UPES
UNIVERSITY OF TOMORROW
Centre for Continuing Education

Programming in C

Code : BC345F

UNIT 1

An Overview of C

Learning Objectives

At the end of this unit, you will be able to:

- Define variables, expressions, and assignment
- Discuss how we implement #define and #include
- Learn about Printf() and scanf()

Introduction

Dennis M. Ritchie created the general-purpose, high-level language C to create the UNIX operating system at Bell Laboratories. On the DEC PDP-11 computer, C was first introduced in 1972.

> The K&R standard, currently known as the first publicly accessible definition of C, was created in 1978 by Brian Kernighan and Dennis Ritchie.

The C programming language was used to create the UNIX operating system, the C compiler, and basically all UNIX application applications.

For a number of reasons, C is now a popular professional language:

- A Simple to learn
- B Structured language
- C It creates effective programmes
- D It is capable of handling simple tasks
- E It can be built on a number of different computer platforms

Why use C?

C was initially utilised for system development tasks, especially for the operating system's programmes. Because C produces code that executes almost as quickly as code written in assembly language,

it was accepted as a system development language.

Here are a few instances where C is used:

Programming and Preparation

The operating system, a collection of unique applications, is installed on the device. Operating systems like MS-DOS, OS/2, and UNIX are frequently accessible. An operating system controls the resources of the computer, offers user software, and serves as a conduit between the hardware and the user. The C compiler and several text editors are only a couple of the numerous software packages that the operating system offers. Vi is the name of the default text editor on UNIX systems.

The text editor and compiler are integrated in certain systems, such Borland C++. We presume that the reader is able to produce C code files using a text editor. These files are known as source files, and on the majority of UNIX systems, they are compiled using the cc command, which launches the C compiler. The compiler is called by the cc command, hence the name of the command also serves as the name of the compiler.

As a result, the terms C compiler and cc compiler are used synonymously. A compiler, roughly speaking, converts source code into executable object code. This compiled code is automatically created in a file called a.out on UND(systems. This compiled code is automatically generated on MS-DOS computers in a file with the same name as the.c file, but with the.exe extension in place of the.c extension.

Variables, Expressions, and Assignment

Let's create a program that translate the kilometers from miles and yards that make up a marathon. The length of a marathon is 26 miles and 385 yards in English units. Integers make up these numbers. We multiply by the conversion factor, 1.609, a real integer, to change miles to kilometers. Computers store integers differently than reals in memory. We divide by 1760.0 to convert yards to miles, and as we will see, it is important to represent this value as a real rather than an integer.

Our conversion program will include variables that can store both real and integer data, In C, the first line of code must declare (or name) the aU variables. An identifier, also known as a variable name, is made up of a string of letters, digits, and underscores but cannot begin with a digit. The selection of identifiers should take into account the program's intended use. They act as documentation in this way, improving software readability.

In file marathon.c

/*' The distance of a marathon in kilometers. */

```
#include <stdio.h>

int main(void)

{

int miles, yards;

float kilometers;

miles 26;

yards 385; ,

kilometers = 1.609 R * (miles + yards I 1760.0);

printf("\nA marathon is %f kilometers.\n\n", kilometers);

return 0;

}
```

Output:

A marathon is 42.185970 kilometers .

When a number has a decimal point, it is a floating-point constant as opposed to an integer constant. Therefore, a program would handle the integers 37 and 37.0 in various ways. Despite the fact that there are three forms of floating—float, double, and long double Floating constants are always of type double, while variables can be declared to be of any of these kinds.

Expressions are typically found as parameters to functions and on the right side of assignment operators. Simplest expressions include solely of constants, like 385 d 1760.0 from the preceding program. A meaningful combination of operators with variables and constants is also an expression, as is the name of a variable by itself.

Conversion rules may be incorporated into the evolution of expressions. This is a crucial idea. Any remainder is removed when two integers are divided, yielding an integer value. As an illustration, the phrase 7/2 has an int value of 3. On the other hand, 7.0/2 is a double divided by an int. The value of the equation 2.1S was immediately changed to a double when the expression 7.0/2 was evaluated, giving 7.0/2 the value 3.5.

kilometers 1.609 * (miles + yards / 1760.0);

is converted to

kilometer = 1.609 * (miles + yards / 1760);

This results in a software bug. Due to the fact that the variable yards is an int and has a value,

yard/1760

results in the integer value O when using integer division. Not what is desired is this. The error is fixed by using the double-type constant 1760.0.

#define and #include

The instructions for the preprocessor are #include and #define. The header files are included in a source code file using the #include command. Programs can declare macros or constants using the #define statement.

Introduction to Define and Include in C

Before the source code is built, the preprocessor in C processes commands known

as preprocessor directives. They do not end with a semicolon and start with the symbol "#." Preprocessor directives used frequently in C include :

#include	To include header files in a source code file, use this directive. The header file's location can be specified using either double quotes ("") or angle brackets (<>).
#define	With this pre-processor directive, a constant or macro can be defined in Cis. You can use it to give a constant value a name or to make a macro that can be used repeatedly throughout the code.
#ifdef, #ifndef, #endif	Conditions are compiled using these directives. Depending on whether a particular symbol is declared or not, they are used to include or omit specific sections of the code.
#pragma	To give the compiler implementation-specific choices, use this directive.
#error	By using this directive, a compile-time error with a specific error message can be produced.

There are other preprocessor directives in C besides these, which are some of the most used ones. Preprocessor directives are handled before the actual compilation, thus it is vital to remember that they are not a component of the C language itself but rather a preprocessor feature.

define in C

C macros are defined using the preprocessor command #define in a C program. Also known as the macros directive, #define The #define directive in a C program is used to declare constant values or expressions with names that can be reused frequently. The defined macros name replaces each time a #define C pre processor directive is encountered with a specific constant value or expression.

Syntax of #define in C

The following is the syntax for the C #define preprocessor directive:

#define CNAME expression

OR

#define CNAME value

CNAME: It is the expression's or constant value's name. Programmers frequently define it in capital letters.

expression: Any mathematical equation or piece of code is acceptable.

value: It can be an int, char, float, or other data type constant.

Example of #define keyword in C

```
#include <stdio.h>

// defines the PI value to be 3.14 in the whole program

#define PI 3.14

int main() {

    float radius, circumference;

    radius = 56.0;

     // PI will be replaced by 3.14 in the below statement

    circumference = 2 * PI * radius;

      printf("Circumference of Circle : %0.2f", circumference);

    return 0;

}
```

Output:

Circumference of Circle : 351.68

Explanation: We used #define in the code above to give "PI" the value 3.14. This will substitute the value 3.14 for "PI" in the program.

Uses of #define keyword in C

Uses of "#define" that are frequent include:

- **Defining constants**: A preprocessor macro can be given a constant value using the #define define C directive, and that constant value can subsequently be used in place of the value throughout the program. For instance, one could use a macro called PI, defined as #define PI 3.14159, instead of using the value 3.14159 throughout the program.
- **Creating custom macros**: The #define c directive allows the programmer to build unique macros that can be used to simplify difficult expressions or give values more meaningful names.
- **Conditional compilation**: If a macro is present, preprocessor directives like #ifdef, #ifndef, #if, #else, and #endif can be used to influence the compilation of specific code sections.
- **Creating shorthands for keywords**: With #define, a programmer can construct shorthands for terms or expressions that are often used in the code.
- **Creating platform-specific code**: The programmer can write platform-specific code for several operating systems or architectures with #define c.
- **Creating custom debug statements**: A single line of code can be changed to activate or disable specific debug statements that the programmer has created using the #define function.

include in C

The "include" command is used in C programming to include header files in a source code file. Function and variable declarations that can be utilised in the source code file are found in the header file. With the preprocessor directive "#include" and the header file's name enclosed in double quotes or angle brackets, the header file is included.

Syntax of #include in C

The following is the syntax for the C #include preprocessor directive:

#include

OR

#include "file_name"

File_name: You wish to add the name of a header file. The term "header file" refers to a C file with the common ".h" file extension that contains declarations and macro definitions that may be shared by several source files.

Example of #include in C

```
#include <stdio.h>

int main() {

    int a, b, product;

    // we can use printf() and scanf() function because

    // these functions are pre-defined in the stdio.h header file

    printf("Enter two numbers to find their sum : ");

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

    product = a*b;

    printf("Sum of %d and %d is : %d", a, b, product);

    return 0;

}
```

Output:

Enter two numbers to find their sum : Sum of 3 and 4 is : 12

Explanation:

The header file "stdio.h," which includes definitions of functions for taking input and creating output, has been included in the code above. Because we included the header file using the #include keyword, we can utilise the scanf() and printf() functions.

Uses of #include in C

Using "#include" frequently in C involves:

- **Including standard library headers**: Headers like "stdio.h" and "stdlib.h" include declarations for several of the standard C library functions, including "printf,""scanf," and "malloc." To make the functions they contain available to the remainder of the program, these headers are normally inserted at the start of a C source file.
- **Including user-defined header files**: The declarations of functions, variables, and other constructs that a programmer uses across numerous source files can be found in their own header files. Programmers can reuse declarations that are common across several source files and lessen code duplication by including these header files.
- **Including third-party headers**: In order to use the functionality offered by third-party libraries or frameworks in a C program, it is frequently essential to include the appropriate header files.
- **Conditional inclusion of headers**: In order to include or exclude header files depending on specific criteria, preprocessor directives like #ifdef, #ifndef, #if, #else, and #endif might be used.
- **Creating custom macros**: Custom macros can be built by the programmer using the #define directive and then added to source files using the #include directive.

Printf() and scanf()

Both the result and user input are displayed using the printf() and scanf() functions,

respectively. In the C programming language, the printf() and scanf() methods are often used. These functions are built-in library functions found in C programming header files.

printf() Function

For output, the C programming language uses the printf() function. The printf() method can accept any number of parameters. Each subsequent argument must be separated from the first by a comma (,) in double quotes. The first argument, "hello," must be enclosed in double quotations.

Important points about printf():

In the header file stdio.h, the printf() function is defined. This function allows us to print data or a message that we define on the monitor (also called the console). A variety of data formats can be printed using printf() on the output string.

We use "\n" in the printf() command to print on a new line on the screen.

C-based programming languages are case-sensitive. For example, printf() and scanf() are treated differently from Printf() and Scanf (). All characters must be lowercase for the built-in methods printf() and scanf() to function properly.

Syntax

```
printf("format specifier",argument_list);
```

%d (integer), %c (character), %s (string), %f (float), %lf (double), and %x (hexadecimal) variables can all be used in the format string for output.

Simple Example of printf() Function

```
#include<stdio.h>

int  main()

{

int num = 450;

// print number

printf("Number is %d \n", num);

return 0;

}
```

Output:

```
Number is 450
```

scanf() Function

Data supplied from the console is read using the scanf() method. The C library includes a built-in function called scanf(). The C language's scanf() function can read character, string, numeric, and other data from the keyboard. When given additional arguments, scanf() will receive formatted data from the user and assign it to the variables. Extra arguments must point to variables with the same datatype as the format of the user input.

Syntax

```
scanf("format specifier",argument_list);
```

Simple Example of scanf() Function

```
#include<stdio.h>

int main()

{
```

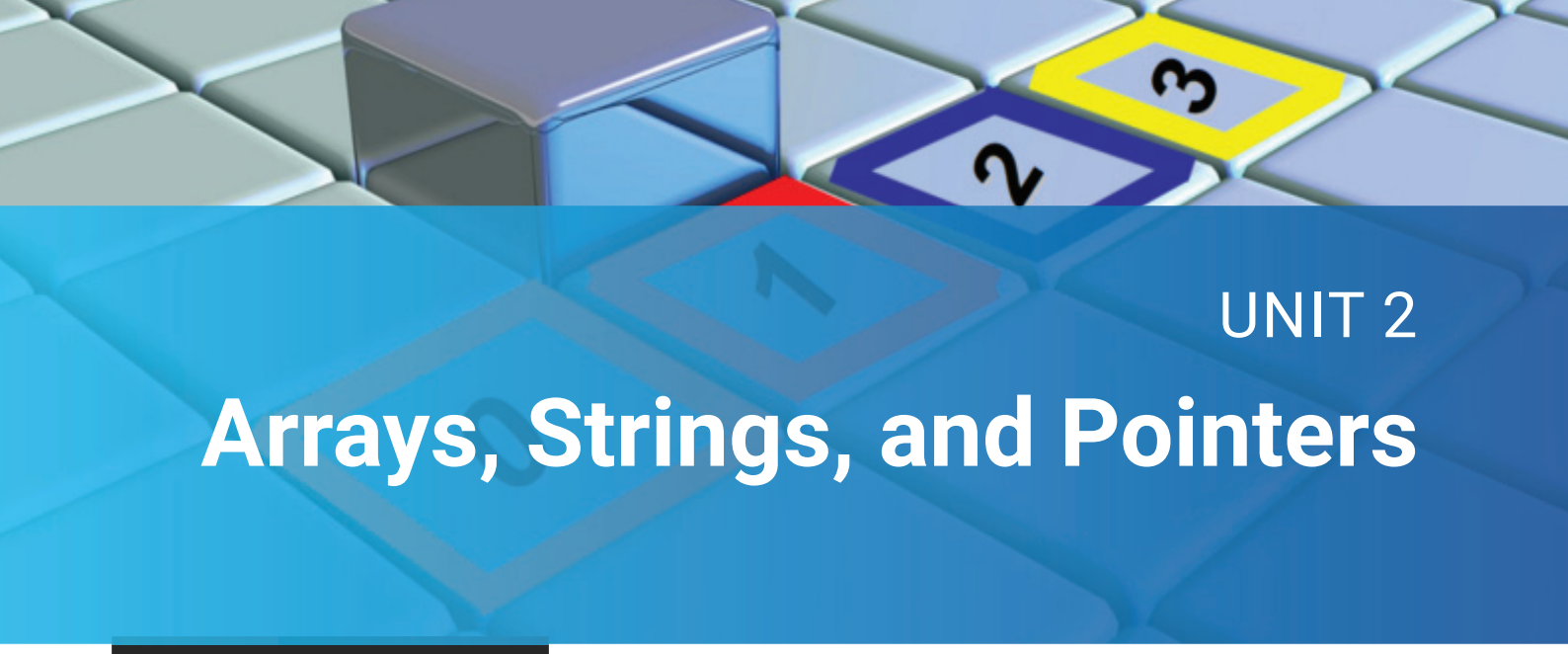
```
int x;

printf("enter the number =");

scanf("%d",&x);

printf("The number is=%d",x);

return 0;

}
```

Output:

```
enter the number =180

The number is=180
```

Summary

- C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972.
- C was initially used for system development work, particularly the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as the code written in assembly language.
- The evolution of expressions can involve conversion rules. This is an important point. The division of two integers results in an integer value, and any remainder is discarded.
- #include and #define are the preprocessor directives. #include is used to include the header files in a source code file. #define is used to define macros or constants in programs.
- In a C program, the preprocessor command #define is used to define C macros. #define also known as macros directive. In a C program, the #define directive is used to declare constant values or expressions with names that may be used repeatedly. Every time a #define C preprocessor directive is encountered, the defined macros name substitutes a specified constant value or expression in its place.



UNIT 2

Arrays, Strings, and Pointers

Learning Objectives

At the end of this unit, you will be able to:

- Recognize arrays and string handling
- Define pointers
- Discuss about operating system considerations

Introduction

An array name alone in C is a pointer, while a string is an array of characters. The ideas of arrays, strings, and pointers are hence closely related. A pointer is nothing more than an object's memory address. Unlike to most other languages, C supports pointer arithmetic. Pointer arithmetic is one of the language's strong features since it allows for the creation of extremely useful pointer expressions.

Arrays

When several variables of the same type are required, arrays are utilized.

For instance, the assertion

int a[3];

gives the three-element array a space to be used. The array's elements, which are of type *int*, are accessed using the symbols [0], [1], and [2]. An array's index, or subscript, always begins at O. The use of an array is demonstrated in the following application. Five scores are read into the computer, sorted, and then printed out in chronological sequence.

For Example: In C

#include<stdio.h>

#define CLASS_SIZE 5

int main (void)

```
{
int i, j, score[CLASS_SIZE]. sum = 0, tmp;
for (i = 0; i < CLASS_SIZE; ++i) {
scanf("%d", &score[i]);
sum += scorer;];
}
for (i 0; i < CLASS_SIZE - 1; ++i) /* bubble sort
*/
for (j = CLASS_SIZE - 1; j > i;--j)
if (score[j-1J < score[j]) { /* check the order */
tmp = score[j-l];
score[j-1] = score[jJ; score[j] = tmp;
}
printf("\nOrdered scores:\n\n");
for (; = 0; i < CLASS_SIZE; ++i)
printf(" score[%d] =%5d\n", i, score[i]);
printf("\n%18d%s\n%18.lf%s\n\n",
sum, " the sum of all the scores",
(double) sum / CLASS_SIZE, " the class
average");
return 0;
}
```

The results will appear on the screen if we run the application and enter the scores 63, 88, 97, 53, and 77 when prompted.

```
Input 5 Scores: 63  88  97  53  77
Ordered score:
Score[0]        =       97
Score[1]        =       88
Score[2]        =       77
Score[3]        =       63
Score[4]        =       53
378 is the sum of all scores
75.6 is the class average
```

The application sorts the scores using a bubble sort. This design is commonly carried out using nested for loops, with a test being made in the inner loop's body to determine whether a pair of elements are in the correct order. When elements are compared that are not in the proper sequence, their values are switched. Here, the code carries out this transaction.

```
mp = score[j-1];
score[j-1] = score[j];
score[j] =: tmp;
```

The variable tmp in the first sentence holds the value of score [j-l]. In the next sentence, the value of score [j] is used in place of the value of score [j-1] that was previously stored in memory. The final sentence substitutes the value of score [j] for the original value of score I, which is now stored in tmp.

By manually executing the program with the given data, the reader will be able to see why this bubble sort construct of two nested far loops results in an array containing sorted elements. With each iteration of the outer loop,

the necessary value among those that need to be processed is bubbled into position, giving bubble sort its name.

While bubble sorts are easy to program, they are not very efficient. Certain sorting techniques operate much more fast. Efficiency is not important when sorting a small number of items seldom, but it becomes important when the number of items is large or the code is used frequently. This word

(double) sum / CLASS_SIZE

which employs a cast operator and appears as an argument in the final printf() statement. The (double) sum function has the effect of casting the sum's int value to a double. The cast is completed before division takes place since a cast operator's precedence is higher than a division operator's. A mixed expression is created when a double is divided by an int. Now, conversion happens automatically. The operation results in a double once the int is upgraded to a double. In the absence of a cast, integer division would have taken place and any fractional parts would have been thrown away. Also, since the outcome would have been an int, the printf() statement's format would have been incorrect.

String Handling

A character array is what C refers to as a string. In addition to demonstrating how to utilise strings in this part, we also want to demonstrate the use of getchar() and putchar (). These are macros from the stdio.h file. A macro is used similarly to a function, despite some technical distinctions. To read characters from the keyboard and print characters on the screen, respectively, use the macros getchar() and putchar().

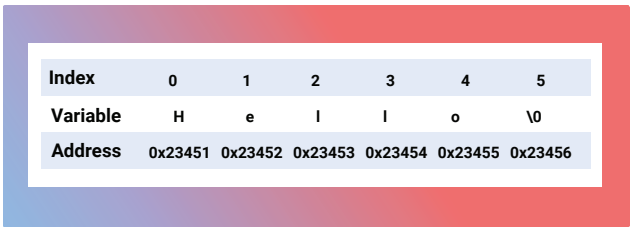
The declaration and initialization that follow produce a string containing the word "Hello". The character array holding the string must be one character larger than the word "Hello" in order to accommodate the null character at the end of the array.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The following sentence can be used to express the above statement if you adhere to the rule of array initialization:

```
char greeting[] = "Hello";
```

The above-described string's memory presentation in C is seen below;



Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

The null character is not actually added to the end of a string constant. After initialising the array, the C compiler automatically appends the '\0' to the end of the string. Let's try printing the string described above:

```
#include <stdio.h>

int main ()

{

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

printf("Output message: %s\n", greeting );

return 0;

}
```

When the aforementioned code is compiled and run, the following outcome is obtained:

Output message: Hello

Several functions that work with null-terminated strings are available in C:

strcpys1, s2	Copies string s2 into string s1.
strcats1, s2;	Concatenates string s2 onto the end of string s1.
strlens1	Returns the length of string s1.
strcmps1, s2	Returns 0 if s1 and s2 are the same; less than 0 if s1s2.
strstrs1, s2	Returns a pointer to the first occurrence of string s2 in string s1.

