Dr.M.Sivasankari earned her Ph.D degree in Computer Science from MS University, Tirunelveli. She was a meritorious student both in academics and non academics during her college days and she won a number of prizes and medals in competitions conducted inside and outside of the college.

On top of that she is a Hat-trick World Record Holder with 40 different awards. She completed her M.Phil (Computer Science) at St.Xavier's College, Palayamkottai, Tirunelveli and M.Sc (Computer Science) & B.Sc (Computer Science) at G.Venkataswamy Naidu College, Kovilpatti. She is currently working as an Assistant Professor of Computer Applications at Don Bosco College of Arts and Science, Keela Eral. She has been actively playing a vital role in organizing State, National, International level Competitions for the college. She has been honored and felicitated in many occasions. She is being invited as a Resource Person for many programs related to her discipline. She is an orator. She has published a number of technical papers at National & International Conferences.

ISBN
No.

TAKE C EASY - Dr. M. SIVASANKARI

A TEXT BOOK OF

TAKE C EASY

Dr. M. SIVASANKARI

A TEXT BOOK OF

TAKE C EASY

B.E. / B.Tech. STUDENTS

As Per Anna University Syllabus

(Computer Science and Engineering) - Regulation – 2017

B.Sc(CS) / B.Sc(IT) / BCA

Manonmaniam Sundaranar University Syllabus

Dr. M. SIVASANKARI

Assistant Professor, Department of Computer Applications

Don Bosco College of Arts and Science,

Keela Eral.

First Edition : November 2020

ISBN No. :

Price : Rs.

Publishers

Dr. M. Sivasankari
Assistant Professor, Department of Computer Applications
Don Bosco College of Arts and Science,
Keela Eral.

Contact for Copies:

Printed By :
M/s. Vinayaga Traders, 330-A, PKSA Arumugam Road,
Sivakasi- 626 189, Cell : 9486357318

Programming in C

ABOUT THE AUTHOR

Dr.M.Sivasankari earned her Ph.D degree in Computer Science from MS University, Tirunelveli. She was a meritorious student both in academics and non academics during her college days and she won a number of prizes and medals in competitions conducted inside and outside of the college. On top of that she is a Hat-trick World Record Holder with 40 different awards. She completed her M.Phil(Computer Science) at St.Xavier’s College, Palayamkottai, Tirunelveli and M.Sc (Computer Science) & B.Sc (Computer Science) at G.Venkataswamy Naidu College, Kovilpatti. She is currently working as an Assistant Professor of Computer Applications at Don Bosco College of Arts and Science, Keela Eral. She has been actively playing a vital role in organizing State, National, International level programmes for the college. She has been honored and felicitated in many occasions. She is being invited as a Resource Person for many programs related to her discipline. She’s an orator. She has published a number of technical papers at National & International Conferences..

Dr.M.Sivasankari

PREFACE

This book is intended for beginners, intermediate level and for all those who want to learn or expand their knowledge in C Program. A systematic approach has been followed from the beginning to the end. Most of the concepts of C language are explained in detail with practical applications. Solved programs have also been provided to help the learners to master the programming language.

A simple approach is used to understand the various concepts of C language. Each Program is thoroughly explained and output is also shown. Each and every program given in the book is perfectly working.

These exercises are meant to test your skills & understanding for solving the problems. If you study this book in a right spirit, you will become an expert in C Programming. Best of Luck!

Utmost care has been taken to write the book in order to make it free of errors. However, if you come across any error, you can feel free to contact me. Your suggestions & feedback may kindly be sent to the following maid id - mvsivasankari@gmail.com.

Dr. M. SIVASANKARI

ACKNOWLEDGEMENT

I would like to thank all those who have encouraged me to write the book.

I express my deep sense of gratitude to my professor Dr. P. Velmani, Assistant Professor of Computer Science for her thorough review of every topic discussed in the book. It was of great help in improving this book.

I dedicate this book to my family and I wholeheartedly thank them for their patience & support extended to me all the times

FOREWORD

Dr. P. Velmani,
Assistant Professor of Computer Science**,**
The M.D.T.Hindu College, Tirunelveli.
way2itcareerseekers.blogspot.com

The author of this study material Dr.M.Sivasankari, has the qualities of high flying such as an orator, artists, organizer. I am very happy to write the foreword of the study material "Take C easy" prepared by her.

C is an ancient and still most popular programming language in industry. Learning a programming language is very easy once you know the basics of that particular language.

The material is prepared in an easy to understand manner. She has explained all basic concepts of C programming in a simple language with syntax, semantics, flow charts and sample programs. Students can make it as a source for self-learning study material. This is suitable for undergraduate students those who start studying on C programming.

Since C is an ever wanted programming language and comes in many versions, I wish the author to continue this work with refined and possibly more example programs.

Don't See the C as Sea. Dive to explore to catch your dream.

(Dr. P. Velmani)

CONTENTS

CHAPTER 01
INTRODUCTION

History of C

- ✓ C is a popular general-purpose programming language. C language has been designed and developed by ***DENNIS RITCHIE*** at Bell Laboratories in ***1972***.
- ✓ C was evolved from ALGOL, BCPL and B languages since it was developed along with the UNIX Operating System.
- ✓ It‟s one of the most popular computer languages today because it is a structured, high level, machine independent language.
- ✓ In 1983, ***American National Standards Institute (ANSI)*** appointed a technical committee to define a standard for C.

Structure of a C Program

```
Include header file section
Global Declaration Section
/* comments */
 main()     //Function name
{
* comments*/
Declaration Part
Executable Part
}
User- defined functions
{
    Function 1
    Function 2
      -
      -
      -
   Function n
}
```

Executing the Program

The following steps are essential in executing a program in „C‟.

a) Creation of Program: Programs should be written in C editor. The default extension is C.
b) Compilation of a program:the source program should be translated into object programs which is suitable for execution by the computer.if there is no error compilation proceeds and translated program is stored in another file with the same file name with extension ".obj".
c) Linking: Linking is also an essential Process. It puts all other program files and functions together that are required by the program.
d) Executing the program: After the compilation the executable object code will be loaded in the computer‟s main memory and the program is executed.

For example: if the programmer is using pow() function, then the object code of this function should be brought from math.h library of the system and linked to the main() program.

- - ☙☙☙ - -

CHAPTER 02
CONSTANTS, VARIABLES AND DATA TYPES

Character Set

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. The characters in C are grouped into the following categories:

- Letters
- Digits
- Special Characters
- White Spaces

Letters Uppercase A…Z Lowercase a…z	**Digits** 0…9
Special Characters , comma . period ; Semicolon : Colon ? Question mark „ Apostrophe “ Quotation mark ! Exclamation Mark | Vertical Bar / Slash \ Backslash ~ Tilde _ Underscore $ Dollar Sign % Percent Sign **White Spaces** Blank Space Horizontal Tab Carriage Return New Line Form Feed	**Special Characters** &Ampersand ^ Caret *Asterisk - Minus sign + Plus sign <Opening angle bracket or less than sign >Closing angle bracket or greater than sign (Left parenthesis) Right Parenthesis [Left Bracket] Right Bracket { Left Brace } Right Brace # Number sign or Hash

TOKENS

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are known as C tokens

C has six types of tokens

- Keywords
- Constants
- Identifiers
- Strings
- Operators
- Special Symbols

Keywords

Keywords serve as basic building blocks for program statements. All keywords must be written in lowercase.

ANSI C Keywords

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

Identifiers

Identifiers refer to the names of variables, function and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character.

Both uppercase and lowercase letters are permitted. The underscore character is also permitted in identifiers.

