CONTENTS

Chapter -1
Introduction to C

History of programming language

The history behind the programming language is given below. Here written what languages are developed up to ANSI C, but what time I maintain which is approximate time.

Remarks	***Language/OS***	***Year***
A team lead by **John W. Backus** developed a numerically orientated language called FORTRAN	**FORTRAN(FORmula TRANSlation)**	1954-57
IBM developed Assembly language	**Assembly Languages**	1956-63
It is a structured programming language was developed by European and American computer scientists	**ALGOL(ALGOrithmic Language)**	1958
Martin Richards wrote BCPL Dennis M. Ritchie joined Bell Labs.	**BCPL (Basic Combined Programming Language)**	1967
Challenged by McIlroy's feat in reproducing TMG, **Ken Thompson** wrote a system programming language called **B**	**B**	1969-70
Dennis M. Ritchie added few more features to NB and C was born.	**C**	1971-73
ANSI X3J11 committee came out with a new and decent standard for C	**ANSI C**	1989
ANSI C standard was also accepted by ISO as ISO/IEC 9899-1990		1990

<u>History of C:</u>

C is a professionally wide-ranging programming language. It is also called as a system programming language because most of the operating systems (system software), application software and compliers were written in this language. It is useful to develop operating system.
At first the idea of the C was from the language BCPL which is developed by Martin Richards. After that through BCPL the language B was developed, which was written by Ken Thomson in 1970 for first UNIX system.

The main Drawback of BCPL and B language are they are "*typeless*" language. To overcome this C provides different data types. The general data types are int, (for integer type), float (for floating point type) with several size and char (character type). Apart from this there

are some derived data types.
The initial development of C occurred at AT&T Bell Labs between 1969 and 1973 by Brian Kernighan and Dennis Ritchie. The development of C was the result of the programmers' desire to play Space Travel on an idle computer in their office. By 1973, the C language had become powerful enough that most of the UNIX kernel was rewritten in C. C is now the basis for all Unix/Linux operating systems.

Why we learn C?

C is very close to the internals of a computer: CPU, Memory, and Input/output. Hence, it gives very good insight into what goes on 'marine' in packages. You can easily understand how a work can control at the time of program execution. C is very commonly used in scientific computing.

At first you will start with basics writing the code, compiling and running the programs. How to do calculations, input/output also. *Learn by solving computational problems in economics*. E.g. linear algebra, (static and dynamic) optimization,

Note: *C programming language that can be applied to any economic problem with or without third-party packages.*

What Do You Mean By Programs

The word *program* is used in two ways: to describe individual *instructions*, or *source code*, created by the programmer, and to describe an entire piece of *executable software*. This distinction can cause enormous confusion, so we will try to distinguish between the source code on one hand, and the executable on the other.

NOTE: A *program* can be defined as either a set of written instructions created by a programmer or an executable piece of software.

Source code can be turned into an executable program in two ways: **Interpreters** translate the source code into computer instructions, and the computer acts on those instructions immediately. Alternatively, **compilers** translate source code into a program, which you can run at a later time. While interpreters are easier to work with, most serious programming is done with compilers because compiled code runs much faster. C is a compiled language.

Procedural, Structured, and Object-Oriented Programming

Until recently, programs were thought of as a series of procedures that acted upon data. A **procedure**, or function, is a set of specific instructions executed one after the other. The data was quite separate from the procedures, and the trick in programming was to keep track of which functions called which other functions, and what data was changed. To make sense of this potentially confusing situation, structured programming was created.

The principle idea behind structured programming is as simple as the idea of *divide* and *conquers*. A computer program can be thought of as consisting of a set of tasks. Any task that is too complex to be described simply would be broken down into a set of smaller component tasks, until the tasks were sufficiently small and self-contained enough that they were easily understood.

As an example, computing the average salary of every employee of a company is a rather complex task. You can, however, break it down into these subtasks:

1. Find out what each person earns.
2. Count how many people you have.
3. Total all the salaries.
4. Divide the total by the number of people you have.

