ITEMNO 5

struct TreeTag
{
    char ItemNo[SIZE_OF_ITEMNO];
    struct TreeTag *left;
    struct TreeTag *right;
};

struct TreeTag *root, *ptr, *newnode;

void getnode()
{
   nawnode=(struct TreeTag*)malloc(sizeof(struct TreeTag));
   printf("\n Type the item number: ");
   scanf("%s",newnode->ItemNo);
   newnode->right=NULL;
   newnode->right=NULL;
}

main()
{
   root = NULL;
   getnode()
   {
      if(root== NULL)
         root =newnode;
      else
         createInsert(root);
   }
}
```

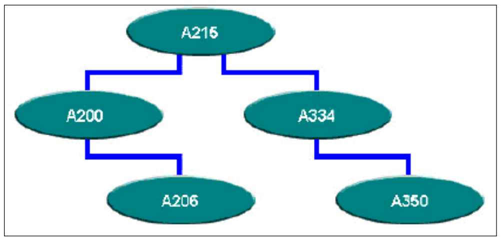
Detailed Explanation

- The variable `itemNo` stores the item codes of the products. The `left` and `right` pointers are used to store the address of the left and right nodes, respectively
- There are three other pointers created in this program. The root pointer is declared to indicate the root node in the binary tree. The ptr pointer is used to traverse the binary tree to find the location where the new node can be inserted.
- A pointer to store the address of the new node should be declared before creating the new node. The pointer new node is declared to point to the node to be inserted in the binary tree.
- To insert a new node in the binary tree, a user-defined function, `getnode`, is declared. This function allocates the memory for the new node by using the `malloc` function.
- Next, the `getnode` function prompts the user to type the item number. The item number will assigned to the new node.
- Finally, the `left` and `right` pointers in the node are assigned the `NULL` value.

- The next step in the process to insert a new item in the binary tree is to compare the new item to be inserted with the existing item in the node pointed to by the pointer.
- If the new item is greater than the item pointed to by `ptr`, the item will be inserted in the right subtree.
- However, if the new item is less than the item pointed to by `ptr`, this new item will be inserted in the left subtree.

Self Review Exercise 137.1

1. *You learned about the concept t insert a node in a binary tree. A diagram representing a binary tree is given below. Identify the path the pointer will select to insert the item A210 in the binary tree.*



A. `root (A215)->node with INFO A334-> node with INFO A350->left node.`
B. `root (A215)->node with INFO A200-> node with INFO A206->left node.`
C. `root (A215)->node with INFO A334-> node with INFO A350->right node.`
D. `root (A215)->node with INFO A200-> node with INFO A206->right node.`

2. *You learned about the code to insert an item in a binary tree. Tick the code sample that can be used to insert an item in a binary tree.*

```
If(strcmp(newnode->ItemNo, ptr->ItemNO)>0)
{
if(ptr->right!=NULL)
CIns(ptr->right);
else
ptr->right=newnode;
}
else
if(strcmp(newnode->ItemNo.ptr->ItemNo)<0)
{
if(ptr->left!=NULL)
CIns(ptr->left);
else
ptr->left=newnode;
}
```
A

```
If(strcmp(newnode->ItemNo, ptr->ItemNO)>0)
{
if(ptr->left!=NULL)
CIns(ptr->left);
else
ptr->left=newnode;
}
else
if(strcmp(newnode->ItemNo.ptr->ItemNo)<0)
{
if(ptr->right!=NULL)
CIns(ptr->right);
else
ptr->right=newnode;
}
```
B

```
If(strcmp(newnode->ItemNo, ptr->ItemNO)>0)
{
if(ptr->left!=NULL)
CIns(ptr->left);
else
ptr->right=newnode;
}
else
if(strcmp(newnode->ItemNo.ptr->ItemNo)<0)
{
if(ptr->right!=NULL)
CIns(ptr->right);
else
ptr->left=newnode;
}
```
C

```
If(strcmp(newnode->ItemNo, ptr->ItemNO)>0)
{
if(ptr->right!=NULL)
CIns(ptr->left);
else
ptr->left=newnode;
}
else
if(strcmp(newnode->ItemNo.ptr->ItemNo)<0)
{
if(ptr->left!=NULL)
CIns(ptr->right);
else
ptr->right=newnode;
}
```
D

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A binary tree might have to be traversed to process or print the information stored in nodes. There are three methods to traverse a binary tree:

- inorder,
- preorder, and
- postorder.

The Preorder Method

In the preorder method, traversing starts from the root node and then moves on to the left subtree. The right subtree is traversed in the end.

A code sample representing the preorder method of traversing is given here. In this code, the order in which the items will be traversed is root node, left subtree, and then the right subtree.

Listing 138.1 : using preorder method of traversing binary trees

```
// Listing 138.1 : This program uses the preorder method to traverse the binary tree

#include<stdio.h>
#include<stdlib.h>

struct TreeTag
{
    char ItemNo[SIZE_OF_ITEMNO];
    struct TreeTag *left;
    struct TreeTag *right;
};

struct TreeTag *root, *ptr, *newnode;

preorder (struct TreeTag *ptr)
{
   if(ptr=NULL)
     return;
   else
   {
     printf("\n Item number %s\n", ptr->ItemNo);
     preorder(ptr->left);
     preorder(ptr->right);
   }
}
```

The Postorder Method

Another method of traversing is the postorder method. In this method, traversing starts from the left subtree and then moves on to the right subtree. The root node is traversed in the end. In the diagram, the order of output would be `a, e, d, h, i`, and `f`.

The code sample given here displays the postorder method of traversing. In this code, the items will be traversed from the left subtree and then the right subtree. Finally, the items in the root node will be traversed.

Listing 138.2 : using postorder method of traversing binary trees

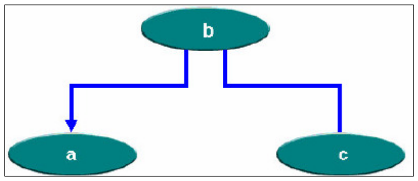
```
// Listing 138.2 : This program uses the postorder method to traverse the binary tree

#include<stdio.h>
#include<stdlib.h>

struct TreeTag
{
    char ItemNo;
    struct TreeTag *left;
    struct TreeTag *right;
};

struct TreeTag *root, *ptr;

postorder (struct TreeTag *ptr)
{
   if(ptr=NULL)
     return;
   else
   {
     postorder(ptr->left);
     postorder(ptr->right);
     printf("\n Item number %s\n", ptr->ItemNo);
   }
}
```

The Inorder Method

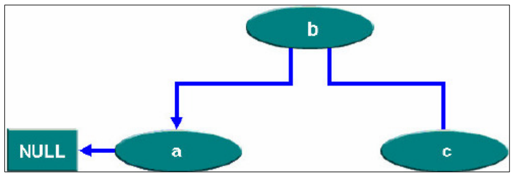
The inorder method is the most common method of traversing a binary tree. The inorder method involves three steps.

The first step for traversing a binary tree by using the inorder method is to traverse the left subtree of the root node.

A diagram representing the inorder method of traversal is given below. If the inorder method is followed for printing the contents of nodes, the output for the example will be a, b, and c.

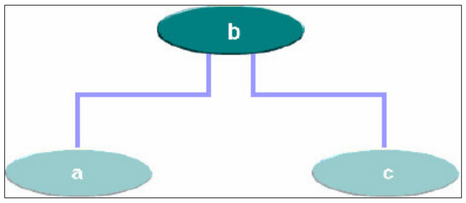


In the example given here, traversal starts at the root node called b. In the first step, inorder traversal begins at the left subtree from root node b. The pointer is advanced in the left subtree until a NULL value is encountered. In the example given here, the left subtree contains value a. traversing further in the left subtree will display a NULL value.

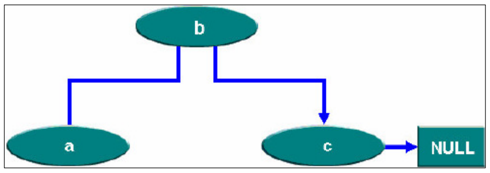


If a NULL pointer is encountered while traversing the left subtree, the traversal in the left subtree is terminated. After completely traversing the left subtree, value a is displayed.

After traversing the left subtree, the root node is traversed. In this example, after printing value a from the left subtree, value b of the root node is printed.



Next, the right subtree is traversed. While traversing the right subtree, the pointer assumes the node in the right subtree to be the new root node. In this example, the new root node is c. This node does not contain any left or right subtrees. Therefore, value c is given.



Consider the example code given here. The program declares a binary tree structure. The left and right pointers are declared to traverse the binary tree in the left and right directions of the binary tree structure.

Listing 138.3 : using inorder method of traversing binary trees

```
// Listing 138.3 : This program uses the inorder method to traverse a binary tree
// structure

#include<stdio.h>
#include<stdlib.h>
struct TreeTag
{
  char ItemNo;
  struct TreeTag *left;
  struct TreeTag *right;
};
struct TreeTag *root, *ptr;

Inorder (struct TreeTag *ptr)
{
   if(ptr=NULL)
     return;
   else
   {
     Inorder(ptr->left);
     printf("\n Item number %s\n", ptr->ItemNo);
     Inorder(ptr->right);
   }
}
```

Detailed Explanation

- The statement in line `14` is the function prototype of the `inorder` function.
- Next, a pointer variable is created to store the storage location of the nodes that are traversed. The initial values assigned to `ptr` is the root node address.
- The statement invoking the `inorder` function with the pointer referencing to the root node of the binary tree is described.
- Next, the function that contains the C instructions to traverse a binary tree is created. The `inorder` function with a parameter called `ptr` is defined.
- It is necessary to pass a parameter to the `inorder` function because the successive subtrees in the binary tree have to be traversed. Passing pointer to the function provides information on each subtree.
- The function to traverse the binary tree checks whether the root node of the subtree contains a `NULL` value. If the root node of the subtree does not contain a `NULL` value, the function traverses the left subtree recursively.
- After traversing the left subtree, the pointer prints the contents of the root node.
- After printing the contents of the root node, the pointer traverses the right subtree recursively.

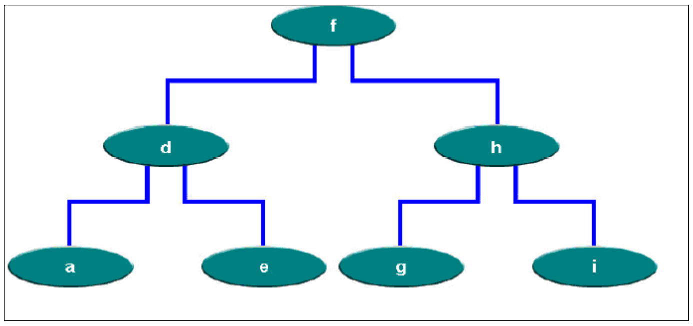
Self Review Exercise 139.1

1. *The first step to traverse a binary tree by using the inorder method of traversal is to traverse the :*

A. Root node
B. Right subtree
C. Left subtree
D. Leaf node of the right subtree

Self Review Exercise 138.2

2. *The first step to traverse a binary tree by using the inorder method of traversal is to traverse the :*



A. `fdaehgi`
B. `adefghi`
C. `hgifdae`
D. `ihgfdae`

3. *You learned about the inorder method for traversing a binary tree. Choose the code sample that will be used to traverse a binary tree by using the inorder method.*

```
inorder(struct TreeTag *ptr)
{
  if(ptr==NULL)
    return;
  else
    inorder(ptr->right);
    inorder(ptr->left);
    printf("\nItem number %s\n",
                   ptr->ItemNo);
}
```
A

```
inorder(struct TreeTag *ptr)
{
  if(ptr==NULL)
    return;
  else
    printf("\nItem number %s\n",
                   ptr->ItemNo);
    inorder(ptr->right);
    inorder(ptr->left);
}
```
B

```
inorder(struct TreeTag *ptr)
{
  if(ptr==NULL)
    return;
  else
    inorder(ptr->left);
    inorder(ptr->right);
    printf("\nItem number %s\n",
                   ptr->ItemNo);
}
```
C

```
inorder(struct TreeTag *ptr)
{
  if(ptr==NULL)
    return;
  else
    inorder(ptr->left);
    printf("\nItem number %s\n",
                   ptr->ItemNo);
    inorder(ptr->right);
}
```
D

LESSON 139 : SEARCHING

Lesson Objectives

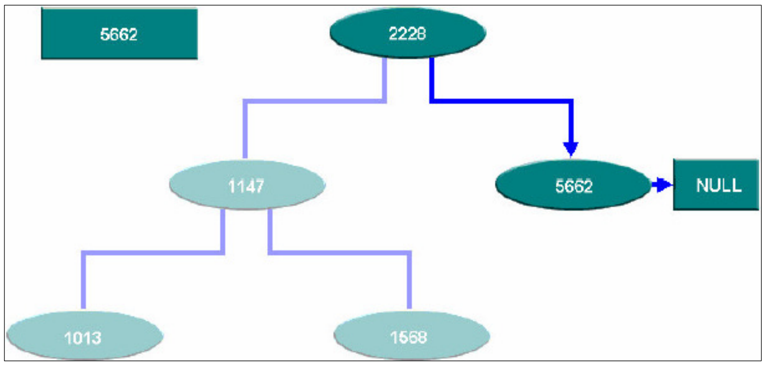
In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The Human Resources division of your organization needs to retrieve the employees management package periodically to search for details on an employee. You can write a query program by using a binary tree structure to make the search process easy

It is easy and quick to search values in a binary tree because data is stored so that the left subtree may contain data values that are lesser than the data value stored in the parent node, whereas the right subtree may contain data values greater than the parent node.

The sorted binary tree allows the search process to be made only along one subtree until a specific data value is located. To search data in a binary tree, there are certain steps that must be followed. The first step is to move the pointer to the root node. The next step is to prompt the user for the search item. For example, if the `EmpID 5662` is to be searched, prompt the user for the specific `EmpID`. After accepting the search information from the user, compare it with the item in the node to which the pointer is pointing.



If the search item is greater then the data value stored in the current node, set the pointer to the right pointer of the node. In the example given here, the `EmpID 5662` is greater than the `EmpID 2228`. therefore, set the pointer to the right pointer of the node. However, if the search item is less than the node the pointer is pointing to, set the pointer to the left pointer of the node. If the pointer returns a `NULL` value, report that the search item dos not exist. If the search item is not found, quit the loop.

A code sample representing the search process in a binary tree is given here.

Listing 139.1 : Binary Search

```
// Listing 139.1 : This program performs a search process in a binary tree.