Rules for Identifiers

1. *First character must be an alphabet (or underscore)*
2. *Must consist of onlyletters, digits or underscore*
3. *Only first 31 characters are significant*
4. *Cannot use a keyword*
5. *Must not contain white space*

Constants

- Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.
- Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*. There are enumeration constants as well.
- Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Constants

An integer constant refers to a sequence of digits. An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some ***examples of integer literals*** −

```
212        /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078        /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of integer literals −

```
85        /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30        /* int */
30u        /* unsigned int */
30l       /* long */
30ul       /* unsigned long */
```

Floating-point Constants

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some ***examples of floating-point literals*** −

3.14159	/* Legal */
314159E-5L	/* Legal */
510E	/* Illegal: incomplete exponent */
210f	/* Illegal: no decimal or exponent */
.e55	/* Illegal: missing integer or fraction */

Character Constants

- Character constants are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of **char** type.
- A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

Backslash Character Constants

The backslash character constants that are used in output functions.

Constant	**Meaning**
„\a‟	Audible alert(bell)
„\b‟	Backspace
„\f‟	Form feed
„\n‟	New line
„\r‟	Carriage return
„\t‟	Horizontal tab
„\v‟	Vertical tab

Constant	**Meaning**
„\‘‘‘‘	Single quote
„\’’‘‘	Double quote
„\?‘‘	Question mark
„\\‘‘	Backslash
„\0‘‘	Null

Example

```
#include <stdio.h>

void main() {
   printf("Hello\tWorld\n\n");
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

Output:

```
Hello World
```

String Constants

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

We can break a long line into multiple lines using string literals and separating them using white spaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
```

Variable Definition in C

A variable is a data name used for storing a data value. Its value may be changed during the program execution. The value of the variable keeps on changing during the execution of a program.

Examples

Average

height

Sum

Rules for Defining Variables

- They must begin with a character without spaces but underscore is permitted.
- The length of the variable varies from compiler to compiler. Generally, most of the compilers support 8 characters excluding extension. However, the **ANSI** standard recognizes the maximum length of a variable up to 31characters.
- The variable should not be a C keyword.
- The variable names may be a combination of uppercase and lowercase characters. For example,suM and sum are not the same.
- The variable name should not start with a digit.

Declaring Variables

The declaration of variables should be done in the declaration part of the program. The variables must be declared before they are used in the program. Declaration provides two things

1. Compiler obtains the variable name
2. It tells to the computer data type of the variable being declared and helps in allocating the memory.

Syntax

Data-type variable name

Example

int age;

char m;

float s;

double k;

int a,b,c;

The int, char, float, double are keywords to represent data types.

Initializing/Assigning the variables

Variables declared can be assigned or initialized using the assignment operator „=‟. The declaration and initialization can also be done in the same line.

Syntax

variable_name = constant;

or

data_type variable _name = constant;

Example

X=2;

int x=2;

Data Types

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows −

Sr.No.	Types & Description
1	**Basic Types** They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.

Sr.No.	Types & Description
2	**Enumerated types** They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
3	**The type void** The type specifier *void* indicates that no value is available.
4	**Derived types** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see the basic types in the following section, where as other types will be covered in the upcoming chapters.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges −

Type	Storage Size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions *sizeof(type)* yields the storage size of the object or type in bytes. Given below is an example to get the size of various type on a machine using different constant defined in limits.h header file −

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

```
#include <limits.h>
#include <float.h>
int main(int argc, char** argv) {
   printf("CHAR_BIT    : %d\n", CHAR_BIT);
   printf("CHAR_MAX    : %d\n", CHAR_MAX);
   printf("CHAR_MIN    : %d\n", CHAR_MIN);
   printf("INT_MAX     : %d\n", INT_MAX);
   printf("INT_MIN     : %d\n", INT_MIN);
   printf("LONG_MAX    : %ld\n", (long) LONG_MAX);
   printf("LONG_MIN    : %ld\n", (long) LONG_MIN);
   printf("SCHAR_MAX : %d\n", SCHAR_MAX);
   printf("SCHAR_MIN : %d\n", SCHAR_MIN);
   printf("SHRT_MAX    : %d\n", SHRT_MAX);
   printf("SHRT_MIN    : %d\n", SHRT_MIN);
   printf("UCHAR_MAX : %d\n", UCHAR_MAX);
   printf("UINT_MAX    : %u\n", (unsigned int) UINT_MAX);
   printf("ULONG_MAX : %lu\n", (unsigned long) ULONG_MAX);
   printf("USHRT_MAX : %d\n", (unsigned short) USHRT_MAX);

   return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux −

```
CHAR_BIT    : 8
CHAR_MAX    :   127
CHAR_MIN    :   -128
INT_MAX     :   2147483647
INT_MIN     :   -2147483648
LONG_MAX    :   9223372036854775807
LONG_MIN    :   -9223372036854775808
SCHAR_MAX   :   127
SCHAR_MIN   :   -128
SHRT_MAX    :   32767
SHRT_MIN    :   -32768
UCHAR_MAX :  255
```

```
UINT_MAX :  4294967295
ULONG_MAX :  18446744073709551615
USHRT_MAX :  65535
```

Floating-Point Types

The following table provides the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range	Precision
Float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
Double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values −

```
#include  <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

int main(int argc, char** argv) {

   printf("Storage size for float : %d \n", sizeof(float));
   printf("FLT_MAX     : %g\n", (float) FLT_MAX);
   printf("FLT_MIN     : %g\n", (float) FLT_MIN);
   printf("-FLT_MAX    : %g\n", (float) -FLT_MAX);
   printf("-FLT_MIN    : %g\n", (float) -FLT_MIN);
   printf("DBL_MAX     : %g\n", (double) DBL_MAX);
   printf("DBL_MIN     : %g\n", (double) DBL_MIN);
   printf("-DBL_MAX     : %g\n", (double) -DBL_MAX);
   printf("Precision value: %d\n", FLT_DIG );

   return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux −

```
Storage size for float: 4

FLT_MAX     : 3.40282e+38

FLT_MIN     :  1.17549e-38
```

-FLT_MAX	:	-3.40282e+38
-FLT_MIN	:	-1.17549e-38
DBL_MAX	:	1.79769e+308
DBL_MIN	:	2.22507e-308
-DBL_MAX	:	-1.79769e+308

Precision value: 6

Void Type

The void type specifies that no value is available. It is used in three kinds of situations −

Sr.No.	Types & Description
1	**Function returns as void** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

Character type

A single character can be defined as a character(char) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The quantifier **signed or unsigned** may be explicitly applied to char. While **unsigned chars** have values between 0 and 25, **signed chars** have values from -128 to 127.

- - ◌◌◌ - -

CHAPTER 03
OPERATORS

Definition

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Increment and Decrement Operators
- Conditional Operators
- Special Operators

Arithmetic Operators

- *C provides all the basic arithmetic operators. These can operate on any built-in data type allowed in c.*
- The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then Show Examples

Operator	**Description**	**Example**
+	Unary Plus or Adds two operands.	A + B = 30
−	Unary minus or Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called integer arithmetic. Integer arithmetic always yields an integer value.

Example

If a and b are integers, then for a=14 and b=4 we have the following results.

*a-b=10, a+b=18, a*b=56, a/b=3(decimal part truncated), a%b = 2(remainder of division).*

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation.

Example

X=6.0/7.0= 0.857143

Y= -2.0/3.0 = -0.666667

The operator % cannot be used with real operands.

Mixed - mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed- mode arithmetic expression.

Example

15/10.0 = 1.5

Whereas 15/10 =1

Relational Operators

We often compare two quantities and depending on their relation, take certain decisions.

Syntax

ae-1 relational operator ae-2

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then −

Show Examples

Operator	Meaning	Description	Example
==	is equal to	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	is not equal to	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	is greater than	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	is less than	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	is greater than or equal to	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	is less than or equal to	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

The logical operators && and || are used when we want to test more than one condition and make decisions.

Example

a>b && x==10

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression.

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

Show Examples

Operator	Meaning	Description	Example
&&	Logical AND	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Logical OR	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Logical NOT	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double.

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Show Examples

Operator	Meaning	Description	Example
&	Bitwise AND	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Bitwise OR	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Bitwise exclusive OR	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Bitwise tilde	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Shift left	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Shift right	Binary Right Shift Operator. The left operands value is	A >> 2 = 15 i.e., 0000 1111

Operator	Meaning	Description	Example
		moved right by the number of bits specified by the right operand.	

Assignment Operators

- Assignment operators are used to assign the result of an expression to a variable.
- C has a set of short hand assignment operators of the form

Syntax

V op=exp;

is equivalent to v=v op (exp);

Three advantages

- What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
- The statement is more concise and easier to read
- The statement is more efficient.

The following table lists the assignment operators supported by the C language −

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right	C /= A is equivalent to C = C / A

Operator	Description	Example
	operand and assigns the result to the left operand.	
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Increment and Decrement Operators

C allows two very useful operators not generally found in other language.

- ***Increment operator ++***
- ***Decrement operator - -***

The operator ++ adds 1 to the operand, while -- subtracts 1. Both are unary operators and takes the following form

++m; or m++;

- -m; or m- -;

++ m is equivalent to m=m+1;

m++ is equivalent to m=m+1;

Rules

- Increment and Decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.

- When prefix ++ (or --) is used in an expression, the variable is incremented (or decremented)first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of ++ and -- operators are the same as those of unary + and unary -.

Example

m=5; y=++m;

In this case, the value of y and m would be 6. A prefix operator first adds 1 to the operand then the result is assigned to the variable on left.

Suppose, if we rewrite the above statements as

m=5; y=m++;

Then the value of y would be 5 and m would be 6. A postfix operator first assigns the value to the variable on left then increments the operator.

Conditional Operator

The ternary operator pair **"? : "** is available in C to construct conditional expressions of the form

Syntax

exp1? exp 2 : exp3

where exp1,exp2and exp3 are expressions

The operator ?works as follows : ep1 is evaluated first. If it is nonzero(true), then the expressionexp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

Example

a=10;

b=15;

x=(a>b) ? a: b;

Special Operators

C supports some special operators of interest such as comma operator, sizeof operator, pointer operators(& and *) and member selection operators(. and ->)

Show Examples

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

The Comma operator

The comma operator can be used to link the related expressions together.

Example

Value = (x=10,y=5, x+y);

Sizeof operator

The sizeof is a compile time operator and,when used with an operand,it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples

M= sizeof(sum);

N=sizeof(long int);

K= sizeof(235L);

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others;

For example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

- - ଔଔଔ - -

CHAPTER 04
MANAGING INPUT AND OUTPUT OPERATIONS

Reading, processing, and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as information or results.

Reading a Character

getchar() function reads character type data from the standard input. It reads one character at a time till the user presses the enter key.

Syntax

variable_name = getchar();

Example

char name;

name = getchar();

it will assign the character „H" to the variable name when we press the key Hon the keyboard. Since getchar is a function, it requires a set of parentheses.

Writing a Character

putchar() function prints one character on the screen at a time which is read by the standard input.

Syntax

putchar (variable_name);

where variable_name is a type char variable containing a character. This statement displays the character contained in the variable name at the terminal.

Example

Answer = „Y";

putchar (Answer);

It will display the character Y on the screen.

Formatted Input

Formatted input refers to an input data that has been arranged in a particular format.

The scanf() Statement

The scanf() Statement reads all types of data values. It is used for runtime assignment of variables. **The scanf() Statement** also requires conversation symbol to identify the data to be read during execution of a program.

Syntax

scanf("control string" , arg1,arg2…argn);

Example

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

The **scanf()** statement requires „&" operator called address operator. The address operator prints the memory location of the variable. Here, in the scanf() statement the role of „&" operator is to indicate the memory location of the variable, so that the value read would be placed at that location.

Inputting integer numbers

The field specification for reading an integer number is:

%wsd

The percentage sign (%) indicates that a conversion specification follows. w is an integer number that specifies the field width of the number to be read and d, known as data type character, indicates that the number to be read is in integer mode.

Example

scanf("%2d %5d", & num1, & num2);

Inputting real numbers

scanf() reads real numbers using the simple specification %f for both the notations, namely decimal point notation and exponential notation.

Example

Scanf(“ %f %f %f ” , &x,&y,&z);

With the input data

475.89 43.21 34.56

Inputting Character Strings

A single character can be read from the terminal using the getchar function. The same can be achieved using the scanf function also.

%ws or %wc

Formatted Output

Formatted output refers to an output data.

printf() statement

The printf()function prints all types of data values to the console. It requires conversion symbol and variable names to print the data. The conversion symbol and variable names should be same in number.

Syntax

printf(“control string” , arg1,arg2…argn);

Example

printf(“%d %f %c” , a,b,c);

Outputting integer numbers

The format specification for printing an integer number is:

%wd

The percentage sign (%) indicates that a conversion specification follows. w specifies the minimum field width for the output, and d, known as data type character, indicates that the number to be printed is in integer mode.

Example

printf("%2d ",9876);

Output of the real numbers

The output of a real number may be displayed in decimal notation using the following format specification.

%wpf

Example

printf(" %f " , number);

Printing a Single Character

A single character can be displayed in a desired position using the format.

%wc

The character will be displayed right-justified in the field of w columns.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the form.

%w.ps

The characters will be displayed right-justified in the field of w columns.

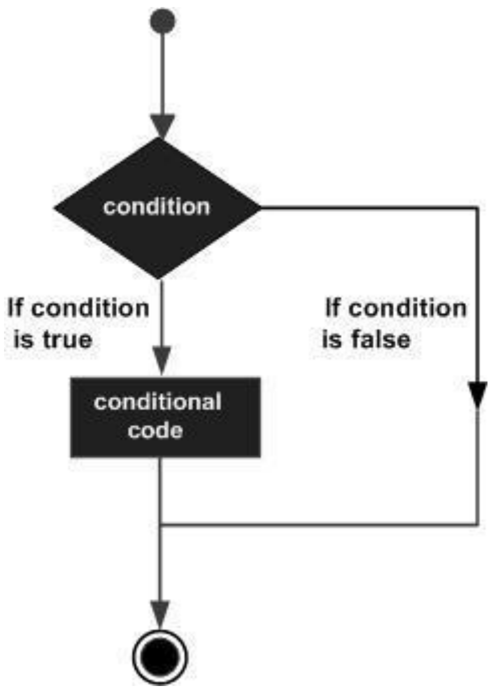
- - ☙☙☙ - -

CHAPTER 05
DECISION MAKING AND BRANCHING

Introduction

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.

Show below is the general form of a typical decision-making structure found in most of the programming languages −



C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides the following types of decision-making statements.

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5	nested switch statements You can use one **switch** statement inside another **switch** statement(s).

if Statement

Definition

C uses the keyword **if** to execute a set of command line or one command line when the logical condition is true. It has only one option.

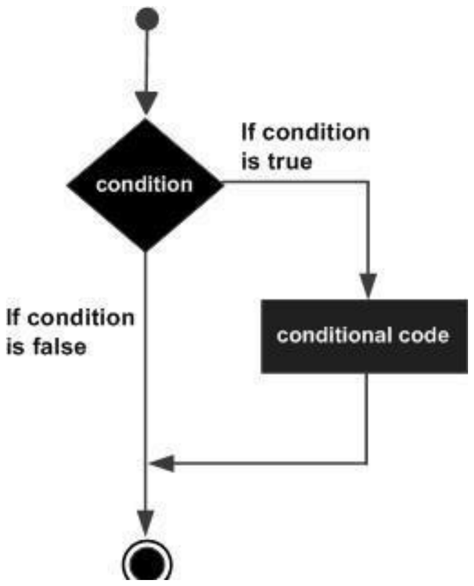
Syntax