Totaling the salaries can be broken down into

1. Get each employee's record.
2. Access the salary.
3. Add the salary to the running total.
4. Get the next employee's record.

In turn, obtaining each employee's record can be broken down into

1. Open the file of employees.
2. Go to the correct record.
3. Read the data from disk.

Structured programming remains an enormously successful approach for dealing with complex problems. By the late 1980s, however, some of the deficiency of structured programming had become all too clear. They are given below:

- First, it is *natural* to think of your data (employee records, for example) and *what* you can do with your data (sort, edit, and so on) as related ideas.
- Second, programmers found themselves constantly *reinventing* new solutions to old problems. This is often called "reinventing the wheel," and is the opposite of **reusability**. The idea behind reusability is to build components that have known properties, and then to be able to stop them into your program as you need them. This is modeled after the hardware world--when an engineer needs a new transistor, he/she doesn't usually invent one, he/she goes to the big bin of transistors and finds one that works the way he/she needs it to, or perhaps modifies it. There was no similar option for a software engineer.

HELLO.c Your First C Program

Traditional programming books begin by writing the words Hello World to the screen, or a variation on that statement. This time-honored tradition is carried on here.

Type the first program directly into your editor, exactly as shown. Once you are certain it is correct, save the file, compile it, link it, and run it. It will print the words Hello World and WelCome To C to your screen. Don't worry too much about how it works, this is really just to get you comfortable with the development cycle. Every aspect of this program will be covered over the next chapters.

NOTE: The following listing contains line numbers on the left. These numbers are for reference within the book. They should not be typed in to your editor. For example, in line 1 , you should enter:

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

HELLO.c, the Hello World program and WelCome To C.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:    printf("Hello World!\n");
6:    printf("WelCome To C\n");
7:  return 0;
8: }
```

Description about “HELLO.c” (getting each Line by Line)

Here I am explaining the first C program that you already been compiled, linked and run successfully and got proper out put. To get each line by line I am writing that HELLO.c again:

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:    printf("Hello World!\n");
6:    printf("WelCome To C\n");
7:  return 0;
8: }
```

The general way for main function declaration is given below:

Process-1:

```
  int main(void){
/* body of the main function*/
return 0;
}
```

Process-2:

```
  int main(){
/* body of the main function*/
return 0;
}
```

Process-3:

```
main(void){ /*default return type is int */
/* body of the main function
return 0;
}
```

Process-4:

```
   int main(int argc, char *argv[]){
/* body of the main function*/
return 0;
}
```

Process-5:

```
main(){/*default return type is int */
/* body of the main function*/
return 0;
}
```

Process-6:

```
main(){
/* body of the main function*/
}
```

Process-7:

```
void main(void){
/* body of the main function*/
}
```

Process-8:

```
void main(){
/* body of the main function*/
}
```

Warning: about /*comment */ please see next chapter.

NOTE: - for a good programmer style one can use Process-1,Process-2 and Process-4.

Remember: So any C program to run we must need a main() function from where a program start execution. There will be only **one** main() function present in any program.

To produce identical output. As previous program.

To print out put as previous we can write as follows:

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
  printf("\nHello");
  printf"World!");
  printf("\nWelCome");
  printf("To C");
return 0;
}
```

The output will same as previous.

Notice that **\n** represents only a single character. An escape sequence, which provides a general and extensible mechanism for representing hard-to-type or invisible characters, it brings the cursor to the new or next line. Among the others that C provides are \t for tab, \b for backspace, \" for the double quote and \\ for the backslash itself. This thing we will discuss later. If I want out put like output like