The example below uses a few of the aforementioned features:

```
#include <stdio.h>

#include <string.h>

int main ()

{

char str1[12] = "Hello";

char str2[12] = "World";

char str3[12];

int len ;

/* copy str1 into str3 */

strcpy(str3, str1);

printf("strcpy( str3, str1) : %s\n", str3 );

/* concatenates str1 and str2 */

strcat( str1, str2);

printf("strcat( str1, str2): %s\n", str1 );

/* total lenghth of str1 after concatenation */

len = strlen(str1);

printf("strlen(str1) : %d\n", len );

return 0;

}
```

Output

```
strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10
```

Pointers

You must have a solid working understanding of pointers if you wish to be effective at developing code in the C programming language. One of those concepts that C beginners find challenging is the idea of pointers. This unit's goal is to introduce pointers and demonstrate how to use them effectively in C programming. Ironically, the terminology used for pointers in C is more of a challenge than the actual idea itself. There are three primary uses for pointers in C.

They are initially employed to create dynamic data structures, which are composed of run-time heap-allocated memory chunks. Second, C handles pointers used as variable parameters in functions. Finally, dealing with strings makes use of pointers in C, which provides an alternative method of accessing data kept in arrays. A typical variable is a location in memory that can be used to store a value.

For instance, four bytes of memory are

reserved for the variable I when you declare it to be an integer. You use the term I to designate that memory region in your program. The four bytes at that place have a memory address at the machine level, where they can each carry an integer value.

A variable that points to another variable is called a pointer. This indicates that it stores the memory location of a different variable. In other words, the pointer contains the address of another variable rather than a value in the conventional sense. By storing its address, it serves as a pointer to the other variable. A pointer contains two components because it stores an address rather than a value.

The address is stored in the pointer itself. Its address denotes a worth. Both the value pointed to and the pointer are present. Pointers can be helpful, strong, and generally trouble-free tools as long as you take care to make sure they always point to appropriate memory regions in your programmes.

Characteristics

The memory of a computer is composed of a series of storage units known as bytes. An address is a number that is connected to each byte. The compiler immediately allots a certain block of memory to keep the value of a variable when we define it in our program. This block of memory will have a distinct starting address because every cell has a distinct address. The permitted range for the variable determines how big this block should be. For instance, an integer variable on a 32-bit computer is 4 bytes in size. In earlier 16-bit computers, an integer was 2 bytes. In C, a variable type like an integer does not necessarily need to have the same size on every type of machine. Running the code given below will provide you with information on the size of the different data kinds on your system.

```
# include <stdio.h>

main()

{

printf("n Size of a int = %d bytes", sizeof(int));

printf("\n Size of a float = %d bytes", sizeof(float));

printf("\n Size of a char = %d bytes", sizeof(char));

}
```

After running the program we will below output:

Size of int = 2 bytes

Size of float = 4 bytes

Size of char = 1 byte

A memory region that may store a value is referred to as an ordinary variable. For instance, the compiler reserves 2 bytes of RAM (depending on the PC) to keep the value of the integer when you declare a variable as an integer. You use the name num to refer to that memory region in your program. There is a memory address for that region at the machine level.

```
int num = 100;
```

Either the memory address or the name num can be used to access the value 100. Addresses can be kept in any other variable because they are just digits. Pointers are those variables that store the addresses of other variables. In other terms, a pointer is just a variable that holds an address,

or a location in memory where another variable is located. By storing its address, a pointer variable "points to" another variable. A pointer has two components because it stores an address rather than a value. The address is stored in the pointer itself. Its address denotes a worth. A pointer and the value it points to are present.

Characteristic features of Pointers:

Pointers are used in programming to:

- When addresses are used to alter the data directly, the application will run more quickly.
- Memory space will be saved.
- Memory access will be quick and effective.
- Allotted memory is dynamic.

Operating System Considerations

Writing and Running a C Program

Operating system, text editor, and compiler are the three variables that determine the precise actions that must be taken to generate a file containing C code, compile it, and execute it. The general process is the same in every situation, though. First, we give a thorough explanation of how it is carried out in a UNIX system. The process is then discussed in an MS-DOS context.

The C compiler will be called using the cc command in the explanation that follows. Nevertheless, in practice, the command is dependent on the compiler being used. For instance, we would use the command tee rather than cc if we were using the Turbo C compiler from Borland's command line interface.

Guidelines for designing and running a C application

- Create a text file with a C program in it, like pgm.c, using an editor. The file must have a name that ends in .c to indicate that it includes C source code. For instance, on a UNIX system, we would issue the command to use the vi editor.

  vi pgm.c

The programmer must be familiar with the proper commands for adding and editing text in order to use an editor.

- Compile the program. The command will enable you to achieve this.

  cc pgm.c

The cc command calls the preprocessor, compiler, and loader individually. By modifying a copy of the source code in compliance with the preprocessing directives, the preprocessor produces a translation unit. The compiler transforms the translation unit into object code. The programmer must return to step 1 and change the source file if mistakes are discovered. Errors that occur at this stage are referred to as syntax errors and compile-time errors. If there are no errors, the loader creates the executable file a.out using both the object code produced by the compiler and object code taken from a number of the system's libraries. The application can now be run since it is ready.

- Execute the program. To accomplish this, use the command.

  a.out

Usually, when the program has finished running, a system prompt will reappear on the screen. Run-time errors are any errors that happen while a program is being executed. The programmer must start over at step 1 if the program needs to be altered for any reason.

The file a.out will be replaced and its prior contents lost if we compile a different application. The executable file a.out needs to be moved or renamed if the contents are to be stored. Let's say we issue the command.

cc sea.c

This results in the automatic writing of executable code into a.out. We can issue the command to save this file.

mv a.out sea

As a result, a.out is shifted out to sea. Now, the command can be used to run the program.

sea

In UNIX, it's customary to omit the.c suffix and give the executable file the same name as the related source file. The cc command's output can be directed if we so choose by using the -0 parameter. For instance, the instruction

cc -0 sea sea.c

causes whatever is in a.out to remain intact while writing the executable output from cc directly into sea.

A software may contain a variety of errors. Run-time errors only appear while a program is being run; syntax problems are captured by the compiler. An attempt to divide by zero, for instance, could be encoded into a program, which could result in a run-time error when the program is run. Typically, a run-time error's error message isn't very helpful in identifying the problem.

So let's think about an MS-DOS environment. Most likely, another text editor would be utilised in this situation. Certain C systems, like Turbo C, offer both an integrated environment and a command line environment. The text editor and compiler are both a part of the integrated environment. (For specifics, consult the Turbo C manuals.) The command that starts the C compiler differs depending on whether MS-DOS or UNIX is being used.

On MS-DOS, a C compiler's executable output is written to a file with the same name as the source file but an.exe extension rather than a.c extension. Think about the situation when we are using the command-line environment for Turbo C. If we provide the instruction

tcc sea.c

then sea.exe will receive the executable code. We issue the command to run the application.

sea.exe or equivalently sea

There is no need to type the.exe extension in order to run the software. The rename command can be used to change the name of this file.

Summary

- In C, a string is an array of characters, and an array name by itself is a pointer. Because

of this, the concepts of arrays, strings, and pointers are intimately related. A pointer is just an address of an object in memory. C, unlike most languages, provides for pointer arithmetic. Because pointer expressions of great utility are possible, pointer arithmetic is one of the strong points of the language.

- The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

UNIT 3

Lexical Elements and Operators

Learning Objectives

At the end of this unit, you will be able to:

- Infer characters and lexical Elements
- Discuss the concept of keywords and identifiers
- Learn about operators and punctuators

Introduction

A language is C. It features an alphabet and rules for how to combine words and punctuation to create proper, or legal, programmes, just like other languages. The syntax of the language is defined by these principles. The compiler is the program that verifies the validity of C code. If there is a mistake, the compiler will stop and produce an error message. If the source code is valid and free of mistakes, the compiler converts it into object code, which the loader then uses to create an executable file. The preprocessor completes its work before the compiler is called. Because of this, we can consider the preprocessor to be a component of the compiler.

This is actually the case on some platforms, but the preprocessor is separate on others. We are not concerned with this in this chapter. But, we must be aware that in addition to the compiler, the preprocessor is another source of error messages. A C program is a series of characters that a C compiler will translate into object code, which is then translated onto a specific machine into a target language.

The target language will typically be a runnable or interpretable kind of machine language. The program must be syntactically sound for this to occur. Tokens, which might be considered the language's core vocabulary, are initially created by the compiler by grouping the characters in the program. There are six different categories of tokens in ANSI C: operators, punctuation, constants, string constants, and keywords. The compiler verifies that the tokens can be combined into valid strings using the language's syntax.

The majority of compilers have highly specific specifications. A C compiler will not offer a translation of a syntactically wrong program, regardless of how little the fault may be, in contrast to human readers of English who are capable of understanding the meaning of a sentence with an extra punctuation mark or a misspelt word. As a result, the programmer needs to develop their accuracy in developing code.

Characters and lexical Elements

The programmer creates a C program first as a string of characters. The following are some examples of characters that can be used in a program:

Characters that can be used in a program

Lowercase letters a b c . . . z

Uppercase letters A B C . . . Z

digits 0 1 2 3 4 5 6 7 8 9

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

white space characters blank, newline, tab, etc

The compiler gathers these characters into tokens, which are syntactic units. Before we move on to a strict description of C syntax, let's take a quick look at a basic program and arbitrarily highlight some of its tokens.

In file sum.c

```
1* Read in two integers and print their sum. *1

#include <stdio.h>

int main(void)

{

int a, b, sum;

printf("Input two integers: ");

scanfC"%d%d", &a, &b);

sum = a + b;

printfC"%d + %d = %d\n", a, b, sum);

return 0;

}
```

Lexical Dissection of the addition Program

```
/* Read in two integers and pri nt thei r sum. */
```

A /* and */ character delimits a comment. Each comment is first replaced by a single blank by the compiler. After that, the compiler either ignores white space or makes use of it to divide tokens.

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

This preprocessing instruction triggers the inclusion of the stdio.h standard header file. It is included because it contains the printf() and scanf function prototypes (). An example of a declaration is a function prototype. Functionprototypes are necessary for the compiler to function.

```
int main(void)

{

int a, b, sum;
```

These characters are divided into four different token types by the compiler. The parenthesis () that come right after main are an operator, and

the function name n is the main identifier. At first, this concept seems contradictory because what you see after main is (void), but it's actually only the parentheses () that make up the operator. The compiler is informed that main is a function by this operator. The punctuation marks "{", "," "," and ";" are identifiers; "a,""b," and "sum" are keywords.

int a, b, sum;

The compiler uses the white space between int and a to distinguish between the two tokens. Nobody is a writer.

int a, b, sum; /* wrong: white space is necessary */

On the other hand, there is no need for any white space after a comma. We had the option to write

int a,b,sum; but not int absum;

Absum would be regarded as an identifier by the compiler.

printfC"Input two integers: ");

scanfC"%d%d" , &a, &b);

The parenthesis that follow the identifiers printf and scanf tell the compiler that they are functions. After the compiler has translated the C code, the loader will attempt to create an executable file. If the programmer hasn't provided the code for printf() and scanf(), it will be taken from the standard library. Normally, a coder wouldn't change these identifiers.

"Input two integers: "

A string constant is a group of characters that are contained in double quotes. This is viewed as a single token by the compiler. Moreover, the compiler supplies RAM for storing the string.

&a, &b

The address operator is the letter &. It is handled as a token by the compiler. The compiler treats each of the characters & and an as a separate token even if they are close to one another. We had the option to write

& a, &b, or &a,&b

but not

&a &b /* the comma punctuator is missing */

a&, &b /* & requires its operand to be on the right */

sum = a + b;

Operators are the letters = and +. White space will not be used in this area, thus we may have

Sum a + b; or sum a + b ;

But not s u m a+b;

The comas would treat each letter on this line as a unique identifier if the latter had been written. The compiler would complain that not all of these identifiers have been declared.

The programmer makes advantage of white space to make the code easier to read. A human reader sees program text as a two-dimensional tableau, whereas the compiler sees it as a single stream of characters.

Keywords and Identifiers

Words that are explicitly reserved and have a precise meaning as separate tokens are known as keywords. They cannot be modified or applied to other situations.

Keywords

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

Figure: Keywords in C

There might be additional keywords in some implementations. These will change depending on the implementation or system. Here are a few other Turbo C keywords as an illustration.

Additional keywords for Borland C

asm	cdecl	far	huge	interrupt	near	pascal

Figure: Additional Keywords in C

In comparison to other popular languages, C contains fewer keywords. For instance, Ada has 62 keywords. C is unique in that it can accomplish a lot with only a small number of special symbols and keywords.

Identifiers

An identifying token consists of a string of characters, numbers, and the unique underscore ( ). A letter or an underscore must be the first character of an identifier. Most C implementations treat the lowercase and uppercase letters differently. It's best programming practice to select identifiers with mnemonic significance so they can improve the program's readability and documentation.

Identifier : : = {letter | underscore}$_1$ {letter | underscore | digit}$_{0+}$

Underscore : : - _

Some examples of identifiers are

K

_id

Iamamidentifier2

So_am_i

But not

Not#me /*special character # not allowed*/

101_south /*must not start with a digit*/

-plus /*do not mistake – for _*/

In order to provide software objects distinctive names, identifiers are made. One way to think of keywords is as reserved identifiers with a certain meaning. The C system is already aware of identifiers like scanf and printf as input/output functions in the standard library. Normally, these names wouldn't be changed. The reason the identifier ma; n is unique is because C programmes always start their execution at the main function.

The length of discriminated identifiers is one of the main differences between operating systems and C compilers. Just the first 8 characters of an

identification that has more than 8 characters will be used on some older systems. Just ignoring the remaining characters. For instance, on such a system, the variable names

I_am_an_identifier and I_am_an_elephant

would be seen as being the same.

An identifier's first 31 characters must be distinguishable in ANSIC. Many C systems are more discriminating.

An important component of good programming is making meaningful name selections.

If you were to create a program to calculate different taxes, you may include identifiers like tax rate, price, and tax so that the assertion

Tax = price * tax_rate;

would be clear in its meaning. The underscore is used to combine a string of words that are often separated by spaces into a single identification. The main rule for good programming style is readability, followed by meaningfulness and avoiding confusion.

Operators and Punctuators

There are numerous special characters in C that have specific meanings. The arithmetic operators are a couple of examples.

+ - * / %

They stand for addition, subtraction, multiplication, division, and modulus—the standard mathematical operations—respectively. Recall that in mathematics, the remainder produced when dividing an expression by an expression yields the value of a modulus b. For instance, 5% 3 has a value of 2, whereas 7% 2 has a value of 1.

Operators in a program can be used to divide identifiers. Although it is customary to surround binary operators with white space to improve readability, this is not necessary.

a + b

a – b

Context can affect the interpretation of some symbols. Consider the % sign in the two statements as an illustration of this.

printf("%d", a); and a = b % 7;

The modulus operator is represented by the second% symbol, whereas the first% symbol marks the beginning of a conversion specification or format. The brackets, commas, semicolons, and parentheses are some examples of punctuation. Take into account this code:

int main(void)

{

int a, b = 2, c

a = 17 * (b + c);

Following mai n, the parenthesis are regarded as an operator. They inform the compiler that the function's name is main. The punctuation marks "{", ",", ";", U(", and ")" come next.

Operators and punctuation are used to demarcate language elements and are both gathered by the compiler as tokens. The intended

usage of some special characters can be inferred from the context in which they are employed. For instance, parentheses are sometimes used as punctuation and other times to denote a function name. The expressions provide yet another illustration.

a + b ++a a += b

They all employ the character +, although ++ and += are only single operators. A tiny symbol set and a terse language result from making a symbol's meaning dependent on context.

Precedence and Associativity of Operators

Expressions are evaluated according to the precedence and associativity rules for operators. Due of the complier's flexibility to alter the evaluation to suit its needs, these rules do not completely dictate evaluation. Parentheses can be used to clarify or alter the order in which operations are carried out because they are evaluated before any expressions outside of them. Think about the phrase

1 + 2 * 3

Because the operator * in C has higher precedence than the operator +, multiplication takes place before addition in the program. Hence, the expression's value is 7. A comparable expression is

1 + (2 * 3)

Yet, because parentheses-enclosed phrases are evaluated first, the

(1 + 2) * 3

The value will be 9.

The precedence and associativity rules for some of the C operators are provided in the table below.

Operator precedence and associativity	
Operator	Associativity
() ++ (*postfix*) -- (*postfix*)	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
= += -= * = /= *etc.*	right to left

Figure: Operator precedence and associativity

Any operator on a line, including '~, I, and%, has equal precedence to all other operators on that line but has higher precedence than all the operators on lines below it. The right side of the table displays the associativity rule for each operator on a particular line. Every time we introduce a new operator, we will state its precedence and associativity rules. Frequently, we will also add to this table to condense the information.

Every C programmer needs to be aware of these guidelines. There is a unary plus in addition to the binary plus, which stands for addition. Both of these operators are denoted by a plus sign. Moreover, the negative symbol has unary and binary interpretations.

Keep in mind that ANSI C marked the introduction of the unary plus. Traditional C only has unary minus; there is no unary plus. We can observe from the above table that unary operators take precedence over binary plus and minus. Within the phrase

-a & b – c

The first and second minus signs are binary and unary, respectively. Using the rules of precedence, we can determine that the following expression is an equal one:

((-a) * b) – c

Summary

- Tokens are the basic syntactic units of C. They include keywords, identifiers, constants, string constants, operators, and punctuators. White space, along with operators and punctuators, can serve to separate tokens. For this reason, white space, operators, and punctuators are collectively called separators. White space, other than serving to separate tokens, is ignored by the compiler.
- A keyword, also called a reserved word, has a strict meaning. There are 32 keywords in C. They cannot be redefined.
- Some identifiers are already known to the system because they are the names of functions in the standard library. These include the input/output functions scanf() and printf() and mathematical functions such as sqrt(), sin(), cos(), and tan().

UNIT 4

The Fundamental Data Types

Learning Objectives

At the end of this unit, you will be able to:

- Define the concepts declarations, expressions, and assignment
- Recognize the use of getchar() and putchar()
- Infer mathematical functions

Introduction

Declarations, Expressions, and Assignment

Fundamental data types are the data types which are predefined in the language and can be directly used to declare a variable in C. Fundamental data types are the basic data types and all other data types are made from these basic data types.

A program works with things called variables and constants. All variables in C must be declared before use. This could be how a program's introduction would appear.

```
#include <stdio.h>

int main(void)

{

int

float

a, b, c;

x, y = 3.3, z -7.7;

printf("Input two integers: ");

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

a = b + c;

x = y + z;
```

Each variable that is declared has a type associated with it, which instructs the compiler to reserve the necessary amount of memory space to retain the values for the variables. It also enables the compiler to precisely instruct the machine to do the specified tasks. At the machine level, using the operator + to two variables of type int in the expression b + c is different from doing it to variables of type float in the equation y + z.

The technical distinctions between the two + operations need not bother the programmer, but the C compiler must be able to do so and produce the appropriate machine instructions.

A block of declarations and statements is enclosed by the braces { and}. The statements must come before any declarations.

Operators, function calls, constants, and variables are logically combined to form expressions. A constant, variable, or function call by itself can also constitute an expression. Several examples of phrases are given below:

a + b

sqrt(7.333)

5.0 * x - tan(9.0 / x)

Phrases frequently contain a value. For instance, the expression "a + b" has a clear value dependent on the values of the variables a and b. If a and b both have values of 1, then a + b has a value of 3.

The equal symbol = is the basic assignment operator in C. An example of an assignment expression is as follows:

i =7

The expression as a whole is given the value 7, and the variable i is given the value 7. An expression becomes a statement, or more specifically, an expression statement, when it is followed by a semicolon. These are a few instances of statements:

i = 7;

printfC"The plot thickens!\n");

The next two assertions are perfectly legal, but they have no purpose. In contrast to others, certain compilers will alert users to these assertions.

3.777;

a + b;

Let's take a look at a sentence that starts with a basic assignment expression and ends with a semicolon. The form will be as follows:

Variable = expr;

The right-hand side of the equal sign of the expression's value is computed first. The value is subsequently assigned to the variable to the left of the equal sign, which acts as the value of the assignment expression as a whole. (Statements have no intrinsic value.) The value of the assignment phrase as a whole is not employed in this case, as is to be noted. That is entirely OK. The value that an expression generates is not necessary to be used by the programmer.

The Fundamental Data Types

Several of the basic data types that C offers have already been seen. The restrictions on what can be kept in each kind must be discussed.

Fundamental data types: long form		
char	signed char	unsigned char
signed short int	signed int	signed long
unsigned short int	unsigned int	unsigned long
float	double long	double

Figure: Fundamental data type: long form

These are all keywords. They are ineffective as variable names. Only char and int are permitted to be used as keywords, despite the fact that they stand for "character" and "integer," respectively. Further data types like arrays, pointers, and structures are derived from the fundamental types.

The word "signed" is not frequently used. For instance, signed int is similar to ; int, and int is frequently used since shorter names are simpler to type. Nonetheless, the type char is unique in this sense. Moreover, the terms "short int,""long int," and "unsigned int" can all be abbreviated to just "short,""long," and "unsigned," respectively. Although the keyword signed by itself is comparable to int in this situation, it is rarely used. With each of these conventions, a new list is produced.

Assume for the moment that the category type is any of the basic types listed in the previous table. We can offer the basic declaration syntax using this category:

declaration :: = type identifler { , identifier }0_+ ;

The fundamental types can be divided into functional categories. The types that can carry integer values are known as integral types, whereas the kinds that can hold real values are known as floating types. They are all mathematically inclined.