```
if(conditon)
 {
   /* statement(s) will execute if the condition is true */
}
```

Explanation

If the condition evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the test condition evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
#include <stdio.h>

void main () {
        /* local variable definition */
        int a = 10;

        /* check the test condition using if statement */

        if( a < 20 ) {
            /* if condition is true then print the following */
            printf("a is less than 20\n" );
        }

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

When the above code is compiled and executed, it produces the following result −

```
a is less than 20

value of a is : 10
```

if…else Statement

The **if…else** statement takes care of true as well as false conditions. It has two blocks. One block is for if and it is executed when the condition is true. The other block is of else and it is executed when the condition is false. The else statement cannot be used without if.

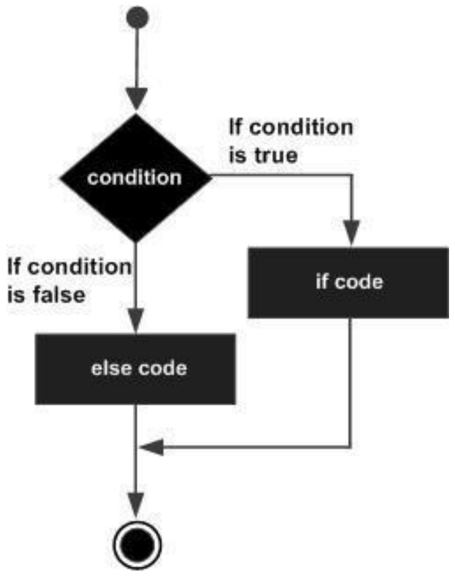
Syntax

```
if( condition)
{
  /* statement(s) will execute if the condition is true */
}
else
 {
  /* statement(s) will execute if the condition is false */
}
```

If the condition evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram



Example

```
#include<stdio.h>

void main (){

        /* local variable definition */
        int a =100;

        /* check the 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);


}
```

When the above code is compiled and executed, it produces the following result −

```
a is not less than 20
value of a is : 100
```

Nested if...else Statement

if any logical condition is true the compiler executes the block followed by if condition otherwise it skips and executes else block. In if .. else statement else block is executed by default after failure of condition. This kind of nesting will be unlimited.

Rules

- Nested if..else can be chained with one another
- If the condition is false control passes to else block where condition is again checked with the if statement. This process continues if there is no if statement in the last else block.
- If one of the if statements satisfies the condition, other nested else…if will not be executed.

Syntax

```
if(condition 1) {
   /* Executes when the condition1 is true */
} else if( condition 2) {
   /* Executes when the condition 2 is true */
} else if(condition 3) {
   /* Executes when the condition 3 is true */
} else {
   /* executes when the none of the above condition is true */
}
```

Example

```
#include<stdio.h>

void main (){

/* local variable definition */
int a =100;

/* check the condition */
if( a ==10){
/* if condition is true then print the following
*/
    printf("Value of a is 10\n");
}elseif( a ==20){
/* if else if condition is true */
    printf("Value of a is 20\n");
}elseif( 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 );
}
```

When the above code is compiled and executed, it produces the following result −

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

The Else if ladder

A multipath decision is a chain of ifs in which the statement associated with each else is an if.

Syntax

```
if(condition 1)
        statement1;
else if(condition 2)
        statement 2;
else if(condition 3)
        statement 3;
 else if(condition n)
        statement n;
else
        default – statement;
statement x;
```

```
#include<stdio.h>

void main (){
```

```
/* local variable definition */
int a =100;

/* check the condition */
if( a ==10){
/* if condition is true then print the following
*/
    printf("Value of a is 10\n");
}elseif( a ==20){
/* if else if condition is true */
    printf("Value of a is 20\n");
}elseif( 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 ); return0;
}
```

When the above code is compiled and executed, it produces the following result −

```
None of the values is matching

Exact value of a is: 100
```

switch statement

The switch statement is a multi-way branch statement. The switch statement evaluates expression and then looks for its value among the case constants. if the value matches with case constant, this particular case statement is executed. If not, default is executed.

Syntax

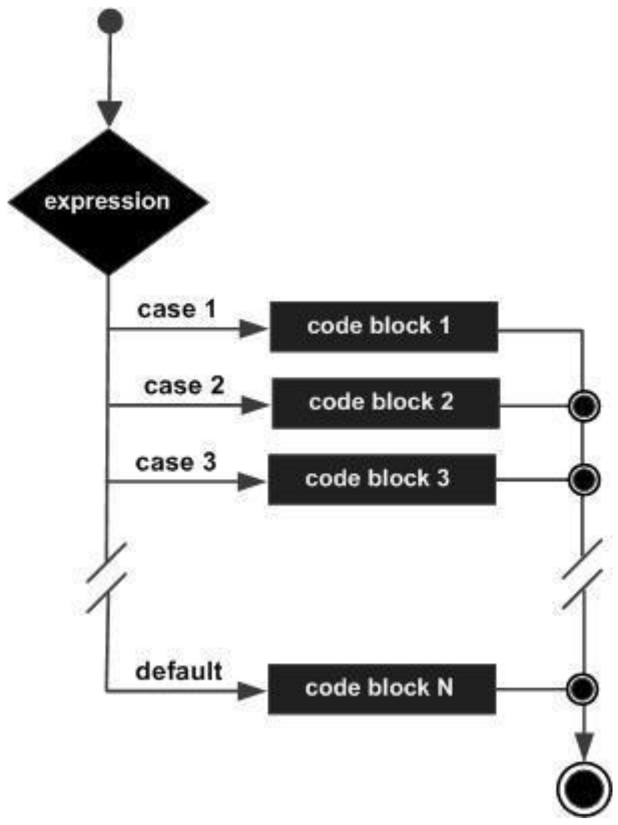
```
switch(expression) {
  case constant-expression :
    statement(s);
    break; /* optional */
  case constant-expression :
    statement(s);
    break; /* optional */
  /* you can have any number of case statements */
  default : /* Optional */
  statement(s);
}
```

Rules

The following rules apply to a **switch** statement −

- 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.

Flow Diagram



Example

```
#include <stdio.h>

void main () {

  /* local variable definition */
  char grade = 'B';

  switch(grade) {

    case 'A' :
      printf("Excellent!\n" );
      break;
    case 'B' :
    case 'C' :
      printf("Well done\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;
}
```

When the above code is compiled and executed, it produces the following result −