#include<stdio.h>
#include<stdlib.h>

struct TreeTag
{
 int info
 struct TreeTag *left;
 struct TreeTag *right;
};

struct TreeTag *root, *ptr;

struct TreeTag *treesearch(int c, struct TreeTag *ptr)
{
 while(ptr!=Null)
  {
   if(c>ptr->info)
   {
      ptr=ptr->right;
   }
   else
   {
    return ptr;
   }
  }
}
return NULL;
}
```

Detailed Explanation

- The `while` loop given here is used to determine if the search item is found. If the item to be searched is greater in value than the node pointed to by `ptr`, set `ptr` to the right pointer.
- The pointer `ptr` is set to the left pointer if the item to be searched is lesser in a value than the node pointed to by `ptr`. If these conditions are not met, the search is terminated.

Self Review Exercise 139.1

1. *You learned about searching for an item in a binary tree. Four code samples are given below. Tick the code sample to search an item in a binary tree.*

```
struct TTag *tsearch(int c, struct TTag
*ptr)
{
While(ptr!=NULL)
{
if(c<ptr->info)
ptr=ptr->right;
else
if(c>ptr->info)
ptr=ptr->left;
else
return ptr;
}
return NULL;
}
```
A

```
struct TTag *tsearch(int c, struct TTag
*ptr)
{
While(ptr!=NULL)
{
if(c=ptr->info)
ptr=ptr->right;
else
if(c<ptr->info)
ptr=ptr->left;
else
return ptr;
}
return NULL;
}
```
B

```
struct TTag *tsearch(int c, struct TTag
*ptr)
{

if(c<ptr->info)
ptr=ptr->right;

else
ptr=ptr->left;
return NULL;

}
```
C

```
struct TTag *tsearch(int c, struct TTag
*ptr)
{
While(ptr!=NULL)
{
if(c>ptr->info)
ptr=ptr->right;
else
if(c<ptr->info)
ptr=ptr->left;
else
return ptr;
}
return NULL;
}
```
D

Lesson Objectives

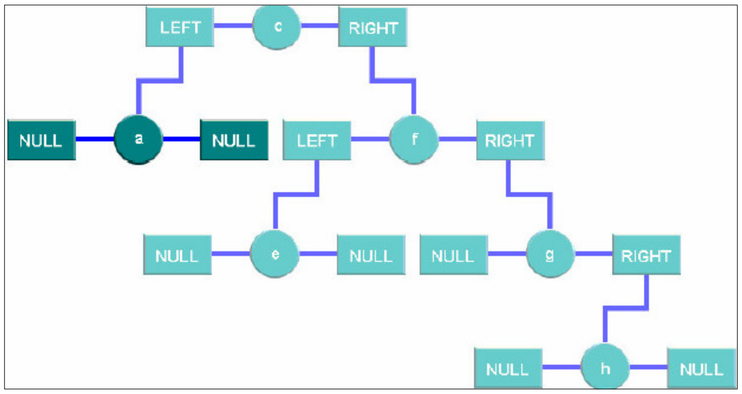
In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Like the procedures of creating and editing nodes in a binary tree, deleting nodes also involves changing pointers. The procedure for deletion is elaborate and complex. In this lesson, you learn about the situations in which a node is deleted and the activities that should be performed to delete a node in a particular situation.

The activities that are performed during the deletion for a node ensure that the links between nodes are not broken. When a node is to be deleted, its left and right subtrees have to be linked to the rest of the binary tree.

During deletions, the node to be deleted may not have any subtrees. Consider the sample diagram given. In the diagram, node a does not contain any subtree.



Detailed Explanation

- If node is deleted, the left pointer of the parent node must be changed to `NULL`. This is because node does not contain any child nodes. If the node being deleted is the last node of the right subtree, the right pointer is the parent node must be changed to `NULL`. Another situation that may occur when deleting a node is that the node to be deleted has only one subtree. For example, the deletion of node `g` can lead to such a situation.
- If the node to be deleted has only one subtree, the left pointer of right pointer of the parent node should be modified to ensure that the parent node of the node to be deleted points to the left node or the right node.
- If node `g` is deleted from the subtree, the structure is the tree will change as displayed in the diagram. In the diagram, the right pointer of node `f` has to point to node `h`. Therefore, the values should be changed to the value of the right pointer of node `g`.

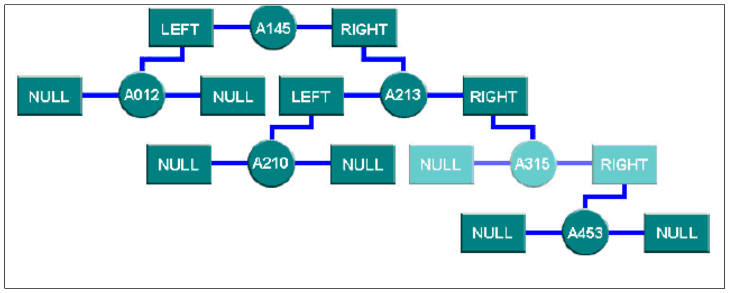
- The third situation for deleting a node from a binary tree is that the node to be deleted has both the left and right subtrees. For example, the deletion of node `f` in the diagram can lead to such a situation.
- The change that would take place if node `f` is deleted is that the entire left subtree of node `f` will be linked under that last left node of the right subtree. For example, node `e` that is in the left subtree will be linked to node `g`.
- After the left subtree is linked to the last node of the right subtree, node `g` that is linked to the right subtree of node `f` has to be linked to node `c`. the right pointer of node `c` should point to the right subtree of node being deleted.
- Another situation that may occur when deleting a node from a binary tree is that the node to be deleted is the root node. The requirement for deleting the root node is to update the address stored in the pointer that points to the root node.
- After the deletion of the root node, the updated pointer will point to the node to the right of the root node that was deleted. The node to the right of the deleted node becomes the new root node.
- The left subtree should be attached to the last left node of the right subtree. In the example given here, the root node c is deleted. The pointer to the root node will be updated to the node that is to the right of the deleted root node and the left subtree is linked to the last node of the right subtree.

Self Review Exercise 140.1

1. *The node that does not contain any child nodes contains the pointer:*

   A. NULL
   B. Left
   C. Right
   D. Root

2. *You learned about the requirements to be considered for deleting a node in a specific situation. A diagram displaying the binary tree structure of item numbers is given here. Tick the activity that must be performed for deleting a node with one subtree.*



A. The corresponding pointer of the parent node must be changed to NULL.
B. The root pointer must be updated so that the rest of the tree is accessible.
C. The left pointer of the right pointer of the parent node should be modified so that the node points to the left subtree or the right subtree of the node being deleted.
D. The left subtree or the right subtree of the node being deleted should be linked to the last terminal node.

LESSON 141 : ACCESS METHODS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

Data management is an important activity for any business application. Data management involves accessing data from various storage media and processing the data. There are three access methods that are used in C to retrieve records from that data files stored on a disk. These methods are sequential access, random or direct access, and indexed access.

In the sequential access method, all the records in a data file are read sequentially until the desired record is reached. The sequential access method is the simplest method for accessing records in a data file.

A disadvantage of using the sequential access method is that it is the least efficient method. If the number of records is very large, such as a few million, a million records may need to be read before a single record is retrieved.

To improve the efficiency and speed of data access, you can use the random access method or the direct access method. This method enables you to use the offset address to access the desired record without traversing through all the records.

The offset address is the relative position of the record from the beginning of the database file. If you know the record number and length, you can calculate the offset address by multiplying the record length with the record number of the previous record. Records are retrieved in the least amount of time by using the random access method.

A disadvantage of using the random access method is that it can be used only for files contain records of fixed lengths. Another disadvantage of using the random access method is that it cannot be used if you do not know the recode number that is to be retrieved. If the record number is not known, the offset address cannot be calculated.

In a situation where that record size and number are not known, the indexed access method is used to retrieve records. The indexed access method is the most commonly used data access method. The indexed access method uses the key field of the required record or records. Data files usually store data as records with several fields. One of these field is used as a key field. The key field contains the data that is used to uniquely identify a record.

In the indexed access method, a table that contains the list of all key fields and their offset positions along with the data file is written to the disk. This table is called the index. The index is sorted in the ascending order of the key field values. If the key field value if known, the offset value can be read from the index and the required records can be retrieved. The indexed access method is faster that the sequential access method. However, it is slower than the random access method.

This lesson covered various methods of data access. The sequential access method is the simplest method, but it is also the least efficient method. The random access method is the fastest method, but it can be used only for data files that contain records of fixed lengths

The indexed access method can be used to access the records in a data file that contains records of variable lengths.

Self Review Exercise 141.1

1. *In the sequential access method:*

   A. records are read most efficiently.
   B. The desired record in the data file is read directly.
   C. The records in the data file are read successively until the desired record is found.
   D. The records in the data file are sorted and then read.

2. *The random access method can be used only for the data files that contain:*

   A. records of fixed lengths.
   B. Records that are sorted.
   C. Records with a unique key value.
   D. A fixed number of records.

3. *The indexed access method:*

   A. Is the fastest data access method.
   B. Is the simplest data access method.
   C. Uses record numbers to retrieve data.
   D. Uses the key field of the required record or records.

LESSON 142 : INDEX --> STORING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The efficiency of data access using the indexed access method depends on the quick and efficient access of index elements. The data structures that are used to store the index influence the speed at which the elements in the index are accessed

In this lesson, you learn about the advantages and disadvantages of using various data structures for storing the index. The data structures that can be used to store the index are the array, the linked list, and the binary tree. The simplest way of storing an index is to use an array. Each element in an array contains two parts, the key value of the records in the data file and the offset position of the record from the beginning of the file.

A major disadvantage of using an array to store an index is that a sequential search is required to locate an element of the array. This makes the retrieval process very show. However, if the array is sorted, you can use binary search to locate an element in the array.

Another disadvantage of using an array for storing an index is the requirement of the periodic sorting of the key values in the array. This is because the addition and deletion of elements in an array result in an unsorted array.

The insertion and deletion of records are difficult to handle in an array because an array needs to be restructured every time an element is inserted or deleted in the array.

You can avoid most of the disadvantages of storing an index in an array by using a linked list to store the index. In a linked list, the key values in the index and offset positions are stored as nodes. An advantage is using a linked list to store an index is that the records in a linked list can be sorted at the time of creating the linked list. Another advantage of using a linked list to store an index is that it is efficient to inert and delete nodes in a linked list. You do not need to restructure a linked list when you insert or delete a node.

To handle the insertion and deletion of nodes in a linked list, you need to manipulate only the address pointers of the nodes before and after the point of insertion or deletion. However, there is a disadvantage is using a linked list to store an index. Linked list need to be searched sequentially to find the required node.

Another data structure that can be used to store an index efficiently is a binary tree. A binary tree stores the key value and the offset position together as a node. There are many advantages of using a binary tree to store an index. A binary tree enables a quick search of data. Another advantage of using a binary tree to store an index is that a binary tree stores sorted data.

In a binary tree index, the search starts at the root node of the tree. The key value of the left node or of the right node is compared t the key value of the node to be located until a match is found.

In addition to storing sorted data, the key values in the left child nodes in a binary tree are of less value than their parent nodes. However, the key values in the right child nodes are greater then those of their parent nodes. This enables you to search for a key value from a binary tree in less time than from an array or linked list.

You learned about the data structures used to store an index. An array is the simplest way to store an index, but it is also the least efficient way. Linked lists and binary trees are more efficient than arrays because it is easy to insert and delete nodes in them.

Self Review Exercise 142.1

1. *Identify the advantage of using arrays to store an index.*

   A. It is easy to insert and delete records.
   B. It is the simplest way of storing an index.
   C. It sorts elements while storing them.
   D. It provides a quick method to retrieve records from a database file.

2. *The advantage of using a linked list to store an index is that :*

   A. Records need not be searched sequentially.
   B. The retrieval of records is faster that for arrays and binary trees.
   C. Records can be inserted and deleted efficiently.
   D. It is the simplest way of storing an index.

LESSON 143 : LINKED LIST INDEX --> CREATING

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

You create an index for a database file to quickly access the records in the database file. This index should be stored in a data structure. A linked list is a data structure that is commonly used to store an index because it is efficient to insert and delete nodes in a linked list.

In this lesson, you learn to crate a linked list index for a data file. The process of creating a linked list index involves four steps. The first step in creating a linked list index is to declare the structures that represent a node in the linked list index.

The structure `RecordData` that stores the key value and the offset position is given below. The structure element key stores the key value. The variable `n` is the size of the key field.

```
structy RecordData
{
char key[n];
long offset;
};
```

The structure `RecordData` is stored in a linked list node. To represent a node of the linked list, a structure needs to be declared. The C code used to declare a structure representing a linked list node that stores the index is given .