Fundamental types grouped by functionality			
Integral types	char short unsigned short	signed chat int unsigned	unsigned char long unsigned long
Floating types	float	double	long double
Arithmetic types	*Intergral types + Floating types*		

Figure: Fundamental types grouped by functionality

These group names are provided for convenience.

The Use of getchar() and putchar()

For character 1/0, the standard C library offers a number of functions and macros. We will now take a look at the getchar and putchar macros. These macros often work in a loop to read/write a series of characters because they only read or write a single character.

A function call and a macro call are comparable. As a result, it consists of a macro name followed by a list of arguments separated by commas and wrapped in two parenthesis. Even though a macro doesn't require any parameters, the pair of parentheses still has to be utilised.

The getchar macro reads a single character from the keyboard's standard input stream. The format of a call to getchar is as follows: getchar (). Prior to providing its value, this macro waits for the key to be depressed. The returned value

can be put into a char variable. Interestingly, this macro actually gets information from the input buffer, which is processed only when the Enter key is hit by the application. Hence, getchar does not return a value before the user presses the Enter key. Interactive programmes cannot use it as a result. An alternative are the getch and getche procedures, which read data directly from the keyboard.

The putchar macro allows a single character to be written to the standard output stream (i.e., display). Putchar (c), where c is an integer expression denoting the character to be written, is the syntax for calling this macro. We generally pass a character to this macro even though it needs an argument of type int. The input value (c) is converted to an unsigned char and written to the standard output stream.

The getch and getche Functions

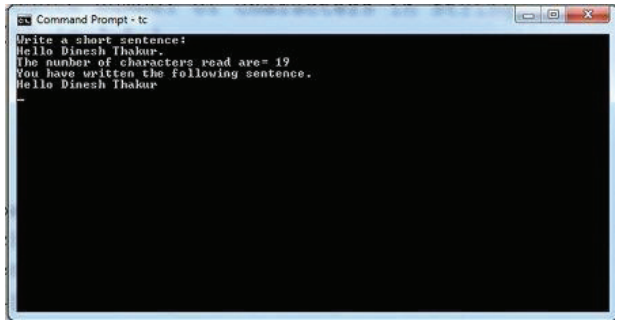
The getchar macro's use of line buffering makes it inappropriate for interactive settings. There isn't a facility in the C standard library that promises to deliver an interactive character input. Because of this, the majority of C compilers include substitute functions for usage in these interactive settings. They consist of the getch and getche functions offered in the console I/O header file, <conio.h>. Calls to these functions take the following forms, just like the getchar function: getch () and getche (). Both Dev-C++ and Turbo C/C++ include these functions. These functions have the names _getch and _getche in Visual C++.

The getch and getche procedures foresee a key press and immediately return the value. The entered character is displayed on the screen using the getche function rather than the getch function. The only difference between these two functions is this.

Example:

```
#include<stdio.h>

void main() {

char ch;

int count =0, i =0;

char Str [33] ;

clrscr();

printf("Write a short sentence:\n");

//The following code counts number of characters in string.

while ( (ch = getchar ()) != '.')  {

Str[i] = ch;

count++;

i++;

}

Str[i] = '\0';

// append null character at end of string

printf("The number of characters read are= %d\n", count);

printf("You have written the following sentence.\n");

puts(Str); //display the string on std <a href="https://Test.com" data-internallinksmanager029f6b8e52c="6" rel="nofollow">output device</a>.

}
```

Output:



According to the while expression, when the character '.' is encountered, the system stops reading the code. The string doesn't even contain the character ".". Reading is character-by-character, therefore this offers a chance to get to know a certain character. Counting characters is thus feasible.

Mathematical Functions

Via the functions defined in the <math. h> header file, we can execute mathematical operations using the C programming language. Many methods for carrying out mathematical operations, including sqrt(), pow(), ceil(), and floor(), are included in the header file <math.h>.

C Math Functions

The header file math.h contains a number of methods. Here are listed some of the math.h header file's frequently used functions.

No .	Function	Description
1)	ceil(number)	rounds up the given number. It returns the integer value which is greater than or equal to given number.
2)	floor(number)	rounds down the given number. It returns the integer value which is less than or equal to given number.
3)	sqrt(number)	returns the square root of given number.
4)	pow(base, exponent)	returns the power of given number.
5)	abs(number)	returns the absolute value of given number.

C Math Example

Let's look at a straightforward example of a math function from the header file math.h.

```
#include<stdio.h>

#include <math.h>

int main(){

printf("\n%f",ceil(3.6));

printf("\n%f",ceil(3.3));

printf("\n%f",floor(3.6));

printf("\n%f",floor(3.2));

printf("\n%f",sqrt(16));

printf("\n%f",sqrt(7));

printf("\n%f",pow(2,4));

printf("\n%f",pow(3,3));

printf("\n%d",abs(-12));

 return 0;

}
```

Output:

```
4.000000
4.000000
3.000000
3.000000
4.000000
2.645751
16.000000
27.000000
12
```

Summary

- The fundamental data types are char, short, int, long, unsigned versions of these, and three floating types. The type char is a I-byte integral type mostly used for representing characters.
- Three floating types, float, double, and long double, are provided to represent real numbers. Typically, a float is stored in 4 bytes and a doub1 e in 8 bytes, the number of bytes used to store along double varies from one compiler to another. However, as compilers get updated, the trend is to store along double in 16 bytes. The type double, not float, is the "wording" type.
- Automatic conversions occur in mixed expressions and across an equal sign. Casts can be used to force explicit conversions.
- Suffixes can be used to explicitly specify the type of a constant. For example, 3U of type unsigned, and 7. 0F is of type float.
- A character constant such as 1 A I is of type int in C, but it is of type char in C++. This is one of the few places where c++ differs from C.

UNIT 5

Operator In C

Learning Objectives

At the end of this unit, you will be able to:

- Define the various operator used in C

Introduction

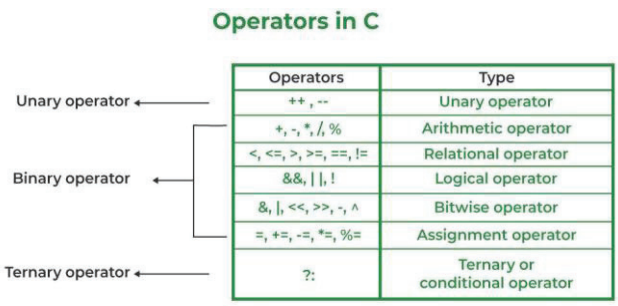
Any programming language's core is its set of operators. Operators are symbols that enable us to carry out particular logical and mathematical operations on operands. The operands are operated by an operator, in other words. In an illustration, the addition operator "+" is employed as follows:

In this formula, "c" is equal to "a + b," where "a" and "b" are the operands and "+" is the addition operator. The compiler is instructed to add both operands, "a" and "b," by the addition operator.

Without the usage of operators, the C programming language cannot work fully.

C can be divided into six kinds and has a large number of built-in operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators



Arithmetic Operators

This operator is used to calculate numbers. They are one of the binary or unary arithmetic operators. where the unary arithmetic operator, such as +,-, ++, --,!, tiled, only required one operand. Addition, subtraction, multiplication, and division are among these operators. In contrast, the binary arithmetic operator has two operands and has the following operators: + (addition), - (subtraction), * (multiplication), / (division), and % (modulus). Yet because there is no exponent operator in C, modulus cannot be applied with a floating point operand.

Addition and subtraction are similar, but unary (+) and unary (-) are distinct. Integer arithmetic is used when both operands are integers, and the outcome is always an integer. Floating arithmetic is known when both operands are floating point, and mixed type or mixed mode arithmetic is known when the operands are both integer and floating point. The outcome is of the float type.

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A *B will give 200
/	Divides numerator by de-numerator	B /A will give 2
%	Modulus Operator and remainder of after an interger devision	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A-- will give 9

Figure: Arithmetic Operator.

C Language code for arithmetic operator

```
#include <stdio.h>

main()

{

int a = 21;

int b = 10;

int c ;

c = a + b;

printf("Line 1 - Value of c is %d\n", c );

c = a - b;

printf("Line 2 - Value of c is %d\n", c );

c = a * b;

printf("Line 3 - Value of c is %d\n", c );

c = a / b;

printf("Line 4 - Value of c is %d\n", c );

c = a % b;

printf("Line 5 - Value of c is %d\n", c );

c = a++;

printf("Line 6 - Value of c is %d\n", c );

c = a--;

printf("Line 7 - Value of c is %d\n", c );

}
```

Output:

The following outcome is produced by compiling and running the aforementioned program:

Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2

Line 5 - Value of c is 1

Line 6 - Value of c is 21

Line 7 - Value of c is 22

Relational Operators

It is employed to compare two expressions' values based on their relationships. Relational expression is an expression that contains a relational operator. The value is assigned in this case based on whether it is true or false.

Examples include less than, more than, equal to, etc. Let's examine each one in turn.

1. **Equal to operator**: Represented as '==', the equal to operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise, it returns false. For example, 5==5 will return true.

2. **Not equal to operator**: Represented as '!=', the not equal to operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise, it returns false. It is the exact boolean complement of the '==' operator. For example, 5!=5 will return false.

3. **Greater than operator**: Represented as '>', the greater than operator checks whether the first operand is greater than the second operand or not. If so, it returns true. Otherwise, it returns false. For example, 6>5 will return true.

4. **Less than operator**: Represented as '<', the less than operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise, it returns false. For example, 6<5 will return false.

5. **Greater than or equal to operator**: Represented as '>=', the greater than or equal to operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true else it returns false. For example, 5>=5 will return true.

6. **Less than or equal to operator**: Represented as '<=', the less than or equal to operator checks whether the first operand is less than or equal to the second operand. If so, it returns true else false. For example, 5<=5 will also return true.

Example:

```
// C program to demonstrate working of relational operators

#include <stdio.h>

int main()

{

int a = 10, b = 4;

    // greater than example
```

```
if (a > b)
printf("a is greater than b\n");
else
printf("a is less than or equal to b\n");
// greater than equal to
if (a >= b)
printf("a is greater than or equal to b\n");
else
printf("a is lesser than b\n");
// less than example
if (a < b)
printf("a is less than b\n");
else
printf("a is greater than or equal to b\n");
// lesser than equal to
if (a <= b)
printf("a is lesser than or equal to b\n");
else
printf("a is greater than b\n");
// equal to
if (a == b)
printf("a is equal to b\n");
else
printf("a and b are not equal\n");
// not equal to
if (a != b)
printf("a is not equal to b\n");
else
printf("a is equal b\n");
return 0;
}
```

Output:

```
a is greater than b
a is greater than or equal to b
a is greater than or equal to b
a is greater than b
a and b are not equal
a is not equal to b
```

Logical Operators

Logical operators can be used to integrate two or more conditions or constraints, as well as to improve the analysis of the initial condition being looked at. The result of a logical operation is a Boolean value that can be either true or false.

For instance, when both of the conditions are satisfied, the logical AND operator, represented by the && operator in C, returns true. False is returned if not. As a result, a && b yields true when both a and b are true (i.e. non-zero).

They can be used to combine two or more restrictions or conditions, or they can be used to

improve the analysis of the initial condition being taken into account. These are described below:

1. Logical AND operator: The '&&' operator returns true when both the conditions under consideration are satisfied. Otherwise, it returns false. For example, a && b returns true when both a and b are true (i.e. non-zero).
2. Logical OR operator: The '||' operator returns true even if one (or both) of the conditions under consideration is satisfied. Otherwise, it returns false. For example, a || b returns true if one of a or b, or both are true (i.e. non-zero). Of course, it returns true when both a and b are true.
3. Logical NOT operator: The '!' operator returns true the condition in consideration is not satisfied. Otherwise, it returns false. For example, !a returns true if a is false, i.e. when a=0.

Example:

```
// C program to demonstrate working of logical operators

#include <stdio.h>

int main()

{

    int a = 10, b = 4, c = 10, d = 20;

    // logical operators

    // logical AND example

    if (a > b && c == d)

    printf("a is greater than b AND c is equal to d\n");

    else

    printf("AND condition not satisfied\n");

    // logical OR example

    if (a > b || c == d)

    printf("a is greater than b OR c is equal to d\n");

    else

    printf("Neither a is greater than b nor c is equal "

    " to d\n");

    // logical NOT example

    if (!a)

    printf("a is zero\n");

    else

    printf("a is not zero");

    return 0;

}
```

Output:

```
AND condition not satisfied

a is greater than b OR c is equal to d

a is not zero
```

Short-Circuiting in Logical Operators:

- In the case of logical AND, the second operand is not evaluated if the first operand

is false. For example, because the first operand of a logical AND is false, Program 1 below does not output "Logical operator."

```
#include <stdbool.h>

#include <stdio.h>

int main()

{

    int a = 10, b = 4;

    bool res = ((a == b) && printf("Logical operator"));

    return 0;

}
```

Output: No Output

- However, because the first operand of a logical AND is true, the program below prints "Logical Operator."

```
#include <stdbool.h>

#include <stdio.h>

int main()

{

    int a = 10, b = 4;

    bool res = ((a != b) && printf("Logical Operator"));

    return 0;

}
```

Output: Logical operator

- In the case of logical OR, the second operand is not considered if the first operand is true. For instance, program 1 below doesn't print "Operator" because the first operand of the logical OR is true by itself.

```
#include <stdbool.h>

#include <stdio.h>

int main()

{

    int a = 10, b = 4;

    bool res = ((a != b) || printf("Operator"));

    return 0;

}
```

Output: No Output

But, because the first operand of the logical OR is false, the following program produces "GeeksQuiz."

```
#include <stdbool.h>

#include <stdio.h>

int main()

{

    int a = 10, b = 4;

    bool res = ((a == b) || printf("Operator"));

    return 0;

    }
```

Output: Operator

Bitwise Operators

Bit-level operations on the operands are carried out using the Bitwise operators. Prior to performing the calculation on the operands, the operators are translated to bit-level. For quicker processing, mathematical operations like addition, subtraction, multiplication, etc. can be carried out at the bit level.

The following are the truth tables for &, |, and ^:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table is a list of the bitwise operators that the C language supports. If variable A has a value of 60 and variable B has a value of 13, then

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eigher operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60, which is 1100 0011
<<	Binary left shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Example:

```
#include <stdio.h>

main()

{

unsigned int a = 60; /* 60 = 0011 1100 */

unsigned int b = 13; /* 13 = 0000 1101 */

int c = 0;

c = a & b; /* 12 = 0000 1100 */

printf("Line 1 - Value of c is %d\n", c );

c = a | b; /* 61 = 0011 1101 */

printf("Line 2 - Value of c is %d\n", c );

c = a ^ b; /* 49 = 0011 0001 */

printf("Line 3 - Value of c is %d\n", c );

c = ~a; /*-61 = 1100 0011 */

printf("Line 4 - Value of c is %d\n", c );

c = a << 2; /* 240 = 1111 0000 */
```

```
printf("Line 5 - Value of c is %d\n", c );

c = a >> 2; /* 15 = 0000 1111 */

printf("Line 6 - Value of c is %d\n", c );

}
```

Output:

```
Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15
```

Assignment Operators

A variable's value is assigned using assignment operators. A variable serves as the assignment operator's left side operand, and a value serves as its right side operand. The variable on the left must have the same data type as the value on the right side in order for the compiler to avoid raising an error.

A variable's value is assigned using assignment operators. A variable serves as the assignment operator's left side operand, and a value serves as its right side operand. The variable on the left must have the same data type as the value on the right side in order for the compiler to avoid raising an error.

Below are examples of various assignment operators:

a). “=”

The simplest assignment operator is this one. The variable on the left is given the value on the right using this operator.

Example:

```
a = 10;

b = 20;

ch = 'y';
```

b) “+=”

This operator is created by combining the "+" and "=" operators. This operator assigns the outcome to the left variable after first adding the left variable's present value to the right variable's value.

Example:

(a += b) can be written as (a = a + b)

If initially value stored in a is 5. Then (a += 6) = 11.

c) “-=”

The '-' and '=' operators are combined to form this operator. This operator assigns the result to the variable on the left after first subtracting the value on the right from the left's current value.

Example:

(a -= b) can be written as (a = a - b)

If initially value stored in a is 8. Then (a -= 6) = 2.

d) “*=”

This operator is made up of the '*' and '=' operators. Prior to assigning the outcome to the

left variable, this operator multiplies the value of the left variable by the value of the right variable.

Example:

(a *= b) can be written as (a = a * b)

If initially, the value stored in a is 5. Then (a *= 6) = 30.

e) "/="

This operator is created by combining the '/' and '=' operators. This operator divides the value of the variable on the left by the value of the variable on the right before assigning the result to the variable on the left.

Example:

(a /= b) can be written as (a = a / b)

If initially, the value stored in a is 6. Then (a /= 2) = 3.

Misc Operators

Apart from the above operators, there are some other operators available in C used to perform some specific tasks. Some of them are discussed here:

a. sizeof operator

- sizeof is much used in the C programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of the unsigned integral type which is usually denoted by size_t.
- Basically, the sizeof the operator is used to compute the size of the variable.

b. Comma Operator

- The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).
- The comma operator has the lowest precedence of any C operator.
- Comma acts as both operator and separator.

c. Conditional Operator

The conditional operator is of the form Expression 1? Expression 2 : Expression 3

- Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is True then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is false then we will execute and return the result of Expression3.
- We may replace the use of if..else statements with conditional operators.

d. dot (.) and arrow (->) Operators

- Member operators are used to reference individual members of classes, structures, and unions.
- The dot operator is applied to the actual object.
- The arrow operator is used with a pointer to an object.

e. Cast Operator

- Casting operators convert one data type to another. For example, int(2.2000) would return 2.
- A cast is a special operator that forces one data type to be converted into another.
- The most general cast supported by most of the C compilers is as follows – [(type) expression].

f. &,* Operator

- Pointer operator & returns the address of a variable. For example &a; will give the actual address of the variable.
- Pointer operator * is a pointer to a variable. For example *var; will pointer to a variable var.

Example:

```
// C Program to Demonstrate the working concept of
// Operators
#include <stdio.h>
int main()
{
    int a = 10, b = 5;
    // Arithmetic operators
    printf("Following are the Arithmetic operators in C\n");
    printf("The value of a + b is %d\n", a + b);
    printf("The value of a - b is %d\n", a - b);
    printf("The value of a * b is %d\n", a * b);
    printf("The value of a / b is %d\n", a / b);
    printf("The value of a % b is %d\n", a % b);
    printf("The value of a++ is %d\n",
    a++); // First print (a) and then increment it
    // by 1
    printf("The value of a-- is %d\n",
    a--); // First print (a+1) and then decrease it
    // by 1
    printf("The value of ++a is %d\n",
    ++a); // Increment (a) by (a+1) and then print
    printf("The value of --a is %d\n",
    --a); // Decrement (a+1) by (a) and then print
    // Assignment Operators --> used to assign values to
    // variables int a =3, b=9; char d='d';
    // Comparison operators
    // Output of all these comparison operators will be (1)
    // if it is true and (0) if it is false
    printf(
```

```
    "\nFollowing are the comparison operators in C\n");

    printf("The value of a == b is %d\n", (a == b));

    printf("The value of a != b is %d\n", (a != b));

    printf("The value of a >= b is %d\n", (a >= b));

    printf("The value of a <= b is %d\n", (a <= b));

    printf("The value of a > b is %d\n", (a > b));

    printf("The value of a < b is %d\n", (a < b));

    // Logical operators

    printf("\nFollowing are the logical operators in C\n");

    printf("The value of this logical and operator ((a==b) "

           "&& (a<b)) is:%d\n",

           ((a == b) && (a < b)));

           printf("The value of this logical or operator ((a==b) "

           "|| (a<b)) is:%d\n",

           ((a == b) || (a < b)));

    printf("The value of this logical not operator "

           "(!(a==b)) is:%d\n",

           (!(a == b)));

           return 0;

    }
```

Output:

The C's arithmetic operators are listed here.

The value of a + b is 15

The value of a - b is 5

The value of a * b is 50

The value of a / b is 2

The value of a % b is 0

The value of a++ is 10

The value of a-- is 11

The value of ++a is 11

The value of --a is 10

The comparison operators in C are listed below.

The value of a == b is 0

The value of a != b is 1

The value of a >= b is 1

The value of a <= b is 0

The value of a > b is 1

The value of a < b is 0

The logical operators in C are listed below.

The value of this logical and operator ((a==b) && (a<b)) is:0

The value of this logical or operator ((a==b) || (a<b)) is:0

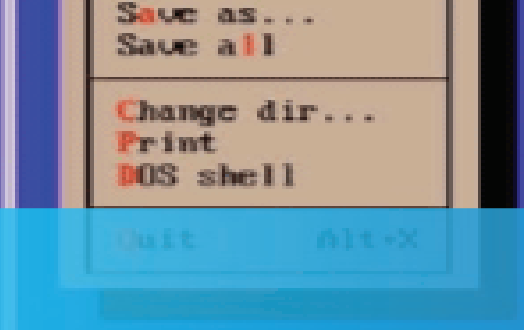
The value of this logical not operator (!(a==b)) is:1

Summary

- Operators are the foundation of any

programming language. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.