```
Well done
Your grade is B
```

The ? : Operator

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form –

Syntax

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this −

- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.
- We may encounter situations, when a block of code needs to be executed several numbers of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that

allow for more complicated execution paths.

- A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages –

Go to Statement

The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred.

Syntax

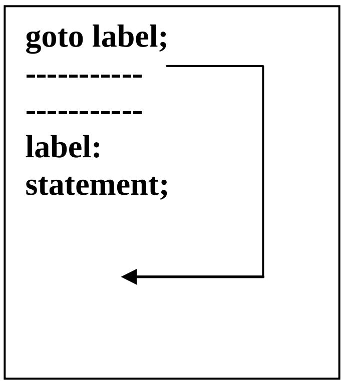


Fig. 5.8 (a) Forward Jump

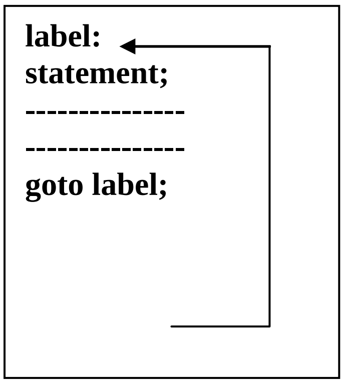


Fig. 5.8 (b) Backward Jump

Example

```
#include<stdio.h>
void main()
{
        double x,y;
        read:
        scanf("%f",&x);
        if(x<0) goto read;
        printf("%f",x);
}
```

- - ଔଔଔ - -

CHAPTER 06

DECISION MAKING AND LOOPING

Introduction

C programming language provides the following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop It is more like a while statement, except that it tests the condition at the end of the loop body.
4	nested loops You can use one or more loops inside any other while, for, or do..while loop.

while statement

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax

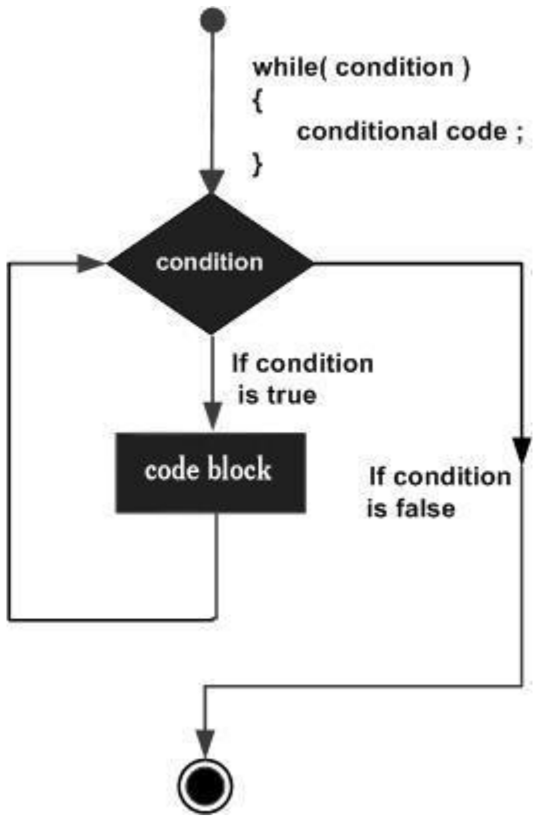
The syntax of a **while** loop in C programming language is −

```
while(condition) {
  statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

Flow Diagram



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#include <stdio.h>

void main () {
   /* local variable definition */
   int a = 10;

   /* while loop execution */
while(a < 20) {
      printf("value of a: %d\n", a);
      a++;
   }

}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is −

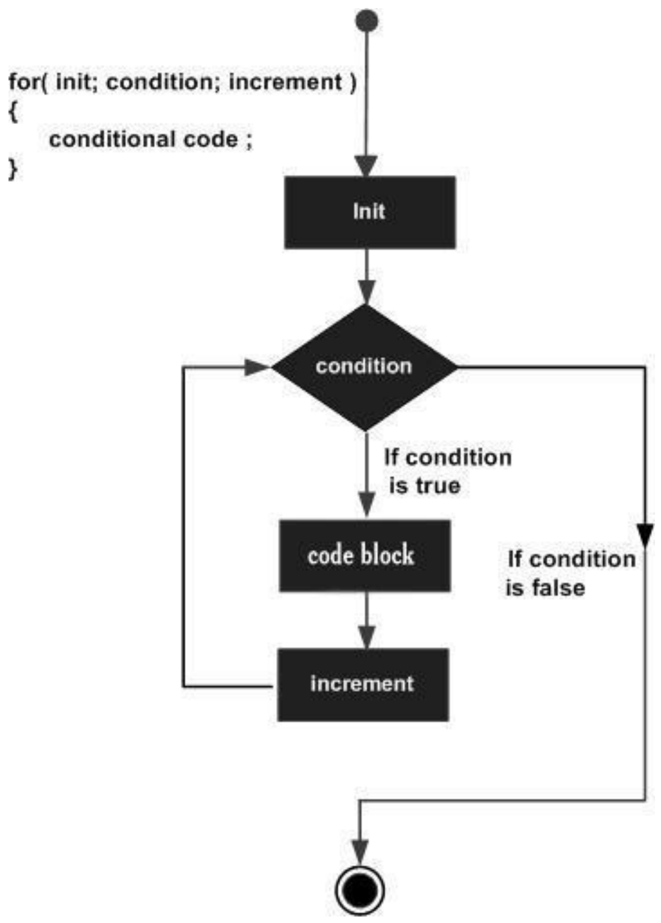
```
for ( init; condition; increment ) {
   statement(s);
}
```

Here is the flow of control in a 'for' loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step,

and then again condition). After the condition becomes false, the 'for' loop terminates.

Flow Diagram



Example

```
#include <stdio.h>

void main () {

   int a;

   /* for loop execution */
   for( a = 10; a < 20; a = a + 1 ){
      printf("value of a: %d\n", a);
   }

}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

do… while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

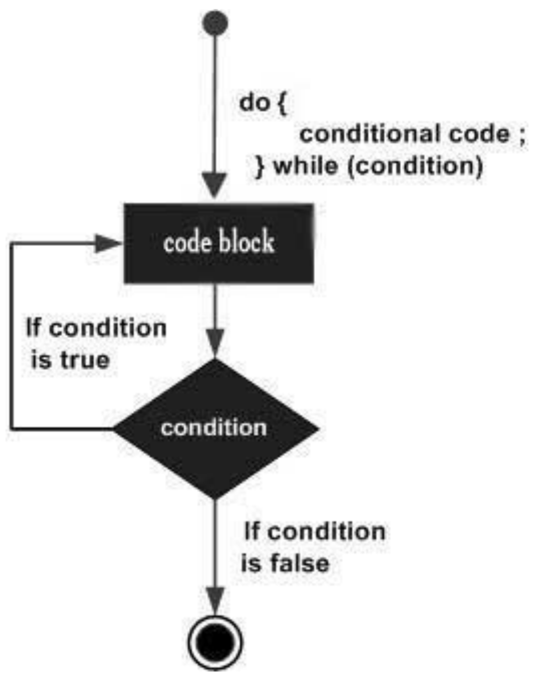
The syntax of a **do...while** loop in C programming language is −

```
do {
  statement(s);
} while(condition);
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



Example

```
#include <stdio.h>

void main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {
      printf("value of a: %d\n", a);
      a = a + 1;
   }while( a < 20 );

}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested for loop

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

```
for ( init; condition; increment ) {
  for ( init; condition; increment ) {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows −

```
while(condition) {
  while(condition) {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows −

```
do {
  statement(s);
  do {
    statement(s);
  }while(condition );
}while(condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

```
#include <stdio.h>

void main () {

   /* local variable definition */
   int i, j;

   for(i = 2; i<100; i++) {

      for(j = 2; j <= (i/j); j++)
      if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime\n", i);
   }


}
```

When the above code is compiled and executed, it produces the following result −

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
```

```
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
    89 s prime
97 s prime
```

- Loop Control Statements
- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- C supports the following control statements.

Sr.No.	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement Transfers control to the labeled statement.

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>

void main () {

  for( ; ; ) {
    printf("This loop will run forever.\n");
  }


}
```

- When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.
- **NOTE** − You can terminate an infinite loop by pressing Ctrl + C keys.