```
Hello
World!
WleCome
To C
```

So I will write like this way.

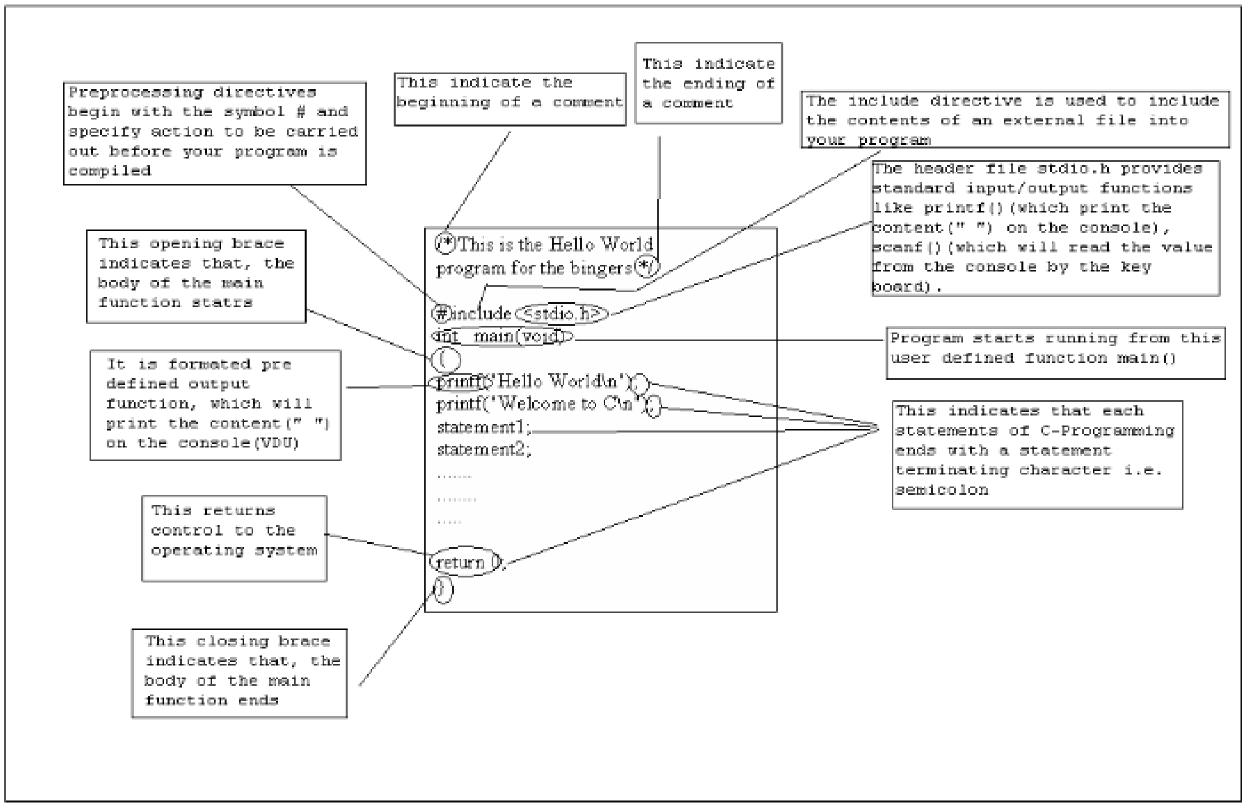
```
#include <stdio.h>

int main(void)
{
  printf("Hello\nWorld!\n");
  printf("WelCome \nTo C\n");
return 0;
}
```

Here the job of ‘\n’, which bring the cursor (-) to the new line or next line. So the out put will be

```
Hello
World!
WelCome
To C
```

Generally printing the character on the screen at the position of cursor, after printing the char the cursor will move right one position, when cursor got ‘\n’ suddenly the cursor will move to the next line. Next character will print on next line at the cursor position.