- This operator used for numeric calculation. These are of either Unary arithmetic operator, Binary arithmetic operator.
- Where Unary arithmetic operator required only one operand such as +,-, ++, --,!, tiled. And these operators are addition, subtraction, multiplication, division. Binary arithmetic operator on other hand required two operand and its operators are +(addition),-(subtraction),*(multiplication), /(division), %(modulus).
- Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false.**
- The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.



UNIT 6

Decision Making in C

Learning Objectives

At the end of this unit, you will be able to:

- Discuss various conditional statement in c
- Infer about switch statement
- Define nested switch statements

Introduction

if statement

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

In an if statement, a boolean expression is followed by one or more sentences.

Syntax

In the C programming language, an if statement has the following syntax:

```
if(boolean_expression)

{

/* statement(s) will execute if the boolean expression is true */

}
```

The code block in the if statement will be executed if the boolean expression returns true. If the boolean expression returns false, the first block of code after the closing curly brace (the conclusion of the if statement) will be executed.

All non-zero and non-null values are taken as true in the C

programming language, but zero or null values are taken as false values.

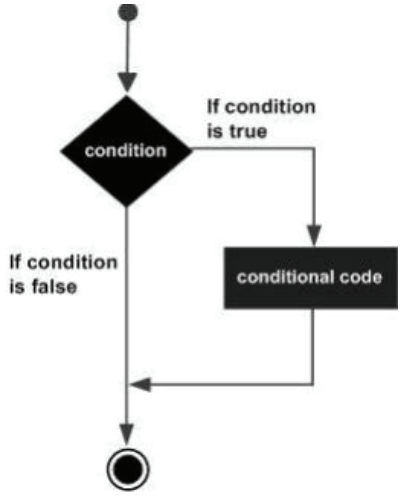


Figure: Flow diagram

Example:

```
#include <stdio.h>

int main ()

{

/* local variable definition */

int a = 110;

/* check the boolean condition using if statement */

if( a < 220 )

{

/* if condition is true then print the following */

printf("a is less than 20\n" );

}

printf("value of a is : %d\n", a);

return 0;

}
```

Output:

a is less than 20;

value of a is : 10

if...else statement

An optional else sentence that runs when the boolean expression is false can be placed after an if statement.

Syntax

In the C programming language, an if...else statement has the following syntax:

```
if(boolean_expression)

{

/* statement(s) will execute if the boolean expression is true */

}

else

{

/* statement(s) will execute if the boolean expression is false */

}
```

The if block of code is run if the boolean statement evaluates to true; else, the else block of code is executed.

All non-zero and non-null values are taken as true in the C programming language, but zero or null values are taken as false values.

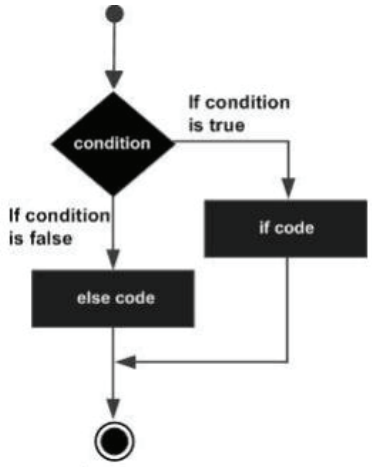


Figure: Flow diagram.

Example:

```
#include <stdio.h>

int main ()

{

/* local variable definition */

int a = 100;

/* check the boolean condition */

if( a < 20 )

{

/* if condition is true then print the following
*/

printf("a is less than 20\n" );

}

else

{

/* if condition is false then print the following
*/

printf("a is not less than 20\n" );

}

printf("value of a is : %d\n", a);

return 0;

}
```

Output:

a is not less than 20;

value of a is : 100

The if...else if...else Statement

A single if...else if statement is particularly helpful for testing several conditions. It can be followed by an optional else if...else statement.

There are a few things to bear in mind while employing if, otherwise, if, else statements:

- An if must come after any other ifs and can have zero or one else's.
- Any number of other ifs can follow an if, but they must appear before the else.
- None of the remaining else if's or else's will be tried after an else if succeeds.

Syntax

In the C programming language, an if...else if...else statement has the following syntax:

```
if(boolean_expression 1)

{

 /* Executes when the boolean expression 1
is true */
```

```
}
else if( boolean_expression 2)
{
 /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
 /* Executes when the boolean expression 3 is true */
}
else
{
 /* executes when the none of the above condition is true */
}
```

Example:

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 100;
/* check the boolean condition */
if( a == 10 )
{
/* if condition is true then print the following */
printf("Value of a is 10\n" );
}
else if( a == 20 )
{
/* if else if condition is true */
printf("Value of a is 20\n" );
}
else if( a == 30 )
{
/* if else if condition is true */
printf("Value of a is 30\n" );
}
else
{
/* if none of the conditions is true */
printf("None of the values is matching\n" );
}
printf("Exact value of a is: %d\n", a );
return 0;
}
```

The following is the outcome of compiling and running the aforementioned code:

```
None of the values is matching
Exact value of a is: 100
```

Nested if statements

Using one if or else if statement inside another if or else if statement is always acceptable in the C programming language (s).

Syntax

A nested if statement has the following syntax:

```
if( boolean_expression 1)
{
/* Executes when the boolean expression 1 is true */
if(boolean_expression 2)
{
/* Executes when the boolean expression 2 is true */
}
}
```

Similar to how you nested an if statement, you can nest an else if...else statement.

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
/* check the boolean condition */
if( a == 100 )
{
/* if condition is true then check the following */
if( b == 200 )
{
/* if condition is true then print the following */
printf("Value of a is 100 and b is 200\n" );
}
}
printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );
return 0;
}
```

The following is the outcome of compiling and running the aforementioned code:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Switch statement

A switch statement enables the comparison of a variable's value to a list of possible values. Each value is referred to as a case, and each switch case checks the variable being turned on.

Syntax

The following is the syntax for a switch statement in the C computer language:

```
switch(expression){

case constant-expression :

statement(s);

break; /* optional */

case constant-expression :

statement(s);

break; /* optional */

/* you can have any number of case statements */

 default : /* Optional */

 statement(s);

}
```

A switch statement must adhere to the following guidelines:

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

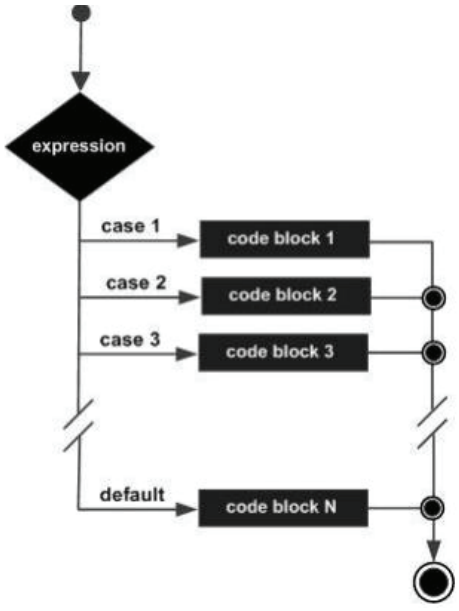


Figure: Flow diagram.

Example:

```
#include <stdio.h>

int main ()

{

/* local variable definition */

char grade = 'B';

switch(grade)

{

case 'A' :

printf("Excellent!\n" );

break;

case 'B' :

case 'C' :

printf("Well done Keep up\n" );

break;

case 'D' :

printf("You passed\n" );

break;

case 'F' :

printf("Better try again\n" );

break;

default :

printf("Invalid grade\n" );

}

printf("Your grade is %c\n", grade );

return 0;

}
```

The following is the outcome of compiling and running the aforementioned code:

```
Well done

Your grade is B
```

Nested switch statements

A switch could be included in the statement chain of an outside switch. No conflicts will develop even if the case constants of the inner and outer switches have similar values.

Syntax

A nested switch statement has the following syntax:

```
switch(ch1) {

case 'A':

printf("This A is part of outer switch" );

switch(ch2) {

case 'A':

printf("This A is part of inner switch" );

break;

case 'B': /* case code */

}

break;

case 'B': /* case code */

}
```

Example

```
#include <stdio.h>

int main ()

{

/* local variable definition */

 int a = 100;

 int b = 200;

 switch(a) {

 case 100:

 printf("This is part of outer switch\n", a );

 switch(b) {

 case 200:

 printf("This is part of inner switch\n", a );

 }

 }

 printf("Exact value of a is : %d\n", a );

 printf("Exact value of b is : %d\n", b );

 return 0;

}
```

The following is the outcome of compiling and running the aforementioned code:

This is part of outer switch

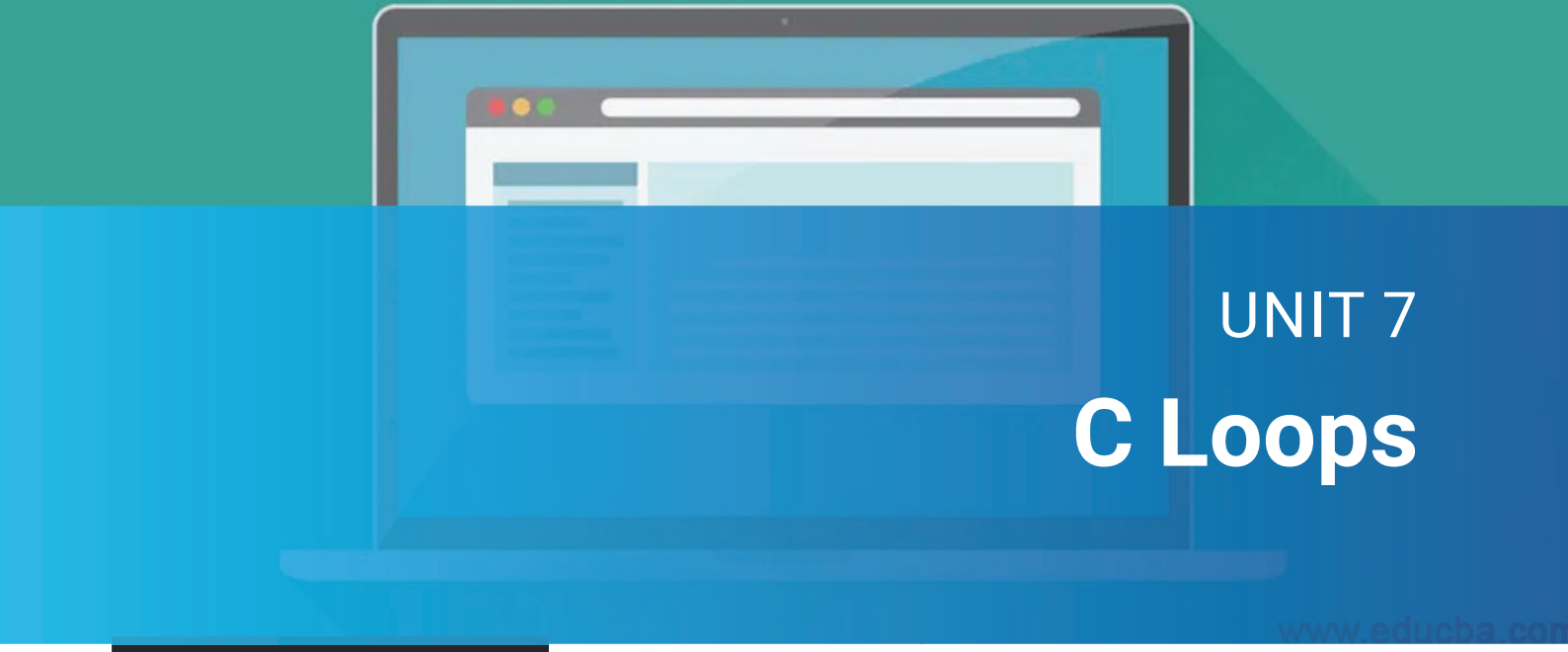
This is part of inner switch

Exact value of a is : 100

Exact value of b is : 200

Summary

- Relational, equality, and logical expressions have the int value 0 or 1.
- Automatic type conversions can occur when two expressions are compared that the operands of a relational, equality, or logical operator.
- An if statement provides a way of choosing whether or not to execute a statement.
- The else part of an if-else statement associates with the nearest available if This resolves the "dangling else" problem.
- The programmer often has to choose between the use of a while or a far statement. In situations where clarity dictates that both the control and the indexing be kept visible at the top of the loop, the for statement is the natural choice.



UNIT 7

C Loops

Learning Objectives

At the end of this unit, you will be able to:

- Learn about while loop in C, for loop in C and do...while loop in C
- Define the concepts of nested loops in C
- Discuss about the break statement in C

Introduction

continue statement in C

C Loop is used to execute the block of code several times according to the condition given in the loop. It means it executes the same code multiple times so it saves code and also helps to traverse the elements of an array.

There may be times when you need to execute a code block repeatedly. A function's first statement is executed first, followed by its second and so on. Most of the time, statements are carried out in order. Programming languages' various control structures allow for more complex execution paths. In the majority of programming languages, a loop statement often takes the following form. We can execute a statement or group of statements repeatedly by using a loop statement.

while loop in C

We have learned from studying the for loop that the number of iterations is known in advance, i.e., how many times the body of the loop must be executed. While loops are employed when we are unsure of the precise number of loop iterations in advance. Based on the results of the test, the loop is stopped.

The initialization expression, test expression, and update expression are the three statements that make up a loop, as we've just mentioned. The arrangement of these three statements varies between the three looping constructs—For, while, and do while—in terms of syntax.

```
initialization expression;

while (test_expression)

{

 // statements

 update_expression;

}
```

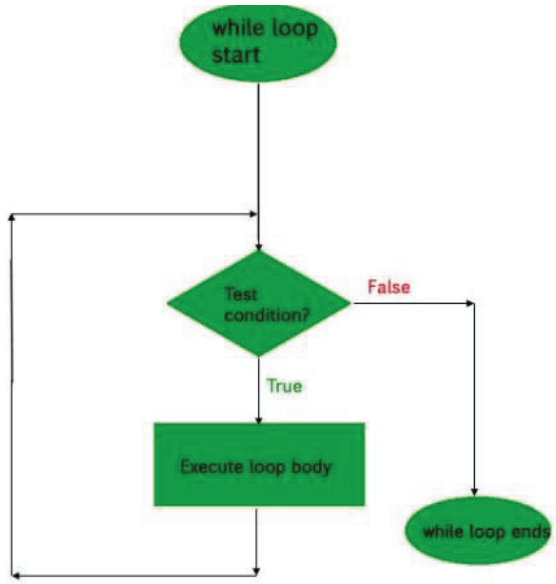


Figure: Flow Diagram.

Example:

```
// C program to illustrate while loop

#include <stdio.h>

int main()

{

 // initialization expression

 int i = 1;

 // test expression

 while (i < 6)

 {

  printf( "Hello India\n");

  // update expression

  i++;

  }

  return 0;

 }
```

Output:

Hello India

Hello India

Hello India

Hello India

Hello India

for loop in C

With the use of a for loop, which is a repetition control structure, we can create a loop that runs a predetermined number of times. The loop permits us to carry out n stages simultaneously in a single line.

Syntax:

```
for (initialization expr; test expr; update expr)

{

 // body of the loop

 // statements we want to execute

}
```

A for loop manages the loop by using a loop variable. Set a value for the loop variable's initial value before determining whether it exceeds or falls short of the counter value. If the assertion

is accurate, the loop's body is executed, and the loop variable is changed. Continue until the exit condition is met.

- **Initialization Expression**: The loop counter must be initialized to a certain value in this expression. for instance, int i=1;
- **Test Expression**: We must verify the condition in this expression. If the condition is true, we will carry out the for loop's body and move on to the update expression; otherwise, we will stop. For instance: i <= 10;
- **Update Expression**: This expression increases or decreases the loop variable by a certain amount after the loop body has run. for instance: i++;

Equivalent flow diagram for loop:

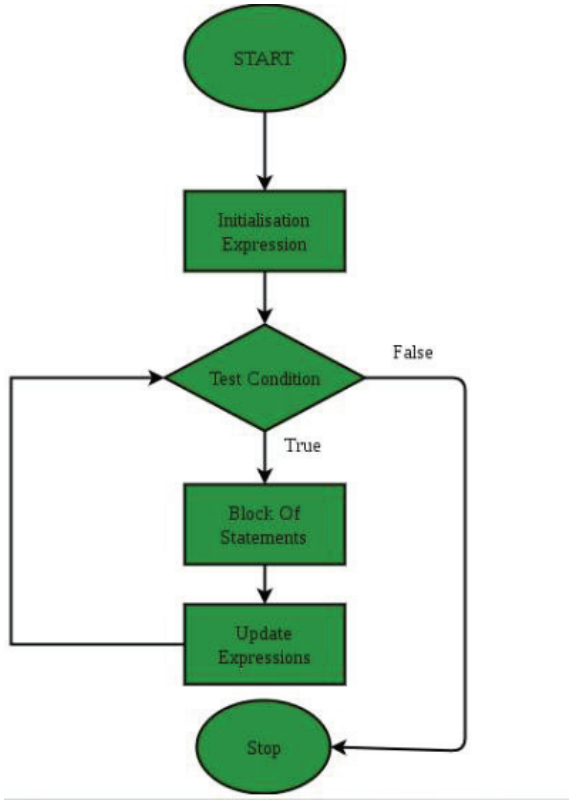


Figure: diagram for loop.

Example:

// C program to illustrate for loop

```
#include <stdio.h>

int main()

{

 int i=0;

 for (i = 1; i <= 10; i++)

 {

 printf( "Hello India\n");

 }

 return 0;

}
```

Output:

Hello India

Hello India

Hello India

Hello India

Hello India

Hello India

Hello India

Hello India

Hello India

Hello India

do...while loop in C

The execution of the loop is stopped in do while loops as well based on the results of the test. While loops are entry controlled, do while loops are exit controlled since the condition is verified at the end of the loop body. This is the major distinction between do while loops and while loops.

Syntax:

```
initialization expression;

do

{

// statements

update_expression;

}while (test_expression);
```

Note: Take note of the semicolon (";") at the loop's end.

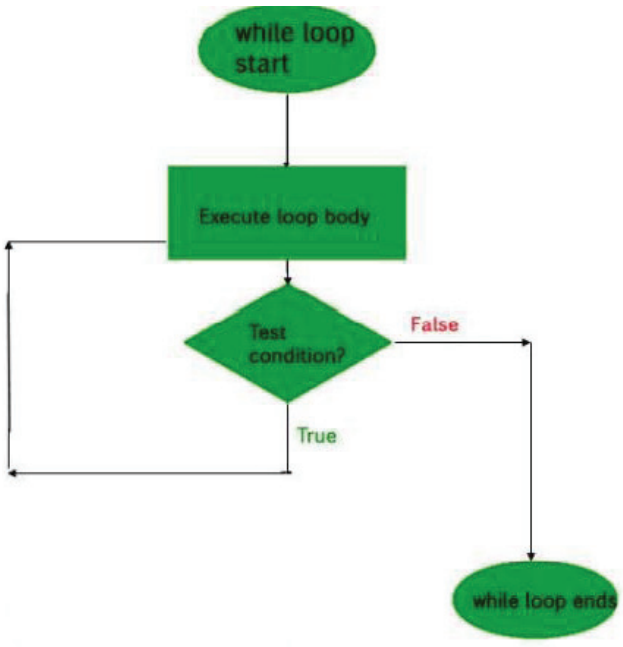


Figure: Flow diagram do while.

Example:

```
// C program to illustrate do-while loop

#include <stdio.h>

int main()

{

 int i = 2; // Initialization expression

 do

 {

 // loop body

 printf( "Hello India\n");

 // update expression

 i++;

 } while (i < 1); // test expression

 return 0;

}
```

Output:

Hello India

In the program above, the test condition (i1) evaluates to false. The loop's body will only execute once, though, because the exit is controlled.

nested loops in C

Loop nesting is supported in C. The C language's nesting of loops feature enables the looping of statements inside of other loops. Let's look at a C nesting loop example.

There is no constraint on the number of loops that can be formed, therefore any number of loops can be defined inside another loop. It is possible to define the nesting level n times. Any type of loop can be defined within of another loop; for instance, a "while" loop can be defined inside of a "for" loop.

Syntax of Nested loop

```
Outer_loop
{
Inner_loop
{
// inner loop statements.
}
// outer loop statements.
}
```

The legitimate loops that can be used as "for" loops, "while" loops, or "do-while" loops are outer_loop and inner_loop.