```
structy list
{
char recordData *rec;
struct list *next;
};
```

The second step is to allocate memory for linked list index nodes. Every node in a linked list index has a data node associated with it. Memory allocation for the linked list node should also include memory allocation for the data node.

You can allocate memory for linked list index nodes and data nodes associated with them by using the `malloc` function. This function is available in the standard ANSI C library.

The third step in creating a linked list index is to read the records from the database file. In addition, the key and offset values of the database record should be assigned to the appropriate elements of the linked list index node.

You can read key values form the database file by using the `fgets` function. The values read using the `fgets` function are assigned to a linked list node. The offset position is assigned to a linked list node by using the `ftell` function.

The final step to crate a linked list index is to insert the node at the sorted position in the linked list. The linked list should be kept sorted in the order of the key field values. The last three steps are repeated until there are no more records in the database file.

Consider an example of a text file naked `datafile` in which each record contains two fields. `itemCode` and `ItemName`. The key field `ItemCode` is a string of five characters, and the field `ItemName` is a string of `12` characters.

Each record in the database is preceded by a single byte for a deleting flag. This single byte is set to D if the record is marked as deleted. Otherwise, it is set to `U`.

To crate a linked list index, first declare a structure to store the key and offset values. The code used to declare the structure that stores the key and offset values is given here. The extra byte in the variable `ItemCode` is reserved for the `NULL` character.

LESSON 143 : LINKED LIST INDEX --> CREATING

Listing 143.1 : creating linked list index

```
//Listing 143.1 : This program creates a linked list index.

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct RecordData
{
char ItemCode[6];
long offset;
};

struct list
{
struct RecordData *rd;
struct list *next;
};
struct list *start;
struct list *newnode;
FILE *fs;
void MakeNode()
{
newnode=(struct list *)malloc(sizeof(struct list));
newnode->rd=(struct RecodData *)malloc(sizeof(struct RecordData));
}
GetData()
{
char fkey[6], rest[14];
newnode->rd->offset-ftell(fs);
fgets(fkey, 6, fs);
fgets(rest, 14, fs);
strcpy(newnode->rd->ItemCode, fkey);
newnode->next=NULL;
}

MakeIndex()
{
char st[2];
fs=fopen(“detafile.txt”,”r”);
rewind(fs);

while(fgets(st, 2, fs))
{
MakeNode();
GetData();
if(st[0]!=’D’)
{
if(start==NULL)
{
start = newnode;
}
else
{
for(ptr=start;(ptr!=NULL)&&strcmp(newnode->rd->ItemCode, ptr->rd->ItemCode)>0;
prev-ptr, ptr-ptr->next);
if(ptr==start)
{
newnode->next=start;
start=newnode;
}
else
{
prev->next=newnode;
newnode-> next=ptr;
}
}
}
}
}
```

The key value and the offset position need to be stored in a linked list. The structure required to declare a node of this inked list is displayed here..

Detailed Explanation

- A linked list index node can be referred to by using a pointer to the structure list. A pointer, start, that indicates the first node of the linked list is declared. Another pointer, `newnode`, that points to the new node that is to be added to the linked list is also declared.
- After declaring the structures that represent a linked list index node, a use-defined function, `MakeIndex`, is declared. This function allocates memory to the nodes, assigns the key and off-set values to the linked list index nodes, and inserts the nodes in the sorted positions.
- The `MakeIndex` function invokes another user-defined function named `MakeNode` to allocate memory for nodes in the linked list. The `MakeNode` function allocates memory to the linked list index node pointed to by the pointer `newnode`.
- First, memory is allocated to the linked list index node that is pointed to by the pointer `newnode` by using the `malloc` function. The structure list is passed as an argument to the `malloc` function.
- Next, memory is allocated to the data elements in the structure `RecordData`. The structure `RecordData` is passed as an argument to the `malloc` function.
- After memory is allocated to the new node, the records in the file `datafile` are read. In addition, the values of the key field and the offset position are assigned to the elements in the linked list index node.
- Before you read the records in the file `datafile`, the records of the file need to be accessed. To access the records in the file, a pointer named `fs` is declared.
- To read the records in the file `datafile` from the beginning, the file `datafile` is opened in the read node and the file pointer `fs` is placed at the beginning of the file.
- The records in the database file are read successively by using the `while` loop given here. The `fgets` function returns the value in the first argument, `st`, if it is able to read record. It returns `NULL` if the end of the file is reached.
- In the `while` loop, invoke the user-defined function named `MakeNode` to allocate memory for the new node.
- Next, read the records in the database file and assign the key and offset position values to the linked list index node. To do this, invoke another user-defined function named `GetData`.
- The user-defined function `GetData` enables you to read the records and assign values to the elements of the linked list node. In this function, two variables, `key` and `rest`, are declared to store the key value and the values in the rest of the record, respectively.

- The offset position is assigned to the offset element of the new node by using the `ftell` function.
- Next, the `fgets` function is used to read the key value. In addition, the remaining part of the record, including the end-of-line character, is read to set the file pointer `fs` to the beginning of the next record. The key field value is assigned to the key element of the linked list node.
- After assigning the offset position and key field values to the new node, the pointer next of the new node is set to `NULL`.
- Finally, the new node needs to be inserted at the sorted location in the linked list. To do this, two pointers, `prev` and `ptr`, are declared to store the addresses of the nodes that are before and after the point in the linked list index where the new node is to be inserted.
- Before inserting the node in the linked list, check the first byte of the record to verify if the record is marked for deletion or not. If the first byte is equal to `D`, the record is marked for deletion. In this situation, do not insert the node in the linked list index.
- If the record is not marked for deletion, verify if the linked list exists by checking the value in the pointer start. If the pointer `start` is equal to `NULL`, the linked list is created by assigning the address of the new node to the pointer `start`.
- If the linked list exists, determine the sorted position of the new node. To do this, compare the key value in the new node to the key value of the node pointed to by the pointer until a node that contains a key value that is greater than that of the new node is found.
- If the value of the pointer `ptr` is equal to `start`, the new node is inserted in the beginning of the linked list. The pointer `next` of the new node is assigned the address stored on the pointer `start`. Then, the pointer `start` is updated.
- If the value of the pointer `ptr` is not equal to `start`, the new node is inserted in the middle or at the end of the linked list.
- The pointer `next` of the node pointed to by the pointer `prev` is assigned the value in the pointer `newnode`. The pointer `next` of the new node is assigned the value in to pointer `ptr`.

Self Review Exercise 143.1

1. *You learned about the steps that are involved in creating a linked list index. Four sequences are given here. Tick the correct sequence of steps used to create a linked list index now.*

A Declare the structure required to represent a node in the linked list index.
Allocate memory for the nodes in the linked list index.
Read records from the database file and assign key and offset values to linked list nodes.
Insert the node at the sorted position in the linked list.

B Read records from the database file and assign key and offset values to linked list nodes.
Declare the structure required to represent a node in the linked list index.
Allocate memory for the nodes in the linked list index.
Insert the node at the sorted position in the linked list.

C Declare the structure required to represent a node in the linked list index.
Read records from the database file and assign key and offset values to linked list nodes.
Allocate memory for the nodes in the linked list index.
Insert the node at the sorted position in the linked list.

D Declare the structure required to represent a node in the linked list index.
Allocate memory for the nodes in the linked list index.
Insert the node at the sorted position in the linked list.
Read records from the database file and assign key and offset values to linked list nodes.

Self Review Exercise 143.2

2. *You learned to declare structures required to create a linked list index. Identify the code used to declare the structures required t create a linked list index for a data file in which the key field ProductID is a string of 4 characters. Tick the correct code.*

```
struct ProductData
{
char product[5];
};
struct ProductList
{
struct ProductData
*prod;
struct product List
*next;
};
```
A

```
struct ProductData
{
char product[5];
long offset;
};
struct ProductList
{
struct ProductData
*prod;
struct productData
*next;
};
```
B

```
struct ProductData
{
char product[5];
long offset;
};
struct ProductList
{
struct ProductData
*prod;
struct product List
*next;
};
```
C

```
struct ProductData
{
char product[5];
long offset;
};
struct ProductList
{
struct ProductData
*prod;
};
```
D

3. *You learned to allocate memory for a linked list index node. Tick the appropriate C code used to allocate memory for the linked list index nodes that are represent by the structure ProductData and ProductList.*

```
struct ProductLIst *newnode;
void MakeNode()
{
     newnode=(stryct ProductList *)malloc(sizeof(struct ProductList));
}
```
A

```
struct ProductLIst *newnode;
void MakeNode()
{
     newnode=(stryct ProductList *)malloc(sizeof(struct ProductList));
     newnode->prod=(struct ProductData *)malloc(sizeof(struct ProductData));
}
```
B

```
struct ProductLIst *newnode;
void MakeNode()
{
     newnode->prod=(struct ProductData *)malloc(sizeof(struct ProductData));
}
```
C

```
void MakeNode()
{
     newnode=(stryct ProductList *)malloc(sizeof(struct ProductList));
     newnode->prod=(struct ProductData *)malloc(sizeof(struct ProductData));
}
```
D

Self Review Exercise 143.3

4. *You learned to create a linked list index. Partial code used to create a linked list index is displayed over here. Four code sample are also given. Tick the appropriate code sample used to assign the key and offset values to a linked list index node.*

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

struct ProdData
{
                char ProductID[6];
                long offset;
};

struct Prodlist
{
                struct ProdData *rd;
                struct Prodlist *next;
};
struct Prodlist *start;
struct Prodlist *nd;
FILE *fs;
void MakeNode()
{
                nd=(struct Prodlist *)malloc(sizeof(struct Prodlist));
                nd->pd=(struct ProdData *)malloc(sizeof(struct ProdData));
                return(nd);
}
MakeIndex()
{
                char st[2];
                fs=fopen("detafile.txt","r");
                rewind(fs);

                while(fgets(st, 2, fs))
                {
                        MakeNode();
                        GetData();
```

```
GetData()
{
char fkey[5], rest[19];
nd->prod->offset=ftell(fs);
fgets(fkey, 5, fs);
strcpy(nd->pd->prodID, fkey);
nd->next=NULL;
}
```
A

```
GetData()
{
char fkey[5], rest[19];
nd->prod->offset=ftell(fs);
fgets(fkey, 5, fs);
fgets(rest, 19, fs);
strcpy(nd->pd->prodID, fkey);
nd->next=NULL;
}
```
B

```
GetData()
{
char fkey[5], rest[19];
nd->pd->offset=ftell(fs);
fgets(fkey, 5, fs);
fgets(rest, 19, fs);
strcpy(nd->pd->prodID, fkey);
nd->next=NULL;
}
```
C

```
GetData()
{
char fkey[5], rest[19];
nd->pd->offset=ftell(fs);
fgets(fkey, 5, fs);
fgets(rest, 19, fs);
strcpy(nd->pd->offset, fkey);
nd->next=NULL;
}
```
D

LESSON 144 : LINKED LIST INDEX --> WRITING TO THE DISK

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

When you create a linked list index, it exists in memory. To access the index file data subsequently, write a copy of the linked list index to an index file on the hard disk of your computer.

To write the linked list index for a data file to an index file on the hard disk of your computer, sequentially access the successive nodes of the linked list index. Assign the key and offset values of each node to an index file.

Consider the example of a linked list index that is represented by the structures `RecordData` and `list`. The structure `RecordData` contains the key field `ItemCode` and the offset value.

Listing 144.1 : Writing a linked list index to files

```
//Listing 144.1 : This program writes a linked list index to a file.
#include<stdlib.h>
#include<stdio.h>

struct RecordData
{
char ItemCode[6];
long offset;
};

struct list
{
struct RecordData *rd;
struct list *next;
};

struct list *start;
struct list *ptr;

FILE *fw;
write Index()
{
ptr=start;
fw=fopen("index.txt","u");
if(start != NULL)
{
while(ptr!=NULL)
{
fprintf(fw,"%s %d\n", ptr->rd->ItemCode, ptr->rd->offset);
ptr=ptr->next;
}
}
else
{
printf("\n The list is empty.");
}
fclose(fw);
}
```

Detailed Explanation

- A pointer named `start` is declared to store the address of the first node of the linked list index. This pointer is required to access the linked list index.
- To sequentially write the key and offset values in the linked list index to an index file on the disk, declare a user-defined function named `WriteIndex`.
- The function `WriteIndex` reads the key and offset values stored in the nodes of the linked list index and writes these values to the index file. To access and traverse the index file, declare a file pointer that points to the beginning of the index file initially.
- Next, declare a pointer named `ptr` that points to the current node in the linked list. This pointer is required to successively access the nodes in the linked list index. To begin the process of reading the nodes of the linked list index successfully, the pointer `ptr` needs to point to the first node. This is done by assigning the value in the pointer `start` to the pointer `ptr`.
- To write the key and offset values in the nodes of the linked list index to the index file, open the index file `I` the write mode. This is done using the `fopen` function.
- Next, verify if the linked list contains nodes or is empty. If the linked list is empty, display a message that the list is empty.
- However, if the linked list is not empty, write the key and offset values to the index file on the disk and advance the pointer `ptr` to the next node. Perform this operation until you reach the last node in the linked list.
- To write the key and offset values to the disk, use the `fprintf` function. This function writes the key and offset values stored in the code pointed to by the pointer `ptr` to the file pointed to by the file pointer `fw`.
- After writing the key and offset values of all the nodes in the linked list to the index file, close the index file by using the function `fclose`.

Self Review Exercise 144.1

1. *The process of writing a linked list index to a file involves accessing the successive nodes on the linked list index and :*

A. converting the key and offset values to long integer.
B. Storing the key and offset values in memory.
C. Assigning the key and offset values to an index file on the disk.
D. Dumping the key and offset values to an array.