```
/*This is the Hello World
program for the bingers*/
#include <stdio.h>
int  main(void)
{
printf("Hello World\n");
printf("Welcome to C\n");
statement1;
statement2;
.......
........
.....
return 0;
}
```
Preprocessing directives begin with the symbol # and specify action to be carried out before your program is compiled
This indicate the beginning of a comment
This indicate the ending of a comment
The include directive is used to include the contents of an external file into your program
The header file stdio.h provides standard input/output functions like printf()(which print the content(" ") on the console), scanf()(which will read the value from the console by the key board).
This opening brace indicates that, the body of the main function statrs
Program starts running from this user defined function main()
It is formated pre defined output function, which will print the content(" ") on the console(VDU)
This indicates that each statements of C-Programming ends with a statement terminating character i.e. semicolon
This returns control to the operating system
This closing brace indicates that, the body of the main function ends

Questions

Q. Can I ignore warning messages from my compiler?

A. Many books hedge on this one, but I'll stake myself to this position: **No!** Get into the habit, from day one, of treating warning messages as errors. C++ uses the compiler to warn you when you are doing something you may not intend. Observe those warnings, and do what is required to make them go away.

Q. What is compile time?

A. Compile time is the time when you run your compiler, as opposed to link time (when you run the linker) or run-time (when running the program). This is just programmer shorthand to identify the three times when errors usually surface.

Quiz

1. What is the difference between an interpreter and a compiler?

2. How do you compile the source code with your compiler?

3. What does the linker do?

4. What are the steps in the normal development cycle?

5. Write a program that will output your name, RegNo, and address using a separate printf() statement for each line of output.

Chapter -2
Key Words, Identifiers and Variables

Key words

The words which are pre defined for specific task in the compiler they are called reserved word or key words. In “C” there are 32 key words are present they are given below. All you have to remember.

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

Identifiers:
Identifiers are the name of variables, name of function etc.
int a=5; (a is the identifier).
void add(){} (add is the identifier)
Basic Data Types
A data type may be defined as a set and the elements of the set are called the values of the type.

C provides a standard, nominal set of basic data types. These are “Built-in” data type sometimes also called "primitive" data types. More complex data structures can be built up from these basic data types.
Primitive data types with range

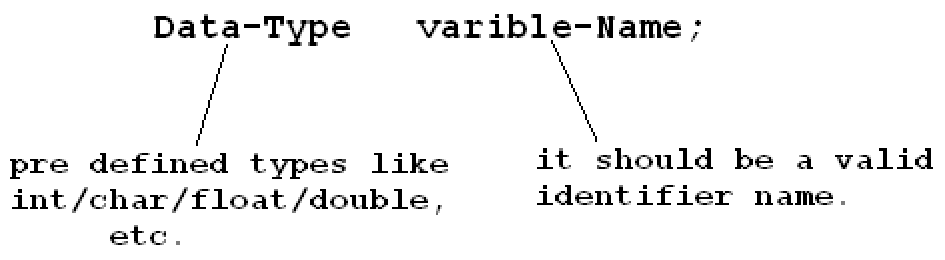
Data Type Category	***Range***	***Length***	***Data Type***
unsigned char	0 to 255	8 bits	**char**
signed char	-128 to 127	8 bits	
enum	-32,768 to 32,767	16 bits	
unsigned int	0 to 65,535	16 bits	
short int	-32,768 to 32,767	16 bits	
int	-32,768 to 32,767	16/32 bits	**int**
unsigned long	0 to 4,294,967,295	32 bits	
long	-2,147,483,648 to 2,147,483,647	32 bits	
float	3.4 * (10^{-38}) to 3.4 * (10^{+38})	32 bits	
double	1.7 * (10^{-308}) to 1.7 * (10^{+308})	64 bits	**float**
long double	3.4 * (10^{-4932}) to 1.1 * (10^{+4932})	80 bits	
Void	Null	0 bits	**void**

Character is specified by ASCII integer value. To see the ASCII value refer the Appendix

Declaration of variables

As in most languages, a variable declaration reserves and names an area in memory at run time to hold a value of particular type. Syntactically, C puts the type first followed by the name of the variable.

Syntax for variable declaration


Data-Type varible-Name;
pre defined types like int/char/float/double, etc.
it should be a valid identifier name.

Datatype: refer to what type of my data is generally, as discussed earlier different types of datatype, char, int , float, double, struct, union;

Name: it refer to the name of the variable same as the in identifier name

Example:

int a;	// a is a integer type data, which can store only integer value
int roll;	// roll is a integer type data which can store integer value.
char c;	// c is a character type data which can store character value.
float f;	//f is floating point data type, which can store floating value.

Constants

Like variables, constants are data storage locations. Unlike variables, and as the name implies, constants don't change. You must initialize a constant when you create it, and you cannot assign a new value later.

char Constants

A char constant is written with single quotes (') like 'a' or 'C'. The char constant ‘a’ is really just a synonym for the ordinary integer value 97, which is the ASCII value for lowercase 'a'. There are special case char constants, such as '\t' for tab, for characters, which are not convenient to type on a keyboard. These special characters are called escape sequences. They are given below:

\'	single quote
\"	double quote
\\	backslash character
\b	backspace

\f	next page/form feed
\n	start a new line
\nnn	treat nnn as an octal number
\000	treat 000 as an octal number
\r	carriage return
\t	move to next tab setting
\0	null character marking the end of a string
\0x	treat 0x as a hexadecimal number

Numbers in the source code such as 234 default to type int. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a long such as 42L. An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal -- 0x10 is way of expressing the number 16. Similarly, a constant may be written in octal by preceding it with "0" 012 is a way of expressing the number 10.

Characters may also be specified using their numeric code value. The general escape sequence \ooo (i.e., a backslash followed by up to three octal digits) is used for this purpose. For example (assuming ASCII):