Nested for loop

Every form of loop that is defined inside the "for" loop is referred to as a "nested for loop."

```
for (initialization; condition; update)
{
for(initialization; condition; update)
{
// inner loop statements.
}
// outer loop statements.
}
```

Example to showcase functionality of nested for loop

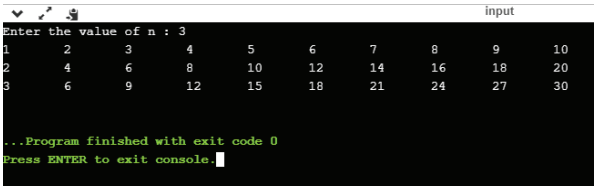
```
#include <stdio.h>
int main()
{
  int n;// variable declaration
  printf("Enter the value of n :");
  // Displaying the n tables.
  for(int i=1;i<=n;i++)  // outer loop
  {
     for(int j=1;j<=10;j++)  // inner loop
     {
        printf("%d\t",(i*j)); // printing the value.
     }
     printf("\n");
}
```

Explanation of the above code

- First, the 'i' variable is initialized to 1 and then program control passes to the i<=n.
- The program control checks whether the condition 'i<=n' is true or not.
- If the condition is true, then the program control passes to the inner loop.
- The inner loop will get executed until the condition is true.

- After the execution of the inner loop, the control moves back to the update of the outer loop, i.e., i++.
- After incrementing the value of the loop counter, the condition is checked again, i.e., i<=n.
- If the condition is true, then the inner loop will be executed again.
- This process will continue until the condition of the outer loop is true.

Output:



Nested while loop

Every form of loop that is defined inside the "while" loop is referred to as a "nested while loop."

```
while(condition)
{
while(condition)
{
// inner loop statements.
}
// outer loop statements.
}
```

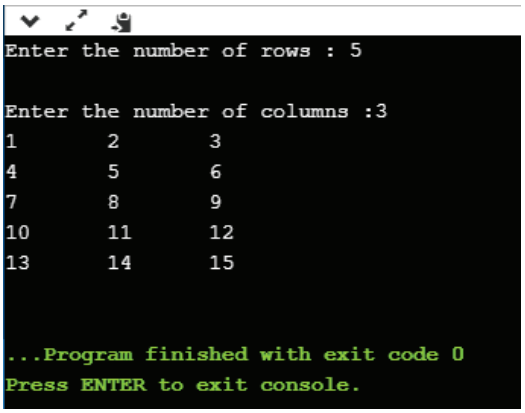
A nested while loop example

```
#include <stdio.h>
int main()
{
    int rows;  // variable declaration
    int columns; // variable declaration
    int k=1; // variable initialization
    printf("Enter the number of rows :");  // input the number of rows.
    scanf("%d",&rows);
    printf("\nEnter the number of columns :"); // input the number of columns.
    scanf("%d",&columns);
     int a[rows][columns]; //2d array declaration
      int i=1;
    while(i<=rows) // outer loop
    {
       int j=1;
      while(j<=columns)  // inner loop
       {
          printf("%d\t",k);  // printing the value of k.
            k++;   // increment counter
            j++;
       }
       i++;
       printf("\n");
    }
}
```

Explanation of the above code.

- We have created the 2d array, i.e., int a[rows][columns].
- The program initializes the 'i' variable by 1.
- Now, control moves to the while loop, and this loop checks whether the condition is true, then the program control moves to the inner loop.
- After the execution of the inner loop, the control moves to the update of the outer loop, i.e., i++.
- After incrementing the value of 'i', the condition (i<=rows) is checked.
- If the condition is true, the control then again moves to the inner loop.
- This process continues until the condition of the outer loop is true.

Output:



Nested do..while loop

Every form of loop that is defined inside the "do..while" loop is referred to as a "nested do.. while loop."

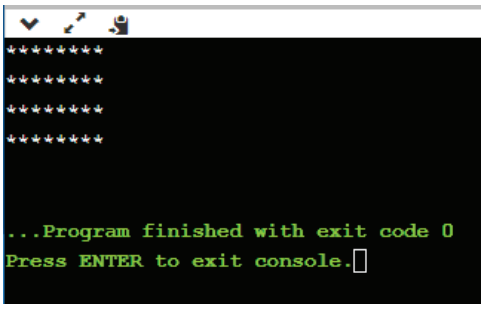
```
do
{
do
{
// inner loop statements.
}while(condition);
// outer loop statements.
}while(condition);
```

Nested do..while loop illustration.

```
#include <stdio.h>
int main()
{
 /*printing the pattern
   ********
   ********
   ********
   ******** */
int i=1;
do        // outer loop
{
  int j=1;
  do      // inner loop
 {
    printf("*");
    j++;
```

```
    }while(j<=8);

    printf("\n");

    i++;

    }while(i<=4);

}
```

Output:



Explanation of the above code:

- First, we initialize the outer loop counter variable, i.e., 'i' by 1.
- As we know that the do..while loop executes once without checking the condition, so the inner loop is executed without checking the condition in the outer loop.
- After the execution of the inner loop, the control moves to the update of the i++.
- When the loop counter value is incremented, the condition is checked. If the condition in the outer loop is true, then the inner loop is executed.
- This process will continue until the condition in the outer loop is true.

break statement in C

There are two applications for the break statement in the C programming language:

1. When a loop encounters the break statement, the loop is instantly broken, and program control moves on to the statement that follows the loop.
2. In the switch statement, it can be used to end a case.

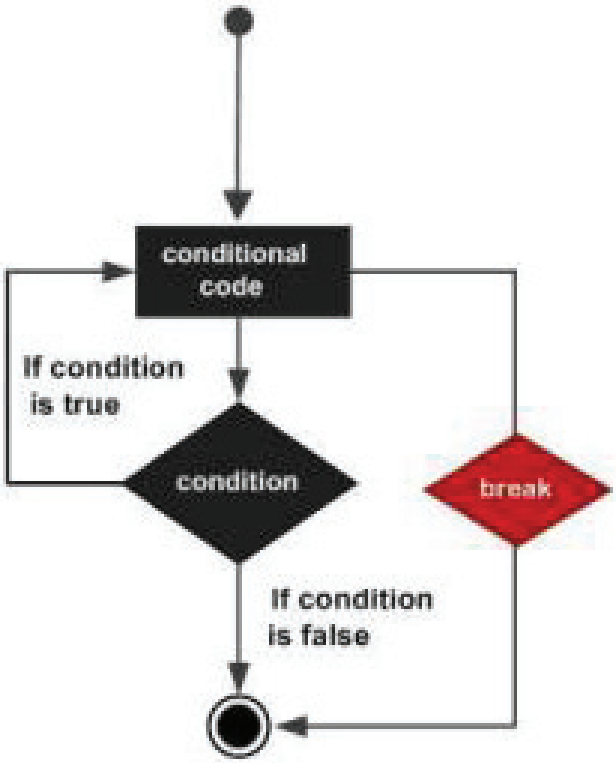
If you are using nested loops, which are loops inside loops, the break statement will stop the innermost loop from running and start the line of code that follows the block.

Syntax

In C, a break statement has the following syntax:

```
break;
```

Flow diagram:



Example

```
#include <stdio.h>

int main ()

{

 /* local variable definition */

 int a = 10;

 /* while loop execution */

 while( a < 20 )

 {

 printf("value of a: %d\n", a);

 a++;

 if( a > 15)

 {

 /* terminate the loop using break statement
*/

 break;

 }

 }

 return 0;

}
```

The following is the outcome of compiling and running the aforementioned code:

```
value of a: 10

 value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15
```

continue statement in C

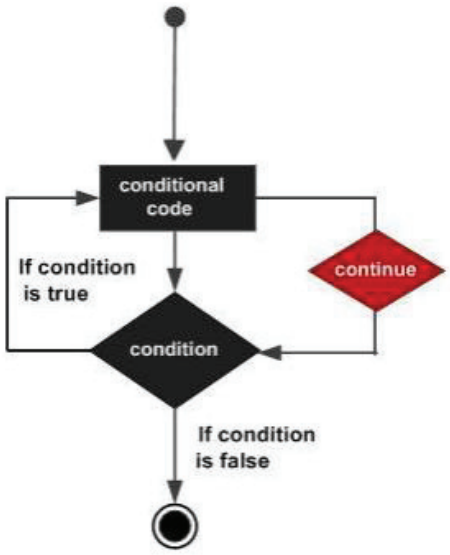
The continue statement and the break statement both serve comparable purposes in the C programming language. However, instead of forcing termination, continue compels the next iteration of the loop to start, skipping any code in between.

The continue statement causes the for loop's conditional test and increment portions to run. The continue statement directs the program control to the while and do...while loops' conditional tests.

Syntax

```
continue;
```

Flow diagram:



Example:

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

```
int main ()
{
/* local variable definition */
int a = 10;
/* do loop execution */
do
{
if( a == 15)
{
/* skip the iteration */
a = a + 1;
continue;
}
printf("value of a: %d\n", a);
a++;
}while( a < 20 );
return 0;
}
```

The following is the outcome of compiling and running the aforementioned code:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Summary

- The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.
- The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.
- In do while loops also the loop execution is terminated on the basis of test condition. The main difference between do while loop and while loop is in do while loop the condition is tested at the end of loop body, i.e do while loop is exit controlled whereas the other two loops are entry controlled loops.
- The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.

UNIT 8

Functions

Learning Objectives

At the end of this unit, you will be able to:

- Define functions and functions prototypes
- Discuss the return statement
- Infer how to declare function from the compiler's viewpoint
- Explain the storage classes and recursion

Introduction

Problem deconstruction is the key to solving problems effectively. Writing huge programmes requires the ability to take an issue and divide it into discrete, manageable parts. This "top-down" approach of programming is implemented in C using the function construct. A program is composed of one or more files, each of which has a main() function and zero or more other functions. The definition of a function is an independent object that cannot be nested. The main() function initiates program execution and may invoke other procedures, including library functions like printf() and sqrt (). Program variables are used for functions to operate, and scope restrictions govern which of these variables are available where in a function.

Function Definition

The C code that specifies what a function does is referred to as the function definition. It should not be confused with a function's declaration. The following is the general form of a function definition:

type function_name(parameter list) { declarations statements }

The function definition's header is made up of everything before the first brace, and its body is made up of everything in between. The declarations are listed in the parameter list, separated by commas. A function definition illustration is

int factorial (int n)

```
{
int i, product = 1;
for (i = 2; i <= n; ++i)
product *= i;
return product;
}
```

The first int instructs the compiler to convert the function's return value to an int if necessary. The declaration int n is included in the argument list. This informs the compiler that the function only accepts a single int parameter. The function is called when an expression like factorial (7) is used. The result is that the function definition's code is executed, using the value 7 for n. Functions thus serve as effective abbreviation systems. Another illustration of a function definition is as follows:

```
Void wrt_address(void)
{
    Printf("%s\n%s\n%s\n%s\n%s\n\n"
"    **************************
"    **      SANTA CLAUS          *"
"    **      NORTH POLE           *""
"    **      EARTH                *""
"    **************************)
)
```

The first void indicates to the compiler that the function returns nothing, while the second void indicates that the function does not accept any arguments. This phrase

```
wrt_adresses()
```

initiates the call to the function. For instance, we can write to call the method three times.

```
for (i = 0; i < 3; ++i) wrt_address 0 ;
```

In a function definition, the type of the function is listed first. If there is no value returned, the type is void. If the type is something other than void, the function will transform the value it returns, if necessary, to this type. A list of parameter declarations in parentheses is placed after the function's name. When the function is called, the parameters serve as placeholders for the values that are supplied. In order to emphasise that these arguments are placeholders, they are occasionally referred to as the formal parameters of the function. Declarations may also be found in the function's body, which is a block or compound statement. These are some illustrations of function definitions:

```
void nothing(void) { }
double twice(double x)
{
return (2.0 * x);
}
int all_add(int a, int b)
{
int c;
return (a + b + c);
}
```

A function's default type is int if the function type is not specified in the definition. The final function definition, for instance, might be provided by

```
all_add(int a, int b) {
```

To clearly state the function type is acceptable programming practice, nevertheless.

A variable is said to be "local" to a function if it is defined in the function's body. Other variables outside of the function can be defined. They are referred to as "global" variables. One of them is

```
#include <stdio.h>

int a = 33; /* a is external and initialized to 33
*/

int main(void)

{

int b = 77;

printf("a = %d\n", a);

printf("b = %d\n", b);

return 0;

}
```

In traditional C, the function definition has a different syntax. The variables in the parameter list are declared immediately following the parameter list itself and just before the first brace. One of them is

```
void f(a, b, c, x, y)

int a, b, c;

double x, y;

{
```

It doesn't matter what order the parameters are declared in. A pair of empty parenthesis is used if there are no parameters. Both this conventional syntax and the more recent syntax are supported by ANSI C compilers. As a result, an ANSI C compiler can still build conventional code.

Writing programmes as collections of numerous functions is advantageous for a number of reasons. Writing a precise tiny function that only performs one task is easier. The authoring and debugging processes are simplified. Also, such a grammar is simpler to update or maintain. Just changing the necessary set of functions and assuming the rest of the code will function properly is possible. Little functions are also frequently highly readable and self-documenting. A smart programming heuristic is to write each function's code so that it can fit on a single page.

The return Statement

An expression might or might not be present in the return statement.

```
return_statement ::= return; return expression;

Many instances are

return;

return ++a;

return (a ,/, b);
```

Although it's optional, parentheses can be used to encapsulate the returned expression.

When a return statement is discovered, the function's execution is completed, and control is sent back to the calling environment. The caller

environment additionally receives the value of any expressions provided in the return statement. Also, based on the function's specification, this value will be modified as necessary to reflect the function's type.

```
float f(char a, char b, char c)
{
    int i;
    .....
return i; /*value returned will be converted to a float*/
}
```

A function may contain zero or more return statements. When the closing brace of the body is encountered, if there is no return statement, control is returned to the caller environment. "Falling off the end" is the term used for this. The use of two return statements is demonstrated by the function definition that follows:

```
double absolute_value(double x)
{
if (x >= 0.0)
return x;
else
return -x;
}
```

A function may return a value, but a program is not required to use it.

```
while ( ..... ) {
getchar(); /* get a char, but do nothing with it */
c = getchar 0 ; /* c will be processed */
.....
}
```

Function Prototypes

Functions should be declared before use. The function prototype, a new syntax for function definition, is provided by ANSI C. A function prototype tells the compiler how many and what kind of inputs to pass to the function, as well as what kind of 0 result the function should return. One of them is

```
double sqrt(double);
```

This instructs the compiler that the sqrt() method accepts a single double-type argument and returns a double. An abstract representation of a function prototype is

```
type function_name ( parameter type list) ;
```

A comma-separated list of kinds normally makes up the parameter type list. Identifiers are not required and have no impact on the prototype. As an illustration, the function prototype

```
void f(char c, int i); is equivalent to
void f(char, int);
```

Identifiers like c and I that exist in parameter type lists for function prototypes are not used by the compiler. They provide documentation for the programmer and other code readers. The keyword void is utilised whenever a function does not take any arguments. The keyword void is also used if the function has no output.

When a function takes more than one argument, the ellipses (...) are utilised. Have a look at the printf() function prototype in the standard header file stdio.h, for instance.

The compiler is able to undertake a more thorough code inspection thanks to function prototypes. Moreover, whenever possible, correct coercion is used to force values supplied to functions. For instance, if the function prototype for sqrt() has been defined, the function call sqrt(4) will result in the desired outcome. Because the compiler is aware that sqrt() needs a double, the int value 4 will be upgraded to a double and the appropriate value will be returned.

Argument type lists are not supported in function declarations in traditional C. For instance, the sqrt() function declaration is as follows:

```
double sqrt(); /* traditional C style */
```

Function prototypes are preferable even if ANSI C compilers will support this coding style. The function call sqrt (4) won't produce the right result with this declaration.

Function Declarations from the Compiler's Viewpoint

Function declarations are produced in a variety of methods for the compiler, including function definition, function invocation, explicit function declarations, and function prototypes. The compiler assumes a default declaration of the form if a function call, such as f(x), occurs before any of its declarations, definitions, or prototypes.

```
Int f();
```

About the function's parameter list, nothing is assumed. Now imagine that the definition of the following function comes first:

```
int f(x)        /* traditional C style */

double x;

{
```

The compiler receives both a declaration and a definition from this. However, the parameter list is not presumed again. It is the obligation of the programmer to only send a single double-type argument. Given that 1 is of type I nt, not double, it is assumed that a function call like f (1) will fail. Now imagine that we use an ANSI C style declaration instead:

```
int f(double x)        /* ANSI C style */

{
```

The argument list is also now known to the compiler. A function call like f(l) should perform correctly in this situation. A double will be created when an int is supplied as an argument.

A function declaration's special case is a function prototype. Giving either the function definition (ANSI C style) or the function prototype, or perhaps both, before a function is utilised, is a good programming practice. The fact that standard header files contain function prototypes is a key justification for their inclusion.

Storage Classes

Storage The features of a variable or function are described using classes. We can trace the existence of a particular variable during the

lifetime of a program thanks to these properties, which mainly include scope, visibility, and lifetime.

The four storage classes used by the C programming language are:

Storage classes in C

Storage Specified	Storage	Initial value	Scope	LIfe
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

1. **auto**: This is the default storage class for all variables specified inside a function or a block. As a result, when developing C language applications, the keyword auto is rarely utilised. Only the block or function in which they were declared can access auto variables (which defines their scope). In the parent block or function where the auto variable was declared, they can, of course, be accessed within nested blocks. But, using the idea of pointers presented here, the variables can also be accessed outside of their intended scope by referring to the particular memory location where they are stored. They always have a garbage value when they are declared.

2. **extern**: The phrase "external storage class" simply means that the definition of the variable is found outside of the block in which it is used. In essence, the value can be changed or rewritten in another block after being assigned to it in the first block. An extern variable is just a global variable that is initialised with a valid value where it is defined in order to be used elsewhere. It is reachable from any block or function. A normal global variable can also be made extern by prefixing its declaration or definition with the term "extern" in any function or block. This simply means that we are only using/accessing the global variable and not initialising a new variable. Using extern variables enables them to be accessed from two separate files that are a part of a huge application, which is their primary use.

3. **static**: Static variables, which are frequently used while building C programmes, can be declared using this storage type. As soon as they leave their scope, static variables continue to hold their value. Static variables therefore maintain the value of their most recent use within their scope. So, we can conclude that they receive a single initialization and remain valid till the program is finished. Because they are not re-declared, no new memory is allocated as a result. The function to which they were specified is the only area of their application. Anything in the program can access global static variables. They are given the value 0 by the compiler by default.

4. **register**: Register variables with the same capability as auto variables are declared by this storage class. The main difference is that the compiler only tries to store these variables in the microprocessor's register if a free registration is available. As a result, using register variables during program execution is substantially quicker than using variables that are kept in memory. In

the absence of a free registration, they are just stored in memory. The register keyword is often used to declare a few variables that a program will utilise repeatedly, which reduces the amount of time the program will take to run. The fact that we cannot use pointers to get the address of a register variable is both significant and intriguing in this context.

The following syntax must be used to specify a variable's storage class:

Syntax:

storage_class var_data_type var_name;

The syntax for functions is the same as that for variables above. For more information, check at the following C example:

```
// A C program to demonstrate different storage
// classes
#include <stdio.h>
// declaring the variable which is to be made extern
// an initial value can also be initialized to x
int x;
void autoStorageClass()
{
printf("\nDemonstrating auto class\n\n");
    // declaring an auto variable (simply
    // writing "int a=32;" works as well)
    auto int a = 32;
    // printing the auto variable 'a'
    printf("Value of the variable 'a'"
           " declared as auto: %d\n",
           a);
    printf("--------------------------------");
}
void registerStorageClass()
{
printf("\nDemonstrating register class\n\n");
    // declaring a register variable
    register char b = 'G';
    // printing the register variable 'b'
    printf("Value of the variable 'b'"
    " declared as register: %d\n",
    b);
    printf("--------------------------------");
}
void externStorageClass()
{
    printf("\nDemonstrating extern class\
n\n");
    // telling the compiler that the variable
    // x is an extern variable and has been
    // defined elsewhere (above the main
    // function)
```

```
        extern int x;
        // printing the extern variables 'x'
        printf("Value of the variable 'x'"
                " declared as extern: %d\n",
                x);
        // value of extern variable x modified
        x= 2;
        // printing the modified values of
        // extern variables 'x'
        printf("Modified value of the variable 'x'"
                " declared as extern: %d\n",
                x);
        printf("------------------------------");
    }
    void staticStorageClass()
    {
        int i = 0;
        printf("\nDemonstrating static class\
n\n");
        // using a static variable 'y'
    printf("Declaring 'y' as static inside the loop.\n"
        "But this declaration will occur only"
        " once as 'y' is static.\n"
        "If not, then every time the value of 'y'"
        "will be the declared value 5"
                " as in the case of variable 'p'\n");
        printf("\nLoop started:\n");
        for (i = 1; i < 5; i++) {
                // Declaring the static variable 'y'
                static int y = 5;
                // Declare a non-static variable 'p'
                int p = 10;
        // Incrementing the value of y and p by 1
                y++;
                p++;
        // printing value of y at each iteration
                printf("\nThe value of 'y', "
                        "declared as static, in %d "
                        "iteration is %d\n",
                        y, i);
        // printing value of p at each iteration
    printf("The value of non-static variable 'p', "
                "in %d iteration is %d\n",
                        p, i);
        }
        printf("\nLoop ended:\n");
        printf("------------------------------");
    }
    int main()
```

```
{
    printf("A program to demonstrate"
            " Storage Classes in C\n\n");
    // To demonstrate auto Storage Class
    autoStorageClass();
    // To demonstrate register Storage Class
    registerStorageClass();
    // To demonstrate extern Storage Class
    externStorageClass();
    // To demonstrate static Storage Class
    staticStorageClass();
    // exiting
    printf("\n\nStorage Classes
demonstrated");
    return 0;
}
```

Output:

```
...ster: 71
------------------------------
Demonstrating extern class
Value of the variable 'x' declared as extern: 0
Modified value of the variable 'x' declared as
extern: 2
------------------------------
Demonstrating static class
```

Declaring 'y' as static inside the loop.