Self Review Exercise 144.2

2. *You learned to write a linked list index to the disk. Partial code used to write a linked list index to the disk is displayed over here. Four code samples are also given. Write the correct code sample used to complete this partial code now.*

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

struct EmployeData
{
       char EmployeeName[25];
       long offset;
};

struct EmployeeList
{
       struct EmployeeData *rec;
       struct EmployeeData *next;
};

struct EmployeeList *start;
FILe *fw;
struct EmployeeList *cur;
WriteIndex()
{
       Cur=start;
       fw=fopen("index","w");
       if(start!=NULL)
{

}
else
{
       printf("The list is empty");
}
fclose(fw);
}
```

```
while(cur!=NULL)
{
fprintf(fw,"%s %ld", cur->rec-> EmployeeName, cur->next);
cur=cur->next;
}
```
A

```
while(cur!=NULL)
{
fprintf(fw,"%s %ld", cur->rec-> EmployeeName, cur->rec->offset);
}
```
B

```
while(cur!=NULL)
{
fprintf(fw,"%s %ld", cur->rec, cur->rec->offset);
cur=cur->next;
}
```
C

```
while(cur!=NULL)
{
fprintf(fw,"%s %ld", cur->rec-> EmployeeName, cur->rec->offset);
cur=cur->next;
}
```
D

Lesson Objectives

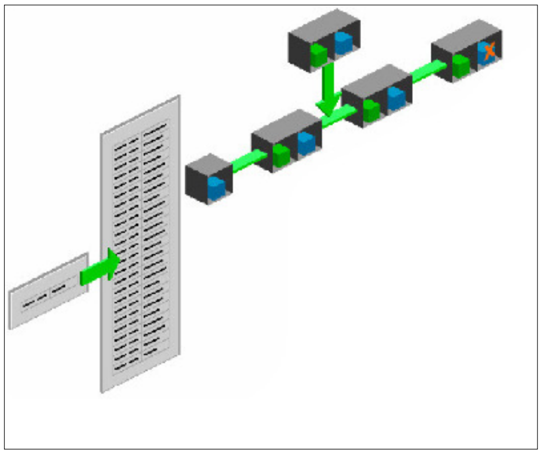
In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A major advantage of using a linked list to store an index is that the processes for inserting and deleting nodes are easy to implement. This topic covers the C code used to insert a node in an existing linked list index when a new record is to be added in the underlying database file.

There are four steps that are involved in the process for inserting a new node in a linked list index hen a new record is to be added in the underlying database file. The first step is to assign the offset position of the beginning of the new record to a new node in the linked list index.

The second step is to assign the key value to the new node in the linked list index. The third step is to write the new record at the end of the database file. The last step in the process for inserting a new node in a linked list index is to insert the new node at the sorted position in the linked list index.



- Assign the offset position of the beginning of the new record to a new node.
- Assign the key value to the new node.
- Write the new record at the end of the database file.
- Insert the new node at the sorted position.

Consider an example of a linked list index that is based on the structures `RecordData` and `list`. The data file underlying this linked list index contains two fields. The key field `ItemCode` is a string of five characters, and the field `ItemName` is a string of `12` characters. A pointer named `start` stores the address of the first node of this linked list index.

LESSON 145 : LINKED LIST INDEX --> HANDLING INSERTIONS

Listing 145.1 : insert a node in a linked list

```
// Listing 145.1 : This program inserts a node in a linked list index.

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct RecordData
{
   char item[6];
   long offset;
};

struct list
{
   struct RecordData *rd;
   struct list *next;
};

struct list *start, *node;
FILE *fs;

void MakeNode()
{
  newnode=(struct list *)malloc(sizeof(struct list));
  newnode->rd=(struct RecordData *)malloc(sizeof(struct RecordData));
}

InsertNode()
{
  MakeNode();
  GetNewNode();
  if(start==NULL)
  {
   start=newnode;
  }
  else
  {
   for(ptr=start;(ptr!=NULL)&&strcmp(newnode->rd->ItemCode,
   ptr->rd->ItemCode)>0; prev=ptr, ptr=ptr->next;
   if(ptr==start)
   {
    newnode->next=start;
    start=newnode;
   }
   else
   {
    prev->next=newnode;
    newnode->next=ptr;
   }
 }
}
```

Detailed Explanation

- Before inserting the new record at the end of the data file, the file must be traversed to reach the end of the file. To traverse the database file, a file pointer named `fs` is declared.
- To insert a new node in the linked list index when a record is added to the underlying database file, a user-defined function, `InsertNode`, is declared.
- First, the `InsertNode` function invokes a user-defined function `MakeNode` to allocate memory for the new node. The `MakeNode` function allocates memory to the structures list and `RecordData` by using the `malloc` function.

- Next, the `InsertNode` function invokes another user-defined function, `GetNewNode`, to assign the key and offset values to the new node.
- To start the process of inserting a new record, the `printf` function is used to prompt the user to type the item code and name. The item code and item name values typed by the user are stored in the `ICode` and `IName` variables, respectively, by using the scanf function.
- Next, the new record should be appended at the end of the file `datafile`. To append the new record at the end of the file, open the file in the read-write mode and place file pointer `fs` at the end of the file.
- Assign the offset position of the new record to the offset element of the new node. The offset position of the new record is equal to `1` byte added to the value returned by the `ftell` function. The extra byte is for the delete flag.
- After assigning the offset position value, assign the key field value of the record to the `ItemCode` element of the new node in the linked list index. To do this, copy the value in the variable `ICode` to the element named `ItemCode` of the new linked list index node.
- Next, write the new record at the end of the data file and close the file. The `fprintf` function is used to write the values in the variables `ICode` and `IName` to the data file. The first byte of the record is set to `U` to indicate that the record is not to be deleted.
- After assigning the key and offset values, the new node is to be inserted at the sorted location in the linked list index.
- To find the sorted position of the new node in the linked list index, two pointers, `prev` and `ptr`, are required. The pointer `prev` points to the node preceding the new node, and the pointer `ptr` points to the node after the new node.
- Before searching the sorted position of the new node, verify whether the linked list index exists. If the pointer `start` is equal to `NULL`, the linked list is empty. In this situation, the address of the new node is assigned to the pointer `start` to create the linked list index.
- If the linked list is not empty, compare the key field `ItemCode` of the new node to the `ItemCode` element of the existing nodes. Perform this process until you find a node in which the value in the `ItemCode` variable is greater than the `ItemCode` value of the new node.
- In addition to comparing the `ItemCode` value in the new node to the `ItemCode` value of the existing nodes, assign the value in the pointer `ptr` to the pointer `prev` and advance the pointer `ptr`.
- If the value of the pointer `ptr` is equal to `start`, the new node is to be inserted at the beginning of the linked index. To insert the new node, assign the value in the pointer `start` to the pointer next of the new node and update the pointer `start`.
- If the pointer `ptr` is not equal to `start`, the new node is to be inserted either in the middle or at the end of the linked list.

- To insert the new node, assign the value in the pointer `newnode` to the pointer `next` of the node pointed to by the pointer `prev`. In addition, assign the value in the pointer `ptr` to the pointer `next` of the new node.

Self Review Exercise 145.1

1. *You learned the steps involved in the process for inserting a new node in a linked list index. Four sequences are given below. Choose the correct sequence of steps to insert a new node in a linked list index.*

Assign the offset position of the new record to the linked list node.
Write the new record at the end of the database file.
Insert the new node at the sorted position in the linked list index.
Assign the key field value to the new linked list node.

A

Assign the offset position of the new record to the linked list node.
Write the new record at the end of the database file.
Assign the key field value to the new linked list node.
Insert the new node at the sorted position in the linked list index.

B

Write the new record at the end of the database file.
Assign the offset position of the new record to the linked list node.
Assign the key field value to the new linked list node.
Insert the new node at the sorted position in the linked list index.

C

Assign the offset position of the new record to the linked list node.
Assign the key field value to the new linked list node.
Write the new record at the end of the database file.
Insert the new node at the sorted position in the linked list index.

D

Self Review Exercise 137.2

2. *You learned to add a new record at the end of a database file and assign the key and offset values to the linked list index node. Partial code used to perform this task is given below. Choose the correct block of code required to complete the code.*

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
struct EmpData{
   char EmpID[5];
   long offset;
};
struct Emplist{
   struct EmpData *rec;
   struct Emplist *next;
};
struct Emplist *start, *node;
FILE *fs;
MakeNode()
{
  newnode=(struct list *)malloc(sizeof(struct list));
  newnode->rd=(struct RecordData *)malloc(sizeof(struct RecordData));
}
GetNewNode()
{
  char EID[5], EName[13];
  printf("Type employee ID:");
  scanf("%s", EID);
  printf("Type employee name:");
  scanf("%s", EName)(;
  fs=fopen("dataFile.txt", "r+");
  close(fs);
}
InsertNode()
{
  MakeNode();
  GetNewNode();
}
```

```
fseek(fs, 0, SEEK_END);
node->rec->offset=ftell(fs)+1;
strcpy(node->rec->EmpID, EID);
fprintf(fs, "U%4s%12s", EID, EName);
```
A

```
fseek(fs, 0, SEEK_END);
node->rec->offset=ftell(fs)+1;
strcpy(node->rec->EmpID, EID);
```
B

```
fseek(fs, 0, SEEK_END);
node->rec->offset=ftell(fs)+1;
fprintf(fs, "U%4s%12s", EID, EName);
```
C

```
fseek(fs, 0, SEEK_END);
strcpy(node->rec->EmpID, EID);
fprintf(fs, "U%4s%12s", EID, EName);
```
D

Self Review Exercise 137.3

3. *You learned to insert a new node in a linked list index. Partial code used to insert a new node in a linked list index represented by the structures* `Empdata` *and* `EmpList` *is given below. Choose the code required to find the sorted position of the node.*

```
struct Emplist *start, *node, *prev, *ptr;
MakeNode()
{
  newnode=(struct list *)malloc(sizeof(struct list));
  newnode->rd=(struct RecordData *)malloc(sizeof(struct RecordData));
}
GetNewNode()
{
  char EID[5], EName[13];
  printf(“Type employee ID:”);
  scanf(“%s”, EID);
  printf(“Type employee name:”);
  scanf(“%s”, EName)(;
  fs=fopen(“dataFile.txt”, “r+”);
  fseek(fs, 0, SEEK_END);
  Node->rec->offset=ftell(fs)+1;
  strcpy(node->rec->EmpID, EID);
  fprintf(fs, “U%4s%12s”, EID, EName);
  close(fs);
}
InsertNode()
{
  MakeNode();
  GetNewNode();
  if(start==NULL)
    Start=node;
  else
{
}
```

```
for(ptr=start;(ptr!=NULL)&&strcmp(node->rec->offset,ptr->rec->offset)>0; prev=ptr, ptr=ptr
->next;
```
A

```
for(ptr=NULL;(ptr!=NULL)&&strcmp(node->rec->EmpID,ptr->rec->EmpID)>0; prev=ptr,    ptr=ptr
->next;
```
B

```
for(ptr=start;(ptr!=NULL)&&strcmp(node->rec->offset,ptr->rec->offset)>0; ptr=ptr->next;
```
C

```
for(ptr=start;(ptr!=NULL)&&strcmp(node->rec->EmpID,ptr->rec->EmpID)>0; prev=ptr,   ptr=ptr
->next;
```
D

Lesson Objectives
In this Lesson you will learn : - *the Genesis of C Language* - *the Stages in the Evolution of C Language*

You can remove records from the database files on the hard disk of your computer. For example, you may want to delete the record of a product that is discontinued from the products database file.

If a record in a database file is deleted, the corresponding node in the linked list index should also be deleted. Otherwise, the linked list index for that database file will store nodes that point to records that do not exist.

To delete a record from a database file, indicate that the record is marked for deletion. This is done by setting the first byte of the record to be deleted as `D`.

However, it is not important to physically remove the records marked for deletion immediately. Periodically, the records marked for deletion are physically removed from the database file by copying all the records referred to by the linked list index to another database file.

After indicating that the record is marked for deletion, delete the corresponding node from the linked list index. This is done to remove the reference to the record marked for deletion.

Consider an example of a text file named `datafile` in which records contain two fields, `ItemCode` and `ItemName`. The key field `ItemCode` is a string of five characters. The field `ItemName` is a string of `12` characters.

A linked list index is created for the file `datafile`. This linked list index is based on the structures `RecordData` and `list`. The pointer `start` stores the address of the first node in the linked list. File pointer `fs` is declared to point to the file `datafile`.