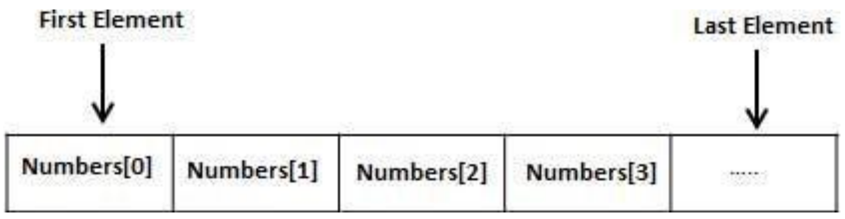
CHAPTER 07
ARRAYS

Definition

An Array is a fixed-size sequenced collection of elements of the same data type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



One – Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a single – subscripted variable or a one – dimensional array.

Declaring Array

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

Syntax

type arrayName [arraySize];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type.

For example, to declare a 10-element array called **balance** of type double, use this statement −

double balance[10];

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

After an array is declared, its element must be initialized. Otherwise, they will contain “garbage”. An array can be initialized at either of the following stages.

- At compile time
- At run time

Compile Time initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

Syntax

Type array_name[size]={ list of values };

Example

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

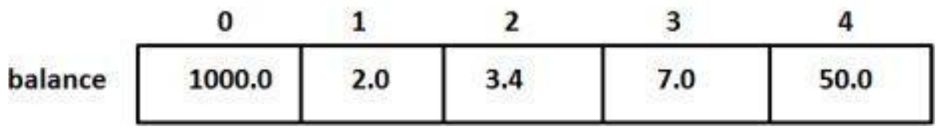
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

Example

int x[3];

scanf(“%d %d%d”, &x[0], &x[1],&x[2]);

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

For example −

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable.

The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

```
#include <stdio.h>

void main () {

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

  /* initialize elements of array n to 0 */
  for ( i = 0; i < 10; i++ ) {
    n[ i ] = i + 100; /* set element at location i to i + 100 */
  }

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


}
```

When the above code is compiled and executed, it produces the following result −

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

Two – Dimensional Array

Two – dimensional array can be thought as a rectangular display of elements with rows and columns. The two – dimensional array is a collection of a number of one - dimensional array can be thought of as a single – dimensional array.

Syntax

Type array_name[row_size][column_size];

Example

int [3][3];

Initializing two – dimensional arrays

Two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Syntax

Type array_name[row_size][column_size]={list of values};

Example

Write a program to display the elements of two –dimensional array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j;
int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
clrscr();
printf("Elements of an Array.\n\n");
for(i=0;i3;i++)
{
for(j=0;j<3;j++)
printf("%5d",a[i][j]);
printf("\n");
}
}
```

Output

```
Elements of an array
1 2 3
4 5 6
7 8 9
```

Three or Multidimensional Arrays

C allows arrays of three or more dimensions. The compiler determines the restriction on it.

Syntax

Type array _name[s1][s2][s3]…s[n];

Example

Write a program to explain the working of three dimensional array.

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

void main()
{
int array_3d[3][3][3];
int a,b,c;
clrscr();
for(a=0;a<3;a++)
for(b=0;b<3;b++)
for(c=0;c<3;c++)
array_3d[a][b][c]=a+b+c;

for(a=0;a<3;a++)
{
printf("\n");
for(b=0;b<3;b++)
{
for(c=0;c<3;c++)
printf("%3d",array _d[a][b][c]);
printf("\n");
}
}
}
```

Output

```
0 1 2
1 2 3
2 3 4

1 2 3
2 3 4
3 4 5

2 3 4
3 4 5
4 5 6
```

- - ☙☙☙ - -

CHAPTER 08

CHARACTER ARRAYS AND STRINGS

Introduction

A string is a sequence of characters that is treated as a single data item. Any group of characters (except double quote sign) defined between double quotation marks is a string constant.

Example

printf("\" Well Done!"\");

Output

"Well Done!"

Declaration and Initialization of Strings

C does not support strings as a data type. however, it allows us to represent strings as character arrays.

Syntax

char string_name[size];

Example

char city[10];

Initialization of String variables

C permits a character array to be initialized in either of the following two forms

Syntax

char string_name[size] = { list of values};

Example

char city[9]="NEW YORK";

Write a Program to Display the Output When the Account of Null Character is not considered

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

char city[8]={„N"‚"E"‚"W", „Y", „O", „R", „K"};

clrscr();

printf("Name1=%s", name1);

}

Output

NEW YORK

String Handling Functions

C library supports a large number of string – handling functions that can be used to carry out many of the string manipulations.

Function	Action
Strcat()	Concatenates two strings
Strcmp()	Compares two strings
Strcpy()	Copies one string over another
Strlen()	Finds a length of a string

strcat() function

The strcat function joins two strings together.

Syntax

strcat(string1,string2);

string1 and string2 are character arrays. When the function strcat is executed, string2 is appended to string1.

strcmp() function

The strcmp function compares two strings identified by the arguments and has a value 0 if they are equal. if they are not , it has the numeric difference between the first non - matching characters in the strings.

Syntax

strcmp(string1, string2);

string1,string2 may be string variables or string constants.

Examples

strcmp(name1, name2);

strcmp(“ROM”, “RAM”);

8.4.3 strcpy() function

strcpy function works almost like a string-assignment operator.

Syntax

strcpy(string1,string2);

And assigns the contents of string2 to string1. string2 may be a character array variable or a string constant.

Example

strcpy(city,”DELHI”);

strcpy(city1,city2);

Will assign the contents of the string variable city2 to the string variable city1. The size of the array city1 should be large enough to receive the contents of city2.

8.4.4. strlen() function

This function counts and returns the number of characters in a string. It takes the form

Syntax

n=strlen(string);

where n is an integer variable, which receives the value of the length of the string. The argument may be a string constant. The continuing ends at the first null character.

Example

Write a program of a given string is palindrome or not

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
void main()
{
        char S1[50],S2[50];
        int x;
        clrscr();
        printf("\n\t\t\t ******** PALINDROME ******** \n\n\n");
        printf("\n\n\n\n\n Enter a string S1 \n\n\n\n\n");
        scanf("%s",&S1);
        strcpy(S2,S1);
        x=strcmp(S2,strrev(S1));
        if(x==0)
                printf("\n\n\n\n\n %s \t is palindrome \n",S1);
        else
                printf("\n\n\n\n\n %s \t is not palindrome \n",S2);
        getch();
}
```

Output:

```
******************PALINDROME******************

Enter String S1:

madam

madam is a palindrome

sir

sir is not palindrome
```

- - ꧁꧁꧁ - -

CHAPTER 09
POINTERS

Definition

A pointer is a memory variable that stores a memory address. Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variables but it is always denoted by „*" operator.

Accessing the address of the variable

Every variable is a memory location and every memory location have its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

Consider the following example, which prints the address of the variables defined −

```
#include <stdio.h>
void main () {
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1 );
   printf("Address of var2 variable: %x\n", &var2 );
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

Pointer Declaration

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

Syntax

type *var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Initialization of Pointer Variables

There are a few important operations, which we will do with the help of pointers very frequently.

a) We define a pointer variable.

b) Assign the address of a variable to a pointer and

c) Finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.