But this declaration will occur only once as 'y' is static.

If not, then every time the value of 'y' will be the declared value 5 as in the case of variable 'p'

Loop started:

The value of 'y', declared as static, in 6 iteration is 1

The value of non-static variable 'p', in 11 iteration is 1

The value of 'y', declared as static, in 7 iteration is 2

The value of non-static variable 'p', in 11 iteration is 2

The value of 'y', declared as static, in 8 iteration is 3

The value of non-static variable 'p', in 11 iteration is 3

The value of 'y', declared as static, in 9 iteration is 4

The value of non-static variable 'p', in 11 iteration is 4

Loop ended:

```
------------------------------
Storage Classes demonstrated
```

Recursion

Recursion happens when a function calls a copy of itself to work on a smaller issue. A recursive function is any function that calls

itself, and recursive calls are calls performed by recursive functions. Recursion involves a large number of recursive calls. It is essential to enforce a recursion termination condition as a result. Recursive code is more cryptic than iterative code, although being shorter.

Recursion cannot be used for every problem, but it is more beneficial for jobs that can be broken down into related subtasks. Recursion can be used to solve sorting, searching, and traversal issues, for instance.

As function calls always incur cost, iterative solutions are typically more efficient than recursive ones. Each issue that may be resolved recursively also lends itself to iterative resolution. Yet, some issues, like the Tower of Hanoi, the Fibonacci sequence, factorial finding, etc., are better handled via recursion.

In the example that follows, recursion is used to get a number's factorial.

```
#include <stdio.h>

int fact (int);

int main()

{

int n,f;

printf("Enter the number whose factorial you want to calculate?");

scanf("%d",&n);

f = fact(n);

printf("factorial = %d",f);

}

int fact(int n)

{

if (n==0)

{

return 0;

}

else if ( n == 1)

{

return 1;

}

else

{

return n*fact(n-1);

}

}
```

Output:

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

The following diagram helps us understand the recursive method call in the program we just looked at:

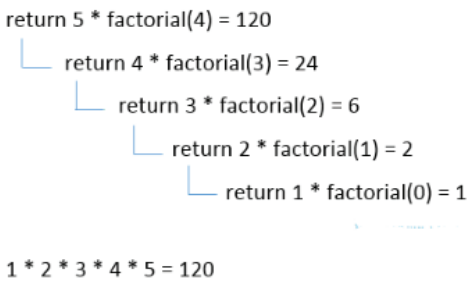


Figure: Recursion.

Recursive Function

Recursive functions carry out tasks by breaking them down into smaller ones. The function has a defined termination condition, which is met by a certain subtask. The recursion ends at this point, and the function returns the ultimate outcome.

The base case is the circumstance in which the function does not recur; in contrast, the recursive case is the circumstance in which the function repeatedly invokes itself to carry out a subtask. It is possible to write all recursive functions in this style.

The following pseudocode can be used to write any recursive function.

```
if (test_for_base)
{
return some_value;
}
else if (test_for_another_base)
{
return some_another_value;
}
else
{
// Statements;
recursive call;
}
```

Example of recursion in C

Here is an illustration of how to determine the nth term in the Fibonacci series.

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
int n,f;
printf("Enter the value of n?");
scanf("%d",&n);
f = fibonacci(n);
printf("%d",f);
}
int fibonacci (int n)
{
if (n==0)
{
return 0;
}
else if (n == 1)
{
return 1;
}
else
{
return fibonacci(n-1)+fibonacci(n-2);
}
}
```

Output:

```
Enter the value of n?12
144
```

Memory allocation of Recursive method

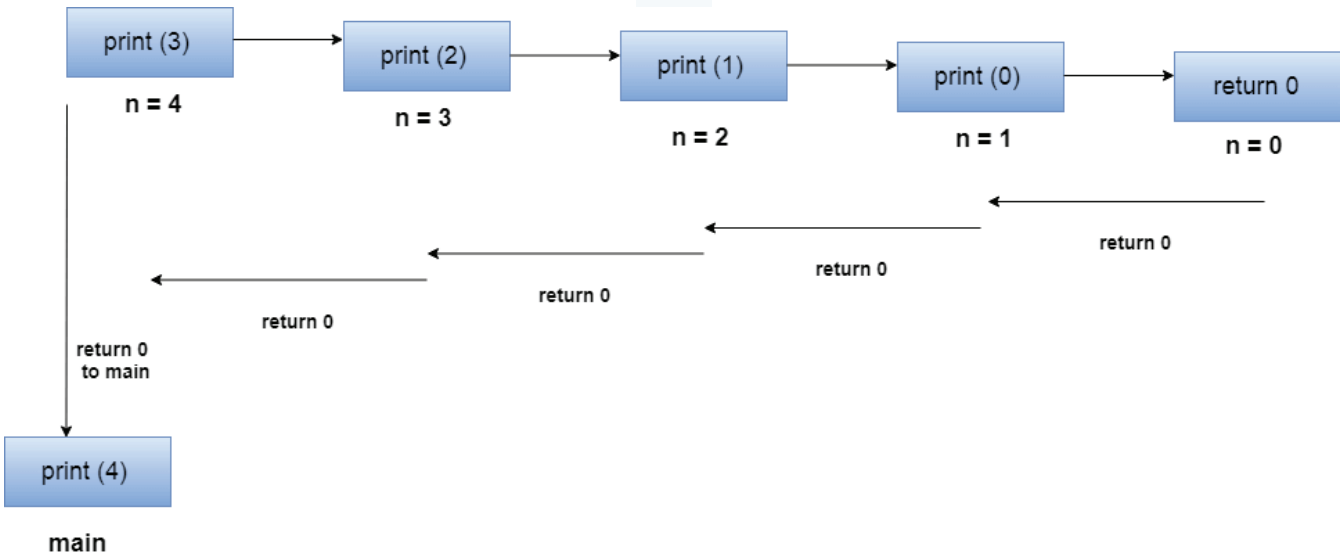
That method is duplicated in memory with each recursive call. The copy is erased from memory after the procedure returns some data. A distinct stack is kept for each recursive call since all variables and other items defined inside functions are saved in the stack. The stack is deleted after the associated function returns its value. Resolving and tracking the values at each recursive call requires a great deal of complexity in recursion. As a result, we must keep the stack up to date and monitor the values of the variables it contains.

To comprehend how recursive functions allocate memory, let's look at the example below.

```
int display (int n)
{
if(n == 0)
return 0; // terminating condition
else
{
printf("%d",n);
return display(n-1); // recursive call
}
}
```

Explanation:

With n set to 4, let's examine this recursive function. First, all stacks are maintained active and show the appropriate value of n until n = 0. Each stack is destroyed one at a time upon attaining the termination condition by returning 0 to the caller stack. Consider the illustration below for further information on the recursive functions' stack traces.



Stack tracing for recursive function call

Summary

- In C, the function construct is used to implement this "top-down" method of programming. A program consists of one or more files, with each file containing zero or more functions, one of them being a main() function. Functions are defined as individual objects that cannot be nested. Program execution begins with main(), which can call other functions, including library functions such as printf() and sqrt().
- Functions operate with program variables, and which of these variables is available at a particular place in a function is determined by scope rules.
- Functions should be declared before they are used. ANSI C provides for a new function declaration syntax called the function prototype. A function prototype tells the compiler the number and type of arguments that are to be passed to the function and the type 0 the value that is to be returned by the function.
- Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.
- Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion.

UNIT 9

Structures and Unions

Learning Objectives

At the end of this unit, you will be able to:

functions and functions prototypes

- Define the concept of Structures
- Discuss how to use structures with functions
- Describe about unions

Introduction

C is a language that is simple to expand. It can be improved by adding functions that are kept in libraries and macros that are kept in header files. It can also be expanded by specifying data types that are constructed from the fundamental types. As an example, the array type is a derived type that is used to represent homogeneous data. On the other hand, the structural type is used to represent heterogeneous data. A structure's members are its individual parts, and they each have a distinct name. Because the parts of a structure could be of various types, a programmer might create data aggregates that are suitable for a particular application.

Structures

A group of member elements and the accompanying data types make up a structure in the C programming language. As a result, the definition of a variable of a structural type is the name of a group of one or more members, who may or may not be of the same data type. In programming, a structural data type is referred to as a record data type, and its components are called fields.

Declaring and Accessing Structure Data

Like with any other data type, we must be able to declare variables of that data type. Particularly for structures, we must specify the names and types of all the structure's fields. As a result, we must define variables of that type and specify the quantity and type of fields in the form of a template in order to construct a structure.

We provide the following example: a program that maintains temperatures in both fahrenheit and celsius. The variable temp, which will be used to maintain the same temperatures in both celsius and fahrenheit, requires two fields that are both integers.

The temperature of one field will be denoted by the letters "ftemp" for fahrenheit and "ctemp" for celsius. The program, which is shown in Figure (Code for simple structure program), reads a temperature from the temp variable's ftemp field, converts it to celsius using the f to c() function, and then stores the result in the ctemp field.

```
/* File: fctemp.c

Program read temperature is Fahrenheit, converts to Celsius, and maintain the equivalen values in a variable of structure type*/

#include <stdio.h>

Main()

{     struct trecd{

                float ftemp;

                floatctemp;

} temp;

double f_to_c(double f)

char c;

printf("***Temprature – Degree F and C***\ n\n");

printf("Enter temperature in degree F : ")

scanf("%f",&temp.ftemp);

temp.ctemp = f_to_c(temp.ftemp);

printf("Temp in degree F =%3. If\n",tenp. ftemp);"

printf("Temp in degree F =%3. If\n",tenp. ftemp);"

}

/* This converts degree F to degree c*/

double f_to_c(doublef)

{

Return((f -32.0)*5.0/9.0);

}
```

Looking at this program, we can observe that the declaration statement for the variable temp's structure type is as follows:

```
struct trecd {

float ftemp;

float ctemp;

} temp;
```

This phrase starts with the word "struct," then goes on to describe the structure's template before ending with the variable name. In our example, a tag (or name), trecd, identifies the template and is followed by a set of field declarations that are wrapped in brackets. The tag is optional. Without having to once more precisely explain the fields, the tag can be used to refer to this structural template within its scope. The bracketed list specifies the type and identification of each field in the structure. Our example shows that this structure has two fields, ftemp and ctemp, both of type float.

The memory cells designated for the variable temp are displayed in Figure (Structure variable temp in memory).

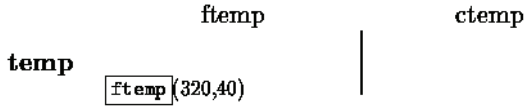


Figure: Structure variable temp in memory.

The field and ctemp are the names of two float cells that have been assigned. The variable name temp refers to the full block of memory. Structure declarations otherwise have the same scope as any other variable declaration, such as an int declaration, and are treated the same way.

The variable name (in this case, temp) is qualified with the "dot" operator (.) and the field name to access the data in a structure:

```
temp.ftemp
temp.ctemp
```

In general, the syntax for accessing a member of a structure is:

_identifier>.<member_identifier>

The same way that other variables are used in programmes, members of structural variables can also be used. The float cell's address, temp. ftemp, is provided as a parameter to scanf() in the aforementioned procedure main() as &temp. ftemp. Since the dot operator takes precedence over the address operator in this case, parenthesis are not required. The argument will be preserved in the temp.ftemp cell, which will also save the numerical value obtained by scanf(). The rest of the program is straightforward. We assigned the double value that was produced by the function f to c to the variable temp.ctemp and printed the results on it.

- Sample Session:***Temperatures - Degrees F and C***
- ***Temperatures - Degrees F and C***
- Enter temperature in degrees F : 78
- Temp in degrees F = 78.0

Temp is degrees C = 25.6

As we've already mentioned, a structural variable's members can be of several sorts. For instance:

struct {

char name[26];

int id_number;

} student;

which generates the structure variable student with the two fields name, a character string, and id, an integer. There is enough contiguous memory allocated to the variable student to store both fields. To find out how much storage a structure has been given, use the sizeof operator. (Be careful that the overall size of a structure variable may not be the same as the sum of the sizes for the fields due to restrictions affecting memory alignment that may vary from computer to computer.

For instance, the start of a memory allocation for an integer could need to be at a machine word boundary, such an even byte address. Due to these alignment requirements, the size of a structural variable may be slightly bigger than the sum of the field sizes.

The identifiers for the field names are specific to variables of that structure type. Different

structure types can have fields specified with the same identifier, but these fields are separate cells that can only be accessed by using the correct structure variable name followed by the field name. Moreover, only the field names that are listed in the structure template can qualify a variable name.

A field name cannot be used alone; it always needs to be followed by the name of an appropriate structure variable. Consider the following structural variable declarations:

```
struct {

char f_name[10];

char m_inits[3];

char l_name[20];

int id_no;

int b_month;

int b_day;

int b_year;

} person, manager;

struct {

int id_no;

float cost;

float price;

} part;
```

In this case, the two variables person and manager are structures with seven fields, some of which are numbers and others of which are strings. Person.id_no and Manager.id_no are independent storage cells in this instance since there are two separate instances of the template that are allocated. We have also defined a part variable, whose template contains the name of the id no field. However part.id_no can also access this other storage location. Nevertheless, it is NOT allowed to refer to the person's cost field (person.cost) or the _day (part.b_day) when using these declarations. Similar to this, referencing f_name or price is prohibited in the absence of a variable name of the proper type. Here are a few legitimate uses of structures:

```
part.id_no = 99;

part.cost = .2239;

if (strcmp(person.f_name, "Helen") == 0)

printf("Last name is %s\n", person.l_name);

printf("This is the cost %d\n",part.cost);

part.price = part.cost * 2.0;
```

As long as the variables are of the same structure type, the only permissible operations that can be performed on a structure variable are accessing its members, copying, or assigning it as a unit, for example:

```
manager = person;
```

Using Structure Tags

In order to declare a structure variable, you must first describe the structure's template and then declare variables of that structure type. These two actions could alternatively be carried out in a program as independent statements. That is, simply define a structure template with a tag, but without declaring any variables. Afterwards, declare variables for the structure type that the

tag indicated. Say, for illustration, the following:

```
struct stdtype {

char name[26];

int id_number;

};
```

provides a template with the tag "stdtype" for a particular structure type. (Note the semicolon for the declaration that follows the statement.) As no variables are declared in such a declaration, RAM is NOT allocated; instead, a template for variables to be declared later is defined. We may then declare stdtype structure variables like: within the scope of the tag declaration.

```
struct stdtype x, y, z;
```

The three variables x, y, and z, all of which are of type structure stdtype and so fit the earlier stated template, are allotted memory by this declaration. Here are some more instances of structure tags and variable declarations:

```
/* named structure template, no variables declared */

struct date {

int month;

int day;

int year;

};

/* named structure template and a variable declared */

struct stu_rec {

char name[30];

char class_id[3];

int test[3];

int project;

int grade;

} student;

struct stu_rec ee_stu, me_stu;

struct date today, birth_day;
```

The key benefit of separating the definition of the template from the declaration of variables is that the template only needs to be declared once and can then be used to declare variables in several locations. When we pass structures to function in the paragraphs below, we will realise how useful this is.

Hence, a structure declaration typically includes the form shown below:

```
item struct [<tag_identifier>] {

item struct [<tag_identifier>] {

<type_specifier> <identifier>;

<type_specifier> <identifier>;

} [<identifier>[, <identifier>
```

where <tag identifiers> and the variable are both optional. Further variables of the structure type may be declared after the definition of a template by:

```
<tag_identifier> <identifier>[, <identifier>
```

Any legal C type, including scalar data types (int, float, etc.), arrays, and structures, may be

used for the structure's fields. Since nested structure types can also be defined, it follows that:

```
struct inventory {
int item_no;
float cost;
float price;
struct date buy_date;
};
struct car_type{
struct inventory part;
struct date ship_date;
int shipment;
} car;
```

Here, the ship date field of the car type structure is a date structure unto itself, and the part field is an inventory structure containing item no, cost, etc. fields, as well as another date structure inside (from above). The members of nested structures can be accessed using dot operators, which should be used sequentially from left to right (the grouping for the dot operator is from left to right). Thus:

```
car.ship_date.month = 5;      /* Lvalue is (car.ship_date).month */
    car.part.buy_date.month = 12;
```

The month field of the variable car's ship date field and the month field of the variable car's buy_date field, respectively, are the subjects of these assignments.

It is common to refer to both structure tags and structure type variables as structures. As a result, we may argue that the variable today is a structure and that date is a structure. Usually, the context makes it clear if a structure tag or a variable of a structure type is intended. Nonetheless, we shall refer to the templates themselves—i.e. tags—as "structures" for the most part, and we will declare that certain variables have a structure type. So, the term "date" refers to a structure, and the term "today" refers to a variable of the structure type, namely the type "struct date."

Structures can be initialised in declarations just like other data types by stating constant values for each member of the structure inside of brackets. Similar to an array, the initializers for structure members are separated by commas. A struct inventory item might be declared as follows:

```
struct inventory part = { 123, 10.00, 13.50 };
```

which sets member's part no to 123, cost to 10.00, and price to 13.50 dollars. Another illustration of a label item declaration is:

```
struct name {
char f_name[10];
char m_inits[3];
char l_name[20];
};
struct address {
char street[30];
char city[15];
char state[15];
```

```
    int zip};
    };
    struct label {
    struct name name;
    struct address address;
    };
    struct label person = { {"Jones", "John", "Paul"},
    {"23 Dole Street", "Honolulu", "Hawaii", 96822}
};
```

Each of the two parts of the structure, label is a structure. Name, the first member, has three members, and address, the second member, has four. Each member structure's initialization is properly nested.

Using Structures with Functions

Like scalar variables, structure variables can be supplied as arguments and returned from functions. Let's look at an illustration that reads and publishes a part's data record. The part number, cost, and retail price are included in the record. Figure displays the code to read and output a single part structure (Code for reading and printing a single part)

```
    #include <stdio.h>
    Struct inventory{
    int part_no;
    float cost;
    float price;
    };
    struct inventory read_part(void);
    void print_part(struct inventort part);
    main( )
    {     struct inventory item;
          printf("***Part Inventory Data***\n\n");
    item = read_part( )
    };
    {
    printf("Part no. =%d, Cost = %5.2f, Reatil price
=%5.2f\n, part.part_no, part.cost, part.price);
    struct inventory read_part(void)
    {     int n;
    float x;
    struct inventory part;
    printf("Part Number: ")
    scanf("%d", &x);
    part.part_no = n;
    printf("Cost: ");
    scanf("%f", &x);
    part.cost = x;
    printf("Price: ");
    scanf("%f", &x);
    part.price = x;
    return part;
    }
```

- You'll see that we've declared the inventory structure template at the top of the source file. The scope of an external declaration is the whole file that follows the declaration. As this structural tag is used by every function in the file, every function must be able to see the template. In order to read data into a structure and return it to be assigned to the variable item, the driver executes read part(). Then, it executes print part() with the argument item, printing each field's value as it is called. The program is simple to follow. Below is an illustration of a session:***Part Inventory Data***
- Part Number: 2341
- Cost: 12.5
- Price: 15
- Part no. = 2341, Cost = 15.00

The consistent use of tags and functions is facilitated by external declarations of structure prototypes and templates. As a rule, we shall externally declare structure templates, typically at the top of the source file. External structure tag declarations may occasionally be contained in a separate header file that is subsequently included in the source file using the include directive.

We can see from this example that utilising structures with functions is the same as using any scalar data type, such as int. But let's think about what actually occurs when the application runs. Memory is allocated for all of the function read_part(local)'s variables, including the struct inventory variable part, when the function is invoked. The appropriate field of the component, accessible with the dot operator, receives each data item as it is read and places it there. After being returned to main(), the value of part is subsequently allocated to the variable item. The return expression's value is transferred back to the calling function, just like it would for a scalar data type. This is a structure, thus every field within the structure is replicated. This isn't too bad for our inventory structure since it only uses two floats and one integer. More values would need to be transferred if the structure were substantially larger, possibly containing nested structures and arrays.

The call to print part has the same effect (). The function in this instance is given a structure for an inventory. As you may remember, in C, the value of each argument expression is transferred from the calling function into the cell corresponding to the parameter of the called function. Once more, this may not be the most effective method of passing data to functions for huge structures. We see a solution to this issue in the next section.

Initialization of Structures

Members of structures cannot be initialized by declaration. The compilation of the following C program, for instance, fails.