```
'\12'        // newline (decimal code = 10)
'\11'        // horizontal tab (decimal code = 9)
'\101'        // 'A' (decimal code = 65)
'\0'        // null (decimal code = 0)
```

Literal Constants

Apart from the character constant C has two other types of constants: **literal** and **symbolic**.
A **literal** constant is a value typed directly into your program wherever it is needed. For example
int myAge = 25;
myAge is a variable of type int; 25 is a literal constant. You can't assign a value to 25, and its value can't be changed.

Symbolic Constants

A **symbolic** constant is a constant that is represented by a name, just as a variable is represented. Unlike a variable, however, after a constant is initialized, its value can't be changed.
If your program has one integer variable named students and other named classes, you could compute how many students you have, given a known number of classes, if you knew there were 15 students per class:
students = classes * 15;

Chapter -3

Expressions and Statements

Introduction

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C integrates operators. Unlike other languages whose operators are mainly keywords, operators in C are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

You do not have to memorize all the content of this chapter. Most details are only provided to serve as a later reference in case you need it.
At its heart, a program is a set of commands executed in sequence. The power in a program comes from its capability to execute one or another set of commands, based on whether a particular condition is true or false. In this lesson you will learn

- What statements are.
- What blocks are.
- What expressions are.
- How to branch your code based on conditions.
- What truth is, and how to act on it.

Statements

In C a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement). All C statements end with a semicolon, even the null statement, which is just the semicolon and nothing else. One of the most common statements is the following assignment statement:

```
x = a + b;
```

Unlike in algebra, this statement does not mean that x equals a+b. This is read, "Assign the value of the sum of a and b to x," or "Assign to x, a+b." Even though this statement is doing two things, it is one statement and thus has one semicolon. The assignment operator assigns whatever is on the right side of the equal sign to whatever is on the left side.
New Term: A *null statement* is a statement that does nothing.

Expressions

Anything that *evaluates* to a value is an expression in C. An expression is said to **return** a value. Thus, 3+2; returns the value 5 and so is an expression. All expressions are statements.
The many pieces of code that qualify as expressions might surprise you. Here are three examples:

```
3.2                 // returns the value 3.2
PI                  // float const that returns the value 3.14
secondsPerMinute        // int const that returns 60
```

Assuming that PI is a constant equal to 3.14 and secondsPerMinute is a constant equal to 60, all

three of these statements are expressions.
The complicated expression
x = a + b;
not only adds a and b and assigns the result to x, but returns the value of that assignment (the value of x) as well. Thus, this statement is also an expression. Because it is an expression, it can be on the right side of an assignment operator:
y = x = a + b;
This line is evaluated in the following order: Add a to b.
Assign the result of the expression a + b to x.

Assign the result of the assignment expression x = a + b to y.

If a, b, x, and y are all integers, and if a has the value 2 and b has the value 5, both x and y will be assigned the value 7.

/*Expression1.c Evaluating complex expressions. */