Listing 146.1 : Deleting nodes from linked list index

```
// Listing 146.1 : This program deletes a node from a linked list index.

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct RecordData
{
   char ItemCode[6];
   long offset;
};

struct list
{
   struct RecordData *rd;
   struct list *next;
};

struct list *start;
FILE *fs;

DeleteNode( )
{
   char item[6];
   long os;
   printf("Type the item code of the record to be deleted:  ");
   scanf("%s",item);

   if(start!=NULL)
   {

    for(ptr=start;(ptr!=NULL)&&strcmp(item,ptr->rd->ItemCode)>0;prev=ptr,ptr=ptr->next);

     if(strcmp(item, ptr->rd->ItemCode==0)
      {

       if(ptr==start)
        {
         start = ptr->next;
        }
       else
        {
         prev->next=ptr->next;
        }

       os=ptr->rd->offset;
       free(ptr);
       fs=fopen("datafile.txt","r+");
       fseek(fs, os-1, SEEK_SET);
       fprintf(fs, "D");
       fclose(fs);
      }

     else
      {
       printf("The key value does not exist.");
      }

    }

   else
    {
     printf("The list is empty.");
    }

}
```

Detailed Explanation

- The variable `item` of the type character is declared to store the value of the key field for the record that is to be deleted. Another variable, `os`, of the type long integer is declared to store the offset position of the record that is to be deleted.
- To identify the record to be deleted, the `keyvalue` of that record is required. The user is prompted to type the `keyvalue` of the record to be deleted by using the `printf` function. The `scanf` function is used to assign the `keyvalue` to the variable `item`.
- Next, to search for the corresponding node in the linked list index, two pointers, `prev` and `ptr`, are declared. The pointer `prev` points to the node preceding the node that is to be deleted, and the pointer `ptr` points to the node that is to be deleted.
- Before you start searching for the linked list node corresponding to the record to be deleted, verify if the list is empty or not. If the list is empty, display a message stating that the list is empty.
- If the list is not empty, locate the node to be deleted by using the `for` loop. Initialize the pointer `ptr` to `start`. Compare the key value assigned to the variable `item` with the key value in the node pointed to by the pointer `ptr`.
- Advance the pointer `ptr` until you find a node that contains the key value in the variable `item`. Before you advance the pointer `ptr`, assign its value to the pointer `prev`.
- If the key value of the node pointed to by the pointer `ptr` is equal to the value in the variable item after quitting the `for` loop, delete the node pointed to by the pointer `ptr`. Otherwise, display a message stating that the key value does not exist.
- Before deleting the node pointed to by the pointer `ptr`, verify whether the pointer `ptr` is equal to `start`. In this situation, the node to be deleted is the first node.
- To delete the first node, assign the value in the pointer `next` of the first node to the pointer `start`. This removes the reference to the first node.
- If the pointer `ptr` is not equal to `start`, the node to be deleted is either in the middle or at the end of the linked list.
- To delete the node corresponding to the record to be deleted, assign the value in the pointer `next` of the node pointed to by the pointer `ptr` to the pointer `next` of the node pointed to by the pointer `prev`.
- Next, assign the offset value in the node pointed to by the pointer `ptr` to variable `os`. The offset value is required to search the record in the file `datafile` that corresponds to the node pointed to by the pointer `ptr`.
- After assigning the value in the pointer `ptr` to variable `os`, release the memory occupied by the node pointed to by the pointer `ptr` by using the `free` function.

- Next, open the text file `datafile` in the read-write mode and place file pointer `fs` in the beginning of the record to be deleted. This is done using the `fseek` function.
- The delete flag of the record pointed to by file pointer `fs` is set to `D`. This indicates that the record is marked for deletion. Finally, the text file `datafile` is closed using the `fclose` function.

Self Review Exercise 146.1

1. *Identify the statement that describes the process of deleting a node from a linked list index.*

A. The record is physically removed from the database file before deleting the corresponding node in the linked list index.
B. The key value in the linked list node that corresponds to the record to be deleted is set to NULL.
C. The delete flag of the database file record to be deleted is set to D, and the corresponding node in the linked list is removed.
D. The next pointer in the linked list index node that corresponds to the record to be deleted is set to NULL.

2. *You learned the code used to find the node that corresponds to the database file record to be deleted. Choose the appropriate piece of code required to complete the C code used to find the node that corresponds to the database file record to be deleted.*

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

struct EmployeeData
{
   char EmployeeID[5];
   long offset;
};

struct list
{
   struct EmployeeData *rec;
   struct list *next;
};

struct list *start, *ptr, *prev;
DeleteNode( )
{
   char empID[5];
   long os;
   printf("Type the employee id of the record to be deleted:  ");
   scanf("%s",empID);

   if(start!=NULL)
   {
```

```
for(ptr=start;(ptr!=NULL)&&strcmp(empID, ptr->rec->EmployeeID)>0;ptr=ptr->next);
```
A

```
for(ptr=start;(ptr!=NULL)&&strcmp(empID, ptr->rec->EmployeeID)==0;ptr=ptr->next);
```
B

```
for(ptr=start;(ptr!=NULL)&&strcmp(empID, ptr->rec->EmployeeID)>0;prev=ptr;ptr=ptr->next);
```
C

```
for(ptr=start;(ptr!=NULL)&&strcmp(empID, ptr->rec->offset)>0;ptr=ptr->next);
```
D

Self Review Exercise 146.2

3. *You learned to delete a linked list index node. A linked list index is base on the structures EmpData and EmpList. Partial code used to delete a linked list index node is given below. Choose the correct block of code used to complete this partial code.*

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
struct EmpData{
   char EmpID[5];
   long offset;
};
struct Emplist{
   struct EmpData *rec;
   struct Emplist *next;
};
struct Emplist *start;
Struct EmpList *ptr, *prev;
FILE *fs;
DeleteNode( ){
   char eid[5];
   long os;
   printf("Type employee ID: ");
   scanf("%s",eid);
   if(start!=NULL)   {
    for(ptr=start;(ptr!=NULL)&&strcmp(eid,ptr->rec->EmpID)>0;prev=ptr, ptr=ptr->next);
     if(strcmp(eid, ptr->rec->EmpID==0)      {
      }
     else      {
       printf("The key value does not exist.");
      }
    }
   else
    {
     printf("The list is empty.");
    }
}
```

```
if(ptr==start)
{
 start = ptr->next;
}
else
 {
  prev->next=ptr->next;
 }
  os=ptr->rec->offset;
  fs=fopen("datafile.txt","r+");
  fseek(fs, os-1, SEEK_SET);
  fprintf(fs, "D");
  fclose(fs);
```
A

```
if(ptr==start)
{
 start = ptr->next;
}
else
 {
  prev->next=ptr->next;
 }
  os=ptr->rec->offset;
  free(ptr);
  fs=fopen("datafile.txt","r+");
  fseek(fs, os-1, SEEK_SET);
  fclose(fs);
```
B

```
if(ptr==start)
{
 start = ptr;
}
else
 {
  prev->next=ptr->next;
 }
  os=ptr->rec->offset;
  free(ptr);
  fs=fopen("datafile.txt","r+");
  fseek(fs, os-1, SEEK_SET);
  fprintf(fs, "D");
  fclose(fs);
```
C

```
if(ptr==start)
{
 start = ptr->next;
}
else
 {
  prev->next=ptr->next;
 }
  os=ptr->rec->offset;
  free(ptr);
  fs=fopen("datafile.txt","r+");
  fseek(fs, os-1, SEEK_SET);
  fprintf(fs, "D");
  fclose(fs);
```
D

LESSON 147 : BINARY TREE INDEX --> READING RECORDS

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

A binary tree index enables you to search a node in less time as compared to a linked list index. This lesson covers the C code used to read the records in a binary tree index.

The inorder traversal algorithm can be used to read records sequentially from the binary tree index. Before reading a record, it should be located. To locate a record in the database file, the node that contains the key value equal to the key value in the record is searched for in the binary tree index.

Next, use the offset value to search the record in the database file.

After locating the record in the database file, use the `fgets` function to read the values in various fields of the record.

Consider an example of a database file in which the records contain two fields, `ItemCode` and `ItemName`. The key field is `ItemCode`. The size of the `ItemCode` field is `4` bytes, and the size of the `ItemName` field is 20 bytes.

A binary tree index is created for the database file whose key field is `ItemCode`. The binary tree index is based on the structures `RecordData` and `TreeNode`.

LESSON 147 : BINARY TREE INDEX --> READING RECORDS

Listing 147.1 : searching records by using a binary tree index

```c
//Listing 147.1 : This program searches for records by using a binary tree index..

#include <stdlib.h>
#include <stdio.h>
struct RecordData
{
char ItemCode[5];
long offset;
};
struct TreeNode
{
struct RecordData *dat;
struct TreeNode *left;
struct TreeNode *right;
};

FILE *fs;
char *ItemCodeval,*prevptr;
FindNode( )
{
char ItemCode[5];
char ItemName[21];
while(prevptr!=NULL)
{
if(strcmp(ItemCodeval,prevptr->dat->ItemCode)<0)
{
prevptr=prevptr->left;
}
else
{
if(strcmp(ItemCodeval,prevptr->dat->ItemCode)>0)
{
prevptr=prevptr->right;
}
else
{
break;
}
}
}
if(prevptr!=NULL)
{
fseek(fs,prevptr->dat->offset-1,0);
fgets(ItemCode,5,fs);
fgets(ItemName,21,fs);
}
}
```

Detailed Explanation

- To search for the key value and the offset position value in the index and sub sequentially locate the record in the database file, certain global variables should be declared. A global variable, `fs`, of the type file pointer is declared to point to the records in the database file.
- Another global variable is `ItemCodeval`, which is a character type pointer. It points to the key value to be searched. A pointer, `prevptr`, that points to the node in the binary tree index is declared. This pointer points to the root node before you start to search the key values.
- After declaring the global variables, a user-defined function named `FindNode` is declared. This function is used to search for the key value in the index and sub sequentially locate and read the record in the database file.

- To read the values stored in the records of the database file, two variables are declared. The variables `ItemCode` and `ItemName` are declared to read the item code and the item name, respectively.
- To locate the keyvalue in the binary tree index, the nodes are traversed and the key value in each node is compared to the keyvalue to be searched until a match is found or the last node is reached.
- If the `keyvalue` of the node pointed to by the pointer `prevptr` is greater than the key value to be searched for, the pointer `prevptr` is advanced to the left.
- If the key value of the node pointed to by the pointer `prevptr` is not greater than the key value to be searched for, verify if the key value to be searched for is greater than the key value of the node pointed to by the pointer `prevptr`.
- If the `keyvalue` to be searched for is greater than the `keyvalue` of the current node, advance the pointer `prevptr` in the right direction.
- If the `keyvalue` to be searched for is equal to the `keyvalue` in the node, quit the `while` loop. Next, verify if the value in `prevptr` is not equal to `NULL`. If the value in `prevptr` is `NULL`, it implies that the searched `keyvalue` was not found.
- However, if the value in the pointer `prevptr` is not `NULL`, the offset value is passed to the `fseek` function and the corresponding record in the database file is located.
- Finally, the `fgets` function is used to read the code and name of the item and assign the values to the variables `ItemCode` and `ItemName`, respectively.

Self Review Exercise 147.1

1. *You learned to search for records by using a binary tree index. Choose .the correct code sample to find a node in a binary tree index and the corresponding record in a database file.*

```
if(strcmp(ProdID, ptr->dat->offset)<0)
ptr=ptr->left;
else
if(strcmp(ProdID,ptr->dat->offset)>0)
ptr=ptr->right;
else
break;
if(ptr!=NULL)
{
fseek(fs,ptr->dat->offset-1,0)
fgets(ProdID, 5,fs);
fgets(ProdName, 21, fs);
}
```
A

```
if(strcmp(ProdID, ptr->dat->ProdID)<0)
ptr=ptr->left;
else
if(strcmp(ProdID,ptr->dat->ProdID)>0)
ptr=ptr->right;
else
break;
if(ptr!=NULL)
{
fseek(fs,ptr->dat->ProdID,0)
fgets(ProdID, 5,fs);
fgets(ProdName, 21, fs);
}
```
B

```
if(strcmp(ProdID, ptr->dat->ProdID)<0)
ptr=ptr->left;
else
if(strcmp(ProdID,ptr->dat->ProdID)>0)
ptr=ptr->right;
else
break;
if(ptr!=NULL)
{
fseek(fs,ptr->dat->offset-1,0)
fgets(ProdID, 5,fs);
fgets(ProdName, 21, fs);
}
```
C

```
if(strcmp(ProdID, ptr->dat->ProdID)>0)
ptr=ptr->left;
else
if(strcmp(ProdID,ptr->dat->ProdID)>0)
ptr=ptr->right;
else
break;
if(ptr!=NULL)
{
fseek(fs,ptr->dat->offset-1,0)
fgets(ProdID, 5,fs);
fgets(ProdName, 21, fs);
}
```
D

Self Review Exercise 147.2

2. *The binary tree index node whose key value matches the key value of the record to be searched for is found. Identify the next step involved.*

   A. The offset value in the node is used t find the record from the database file.
   B. The value of the other fields are read from the binary tree index node.
   C. The key value of the node is used to search the other fields in the database file.
   D. The contents of the node are copied to the database file.

3. *You learned to declare a binary tree node that can be used to create a binary tree index. Four code samples are given below. Choose the correct code sample used to create a binary tree node index.*

```
#include<stdio.h>
#include<stdlib>
Struct productdata
{
char productID[5];
Struct TreeNode *left;
struct TreeNode *right;
};
```
A

```
#include<stdio.h>
#include<stdlib>
Struct productdata
{
char productID[5];
long offset;
};
struct TreeNode
{
Struct productdata *dat;
struct TreeNode *next;
};
```
B

```
#include<stdio.h>
#include<stdlib>
Struct productdata
{
char productID[5];
long offset;
};
struct TreeNode
{
Struct productdata *dat;
struct TreeNode *left;
};
```
C

```
#include<stdio.h>
#include<stdlib>
Struct productdata
{
char productID[5];
long offset;
};
struct TreeNode
{
Struct productdata *dat;
struct TreeNode *left;
struct TreeNode *right;
};
```
D

1 .Choose the code that searches for an `item` *number in a binary tree.*

A.
```
struct TTag *ts(int c, struct TTag *ptr)
{
while (ptr!=NULL)
{
if(strcmp(c,ptr-> Item)=0)
ptr=ptr->right;
else
if(strcmp(c, ptr->Item)<0)
ptr=ptr-> left;
else
return Ptr ;
}
return Null;
}
```

B.
```
struct TTag *ts(int c, struct TTag *ptr)
{
while (ptr==NULL)
{
if(strcmp(c,ptr-> Item)>0)
ptr=ptr-> left;
else
if(strcmp(c, ptr->Item)<0)
ptr=ptr-> right;
else
return Ptr ;
}
return Null;
}
```

C.
```
struct TTag *ts(int c, struct TTag *ptr)
{
while (ptr!=NULL)
{
if(strcmp(c,ptr-> Item)>0)
ptr=ptr->left;
else
if(strcmp(c, ptr->Item)<0)
ptr=ptr-> right;
else
return Ptr ;
}
return Null;
}
```

D.
```
struct TTag *ts(int c, struct TTag *ptr)
{
while (ptr!=NULL)
{
if(strcmp(c,ptr-> Item)>0)
ptr=ptr->right;
else
if(strcmp(c, ptr->Item)<0)
ptr=ptr-> left;
else
return Ptr ;
}
return Null;
}
```

2. Identify the code used to read the nodes in a binary tree index. Partial code used to read the records in a binary tree index is given. Choose the appropriate partial code required to complete this code.