```
struct Point

{

int x = 0; // COMPILER ERROR: cannot initialize members here

int y = 0; // COMPILER ERROR: cannot initialize members here

};
```

Simple lack of memory allocation when a datatype is declared is the cause of the aforementioned error. Only when variables are created is memory allocated.

Curly braces, or "{}" can be used to initialize structure members. The initialization that follows, for instance, is acceptable.

```
struct Point
{
int x, y;
};
int main()
{
// A valid initialization. member x gets value 0 and y
// gets value 1. The order of declaration is followed.
struct Point p1 = {0, 1};
}
```

Unions

A user-defined data type called a union is a grouping of various variables of various data types stored in the same memory address. The union can also be thought of as having numerous members, but at any given time, only one of those members can hold a value.

The user-defined data type union shares the same memory address with structures, in contrast.

Let's use an example to better grasp this.

```
struct abc
{
int a;
char b;
}
```

The user-defined structure in the aforementioned code has two members, namely "a" of type int and "b" of type character. The addresses of "a" and "b" are different, as we discovered when we checked them. As a result, we draw the conclusion that the structure's members do not share the same memory address.

After defining the union, we found that its definition is essentially the same as that of a structure, with the exception that the union keyword is used to specify the union data type and the struct keyword to specify a structure. The data elements "a" and "b" that make up the union are identical when we examined the addresses of the two variables. It suggests that each individual in the union has access to the same region of memory.

Deciding the size of the union

The largest member of the union determines the size of the union as a whole.

Let's use an illustration to clarify.

```
union abc{
int a;
char b;
```

```
float c;

double d;

};

int main()

{

printf("Size of union abc is %d", sizeof(union abc));

return 0;

}
```

As is common knowledge, an int is 4 bytes, a char is 1 byte, a float is 4 bytes, and a double is 8 bytes in size. A total of 8 bytes will be allocated in the RAM because the double variable uses the most memory of the four variables. As a result, 8 bytes would be the output of the aforementioned program.

Why do we need C unions?

To comprehend the necessity of C unions, consider one illustration. Consider a shop that sells two things:

Books

Shirts

Shop owners desire to keep a record of the two items described above as well as the pertinent data. For instance, shirts feature colour, design, size, and price whereas books have title, author, number of pages, and price. In both items, the 'price' property is present. How would the store owner store the records after deciding to store the properties?

They initially chose to arrange the records in the manner depicted below:

```
struct store

{

double price;

char *title;

char *author;

int number_pages;

int color;

int size;

char *design;

};
```

All the products that the business owner wants to store are included in the structure above. The aforementioned structure is entirely functional, however only the price is shared by both goods, while the rest are separate. Books have attributes like price, *title, *author, and number_pages, whereas shirts have properties like color, size, and *design.

Let's see how we can get to the structure's members.

```
int main()

{

struct store book;

book.title = "C programming";

book.author = "Paulo Cohelo";

book.number_pages = 190;
```

```
book.price = 205;

printf("Size is : %ld bytes", sizeof(book));

return 0;

}
```

We have established a variable of type store in the code above. Title, Author, Number_Pages, and Price variables have values assigned to them, however the book variable lacks attributes like size, colour, and design. Hence, it wastes memory. The aforementioned structure would be 44 bytes in size.

Unions allow us to save a lot of space.

```
#include <stdio.h>

struct store

{

double price;

union

{

struct{

char *title;

char *author;

int number_pages;

} book;

struct {

int color;

int size;

char *design;

} shirt;

}item;

};

int main()

{

struct store s;

s.item.book.title = "C programming";

s.item.book.author = "John";

s.item.book.number_pages = 189;

printf("Size is %ld", sizeof(s));

return 0;

}
```

We have established a variable of type store in the code above. When unions were used in the code above, the variable's biggest memory footprint would be taken into account while allocating RAM. The program mentioned above outputs 32 bytes. When it comes to structures, we got a size of 44 bytes, whereas when it comes to unions, we got a size of 44 bytes. Hence, 44 bytes save a lot of memory space since they are larger than 32 bytes.

Summary

- In C, a **structure** is a derived data type consisting of a collection of member elements and their data types. Thus, a variable of a structure type is the name of a group of one or more **members** which may or may not be of the same data type.

In programming terminology, a structure data type is referred to as a **record data type** and the members are called fields.

- In a program, members of a structure variable may be used just like other variables.
- Structure variables may be passed as arguments and returned from functions just like scalar variables.
- Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

UNIT 10

Input/Output

Learning Objectives

At the end of this unit, you will be able to:

functions and functions prototypes

- Discuss the output function printf()
- Describe the input function scanf()
- Explain the functions fprintf(), fscanf(), sprint(), and sscanf()
- Recognize the functions fopen() and fclose()

Introduction

The Output Function printf()

When we say Input, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say Output, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

When completing calculations, you typically wish to display your findings (sometimes known as "the answer") on the screen. The printf function, which enables you to display messages on the computer screen, is part of the standard input and output library, which is installed by default on all computers with a C compiler. The line below must be the first line in your program to access the standard input and output library:

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

The printf function's common format is

printf(control string, other arguments);

With two exceptions, the control string will be written exactly as it appears in the program when it is executed. Each conversion

specification will initially be replaced by the value supplied in the other arguments part of the printf statement by the computer. Second, distinctive, difficult-to-print characters will be used in place of escape sequences. It's crucial to keep in mind that the control string contains both conversion specifications and escape sequences.

Fill-in-the-blank style placeholders in the control string are the first sort of replacement, known as conversion specifications. A conversion specification has an a% at the start and a conversion character at the end. Your choice of converting character will depend on the kind of content you want to see on the screen. The table below is a list of the various conversion characters. (This table can also be found on page 494 of your work, A Book on C.)

Conversion Character.	How the corresponding argument is printed .
c	as a character
d	as a decimal integer
e	as a floating point number; example 7.123000e+00
f	as a floating point number; example 7.123000
g	in the e-format or f-format, whichever is shorter
s	as a string

Consider the following code as an illustration:

```
#include <stdio.h>

int main(void)

{

int   i=97;

float x=3.14;

printf("My favorite numbers are %d and %f.\n",i,x);

return 0;

}
```

In this illustration, we wish to print an int value that is contained in the variable i. As "d" is the conversion character for decimal integers, we should select the conversion specification%d. As "f" is the conversion character for floating point values, we wish to utilise the conversion specification%f since the second value is of type float. Here is an example of the code's output after compilation and execution:

My favorite numbers are 97 and 3.140000.

What happens if we swap out the printf statement above with the one below? Characters are "secretly" stored as integers.

printf("My favorite numbers are %c and %f.\n",i,x);

The conversion character c has been used in place of the conversion character d, which shows integers (which displays characters). Here is what we observe after compiling and running the program:

My favorite numbers are a and 3.140000.

Computers are obedient despite their lack of intelligence (is the letter "a" your favourite number?). We instructed the computer to display the character 97, and the ASCII table on page 703 of A Book on C confirms that the character 'a' does indeed have the ASCII code 97. Printf interprets the information supplied in the other parameters in accordance with the conversion character.

Escape sequences, the second sort of replacement, are used to display non-printing and challenging-to-print characters. Certain characters, like as newlines and tabs, generally determine where text appears on the screen. The issue with some of these characters is that they move the editor's cursor when you put them in your program when what you really want to do is change how the text will be laid out after the program is run. The following table lists a number of popular escape sequences together with the corresponding special characters:

Special Character.	Escape Sequence.
alert (beep)	\a
backslash	\\
backspace	\b
carriage return	\r
double quote	\"
formfeed	\f
horizontal tab	\t
newline	\n
null character	\0
single quote	\'
vertical tab	\v
question mark	\?

A different outcome will occur if the escape sequences are added to the control string. Have a look at the original code with the printf statement changed to a different control string:

printf("My\tfavorite\nnumbers are \t%d and \t%f.\n",i,x);

Even if the escape sequences and some of the control string words are combined (such as the letters \t and favourite), the result is the same text with additional horizontal tabs and newlines:

My favorite

numbers are 2 and 3.140000.

The Input Function scanf()

Scan Formatted String is what the function scanf in the C programming language stands for. It receives information from the standard input stream (stdin), which is often the keyboard, and then writes the outcome into the specified parameters.

- It accepts user-provided character, string, and numeric data via standard input.
- Like printf, Scanf also makes use of format specifiers.

Syntax:int scanf(const char *format, …);

Here

- The return type is int.
- The type specifiers are contained in a string called format (s).
- The prefix "..." denotes that the function takes an arbitrary number of parameters.

Example type specifiers recognized by scanf:

%d to accept input of integers.

%ld to accept input of long integers

%lld to accept input of long long integers

%f to accept input of real number.

%c to accept input of character types.

%s to accept input of a string.

Three conversion characters are of a unique kind, and one of them, [...], isn't even a character but is still given character treatment.

scanf() conversion characters	
Conversion character	**Remarks**
n	No characters in the input stream are matched. The corresponding argument is a pointer to an integer, into which gets stored the number of characters read so far.
%	A single % character in the input stream is matched. There is no corresponding argument.
[···]	The set of characters inside the brackets [ ], us cakked the scan set. It determines what gets matched. (See the following explanation.) The corresponding argument is a pointer to the base of an array of characters that is large enough to hold the characters that are matched, including a terminating null character that is appended automatically.

Example:

int var;
scanf(“%d”, &var);

The user-inputted value will be written by the scanf into the integer variable var.

Return Values:

>0: The number of correctly assigned and converted values.

0: No value was assigned.

<0: Read error or end-of-file (EOF) reached before any assignment was made.

Why &?

Scanf has to save the input data somewhere while it scans the input. The memory location of a variable must be known for scanf to be able to store this input data. And now the ampersand steps in to save the day.

• The operator's address is also referred to as &.

• For instance, the address of var is &var.

Example:

```
// C program to implement
    // scanf
    #include <stdio.h>
    // Driver code
    int main()
    {
        int a, b;
        printf("Enter first number: ");
        scanf("%d", &a);
        printf("Enter second number: ");
```

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

        printf("A : %d \t B : %d" ,

                a , b);

        return 0;

}
```

Output:

Enter first number: 10

Enter second number: 8

A : 10 B: 8

The Functions fprintf(), fscanf(), sprint(), and sscanf()

File equivalents of the functions printf() and scanf() are the functions fprintf() and fscanf(), respectively. We need to understand how C handles files before we can talk about how to use them.

A specific structure with components that indicate the current state of a file is defined as the identifier FILE in the file's header, or stdio.h. A programmer does not need to be an expert on this structure to use files. The three file pointers stdin, stdout, and stderr are also specified in stdio.h. We occasionally refer to them as files even though they are actually pointers.

Written in C	Name	Remark
stdin	standard input file	connected to the keyboard
stdout	standard output file	connected to the screen
stderr	standard error file	connected to the screen

The function prototypes for file handling functions are given in stdio.h. Here are the prototypes for fprintf() and fscanf():

```
int fprintf(FILE *fp, canst char *format, ... );

int fscanf(FILE *fp, canst char *format, ... );
```

A statement of the form

fprintf(file_ptr, con troL string , other_ arguments);

writes to the file pointed to by file_ptr. The conventions for controLstring and other_ arguments conform to those of printf(). In particular,

fprintf(stdout, ...); is equivalent to printf(...);

In a similar fashion, a statement of the form

fscanf (file_ptr, control string , other_ arguments);

reads from the file pointed to by file_ptr. In particular,

fscanf(stdi n, ...); is equivalent to scanf(...);

The functions sprint() and sscanf() are string versions of the functions printf() and scanf(), respectively. Their function prototypes, found in stdio.h, are

```
int sprintf(char *s, const char *format, ... );
```

int sscanf(const char *s, const char *format, ...);

Instead of writing to the screen, the sprint()

function writes to its first parameter, a reference to char (string). The rest of its arguments line up with those for printf (). Instead of reading from the keyboard, the function sscanf() reads from its first parameter. The remainder of its inputs support scanf (). Think about the code.

```
char str1[] :=: "1 2 3 go", str2[lee] , tmp[l00];

int a, b, c;

sscanf(str1, "%d%d%d%s", &a, &b, &c, tmp);

sprintf(str2, "%s %s %d %d %d\n", tmp, tmp, a, b, c);

printf ("%s", str2)
```

Strl is used as the function sscanf(input.)'s It reads a string, three decimal integers, and puts them in the appropriate places (a, b, c, and tmp). The sprint() function writes to str2. To be more explicit, it writes characters starting at address str2 into memory. Two strings and three decimal integers make up its output. We use printf() to display the contents of str2. The screen displays the following text:

```
go go 1 2 3
```

The Functions fopen() and fclose()

A file is described as a sequence of consecutive bytes that has an end-of-file marker. When a file is opened, the stream is connected to it. When program execution begins, the standard input, standard output, and standard error files and streams are automatically opened. Streams provide a means of exchanging data between files and applications.

For instance, a program can use the standard input stream to read data from the keyboard and the standard output stream to display that data on the screen. When a file is opened, a pointer to a FILE structure (specified in) containing information about the file's size, type, current file pointer location, and other specifics is returned. This structure also includes a file descriptor, an integer that acts as an index into the open file table of the operating system. The operating system employs a block in this table called a file control block (FCB) to handle each particular file.

Standard input, standard output, and standard error are controlled by the file pointers stdin, stdout, and stderr. The group of operations that we will now explain fall under the umbrella of the buffered file system. Because the procedures automatically maintain all the disc buffers needed for reading and writing, this file system is known as a buffered file system.

We must first declare a pointer to the FILE structure and then link it to the specific file in order to access any file. The declaration of this pointer, which is known as a file pointer, is as follows:

```
FILE *fp
```

fopen()

The following action is to open a file after a file pointer variable has been declared. The fopen() function creates a link between a file and a stream and makes it available for use. A file pointer is returned by this function.

The following is the syntax:

```
FILE *fopen(char *filename,*mode);
```

where mode is the intended open state as a string. The filename must be a string of characters that

gives the operating system a valid file name and may also include a path. The permissible values for the fopen() mode parameter are displayed in the table below.

Table: Legal values to the fopen() mode parameter

MODE:	MEANING
"r" / "rt"	opens a text file for read only access
"w" / "wt"	creates a text file for write only access
"a" / "at"	text file for appending to a file
"r+t"	open a text file for read and write access
"w+t"	creates a text file for read and write access,
"a+t"	opens or creates a text file and read access
"rb"	opens a binary file for read only access
"wb"	create a binary file for write only access
"ab"	binary file for appending to a file
"r+b"	opens a binary or read and write access
"w+b"	creates a binary or read and write access,
"a+b"	open or binary file and read access

How to open a file for reading is described in the following bit of code.

Code Fragment 1

```
#include <stdio.h>
main()
{
FILE *fp;
if ((fp=fopen("file1.dat", "r"))==NULL)
{
printf("FILE DOES NOT EXIST\n");
exit(0);
}
}
```

A file pointer is the value that the fopen() function returns. This pointer's value is NULL, a constant defined in, in the event that an error arises while opening the file. Always keep an eye out for this possibility, like in the case above.

fclose()

The fclose() method, whose syntax is as follows, should be used to close the file after processing is complete.

```
int fclose(FILE *fptr);
```

The stream is terminated, any automatically allocated buffers are released, any unwritten data for the stream is flushed, any input that has been buffered but not yet read is discarded. The return value is a constant EOF, a file end-of-file marker, if an error occurred; otherwise, it is 0. Furthermore, specifies this constant in. If the function fclose() is not explicitly used, the operating system will normally close the file when the program execution is complete.

How to close a file is demonstrated in the following section of code.

```
# include <stdio.h>
main()
{
FILE *fp;
if ((fp=fopen("file1.dat", "r"))==NULL)
{
printf("FILE DOES NOT EXIST\n");
exit(0);
}
```

```
……..

……..

……..

……

/* close the file */

fclose(fp);

}
```

The file cannot be used again once it has been closed. It can be accessed in the same mode or another if necessary.

Summary

- The functions printf() and scanf(), and the related file and string versions of these functions, all use conversion specifications in a control string to deal with a list of arguments of variable length.
- To open and close files, we use fopen() and fclose(), respectively. After a file has been opened, the file pointer is used to refer to the file.
- The system opens the three standard files stdin, stdout, and stderr at the beginning of each program. The function printf() writes to stdout. The function scanf() reads from stdi n. The files stdout and stderr are usually connected to the screen. The file stdin is usually connected to the keyboard. Redirection causes the operating system to make other connections.
- A set of functions that use file descriptors is available in most systems, even though these functions are not part of ANSI C. They require user-defined buffers. The file descriptors of stdin, stdout, and stderr are 0, 1, and 2, respectively.

UNIT 11

The Operating System

Learning Objectives

At the end of this unit, you will be able to:

- Explain how to execute commands from within a C program
- Infer about the environment variables
- Discuss the concept of the C compiler
- Define the concept of libraries used in C

Introduction

File Descriptor Input/Output

C was invented to write an operating system called UNIX. C is a successor of B language which was introduced around the early 1970s. The language was formalized in 1988 by the American National Standard Institute (ANSI). The UNIX OS was totally written in C.

A nonnegative integer connected to a file is referred to as a file descriptor. The library functions that are utilized with file descriptors are discussed in this section. Despite not being ANSI C-compliant, these functions are present on the majority of C systems under MS-DOS and UNIX. Care must be used while transferring code from UNIX to MS-DOS or vice versa due to minor differences.

FD	PURPOSE
0	Standard input
1	Standard output
3	Standard error

The majority of standard library functions that use a reference to FILE are buffered. Contrarily, file descriptor-using functions might call for programmer-specified buffers. Let's use a program that reads from one file and writes to another while changing the case of each letter to demonstrate how file descriptors are used.

In file change_case.c

```
/* Change the case of letters in a file. */
```

```
#include <ctype.h>

#include <fcntl.h>

#include <unistd.h>    /* use io.h in MS_
DOS */

#define BUFSIZE 1024

int main(int argc, char **argv)

{

char mybuf[BUFSIZE], *p;

int in_fd, out_fd, n;

in_fd = open(argv[lJ, O_RDONLY);

out_fd = open(argv[2] , O_WRONLY I O_
EXCL | 0_CREAT, 0600;

while ((n = read(in_fd, mybuf, BUFSIZE))
> 0) {

for (p = mybuf; p mybuf < n; ++p)

f (islower(*))

*p = toupper(*p);

else if (isupper(*p))

*p = tolower(*p);

write(out_fd, mybuf, n);

}

close(in_fd);

close(ouLfd);

return 0;

}
```

Program analysis for change_case

```
#include <ctype.h>

#include <fcntl.h>

#include <unistd.h>
```

We'll make advantage of the fcntl.h header file's symbolic constants. The header file unistd.h contains prototypes for the open () and read () functions. Instead, we would add io.h to MS-DOS.

```
in_fd = open(argv[lJ, O_RDONLY);
```

The first input to open() is a file name, and the second one specifies how to open the. If there are no issues, the function returns a file descriptor; otherwise, it returns the value -1. The mnemonic "in fd" stands for "in file descriptor." The cntl.h file presents the symbolic constant O RDONL Y, which works with both MS-DOS and UNIX platforms. As a mnemonic, it denotes "available for reading only."

```
Out_fd = open(argv [2], 0_WRONLY | 0_
EXCL | 0_CREAT, 0600);
```

The symbolic constants in fcntl.h that are used to open a file can be combined using the bitwise OR operator. Here, we specify that the file should only be opened for writing (i.e., if the file already exists, it is a mistake), that it should only be opened for writing, and that if the file does not already exist, it should be created. O EXCL can only be obtained through O CREAT. If the file is created, the third parameter sets the file permissions; otherwise, it has no effect. Further information on file permissions will be provided below.

```
while ((n = read(in_fd, mybuf, BUFSIZE)) > 0) {
```

A maximum of BUFSIZE characters from the stream connected to in fd are read and then written to mybuf. The total number of characters read is returned. The body of the while loop is executed so long as read 0 is able to read characters from the stream. Throughout the body of the loop, the letters in mybuf are changed to the reverse case.

```
write(out_fd, mybuf, n);
```

n characters in mybuf are written to the stream indicated by out_fd.

```
close(in_fd);

close(out_fd);
```

The two files are then closed. The system will close the files upon program exit if the programmer has not expressly instructed it to do so.

File Access Permissions

In UNIX, a file is created with the appropriate access permissions. The permissions govern who has access to a file, including the owner, the group, and other users. It is possible to combine read, write, execute, or none of these access capabilities. When a file is created by calling open 0, the third parameter can be a three-digit octal integer to set the permissions. Each octal digit controls the read, write, and execute permissions. A user's permissions are controlled by the first octal digit, a group's permissions are controlled by the second octal digit, and others' permissions are controlled by the third octal digit. (Everyone is included in "others").

Meaning of each octal digit in the file permissions		
Mnemonic	**Bit representation**	**Octal representation**
r--	100	04
-w-	010	02
--x	001	01
rw-	110	06
r-x	101	05
-wx	011	03
rwx	111	07

Now, we can calculate the file access permissions by combining three octal digits into a single integer. The most straightforward to remember is the mnemonic representation. The user, the group, and others are denoted, respectively, by the first, second, and third groups of three letters.