The following example makes use of these operations –

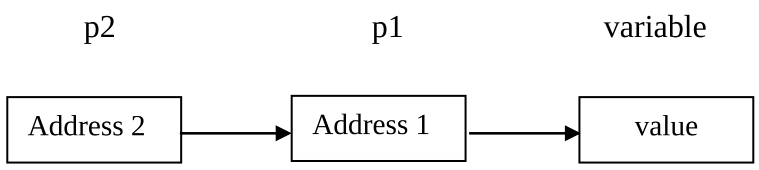
```
#include <stdio.h>

void main () {
   int var = 20; /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var; /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );
   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );
   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Chain of pointers

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown



Here, the pointer variable p2 contains the address of the pointer variable p1, which points to the location that contains the desired value. This is known as multiple indirections.

Example

int **p2;

This declaration tells the compiler that p2 is a pointer to a pointer of int type.

Pointer expression

Pointer variable can be used in expressions.

Example

Y=*p1 * *p2;

Pointers and Arrays

Array name by itself an address or pointer. It points to the address of the first element (0th element of an array).

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.

Example

Write a program to display elements of an array. Start element counting from 1 instead of 0.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x[]={2,4,6,8,10}, k=1;
clrscr();
while(k<=5)
{
printf("%3d",k[x-1]);
k++;
}
}
```

Output

```
2 4 6 8 10
```

Array of Pointers

C language also supports array of pointers. It is nothing but a collection of addresses.

Example

Write a program to store addresses of different elements of an array using array of pointers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int *ap[3];
int at[3]={5,10,15}k;
for(=0;k<3;k++)
ap[k]=at+k;
clrscr();
printf("\n\t Address Element\n");
for(k=0;k<3;k++)
{
printf("\t%u",ap[k]);
printf("\t%7d\n",*(ap[k]);
}
}
```

OUTPUT

```
Address        Element
4060           5
4062           10
4064           15
```

Pointers and Structures

The name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variable.

Example

```
struct inventory
{
char name[30];
int number;
float price;
}
product[2],*ptr;
```

This statement declares product as an array of two elements. The pointer ptr will now point to product[0]. Its members can be accessed using the following notation.

```
ptr →name

ptr →number

ptr →price
```

The symbol→ is called the arrow operator(also known as member selection operator)and is made up of a minus sign and a greater than sign.

Pointers to Functions

A function, like a variable, has a type and an address location in the memory. it is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

Syntax

type(*fptr)();

This tells the compiler that fptr is a pointer to a function, which returns type value. The parentheses around *fptr are necessary.

Syntax

type *gptr();

would declare gptr as a function returning a pointer to type.

Example

double mul(int, int);

double (*p1)();

p1=mul;

POINTER INCREMENTS AND

SCALE FACTOR

When we increment a pointer, its value is increased by the 'length' of the data type that is points to. This length called the scale factor.
Example
P1++

We cause the pointer p1 to point to the next value of its type.

For example , if p1 is an integer pointer with an initial value , say 2800, then after the operation p1=p1+1, the value of p1 will be 2802 and not 2801.

The number of bytes used to store various data types depends on the system and can be found by making use of sizeof operator.

POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program

```
#include <stdio.h>
void main () {
  int *ptr = NULL;
  printf("The value of ptr is : %x\n", ptr );
}
```

When the above code is compiled and executed, it produces the following result −

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

- - ଔଔଔ - -

CHAPTER 10
FUNCTION

Introduction

Function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

A function is a self – contained block or a sub – program of one or more statements that performs a special task when called.

The general form of a function definition in C programming language is as follows –

Syntax

```
return_type function_name( parameter list ) {
   body of the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
```

```
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration is also known as function prototype

A function declaration has the following parts –

Syntax

return_type function_name(parameter list);

For the above defined function max(), the function declaration is as follows −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Return values and their types

A function may or may not send back any value to the calling function. If it does, it is done through the return statement. When it is possible to pass to the called function any number of values, the called function can only return on value per call, at the most.

Syntax

```
return;
or
return(expression);
```

Example

```
int mul(int x, int y)
{
int p;
p=x*y;
return(p);
}
```

returns the value of p which is the product of the values of x and y.

Function Calls

A function can be called by simply using the function name followed by a list of actual parameters (or arguments), if any, enclosed in parentheses.

Example

```
main()
{
int y;
y= mul(10,5); /* function call*/
printf("%d\n",y);
}
```

When the compiler encounters a function call, the control is transferred to the function mul(). This function is then executed line by line as described and a valueis returned when a return statement is encountered. This value is assigned to y.

Recursion

A function is called repetitively by itself. The recursion can be used directly or indirectly.

- The direct recursion function calls itself till the condition is true

- In indirect recursion a function calls another function then the called function calls the calling function

Example

Write a program to call main() function recursively and perform sum of 1 to 5 numbers.

```
#include<stdio.h>
#include<conio.h>
int x,s;
void main(int);
void main(x)
{
s=s+x;
printf("\n x=x%d s=%d",x,s);
if(x==5)
exit(0);
main(++x);
}
```

OUTPUT

```
x=1 s=1
x=2 s=3
x=3 s=6
x=4 s=10
x=5 s=15
```

Categories of Function

A function, depending on whether arguments are present or not and whether a value is returned or not, ma belong to one of the following categories.

Category 1: Functions with no arguments and no return values

Category 2: Functions with arguments but no return values

Category 3: Functions with arguments and one return value

Category 4: Functions with no arguments but return a value

Category 5: Functions that return multiple values.

No Arguments and no Return values

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function.

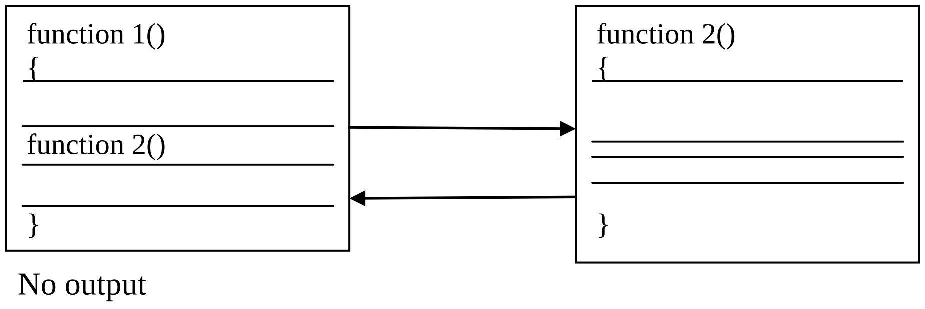


Figure 10.7.1 No data communication between functions

```
#include<stdio.h>
void introduction()
{
printf("Hi\n");
}
int main()
{
introduction();
return 0;
}
```

Output

```
Hi
```

Arguments but no Return values

- The main function has no control over the way the functions receive input data.
- The nature of data communication between the calling function and the called function with arguments but no return value.

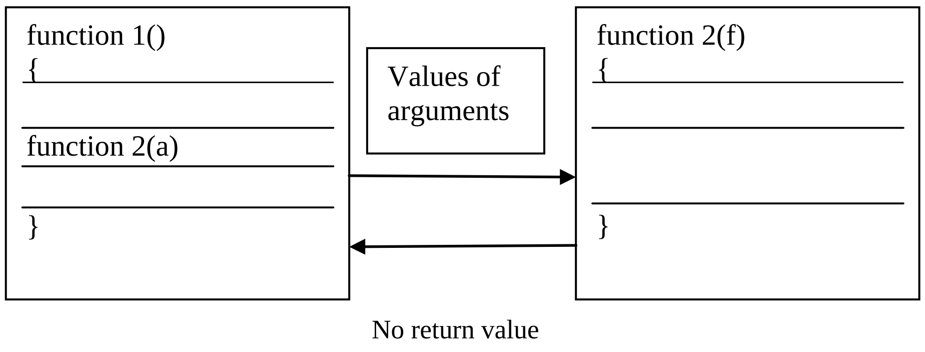


Figure 10.7. 2. one-way data communication

```
#include<stdio.h>

int sum(int a,int b)

{

 int c=a+b;

}

void main()

{

int var1=10,var2=20;

int var3=sum(var1,var2);

printf("%d",var3);

}
```

The arguments ch, p, r and n are called the formal arguments. The calling function values to these arguments using function calls containing appropriate arguments.