A.
```
if(strcmp(keyval, ptr->dat->EmpID)<0)
ptr=ptr->right;
else
if(strcmp(keyval, ptr->dat->EmpID)>0)
ptr=ptr->left;
else
break;
if(ptr!=NULL)
{
fseek(fd,ptr->dat->offset-1,0);
fgets(EmpID,5,fd);
fgets(EmpName, 21, fd);
```

B.
```
if(strcmp(ptr, ptr->dat->EmpID)<0)
ptr=ptr->left;
else
if(strcmp(ptr, ptr->dat->EmpID)>0)
ptr=ptr->right;
else
break;
if(ptr!=NULL)
{
fseek(fd,ptr->dat->offset-1,0);
fgets(EmpID,5,fd);
fgets(EmpName, 21, fd);
```

C.
```
if(strcmp(keyval, ptr->dat->offset)<0)
ptr=ptr->left;
else
if(strcmp(keyval, ptr->dat->EmpID)>0)
ptr=ptr->right;
else
break;
if(ptr!=NULL)
{
fseek(fd,ptr->dat->EmpID-1,0);
fgets(EmpID,5,fd);
fgets(EmpName, 21, fd);
}
```

D.
```
if(strcmp(keyval, ptr->dat->EmpID)<0)
ptr=ptr->left;
else
if(strcmp(keyval, ptr->dat->EmpID)>0)
ptr=ptr->right;
else
break;
if(ptr!=NULL)
{
fseek(fd,ptr->dat->offset-1,0);
fgets(EmpID,5,fd);
fgets(EmpName, 21, fd);
}
```

REVIEW EXERCISE

3. Identify the code that is used to define a binary tree. Tick the appropriate code.

A.
```
#include<stdlib.h>
struct TreeTag
{
char ItemNo[SIZE_OF_ITEMNO);
struct TreeTag *right;
};
struct TreeTag *root;
```

B.
```
#include<stdlib.h>
struct TreeTag
{

char ItemNo[SIZE_OF_ITEMNO);
struct TreeTag *left;
struct TreeTag *right;
};
struct TreeTag *root;
```

C.
```
#include<stdlib.h>
struct TreeTag
{

char ItemNo[SIZE_OF_ITEMNO);
struct TreeTag *left;
};
struct TreeTag *root;
```

D.
```
#include<stdlib.h>
struct TreeTag
{

char ItemNo[SIZE_OF_ITEMNO);
};
struct TreeTag *leftroot;
```

4. Choose the code sample that will be used to insert a node in a binary tree structure,

A.
```
CInsert(struct TreeTag *ptr)
{
if(strcmp(NNode->INo,ptr->INo)>0)
if(ptr->right!=NULL)
CInsert(ptr->right);
else
ptr->right=NNode;
else
if(strcmp(NNode->INo,ptr->INo)<0)
if(ptr->left!=NULL)
CInsert(ptr->left);
else
ptr->left=NNode;
}
```

B.
```
CInsert(struct TreeTag *ptr)
{
if(strcmp(NNode->INo,ptr->INo)<0)
if(ptr->right!=NULL)
CInsert(ptr->right);
else
ptr->right=NNode;
else
if(strcmp(NNode->INo,ptr->INo)>0)
if(ptr->left!=NULL)
CInsert(ptr->left);
else
ptr->left=NNode;
}
```

C.
```
CInsert(struct TreeTag *ptr)
{
if(strcmp(NNode->INo,ptr->INo)>0)
if(ptr->right==NULL)
CInsert(ptr->right);
else
ptr->right=NNode;
else
if(strcmp(NNode->INo,ptr->INo)<0)
if(ptr->left!=NULL)
CInsert(ptr->left);
else
ptr->left=NNode;
}
```

D.
```
CInsert(struct TreeTag *ptr)
{
if(strcmp(NNode->INo,ptr->INo)>0)
if(ptr->right!=NULL)
ptr->right=NNode;
else
CInsert(ptr->right);
else
if(strcmp(NNode->INo,ptr->INo)<0)
if(ptr->left!=NULL)
ptr->left=NNode;
else
CInsert(ptr->left);
}
```

5. Match the data structures to store an index with their advantages.

A.	*Array*		*A.*	*It is an efficient method to insert and delete*
B.	*Linked list*		*B.*	*It helps you to search for the desired key value in the least..*
C.	*Binary Tree*		*C.*	*It is the simplest method of storing an index.*

6. Partial code used to write a linked list index that is located in the primary of an index file on the disk is given. Identify the code sample used to complete this code. Choose the appropriate code.

```
#include<stdio.h>
#include<stdlib.h>
struct ProductData
{
char ProductName[5];
long offset;
};
struct ProductList
{
struct ProductData *rec;
struct productList *next;
};
struct ProductList *strart;
FILE *fw;
struct ProductList *cur;
WreteIndex()
{
cur=start;
fw=fopen("index", "w");
if(start!=NULL)
{

}
else
{
printf("The list is empty.");
}
fclose(fw);
}
```

A.
```
while(cur!=NULL)
{
fprintf(fw,"%s  %d", cur->rec->ProductName.cur->next);
cur=cur->next;
}
```

B.
```
while(cur!=NULL)
{
fprintf(fw,"%s  %d", cur->rec->ProductName.cur->rec->offset);
}
```

C.
```
while(cur!=NULL)
{
fprintf(fw,"%s  %d", cur->rec->ProductName.cur-> rec->offset);
cur=cur->next;
}
```

D.
```
while(cur!=NULL)
{
fprintf(fw,"%s  %d", cur->rec, cur-> rec->offset);
cur=cur->next;
}
```

7. A linked list index is based on the structures given. The key file `ProdID` *of the underlying data file is* 4 *bytes long. The size of the rest of the records is* 19 *bytes. Tick the appropriate partial code used to complete the code to delete a node from the index.*

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
struct ProductData
{
char ProdID[5];
long offset;
};
struct ProductList
{
struct ProductData *rd;
struct ProductList *next;
};
struct ProductList *start;
struct ProductList *ptr, *prev;
FILE *fs;
DeleteNode()
{
char pd[5];
long os;
printf("Type product ID: ");
scanf("%s",pd);
if(start!=NULL)
{
for(ptr=start; (ptr!=NULL)&&strcmp(pd,ptr->rd->ProdID)>0;prev=ptr, ptr=ptr->next);
if(strcmp(pd, ptr->rd->ProdID)==0)
{
}
else
printf("Key not found");
}
else
printf("The list is empty");
}
```

A.
```
if(ptr==start)
{
start=ptr->next;
}
else
{
prev->next=ptr->next;
}
os=ptr->rd->offset;
fs=fopen("datafile.txt", "r+");
fseek(fs, os-1, SEEK_SET);
fprintf(fs, "D");
```

B.
```
if(ptr==start)
{
start=ptr->next;
}
else
{
prev->next=ptr->next;
}
os=ptr->rd->offset;
free(ptr);
fs=fopen("datafile.txt", "r+");
fseek(fs, os-1, SEEK_SET);
```

C.
```
if(ptr==start)
{
start=ptr->next;
}
else
{
prev->next=ptr->next;
}
os=ptr->rd->offset;
free(ptr);
fs=fopen("datafile.txt", "r+");
fseek(fs, os-1, SEEK_SET);
fprintf(fs, "D");
fclose(fs);
```

D.
```
if(ptr==start)
{
start=ptr->next;
}
else
{
prev->next=ptr;
}
os=ptr->rd->offset;
free(ptr);
fs=fopen("datafile.txt", "r+");
fseek(fs, os-1, SEEK_SET);
fprintf(fs, "D");
fclose(fs);
```

8. Match data access methods with their features. Select the letters and put them in matching answer box in front of the correct answer.

A. Random Access		A.	Sequential AccessEach record on the disk is read until the desired record is reached. Records are accessed using the simplest method.
B. Indexed Access		B.	The offset address of a desired record is used to retrieve the record. Records are retrieved in the last amount of time by using this method.
C. Sequential Access		C.	The key field value of the desired record is used to access the record. It enables you to access records in a data file that has records of variables sizes.

9. Four code samples are given. Choose the code to traverse a binary tree based on the `inorder` method.

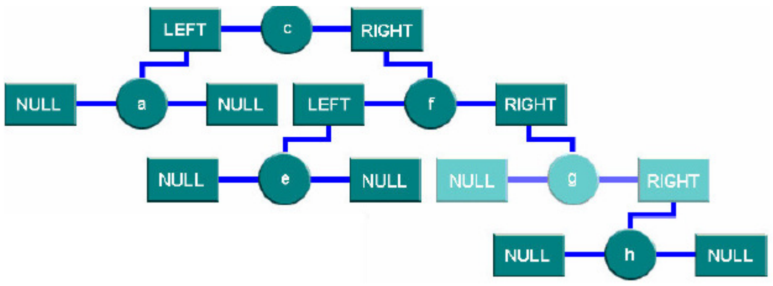
A.
```
inorder(struct TreeTag *ptr)
{
if(ptr==NULL)
return;
else
{
printf("\n Item number %s\n", ptr->INo);
inorder(ptr->left);
inorder(ptr->right);
}
}
```

B.
```
inorder(struct TreeTag *ptr)
{
if(ptr==NULL)
return;
else
{
inorder(ptr->left);
printf("\n Item number %s\n", ptr->INo);
inorder(ptr->right);
}
}
```

C.
```
inorder(struct TreeTag *ptr)
{
if(ptr==NULL)
return;
else
{
inorder(ptr->left);
inorder(ptr->right);
printf("\n Item number %s\n", ptr->INo);
}
}
```

D.
```
inorder(struct TreeTag *ptr)
{
if(ptr==NULL)
return;
else
{
inorder(ptr->right);
inorder(ptr->left);
printf("\n Item number %s\n", ptr->INo);
}
}
```

10. A diagrammatic representation of binary tree is given below. The node g is to be deleted from the binary tree structure. Choose the actively that should be performed to delete the node.



A. The right pointer of the parent node must be changed to NULL.

B. The entire left subtree of the node being deleted should be linked under the last left terminal node of the right subtree.

C. The root pointer must be updated so that the rest of the tree is accessible.

D. The left subtree or the right subtree of the node being deleted should be linked to the parent node of the node being deleted

11. Partial code used to insert a new node in a linked list index is given. Tick the correct code sample used to assign the key and offset values to the new node and write the new record at the end of the underlying data file.

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
struct CustData
{
char CustID[5];
long offset;
};
struct custList
{
struct CustData *rec;
struct CustList *next;
};
struct CustList *start, *nd;
FILE *fp;
MakeNode()
{
nd=(struct CustList *)nalloc(sizeof(struct CustList));
nd->rec=(struct CustData *)malloc(sizeof(struct CustData));
}
GetNewNode()
{
char CustID[5],CustName[13];
printf("Type the customer ID: ");
scanf("%4s", CustID);
printf("Type the customer name: ");
scanf("%12s",CustName);
fp=fopen("datafile.txt","r+");
fseek(fp,0,SEEK_END);
fclose(fp);
}
InsertNode
{
MakeNode();
GetNewNode();
```

A.
```
nd->rec->offset=ftell(fp)+1;
strcpy(nd->rec->CustID,CustID);
fprintf(fp,"U%4s%12s\n",CustID,CustName);
```

B.
```
nd->next=ftell(fp)+1;
strcpy(nd->rec->CustID,CustID);
fprintf(fp,"U%4s%12s\n",CustID,CustName);
```

C.
```
nd->rec->offset=ftell(fp)+1;
strcpy(nd->rec->CustID,CustID);
fprintf("datafile.txt","U%4s%12s\n",CustID,CustName);
```

D.
```
nd->rec->offset=ftell(fp)+1;
strcpy(nd->rec->offset,CustID);
fprintf(fp,"U%4s%12s\n",CustID,CustName);
```

12. A linked list index is based on the structures `CustData` *and* `CustName`*. The linked list node is created by allocating memory and assigning data. Tick the code sample used to insert nodes at a position sorted in an ascending order or key values in the linked index*

A.
```
if(start==NULL)
start=nd;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
    (nd->rec->CustID,ptr->rec->CustID)>0;
     ptr=ptr->next);
if(ptr==start)
{
nd->next=start;
start=nd;
}
else
{
prev->next=nd;
nd->next=ptr;
}
}
```

B.
```
if(start==NULL)
start=nd;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
    (nd->rec->CustID,ptr->rec->CustID)>0;
     ptr=ptr->next);
if(ptr==start)
{
nd->next=start;
start=NULL;
}
else
{
prev->next=nd;
nd->next=ptr;
}
}
```

C.
```
if(start==NULL)
start=nd;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
    (nd->rec->CustID,ptr->rec->CustID)0;
     ptev=ptr,ptr=ptr->next);
if(ptr==start)
{
nd->next=start;
start=nd;
}
else
{
prev->next=nd;
nd->next=prev;
}
}
```

D.
```
if(start==NULL)
start=nd;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
    (nd->rec->CustID,ptr->rec->CustID)>0;
    prev=ptr,ptr=ptr->next);
if(ptr==start)
{
nd->next=start;
start=nd;
}
else
{
prev->next=nd;
nd->next=ptr;
}
}
```

13. Create a linked list index for the database file in which the key field EmpID is a string of five characters. The size of the remaining data in the record is 19 bytes. Tick the code sample used to declare the structures required to create the index.