Examples of file access permissions	
Mnemonic	**Octal representation**
rw-------	0600
rw----r--	0604
rwxr-xr-x	0755
rwxrwxrwx	0777

The rights rwxr-xr-x signify that the owner, the group, and others may all read and execute the file. They also signify that the owner can read, write, and execute the file. The Is -1 command in UNIX displays the mnemonic file access permissions. File permissions are available in MS-DOS, but only for everyone.

Executing Commands from Within a C Program

Access to operating system commands is provided via the library function system(). The command date causes the current date to be printed on the screen in both MS-DOS and UNIX. From within a program, we can write code to print this information on the screen.

```
system("date");
```

System() treats the string as an operating system command. Control is transferred to the operating system when the statement is executed, where it is used to carry out the instruction before being returned to the program.

In UNIX, Vi is a well-liked text editor. Let's imagine that from within a program, we want to utilise vi to edit a file that has been supplied as a command line argument. Our ability to write

```
char    command[MAXSTRING];

        sprintf(command, "vi %S", argv[l])'

        printf("vi on the file %5 is coming up ... \n",
        argv[l]);

system(command);
```

If we use an editor that is available on that system in place of vi, a similar example will function in MS-DOS.

As a last illustration, imagine that we have had enough of the capital letters that the dir command on our MS-DOS system produces. A software that uses this command and only prints lowercase letters on the screen can be created.

In file lower_case.c

```
        /* Write only lowercase on the screen. */

        #include <ctype.h>

        #include <stdio.h>

        #include <stdlib.h>

        #define MAXSTRING 100

        int main(void)

        {

        char    command [MAXSTRING], *tmp_
        filename;

        int     c;

        FILE    *ifp;

        tmp_filename = tmpnam(NULL);

        sprintf(command , "dir > %s", tmp_
        filename);

        system(command)j

        ifp = fopen(tmp_fil ename, n r");

        while ((c = getc(ifp)) != EOF)

        putchar(tolower(c));

        remove(tmp_filename);

        return 0;

        }
```

Then, we use the library function tmpname to create a temporary file name (). The system() function is then used to pass the output of the dir command into the temporary file. After that, the file's contents are displayed on the screen with all uppercase letters changed to lowercase. After we have finished using the temporary file, we call the library method remove() to delete it..

Environment Variables

Both UNIX and MS-DOS support environment variables. To print them on the screen, run the next program:

```
        #include <stdio.h>

        int main(int argc, char *argv[], char *env[])

        {
```

```
int i;

for (i 0; env[i] ! NULL; ++i)

printfC"%s\n" , env[i]);

return 0;

}
```

The third argument to the main() method is a pointer to a pointer to char, often known as an array of strings or pointers to char. The system provides the strings and the storage space for them. Env, the last element of the array, is a NULL pointer. Our UNIX system prints using this program.

```
HOME=/c/c/blufox/center_manifold

SHELL=/bin/csh

TERM=vt102

USER=blufox
```

The environment variable is to the left of the equal sign, and its value, which should be viewed as a string, is to the right of the equal sign. This software prints on our MS-DOS machine.

```
COMSPEC=C:\COMMAND.COM

BASE=d:\base

INCLUDE=d:\msc\include
```

The command to display the environment variables is provided by the UNIX system, however it depends on the shell you are currently using. The command is set in MS-DOS and the Bourne shell while printenv is used in the C shell. The command's output and our program's output are identical.

Environment variables are typically capitalised as a matter of convention. The library function getenv() can be used in a C program to get the value of an environment variable that has been supplied as an argument. Here is an illustration of how to use getenv():

```
printf("%s%s\n%s%s\n%s%s\n%s%s\n",

"      Name: ", getenv("NAME"),

“      User: ", getenv("USER"),

“      Shell: ", getenv("SHELL"),

"      Home directory t:
,”getenv("HOME"));
```

In stdlib.h, the function prototype is available. The NULL pointer is returned if the string supplied as an input is not an environment variable.

The C Compiler

There are numerous C compilers, and any number of them may be offered by an operating system. Just a few of the options are listed below:

Command	The C compiler that gets invoked
cc	The system supplied native C compiler
acc	An early version of an ANSI C compiler from Sun Microsystems
bc	Borland C/C++ compiler, integrated environment
bcc	Borland C/C++ compiler, command line version
gcc	GNU C compiler from the Free Software Foundation
hc	High C compiler from Metaware
occ	Oregon C compiler from Oregon Software
qc	Quick C compiler from Microsoft
tc	Turbo C compiler, integrated environment, from Borland
tcc	Turbo C compiler, command line version, from Borland

In this part, we go over a few of the UNIX systems' cc command's available options. Similar options are offered by other compilers. If an entire

program, such as pgm.c, is stored in a single file, then the command

Cc pgm.c

creates executable object code from the C code in pgm.c and writes it to the file a.out. (Pgm. exe is the executable file's name on MS-DOS.) The program is run with the command a.out. Whatever is in a.out will be overwritten by the next cc command. If we issue the order

cc -0 pgm pgm.c

A.out won't be affected, and the executable code will instead be written directly into the file pgm.

In reality, the cc command completes its task in three steps: The preprocessor is called first, followed by the compiler and then the loader. The final executable file is created by the loader, also known as the linker. You can only invoke the preprocessor and compiler with the -c option; you cannot call the loader. If we have a program that is written in multiple files, this is helpful. Considering the order

cc –c main.c file1.c file2.c

Corresponding object files with a .0 extension will be created if there are no mistakes. We can compile a combination of.C and.0 files to produce an executable file. Imagine, for instance, that main.c contains a mistake. After fixing the error in main.c, we may issue the command.

cc -0 pgm main.c file 7.0 file2.0

Using .0 files instead of.e files speeds up compilation,

Some useful options to the compiler	
v	Compile only, generate corresponding *.o* files.
-g	Generate code suitable for the debugger.
-o *name*	Put executable output code in *name.*
-p	Generate code suitable for the profiler.
-v	Verbose option, generates a lot of information.
-D *name -def*	Place at the top of each *.c* file the line.
	#define *name def.*
-E	Invoke the preprocessor but not the compiler.
-I *dir*	Look for #include files in the directory *dir.*
-M	Make a make file.
-O	Attempt code optimization.
-S	Generate assembler code in corresponding *.s* files.

There's a chance your compiler will offer additional options in addition to those listed below. Alternatively it might employ various flag characters. Several memory models are often supported by MS-DOS compilers. For a comprehensive list of options, consult the documentation for your compiler.

Recommendation: If you've never used the -v option, give it a shot. When trying to understand every aspect of the compilation process, certain compilers generate a tonne of information that can be quite helpful.

Libraries

A utility for creating and managing libraries is offered by several operating systems. The program is known as the archiver in UNIX and is accessed using the ar command. This tool, known as the librarian in the Ms-DOS universe, is an add-on function. For instance, the Turbo C librarian is tUb, whereas the Microsoft librarian is lib. By convention, files in a library end in.a in UNIX and.lib in MS-DOS. The topic will be discussed in

relation to UNIX, but the main concepts apply to any librarian.

The standard C library for UNIX is often located in the file /lib/libc.a, but it can also exist in whole or in part in other files. Try the command if UNIX is available to you.

```
art /usr/lib/llbe.a
```

The title, or name, of each file in the library is displayed using the key t. More than you care to look at exist. You can use the command to count them.

```
art /usr/lib/libe.a I we
```

By connecting the output of the ar command to the input of the we command, this counts the number of lines, words, and characters. It is not very remarkable that the standard library keeps growing over time. A DEC VAX 11/780 from the 1980s has 311 object files in its standard library. In 1990, a moderately new Sun computer contained 498 object files in its standard library. The Sun machine we are using right now has 563 object files.

Let's give an example of how programmers can build and utilise their own libraries. This will be done as part of building a "graceful library." We currently have 11 such gracious functions in a directory called g lib, and we regularly add more. They include procedures like gfclose(), gcalloc(), gmalloc(), and others. Although each is written in a distinct file, they may all very well be in one file for the sake of establishing a library. We provide the gcalloc() code to further illustrate the concept of functions:

```
#inc1ude <stdio.h>

#inc1ude <stdlib.h>

void *gca11oc(int n, unsigned sizeof_
something)

{

void *p;

if ((p = calloc(n, sizeof_something))
NULL) {

fprintf(stderr, "\nERROR: ca11ocO failed -
bye.\n\n");

exit(l);

}

return p;

}
```

Before we can create our library, we must first compile the.e files to create the corresponding.0 files. After that, we give the two directives.

```
ar ruv fl_lib.a gfopen.o gfclose.o flealloc.o
...

ranlib fl_lib.a
```

The keys ruv in the first command stand for replace, update, and verbose, respectively. This action generates the library g lib.a if it doesn't already exist. If so, the named.0 files replace any other identically named files that are currently present in the library. If the named.0 files don't already exist in the library, they are added.

The ranlib tool is used to randomise the library so that the loader may use it. Imagine that main.c and two other.e files will make up the software

we are developing. If our software calls gfopen(), we have to make our library available to the compiler. The following command makes this happen:

CC -0 pgm main.e file I.e file2.c g_lib.a

If a function is called by our application without providing the function definition, the standard library will be searched before g_lib.a. The final executable file will only contain the functions that are actually required.

Summary

- An operating system command can be executed from within a program by invoking system(). In MS-DOS, the statement

  System(“dir");

  will cause a list of directories and files to be listed on the screen.

- Many operating systems provide a utility to create and manage libraries. In UNIX the utility is called, archiver, and it is invoked with the arc command. In the MS-DOS world, this utility IS called the librarian, and it is an add-on feature.

- The make utility can be used to keep track of source files and to provide convenient access to libraries and associated header files.

UNIT 12

Moving from C to C++

Learning Objectives

At the end of this unit, you will be able to:

- Define the concept of classes and abstract data types
- Discuss constructors and destructors
- Infer the concept of object-oriented programming, inheritance, and polymorphism

Introduction

Functions

The new function prototype syntax used by standard C compilers was inspired by the syntax of functions in C++. In essence, a list of the various argument types is contained within the header parentheses. By explicitly declaring the type and number of arguments in C++, strong type checking and conversions that are consistent with assignments are possible.

Function parameters may be called directly by reference in C++. Declaring call-by-reference parameters requires the following syntax:

type& identifier

The default settings for C++ function arguments are also an option. They are provided in the parameter list by the function declaration.

= expression

placed following the parameter.

The example below demonstrates these ideas:

In file add3.cpp

```
// Use of a default value

        #include <iostream.h>

        inline void add3(int& s, int a, int b, int c=0)
```

```
{

s = a + b + c;

}

inline double average(int s) { return s / 3.0; }

int main()

{

int score_I, score_2, score_3, sum;

count <<"\nEnter 3 scores: ";

Cln >> score_I >> score_2 >> score_3;

add3(sum, score_I, score_2, score_3);

cout<< \nSum = "<< sum;

cout << It\nAverage = "<<average(sum) << end1'

add3(sum:. 2 * score_I, score_2); // use of default value 0

cout<< \nWeighted Sum "<< sum << // If,

cout <<"\nWeighted Average = It << average(sum) <<".\n";

}
```

Here, invoking add 30 only requires three actual parameters. The default value for the fourth argument is zero.

Classes and Abstract Data Types

The c1 ass aggregate type is what makes c++ new. A c1 ass is a development of the struct concept in conventional C. The mechanism for implementing a user-defined data type and any associated operators and functions are provided by a c1 ass. Hence an ADT can be implemented with a c1 ass. To implement a limited form of string, let's create a C1 ass called string.

In file my _string.cpp

// An elementary implementation of type string.

```
#inc1ude <string.h>

#include <iostream.h>

canst int max_len = 255;

class string {

public: // universal access

void assign(const char* st)

{ strcpy(s, st); len = strlen(st); }

int 1ength() { return len; }

void printO { cout« s« "\nLength: // <<le<<"\n"; }

private: // restricted access to member functions

char s[max_len]; // implementation by character array

int 1en;

} ;
```

The typical C structure idea is expanded in this example in two key ways: It has members that are functions like assign as well as members that are both public and private. The term pub 1 I c designates the visibility of the following members. Without this keyword, the members are private to the class. Private members can only

be accessed by other member functions of the class. Public members can be accessed by any method that is covered by the class declaration. The implementation of a class type may be "hidden" for privacy reasons. This restriction prevents unanticipated modifications to the data structure. Object-oriented programming is characterised by data hiding or limited access.

The declaration of member functions enables the ADT to assign particular functions to execute on its private representation. The length of the string is determined by the characters up to but excluding the first character with a zero value, and is returned by the member function 1ength as an example. The member function print generates both the string and its length (). By using the member function assign (), a character string is saved in the hidden variable s, and its length is determined and saved in the hidden variable 1en.

This data type string can now be used just like a language's fundamental types. It complies with C's common block structure and scope requirements. This class is also used by client code. Only the public members can be used by the client to manipulate string variables.

```
// Test of the class string.

    int mainO

    {

    string one, two;

    char three [40] {liMy name is Charles Babbage."};

    one.assign("My name ;s Alan Turing.");

    two.ass;gn(three);

    cout << three;

    cout <<"\nLength: // << strlen(three) << endl; // Print shorter of one and two.

    if (one.length() <= two.length())

    one.print ();

    else

    two.print();

    }
```

The first and second variables are of the string type. The variable three is incompatible with strings because it is of the pointer to char type. The dot operator, also known as the "structure member operator," is used to call the member functions. It is clear from their definitions that member functions work with the secret private member fields of the designated variables. In order to access this member, one cannot be written within of main. The result of this example program is

```
    My name is Charles Babbage.

    Length: 27

    My name id Alan Turing.

    Length: 23
```

Constructors and Destructors

A constructor is one of a class's member functions with the same name as the class itself. Initializing a class's object is helpful. It has the option to accept or reject the arguments. It is employed to allocate memory to a class object. Every time a class instance is created, it is called. It can be manually defined both with and without

arguments. A class may include a large number of constructors. It cannot be inherited or virtual, but it can be overloaded. To initialise an item from another object, one can utilise the copy constructor idea.

Syntax:

```
ClassName()
  {
    //Constructor's Body
  }
```

Destructor:

Like a function Object() { [native code] }, a destructor is a member function of a class whose name is the class name followed by the tilde() operator. The memory of an object can be dealt with. When the class's object is released or deleted, it is called. A class always has a single destructor that has no parameters, preventing overloading. It is always invoked in the constructor's reverse order. If a class inherits another class and both classes have a destructor, the destructor of the child class is called first, followed by the destructor of the parent or base class.

```
 ~ClassName()
  {
     //Destructor's Body
          }
```

Example:

```
#include <iostream>
using namespace std;
class Z
{
public:
        // constructor
        Z()
        {
cout<<"Constructor called"<<endl;
        }
        // destructor
        ~Z()
        {
cout<<"Destructor called"<<endl;
        }
};
int main()
{
        Z z1; // Constructor Called
        int a = 1;
        if(a==1)
        {
                Z z2; // Constructor Called
        } // Destructor Called for z2
} // Destructor called for z1
```

Output:

Constructor called

Constructor called

Destructor called

Destructor called

Object-oriented Programming and Inheritance

The inheritance system is an innovative OOP concept. This is the process for deriving a new class from a base class that already exists. The members of the inherited base class are increased or changed by the derived class. This is used to establish a hierarchy of related kinds and share code and interface.

The use of hierarchy helps people manage complexity. It assigns items to different categories. The periodic table of elements, for instance, has elements that are gases. They have characteristics that all items in that categorization share. An essential subclass of gases are inert gases. Here, the hierarchy looks like this: A gas, like argon, is an element since it is an inert gas. The behavior of inert gases can be easily understood using this hierarchy. Since all elements match this characteristic, we are aware that they are made up of protons and electrons. Since all gases behave in the same way at ambient temperature, we can be certain they are in a gaseous state. While this is a characteristic of all inert gases, we know they do not combine with other elements in typical chemical processes.

Think about creating a database for a college. The registrar must keep tabs on various kinds of students. We must create a base class that captures the definition of "student." Graduate and undergraduate students are the two main subgroups.

The OOP design methodology is as follows:

OOP Design Methodology

- Choose a suitable assortment of sorts.
- Design in their connection.
- To share code, use inheritance.

This is an illustration of deriving a class:

```
enum support { ta, ra, fellowship, other };

enum year { fresh, soph, junior, senior, grad };

class student {

public: student(char* nm, int id, double g, year x);

voi d pri ntO ;

private: int student_id;

double gpa;

year y;

char name[30];

} ;

class grad_student: public student {

public: grad_student (char* nm, int id, double g, year x, support t,

char* d, char* th);
```

```
void printO;

private:

support s;

char dept[10];

char  thesis[80];

} ;
```

In this example above, the base class is student, and the derived class is graduate student. Because the word "public" is inserted after the colon in the derived class header, the public members of student are to be inherited as public members of graduate student. The private members of the base class are not accessible to the derived class. Due to public inheritance, the derived class grad student IS a subtype of student.

An inheritance structure offers the overall system a design. For instance, from the base class member, a database containing all of the students at a college may be created. Extension students could be included in the student-grad_ student relation as an additional significant category of objects. Similar to how other employment categories may use person as their basis class,

Polymorphism

The forms of a polymorphic function are varied. The division operator is an illustration of Standard C. Integer division is used when the arguments to the division operator are integral. But, floating-point division is used if either one or both parameters are.

A function name or operator can be overloaded in C++. A function's signature, which is the list of argument types in its parameter list, determines how it is called.

```
a / b    // divide behavior determined by native coercions

cout<< a          // overloading « the shift operator for output
```

The outcome of the division statement depends on the arguments being forced by default to the widest type. Hence, the outcome is an integer division if both arguments are integers. The outcome is floating-paint if either one or both parameters are floating-point. The shift operator <<invokes a function that can output an object of type an in the output statement.

Localizing behavioral responsibility through polymorphism. When new functionality is added to the system thanks to code advancements from ADT, the client code frequently doesn't need to be revised.

A thorough structural description of any shape is a prerequisite for the C method for constructing a set of routines to produce an ADT shape.

```
struct shape {

enum{CIRCLE, ..... } e_val;

double center, radius;

};
```

would include an enumerator value to help identify it as well as all the members required for any shape that is currently drawable in our system. Then, the area procedure would be expressed as

```
double area(shape* s)

{

switch(s -> e_val) {

case CIRCLE: return (PI * s -> radius * s -> radius);

case RECTANGLE: return (s -> height*S -> width);

}
```

A form hierarchy is used by oop coding techniques in C++ to solve the same issue. The hierarchy is the one where the shape of the circle and rectangle comes first. A new derived class is created throughout the revision process to accommodate code updates, localising additional description. The meaning of any modified routines—in this case, the new area calculation—is overridden by the programmer. The new type is not used by client code, therefore it is unaffected. Client code that benefits from the new type often undergoes minimal change.

This design employs shape as an abstract base class in C++ programmes. A single or more pure virtual functions can be found in this class. There is no definition for a pure virtual function. A derived class is where the definition is located.

```
// shape is an abstract base class

class shape {

public:

virtual double area() = 0; // pure virtual function

};

class rectangle: public shape {

public:

II pure virtual function

rectangle(double h, double w): height(h), width(w) {}

double area() { return (height * width);} // overridden

private:

double height, width;

} ;

class circle: public shape {

public:

circle(double r): radius(r) {}

double area() { return ( 3.14159 * radius * radius); }

private:

double radius;

} ;
```

The client code is polymorphic and computes an arbitrary area. At runtime, the proper area() function is chosen.

```
shape'" ptr _shape;

cout <<" area = "<< ptr _shape-> area();
```

Imagine creating a square class to enhance our hierarchy of types.

```
class square: public rectangle {
```

```
public:

square(double h): rectangle(h,h) {}

double area() { return rectangle::area(); }

} ;
```

The client-side code is unaltered. With non-OOP code, this would not have been the case.

Summary

- c++ reduces C's traditional reliance on the preprocessor. Instead of using define, special constants are assigned to variables specified as canst. The new keyword inline specifies that a function is to be compiled inline to avoid function call overhead. As a rule, this should be done sparingly and only on short functions.
- What is novel about C++ is the aggregate type class. A class is an extension of the idea of struct in traditional C. Its use is a way of implementing a data type and associated functions and operators. Therefore, a class is an implementation of an abstract data type (ADT). are two important additions to the structure concept: first, it includes members that are functions, and second, it employs access keywords public, private, and protected. These keywords indicate the visibility of the members that follow. Public members are available to any function within the scope of the class declaration. Private members are available for use only by other member functions of the class. Protected members are available for use only by other member functions of the class and by derived classes. Privacy allows part of the implementation of a class type to be "hidden."
- A polymorphic function has many forms. A virtual function allows run-time selection from a group of functions overridden within a type hierarchy. An example in the text is the area calculation within the shape hierarchy. Client code for computing an arbitrary area is polymorphic. The appropriate area() function is selected at run-time.