```
1:   #include <stdio.h>
2:   int main(void)
3:     {
4:      int a=0, b=0, x=0, y=35;
5:      printf("a:%d\tb:%d\n",a,b);
6:      printf("x:%d\ty:%d\n",x,y);
7:       a = 9;
8:       b = 7;
9:      y = x = a+b;
10:       printf("a:%d\tb:%d\n",a,b);
11:       printf("x:%d\ty:%d\n",x,y);
12:   return 0;
13: }
```

Output:

```
a: 0 b: 0
x: 0 y: 35
a: 9 b: 7
x: 16 y: 16
```

Analysis: On line 4, the four variables are declared and initialized. Their values are printed on lines 5 and 6. On line 7, a is assigned the value 9. One line 8, b is assigned the value 7. On line 9, the values of a and b are summed and the result is assigned to x. This expression (x = a+b) evaluates to a value (the sum of a + b), and that value is in turn assigned to y.

Operators
An operator is a character or group of characters (generally called symbol) that causes the compiler to take an action. Operators act on operands, and in C all operands are expressions. In C there are several different categories of operators.

5.4.1. Assignment Operator
The assignment operator (=) causes the operand on the left side of the assignment operator to have its value changed to the value on the right side of the assignment operator. The expression

```
int   a;
a  =  30;
```

This statement assigns the integer value 30 to the variable a. The part at the left of the assignation operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The *lvalue* has to be a variable whereas the *rvalue* can be a constant, a variable, and the result of an operation or any combination of these.

The most important rule of assignation is the *right-to-left* rule: The assignation operation always takes place from right to left, and never the other way:

```
b=a;
```

This statement assigns the value present in “a” (the rvalue) to the varible “b” (the lvalue). The value that was stored until this moment in “a” is not considered at all in this operation.

Consider also that we are only assigning the value of “a” to “b” at the moment of the assignation. Therefore “b” later change of “a” will not affect the new value of “b”.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

Assignmentoperator.c Example

```
#include<stdio.h>


int main (void)
{
  int a, b;         // a:?,  b:?
 a = 10;           // a:10, b:?
 b = 4;            // a:10, b:4
 a = b;            // a:4,  b:4
 b = 7;            // a:4,  b:7
 printf(“a:%d",a);
 printf(" b:%d",b);
 return 0;
}
```

Out put:

```
a:4
b:7
```

Arithmetic Operators

There are five mathematical operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulation

Addition and subtraction work as you would expect, although subtraction with unsigned integers can lead to surprising results, if the result is a negative number. when variable overflow was described. i**nteger_overflow.c** shows what happens when you subtract a large unsigned number from a small unsigned number.

integer_overflow.c A demonstration of subtraction and integer overflow.

```
1: #include <stdio.h>
2:
3:
4:
5: int main()
6: {
7:    unsigned int difference;
8:    unsigned int bigNumber = 100;
9:    unsigned int smallNumber = 50;
10:    difference = bigNumber - smallNumber;
11:   printf("Difference is:%u\n",difference);
12:    difference = smallNumber - bigNumber;
13:    printf("\nNow difference is:%u\n",difference);
14:    return 0;
15: }
```

Output: Difference is: 50
Now difference is: 4294967246

Analysis: The subtraction operator is invoked on line 10, and the result is printed on line 11, much as we might expect. The subtraction operator is called again on line 12, but this time a large unsigned number is subtracted from a small unsigned number. The result would be negative, but because it is evaluated (and printed) as an unsigned number, the result is an overflow.

Integer Division and Modulus

Integer division is somewhat different from everyday division. When you divide 21 by 4, the result is a real number (a number with a fraction). Integers don't have fractions, and so the "remainder" is **lopped off**. The answer is therefore 5. To get the remainder, you take 21 modulus 4 (21 % 4) and the result is 1. The modulus operator tells you the remainder after an integer division.

Finding the modulus can be very useful. For example, you might want to print a statement on

every 10th action. Any number whose value is 0 when you modulus 10 with that number is an exact multiple of 10. Thus 1 % 10 is 1, 2 % 10 is 2, and so forth, until 10 % 10, whose result is 0. 11 % 10 is back to 1, and this pattern continues until the next multiple of 10, which is 20. We'll use this technique when looping is discussed on latter lessons.
So integer division always results in an integer outcome (i.e., the result is always rounded down). For example
For example:

```
9 / 2        // gives 4, not 4.5!
-9 / 2       // gives -4!
```

Unintended integer divisions are a common source of programming errors. To obtain a real division when both operands are integers, you should cast one of the operands to be real:

```
int          cost = 100;
int          volume = 80;
double    unitPrice = cost / (double) volume;          //
gives 1.25
```

The remainder operator (%) expects integers for both of its operands. It returns the remainder of integer-dividing the operands. For example 13%3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.
It is possible for the outcome of an arithmetic operation to be too large for storing in a designated variable. This situation is called an **overflow**. The outcome of an overflow is machine-dependent and therefore undefined. For example:

```
unsigned char      k = 10 * 92;          // overflow: 920 > 255
```

It is *illegal* to divide a number by zero. This results in a run-time *division-by-zero* failure which typically causes the program to terminate
Many beginner C programmers inadvertently put a semicolon after their if statements:

```
if(someValue < 10);
someValue = 10;
```

What was intended here was to test whether someValue is less than 10, and if so, to set it to 10, making 10 the minimum value for someValue. Running this code snippet will show that someValue is always set to 10! Why? The if statement terminates with the semicolon (the do-nothing operator). Remember that indentation has no meaning to the compiler. This snippet could more accurately have been written as:

```
if (someValue < 10)  // test
;  // do nothing
someValue = 10;  // assign
```

Removing the semicolon will make the final line part of the if statement and will make this code do what was intended.

Combining the Assignment and Mathematical Operators (short hand Operator)

Called compound assignation or short hand operator (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
It is not uncommon to want to add a value to a variable, and then to assign the result back into the variable. If you have a variable myAge and you want to increase the value by two, you can write

```
int myAge = 5;
int temp;
temp = myAge + 2;  // add 5 + 2 and put it in temp
myAge = temp;            // put it back in myAge
```

This method, however, is terribly convoluted and wasteful. In C, you can put the same variable on both sides of the assignment operator, and thus the preceding becomes

```
myAge = myAge + 2;
```

which is much better. In algebra this expression would be meaningless, but in C it is read as "add two to the value in myAge and assign the result to myAge."
Even simpler to write, but perhaps a bit harder to read is

```
myAge += 2;
```

The self-assigned addition operator (+=) adds the rvalue to the lvalue and then reassigns the result into the lvalue. This operator is pronounced "plus-equals." The statement would be read "myAge plus-equals two." If myAge had the value 4 to start, it would have 6 after this statement.
Here all short hand operators are given:
Table 5.1: short hand operator use

Equivalent To	Example	Operator
	n = 25	=
n = n + 25	n += 25	+=
n = n - 25	n -= 25	-=
n = n * 25	n *= 25	*=
n = n / 25	n /= 25	/=
n = n % 25	n %= 25	%=
n = n & 0xF2F2	n &= 0xF2F2	&=
n = n | 0xF2F2	n |= 0xF2F2	|=
n = n ^ 0xF2F2	n ^= 0xF2F2	^=
n = n << 4	n <<= 4	<<=
n = n >> 4	n >>= 4	>>=

Unary operator(++ and --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
a ++;
a += 1;
a = a + 1;
```

Here all three statements are equal. The outputs of all statements are equal.

In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, the compiler generally does this type of code optimization automatically, thus the three expressions should produce exactly the same executable code.