A.
```
struct EmpData
{
char EmpID[6];
long offset;
};
struct EmpList
{
struct EmpData *er;
struct EmpList *next;
};
```

B.
```
struct EmpData
{
char EmpID[6];
char EmpName[14];
};
struct EmpList
{
struct EmpData *er;
struct EmpList *next;
};
```

C.
```
struct EmpData
{
char EmpID[6];
long offset;
};
struct EmpList
{
struct EmpData *er;
struct EmpData *next;
};
```

D.
```
struct EmpData
{
char EmpID[6];
long offset;
};
struct EmpList
{
long er;
struct EmpList *next;
};
```

REVIEW EXERCISE

14. Next, select the code sample used to allocate the memory for a node of the linked list. Tick the correct code sample.

A.
```
void MakeNode()
{
newnode=(struct EmpList *) malloc(sizeof(struct EmpList));
newnode=(struct EmpData *) malloc(sizeof(struct EmpData));
}
```

B.
```
void MakeNode()
{
newnode=(struct EmpList *) malloc(sizeof(struct EmpList));
}
```

C.
```
void MakeNode()
{
newnode=(struct EmpList *) malloc(sizeof(struct EmpList));
newnode->er-(struct EmpData *) malloc(sizeof(struct EmpData));
}
```

D.
```
void MakeNode()
{
newnode->er-(struct EmpData *) malloc(sizeof(struct EmpData));
```

15. After allocating memory, read the record from the database file and assign the key and offset values of the database record to the node in the linked list. The file pointer `fs` *that points to the database file is declared for you. Tick the correct code sample.*

A.
```
getdata(0
{
char fkey[6],rest[19];
newnode->er->offset=ftell(fs);
fgets(fkey,6,fs);
strcpy(newnode->er->EmpID,fkey);
newnode->next=NULL;
}
```

B.
```
getdata(0
{
char fkey[6],rest[19];
fgets(fkey,6,fs);
fgets(rest,19,fs);
strcpy(newnode->er->EmpID,fkey);
newnode->next=NULL;
}
```

C.
```
getdata(0
{
char fkey[6],rest[19];
newnode->next=ftell(fs);
fgets(fkey,6,fs);
fgets(rest,19,fs);
strcpy(newnode->er->EmpID,fkey);
newnode->next=NULL;
}
```

D.
```
getdata(0
{
char fkey[6],rest[19];
newnode->er->offset=ftell(fs);
fgets(fkey,6,fs);
fgets(rest,19,fs);
strcpy(newnode->er->EmpID,fkey);
newnode->next=NULL;
}
```

16. Finally, select the code sample used to insert the node in the linked list that is sorted in the ascending order of the key values. The pointers `prev`*,* `ptr`*, and* `start` *are declared, and the data file is open in the read mode. Tick the correct code sample.*

A.
```
if(start==NULL)
start=newnode;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
   (newnode->er->EmpID,ptr->er->EmpID)>0;
   ptr=ptr->next);
if(ptr==start)
{
newnode->next=start;
start=newnode;
}
else
{
prev->next=newnode;
newnode->next=ptr;
}
}
```

B.
```
if(start==NULL)
start=newnode;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
   (newnode->er->EmpID,ptr->er->EmpID)>0;
    prev-ptr,ptr=ptr->next);
if(ptr==start)
{
newnode->next=start;
start=newnode;
}
else
{
prev->next=newnode;
newnode->next=ptr;
}
}
```

C.
```
if(start==NULL)
start=newnode;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
   (newnode->er->EmpID,ptr->er->EmpID)>0;
    prev=ptr,ptr=ptr->next);
if(ptr==start)
{
newnode->next=start;
}
else
{
prev->next=newnode;
newnode->next=ptr;
}
}
```

D.
```
if(start==NULL)
start=newnode;
else
{
for(ptr=start;(ptr!=NULL)&&strcmp
   (newnode->er->EmpID,ptr->er->EmpID)>0;
    prev=ptr,ptr=ptr->next);
if(ptr==start)
{
newnode->next=start;
start=newnode;
}
else
{
newnode->next=ptr;
}
}
```

LESSON 148 : LINKER --> ROLES

UNIT 39 : CONCEPTS OF ANSI C LIBRARY *STUDY AREA : ANSI C LIBRARY*

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

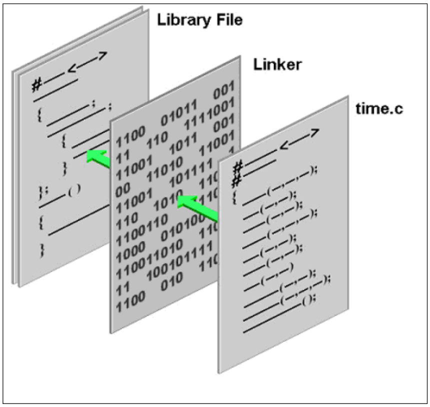
In a business application environment, the modules developed by several programmers need to be integrated together to build a large application. Before integration, the modules are compiled separately. The successful compilation of each module creates an object file for the compiled code.

An object file contains references to the functions present in library files. Therefore, the object file has to be linked to the library files that contain the functions referred to in the program.

To link object files to library files, you use a linker. Another role of the linker is to resolve external references. When a function is called from outside a source code file, it is called an external reference.

A linker resolves the external reference by using the address of the referred function in place of the call to that function.

Consider the example below. In this example, the `time.c` program contains the display function that displays the current system time.



When the display function is called from another source file in the program the function is resolved using the linker. It is not necessary to write the code for resolving the address of the function separately.

Self Review Exercise 148.1

1. *Identify the role of the linker.*

A. It names program files.
B. It compiles sources files.
C. It resolves external references.
D. It generates two object code files

LESSON 149 : LIBRARY FILES VS OBJECT FILES

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

In a programming environment, a function used in a program may be required in other programs also. To make all functions accessible to programs, the C language provides two types of compiled files, library files and object files.

Library files and object files differ in their content and functionality. The contents of a library file include functions while an object file contains references to the functions contained in a library file. A library file includes the name, code, and the location of a function. Unlike a library file, an object file stores only a reference to a function.

Another difference between library files and object files is that when a library file is used in a program, only the functions required for the program are included during program compilation. However, all the functions in an object file are included in a program.

For example, when the library file `main.lib` is used in the `myprog` program, only the function that is called is included in the program. This saves memory. If all the functions are included in a program, the size of the program file increases. This increases the amount of memory used by the program file.

This lesson covered the differences between library files and object files. These files provide functions that can be used by all other programs.

Self Review Exercise 149.1

1. *Identify a different between library files and object files.*

   A. Library files contain the declarations of subprocedures while object files contain the definitions of the prototype of function.
   B. Library files contain standard library functions while object files contain user-defined functions and references to library functions.
   C. Library files increase the amount of space used by program files unlike object files.
   D. All the files from an object file are included in a program during its compilation while only the relevant functions from a library file are included in a program.

LESSON 150 : HEADER FILES --> PURPOSES

Lesson Objectives
In this Lesson you will learn : ☛ *the Genesis of C Language* ☛ *the Stages in the Evolution of C Language*

Header files are the files that contain the prototypes of functions, the definitions of structures, external variables, and macros. To use certain functions and macros in a program, you should include the header file that contains these functions and macros in the program.

ANSI C supports header files. Some of the header files supported by ANSI C are assert.h, math.h, stdarg.h, stdio.h, stdlib.h, string.h, and time.h. In most compilers, header files have the extension .h.

The assert.h header file provides diagnostics and debugging utilities for the source code. These utilities help to determine whether or not a program is operating correctly.

The math.h header file is another ANSI C standard header file. The math.h header file provides functions that perform mathematical operations on the float, double, and long double data types.

Header Files	*Explanation*
`assert.h`	▪ Provides Diagnostics and debugging utilities for the source code

`math.h`	▪ Provides functions that perform mathematical operations on data
	▪ Many mathematical functions, such as log, sqrt, sinh, and exp, are defined in the math.h header file. To use these functions in a program, the math.h header file should be included in the program. If the math.h header file is not included, the program will generate an error.
	`float` | `log`
	`double` | `sqrt`
	`long double` | `sinh`
	| `exp`

`stdarg.h`	▪ Provides support for variable length argument lists
	▪ The stdarg.h header file supports macros such as va_start, va_arg, and va_end. To use any of these macros, the stdarg header file should be included in the program.
	▪ The stdarg.h header file supports macros such as va_start, va_arg, and va_end. To use any of these macros, the stdarg header file should be included in the program.
	▪ The functions va_start, va_arg, and va_end are used when the number of arguments accepted by a function in the program is not defined or fixed.
	`va_start`
	`va_arg`
	`va_end`

LESSON 150 : HEADER FILES --> PURPOSES

Header Files	*Explanation*

`stdio.h`	• Provides functions for file and stream input and output operations
	• Functions such as getchar, putchar, gets, printf, and puts we provided by the stdio.h header file. To use these input and output functions, include the stdio.h header file in the program.
	`getchar`
	`putchar`
	`getc`
	`printf`
	`putc`

`stdlib.h`	• Contains the prototypes of utility functions • The stdlib.h header file contains the prototypes of utility functions, such as the routines used for string conversion, memory allocation, and random number generation.

`string.h`	• provides run-time functions for all string and memory management
	• The strcpy, strcat, and strchr functions are some of the string-related functions provided by the string.h header file. The memory management functions provided by the string.h header file are memchr, memcpy, and memmove.
	`strcpy`
	`strcat`
	`strchr`
	`memchr`
	`memcpy`
	`memmove`

`time.h`	• Provides functions to access and manipulate the system date and time
	• Some of the functions provided by the time.h header file are ctime, gmtime, localtime, and time.
	• ANSI C supports additional header files that can be used in programs.
	`ctime`
	`gmtime`
	`localtime`
	`time`

`ctype.h`	• Provides functions that process characters
`errno.h`	• Provides files that contain error values
`float.h`	• Defines the implementation of floating-type values • It allows you to write a program by using restricted constant expressions to make the program portable.
`limits.h`	• Defines the implementation of the integral data type • For example, the data type CHAR-BIT in the limits header file defines the number of bits in a character.
`locale.h`	• Accesses and modifies the current locale information, such as the formats of numbers and the currency for a specific country.

Header Files	*Explanation*
`setjmp.h`	▪ Handles the transfer of control from a calling function to another function ▪ It allows unconditional branching between functions by saving the return state of the computer before branching occurs.
`signal.h`	▪ provides functions to handle interrupts
`stddef.h`	▪ Provides the definitions for commonly used types and macros

Self Review Exercise 150.1

1. *Identify the purpose of the stdarg.h header file.*

   A. Provides mathematical functions
   B. Porvides functions that support variable length argument lists
   C. Provide error-handling functions
   D. Provides file stream input and output functions

2. *Identify the header file that contains the prototypes of utility functions.*

   A. `stdlib.h`
   B. `stdio.h`
   C. `stdarg.h`
   D. `assert.h`

LESSON 151 : ANSI C HEADER FILES --> `stdarg.h`

Lesson Objectives
In this Lesson you will learn : *the Genesis of C Language* *the Stages in the Evolution of C Language*

The C library provides the standard header file `stdarg.h`. This header file enables a programmer to define functions that have a variable number of arguments.

The `stdarg.h` header file contains one `typedef` statement and three macros that support variable length argument lists. The three macros contained in the header file are `va_start`, `va_arg`, and `va_end`.

Macros in the `stdarg.h` *Header File*
`va_start`
`va_arg`
`va_end`

The implementation of the macros in the `stdarg.h` header file is system dependent. The prototypes of these macros are given.

Prototypes of the Macros in the `stdarg.h` *Header File*
`void va_start(va_list ap, lastfix);`
`type va_arg(va_list ap, type);`
`void va_end(va_list ap);`

- The first macro `va_start` provides the setup for the retrieval of arguments from a function. In the prototype, the `va_start` macro contains pointer `ap` and the variable `lastfix`.
- The `ap` pointer is declared as a local variable of the `va_list` data type and points to the first argument in an argument list. This pointer is used to step through each argument in the argument list of a function that accepts a variable number of arguments.
- The `va_start` macro initializes the argument pointer ap. The argument pointer `ap` contains the stack memory address of the first argument after `va_start` macro is called.
- In the macro `va_start`, the variable `lastfix` is the name of the last fixed variable of an integer type. The `lastfix` variable holds the addresses of each subsequent variable argument when the macro `va_arg` is invoked.
- The next macro is the `va_arg` macro. The parameters in this macro are an argument pointer and its type, `va_list`. In the example displayed, the parameters of the macro are represented by `ap` and type, respectively.
- The argument pointer is used to access each argument in the argument list.

- The type parameter specifies the data type of the next argument.
- The last macro, `va_end`, performs the cleanup operations, such as freeing memory, before the function terminates. The `va_end` macro reinitializes the pointer `ap` if it is necessary to reprocess the variable arguments on the stack.
- The `va_end` macro must be called after all the arguments are processed but before the function that uses arguments terminates.