Arguments with Return values

It receives data from the calling function through arguments, but does not send back a value. A self-contained and independent function should behave like a „black box‟ that receives a predefined form of inputs and outputs a desired value. Such functions will have to – way data communication. The use of to – ay data communication between the calling and the called functions.

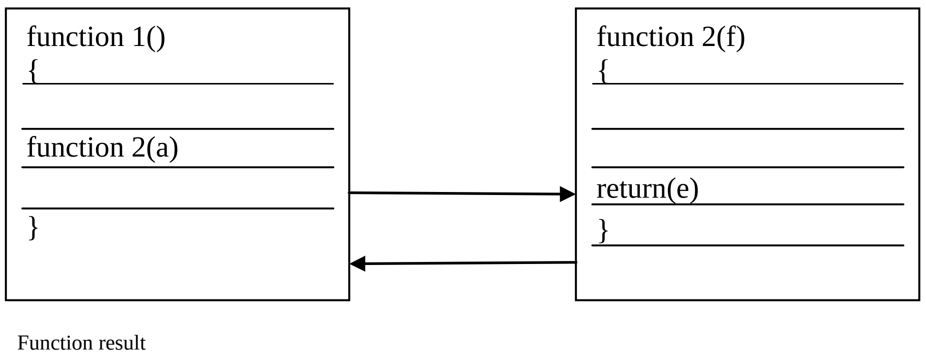


Figure 10.7.3. two-way data communication

Example

```
#include<stdio.h>
int sum(int a,int b)
{
 int c=a+b;
return c;
}
int main()
{
int var1=10,var2=20;
int var3=sum(var1,var2);
printf("%d",var3);
return 0;
}
```

No Arguments but Return a value

It may not take any arguments but returns a value to the calling function.

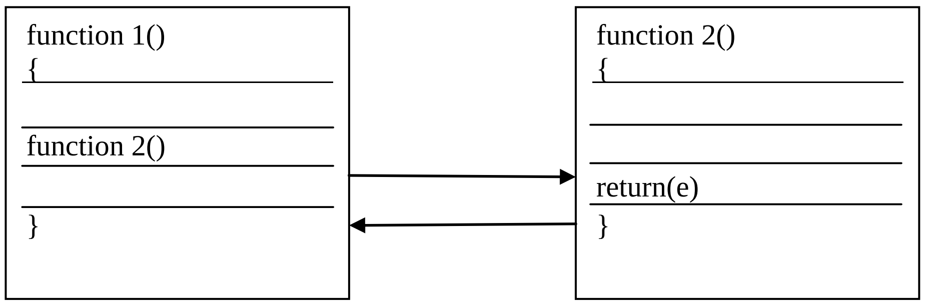


Figure 10.7.4. one-way data communication

Example

```
        int get_number(void);
main()
{
int m= get_number();
printf("%d",m);
}
int get_number(void)
{
int number;
scanf("%d",&number);
return(number);
}
```

Functions that Return Multiple values

We have functions that return just one value using a return statement. We can also force the function to return more values per call. It is possible to call by the reference method.

The mechanism of sending back information through arguments through arguments is achieved using what are known as the address operator(&) and indirection operator(*).

Example

Write a program to pass arguments to user- defined function by value and reference.

```
#include<stdio.h>
#include<conio.h>
main()
{
int k,m,other(int,int*);
clrscr();
printf("\n Address of k & m in main(): %u%u", &k,&m);
other(k,&m);
return0;
}
other(intk,int *m)
{
printf("\n Address of k & m in other(): %u%u",&k,&m);
}
```

OUTPUT

```
Address of k & m in main() : 65524 65522
Address of k & m in other() : 65518 65522
```

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

For example –

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

We have kept max() along with main() and compiled the source code.

While running the final executable, it would produce the following result –

```
Max value is : 200
```

- - ൽൽൽ - -

CHAPTER 11
STRUCTURE

Introduction

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly, **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows.

Syntax

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int  book_id;
} book;
```

11.3 Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type.

The following example shows how to use a structure in a program −

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

struct Books {
   char title[50];
   char  author[50];
   char subject[100];
   int  book_id;
};

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

We can pass a structure as a function argument in the same way as you pass any other variable or pointer.

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

struct Books {
   char title[50];
   char  author[50];
   char subject[100];
   int  book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printBook( Book1 );

   /* Print Book2 info */
   printBook( Book2 );

   return 0;
}

void printBook( struct Books book ) {

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

We can define pointers to structures in the same way as you define pointer to any other variable −

struct Books *struct_pointer;

Now, we can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, we must use the → operator as follows −

```
struct_pointer->title;
```

Let us re-write the above example using structure pointer.

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

struct Books {
   char title[50];
   char  author[50];
   char subject[100];
   int  book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info by passing address of Book1 */
   printBook( &Book1 );

   /* print Book2 info by passing address of Book2 */
   printBook( &Book2 );

   return 0;
}

void printBook( struct Books *book ) {

   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include −

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in, e.g., 9-bit integers.

C allows us to do this in a structure definition by putting: bit length after the variable. For example −

```
struct packed_struct {
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
   unsigned int type:4;
   unsigned int my_int:9;
} pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4-bit type and a 9-bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next word.

Unions

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. a union can be declared using the keyword union as follows.

```
union item
{
        int m;
        float x;
        char c;
}
code;
```

This declares the variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

- - ଔଔଔ - -

CHAPTER 12
FILES

Introduction

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

Opening Files

We can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows.

Syntax

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

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values −

Sr.No.	Mode & Description
1	**r** Opens an existing text file for reading purpose.
2	**w** Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
3	**a** Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
4	**r+** Opens a text file for both reading and writing.
5	**w+**

Sr.No.	Mode & Description
	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
6	**a+** Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones −

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is −

Syntax

int fclose(FILE *fp);

The **fclose()** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

12.4 Writing a File

Following is the simplest function to write individual characters to a stream −

Syntax

int fputc(int c, FILE *fp);

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. We can use the following functions to write a null-terminated string to a stream −

int fputs(const char *s, FILE *fp);

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example.

Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>
main() {
  FILE *fp;

  fp = fopen("/tmp/test.txt", "w+");
  fprintf(fp, "This is testing for fprintf...\n");
  fputs("This is testing for fputs...\n", fp);
  fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

12.5. Reading a File

The **fgetc()** function reads a character from the input file referenced by fp.Given below is the simplest function to read a single character from a file −

Syntax

int fgetc(FILE * fp);

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream –

char *fgets(char *buf, int n, FILE *fp);

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

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

   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result −

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more in detail about what happened here. First, **fscanf()** read just This because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

12.6 Binary I/O Functions

There are two functions, that can be used for binary input and output −

```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements,
FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t
number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

References

- ❖ Programming with ANSI and Turbo C – Ashok N.Kamthane.
- ❖ Programming in ANSI C – E. Balagurusamy, Sixth Edtion.
- ❖ www. Tutorialpoint.com

- - ☙☙☙ - -

ROUGH WORK

ROUGH WORK