Self Review Exercise 118.1

1. You learned about the macros in the `stdarg.h` *header file. Choose the macro name that performs cleanup operations now.*

A. `va_end`
B. `Va_start`
C. `Va-arg`

LESSON 152 : ANSI C HEADER FILES --> `stdio.h`

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

In ANSI C, the stdio.h header file contains the declarations of the file operation functions. It also declares the standard input/output (I/O) functions that handle the flow of information between memory and standard I/O devices.

The stdio.h header file contains macros, type definitions, and the prototypes of the functions that are used to perform file operations. Various actions, such as opening and closing a file, can be performed using the functions in the stdio.h header file. The functions that are used for opening and closing a file are `fopen` and `fclose`, respectively.

The stdio.h header file also contains the prototypes of the getc, `fgetc`, and `fgets` functions, which are used to read character input. It also contains the prototypes of the functions `fputc` and `ungetc`, which are used to write character output.

The prototypes of the functions used to read and write formatted input and output are also provided in the stdio.h header file. These functions are `fscanf`, `scanf`, `sscanf`, `fprintf`, `printf`, `sprintf`, and `vprintf`.

Error-handling functions are also provided in the stdio.h header file. The error-handling functions are `clearer`, `feof`, `ferror`, and `perror`.

The error-handling functions are used to test and clear the error and end-of-file indicators in a program.

Another set of functions in the stdio.h header file includes the functions that are used to perform direct input and output operations on binary files. The functions `fread` and `fwrite` are used to read from and write to binary files, respectively.

The stdio.h header also contains the functions remove and rename. The remove function is used to remove a file from the file system. The rename function in the stdio.h header file is used to rename a file is used to rename a file.

Functions in the stdio.h Header File	
`fopen` *and* `fcose`	*opening and closing a file*
`fgetc, fgetc,` *and* `fgets`	*Reading character input*
`fputc` *and* `ungetc`	*Writing character output*
`fscanf, scanf, sscanf, fprintf, printf, sprintf,` *and* `vprintf`	*Reading and writing formatted input an output*
`clearer, feof, ferror,` *and* `perror`	*Testing and clearing the error and end. of .file indicators*
`fread and fwrite`	*Reading and writing to binary files*
`remove` *and* `rename`	*Removing and renaming files*

Self Review Exercise 152.1

1. *Identify the set of functions used to read formatted input.*

   A. `fopen and fclose`
   B. `feof and clearer`
   C. `fread and fwrite`
   D. `fscanf, scanf, and sscanf`

LESSON 153 : ANSI C HEADER FILES --> `stdlib.h`

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The stdlib.h header file provides functions for searching and sorting data, memory allocation, and number conversions.

Some of the functions included in the stdlib.h header file are bsearch, qsort, getenv, rand, system, abort, atexit, and exit.

Functions in the `stdlib.h` ***header File***

- `bsearch`
- `qsort`
- `getenv`
- `rand`
- `system`
- `abort`
- `atexit`
- `exit`

The purpose of the bsearch function is to search a sorted array by using a binary search method. The bsearch function uses the binary search method to search for an element in a sorted array. The element that is searched for is pointed to by a pointer. If a match for the pointer is found, the address of the element in the sorted array is returned. Otherwise, the value NULL is returned. The qsort function is also declared in the stdlib.h header file. This function is used to sort an array in ascending order. The array is returned to by a pointer declared in the qsort function.

The getenv function is another function in the stdlib.h header. It is used to search a list of environment variables provided by the operating system.

Another function included in the stdlib.h header file is rand. This function is used to generate and return a random number. Repeated calls to the rand function generate a randomly distributed sequence of numbers.

The system function in the stdlib.h header file passes a string containing a command to be processed by the command interpreter shell of the operating system.

The abort function in the stdlib.h header file is used to terminate a program abnormally.

The atexit function in the stdlib.h header file allows you to register a function that should be processed when a program terminates normally.

The execution of registered functions is carried out in the reverse order or their registration. This implies that the function that is registered last is processed first. These functions can use only global variables.

The exit function is another function in the stdlib.h header file. This function causes a normal termination of a program.

When the exit function is invoked, the functions registered by the atexit function are invoked, buffered streams are flushed, files are closed, and temporary files are removed.

In this lesson, you learned about the functions in the stdlib.h header file. This header file contains the functions that can be used to search and sort data and perform string conversions.

Self Review Exercise 153.1

1. *Identify the function in the stdlib.h header file, which is used to search a list of environment variable provided by the operating system.*

   A. `rand`
   B. `getenv`
   C. `bsearch`
   D. `Qsort`

2. *Identify the function that is used to register a program that should be proceed when a program terminates normally.*

   A. `atexit`
   B. `exit`
   C. `abort`
   D. `getenv`

LESSON 154 : ANSI C HEADER FILES --> `time.h`

Lesson Objectives

In this Lesson you will learn :

- *the Genesis of C Language*
- *the Stages in the Evolution of C Language*

The `time.h` header file of the ANSI C library provides the functions for accessing the system time and date.

To access the system clock, the `time.h` header file provides the clock function. The syntax for the clock function is displayed. In the syntax, `clock_t` is an integer type representing time. This function returns the number of system clock ticks since program execution.

The clock function provides access to the underlying machine clock on most operating systems.

```
clock_t clock(void)
```

In ANSI C, the value of time is returned by various functions either as calendar time or as broken-down time. Calendar time is expressed as an integer.

The integer indicating the calendar time represents the number of second that have elapsed since the midnight of January 1, 1970. the calendar time is encode according to Universal Time Coordinated (UTC).

The other version of time in ANSI C is the broken-down time that is expressed as a structure of the type struct tm. The members of the structure are the integers `tm_sec`, `tm_min`, `tm_hour`, `tm_mday`, `tm_mon`, `tm_year`, `tm_wday`, `tm_yday`, and `tm_isdst`.

```
struct tm{
int tm_sec;
int tm_min;
int m_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

The structure members `tm_sec`, `tm_min`, and `tm_hour` represent the time stored in the structure struct tm in seconds, minutes, and hours, respectively. The date stored in the structure is represented by `tm_mday`, `tm_mon`, `tm_year`, `tm_wday`, and `tm_yday`. These represent the day of the month, the month, the year since 1900, the day of the week, and the day of the year since January 1, 1970 respectively. The structure member `tm_isdst` indicates whether the structure represents the standard time or the daylight saving time.

***Functions in* `time.h`**

Various functions are available in the `time.h` header file to access time.

Functions in `time.h`	*Explanation*
`time`	▪ This function returns the current calendar time expressed as the number of seconds that have passed since January 1, 1970. ▪ If a time value is not returned, the time function returns the integer value -1. In the time function, a pointer to the system time is assigned. If the pointer is not NULL, the return value of the time function is assigned to this pointer.
`asctime`	▪ Converts the broken.down time into a string ▪ Another function in the `time.h` header file is the `asctime` function. The syntax representing the `asctime` function is given. The `asctime` function converts broken-down time into a string. In the syntax given, the broken-down time represented by pointer tp is converted into a string. ▪ Another function in the `time.h` header file is the `asctime` function. The syntax representing the `asctime` function is given. The `asctime` function converts broken-down time into a string. In the syntax given, the broken-down time represented by pointer tp is converted into a string. ▪ The `asctime` function also returns the base address of the string representing the broken-down time. `Char *asctime(struct tm *tp);`
`ctime`	▪ The `ctime` function is another function in the `time.h` header file. The syntax of this function is given. ▪ The `ctime` function converts a calendar time pointer into a string provided by the system. In the syntax displayed on the screen, the calendar time pointer, `t_ptr,` is converted to a string by the `ctime` function. ▪ In addition to converting the calendar time into a string, the `ctime` function returns the base address of this string. `Char *ctime(constant time_t *t_ptr);`
`difftime`	▪ Another function in the `time.h` header file is the `difftime` function. The `difftime` function computes the difference between two time values, such as t0 and t1, where t1 is greater than or equal to t0 and the time interval is system-dependent. `double difftime(time_t t0, time_t t1);`
`gmtime`	▪ The function `gmtime` is another function declared in the `time.h` header file. The syntax of the `gmtime` function is given below. ▪ The `gmtime` function converts the calendar time pointed to by a pointer, `t_ptr`, into a broken-down time and stores it in a structure of the type struct tm. ▪ The address of the structure struct tm is also returned by the `gmtime` function. The broken -down time is computed with reference to UTC. `struct tm*gmtime(const time_t*t_ptr);`
`localtime`	▪ The `localtime` function is another `time.h` header file function. The syntax of this function is given. ▪ The locatime function converts the calendar time pointed to by a pointer into a broken-down local time and stores it in a structure of the type, struct tm, as given. The address of the structure is also returned. `struct tm *localtime(const time_t*t_ptr);`

mktime	▪ Another function of the time.h header file is the mktime function. The syntax of this function is given. ▪ In the syntax , the mktime function converts the broken-down local time in the structure pointed to by the pointer tp into the corresponding calendar time. ▪ If the conversion is successful, the calendar time is returned by the mktime function. Otherwise, the integer value -1 is returned. `Time_t mktime(struct tm*tp);`
strftime	▪ The strftime function is another time.h header file function. The syntax of the strftime function is. ▪ The strftime function formats the date and time as a string by using a conversion rule system similar to the printf function. The formatted structure is determined by the string that is passed as a parameter to the strftime function. ▪ The value of the broken-down time is written to the string pointed to by pointer s. if the number of characters to be written in the string is more than n, the strftime function returns zero. Otherwise, it returns the length of the string pointed to by pointer s. `Size_t strftime(char *s, size_tn, const char *const_str, const struct tm *tp);`

Self Review Exercise 154.1

1. *Identify the purpose of the clock function.*

   A. Returns the number of system clock ticks since program execution
   B. Returning the broken-down time
   C. Returns the number of seconds that have elapsed since January 1, 1970
   D. Returns a string type variable that represents time.

2. *Identify the purpose of the asctime function.*

   A. Returns the processor time used since the beginning of the exection of the p.
   B. Converts the broken-down time into a string provided by the system.
   C. Converts the difference between the two time values into the local time.
   D. Returns the calendar time that is represented as the local time.

3. *Identify the purpose of the mktime function.*

   A. Formats the date and time as a string by using rule system similar to the printf function.
   B. converts the broken-down local time pointer to by the pointer tp into the corresponding calendar time.
   C. Converts the calendar time pointed to by a pointer into a broken-down time and stores it in a structure of the type structure.
   D. Converts the calendar time into a broken-down local time.

REVIEW EXERCISE

1. Match the functions in the `time.h` *header file with their purposes. Place the label in front of the function name to the red box in front of the purpose statement.*

A.	mktime()		A.	Returns the current calendar time expressed as the number of seconds that have passed since January 1,1970(UTC)
B.	strftime()		B.	Converts the broken-down time into a string and returns the base address of the string
C.	localtime()		C.	Converts the calendar time pointer into a string and returns the base address of the string
D.	difftime()		D.	Returns the difference between time 1and time 2 where time 2 must be greater than or equal to time 1 and the time interval is system dependent.
E.	ctime()		E.	Converts the calendar the calendar time into a broken-down UTC time and stores it in a standard structure format.
F.	time()		F.	Converts the members of a tm date-time structure to a time_t value, which is the number of seconds passed since January 1, 1970
G.	asctime()		G.	Formats the date and time as string by using a conversion rule system similar to the printf function

2. The statements describing the features of library files and object files are given. Identify the statement that describes an object file.

A. It stores the procedures to install the C standard compiler.
B. It stores the compiler information.
C. It stores the relocation information and object code of a function.
D. It stores the relocation information and object code of a function.
E. It stores only the references to functions.

3. Match the purposes of the functions in the `stdlib.h` *header file with the functions.*

A.	qsort()		A.	Performing a binary search on a sorted array
B.	bsearch()		B.	Generating and returning a random integer
C.	rand()		C.	Searching a list of the environment variable of the operating system
D.	exit()		D.	Terminating a program normally
E.	atexit()		E.	Passing a string as a command to be processed the shell
F.	Getenv()		F.	Registering a function that is processed when the program quits
G.	System()		G.	Arranging the array declared in the function in ascending order

4. The function of the `va_end` *macro in the* `stdarg.h` *header file is to:*

A. clean up the list.
B. accept multiple arguments
C. access the next argument in the list.
D. initialize the argument pointer.

5. What are the roles of a linker?

A. Linking code to run at an absolute location
B. Combining the modules into one program file
C. Generating the object code file.
D. Resolving external references

6. Match the purposes of the functions in the `stdio.h` *header file with the functions.*

A.	fread() and fwrite()		A.	Reading and writing formatted input and output
B.	getc(), gets(), fgetc(), fgets() fputc(), puts(), and ungetc()		B.	Testing and cleaing end.of.file and error indicators
C.	fscanf(), scanf(), sscanf(), fprintf(), prinf(), sprintf (), and vprintf()		C.	Reading and writing character input and output
D.	clearer(),feof(), ferror(), and perror()		D.	Opening
E.	remove() and rename()		E.	Deleting and renaming files
F.	fopen() and fclose()		F.	Performing direct input and output operations on binary files

7. Match the ANSI C header files with their purpose.

A.	stdlib.h		A.	Provides declaration of functions for number conversions
B.	math.h		B.	Provide support for variable length argument list
C.	stdio.h		C.	Provides functions that handle common mathematical operations
D.	string.h		D.	Provides run time functions for all string operations
E.	stdarg.h		E.	Provides functions to access and manipulate the system date
F.	time.h		F.	Provides diagnostics and debugging utilities for the source..
G.	assert.h		G.	Provides functions for file and stream input and output