A characteristic of this operator is that it can be used both as a **prefix** and as a **postfix**. That means that it can be written either **before** the variable identifier (++a) or **after** it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a postfix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1

```
B=3;
A=++B;     // A contains 4, B contains 4
```

Example 2

```
B=3;
A=B++; // A contains 3, B contains 4
```

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

inc_dec.c A demonstration over ++ and -- operator

```
#include <stdio.h>



int main(void)
{
int a=10;
printf(“%d\n”,a++); //output=10 and “a” increase to  11
printf(“%d\n”,a++); //output=11 and “a” increase to  12
printf(“%d\n”,++a); // “a” increase to  13 and output=13
printf(“%d\n”,a--); //output=13 and “a” decrease to  12
printf(“%d\n”,--a); // “a” decrease to  11 and output=11
printf(“%d\n”,a++); //output=11 and “a” increase to  12
printf(“%d\n”,a--);   //output=12 and “a” value is  11
printf(“%d\n”,--a); // “a” decrease to  10 and output=10
printf(“%d\n”,++a); // “a” increase to  11 and output=11
printf(“%d\n”,a++); //output=11 and “a” increase to  12
printf(“%d\n”,a--);//output=12 and “a” decrease to  11
printf(“%d\n”,a++); //output=11 and “a” increase to  12
printf(“%d\n”,++a + ++a); // “a” increase to  14  and output=28
printf(“%d\n”,++a + a++); //first “a” increase to 15 and output=30 then a=16
printf(“%d\n”,a++ + a++); // output=32 and ‘a’ inc to 18
printf(“%d\n”,a++ + ++a);//at first “a” inc to 19 ,out put=38 and ‘a’ inc to 20
return 0;
}
```

The Pre/Post variation has to do with nesting a variable with the increment or decrement operator inside an expression -- should the entire expression represent the value of the variable before or after the change? I never use the operators in this way (see below), but an example looks like...

```
int i = 42;
int j;
j = (i++ + 10);
// i is now 43
// j is now 52 (NOT 53)
j = (++i + 10)
// i is now 44
// j is now 54
```

Relational Operators

The relational operators are used to determine whether two numbers are *equal*, or if one is *greater* or *less* than the other. Every relational statement evaluates to either 1 (TRUE) or 0 (FALSE). The relational operators are presented later, in **Table**.
If the integer variable myAge has the value 39, and the integer variable yourAge has the value 40, you can determine whether they are equal by using the relational "equals" operator:
myAge == yourAge; // is the value in myAge the same as in yourAge?
This expression evaluates to 0, or false, because the variables are not equal. The expression
myAge > yourAge; // is myAge greater than yourAge?
evaluates to 0 or false.
There are six relational operators: equals (==), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), and not equals (!=). **Table** shows each relational operator, its use, and a sample code use.

Table The Relational Operators.

Evaluates	*Sample*	*Operator*	*Name*
false	100 == 50;	==	Equals
true	50 == 50;		
true	100 != 50;	!=	Not Equals
false	50 != 50;		
true	100 > 50;	>	Greater Than
false	50 > 50;		
true	100 >= 50;	>=	Greater Than
true	50 >= 50;		or Equals
false	100 < 50;	<	Less Than
false	50 < 50;		
false	100 <= 50;	<=	Less Than
true	50 <= 50;		or Equals

Difference between assignment (=) and logical equal (==) operators

An absolutely classic pitfall is to write assignment (=) when you mean comparison (==). This would not be such a problem, except the incorrect assignment version compiles fine because the compiler assumes you mean to use the value returned by the assignment. This is rarely what you want

```
if (x = 3) ...
```

This does5.19 not test if x is 3. This sets x to the value 3, and then returns the 3 to the if for testing. 3 is not 0, so it counts as "true" every time. This is probably the single most common error made by beginning C programmers. The problem is that the compiler is no help -- it thinks both forms are fine, so the only defense is extreme attention when coding. Or write "= ==" in big letters on the back of your hand before coding. This mistake is an absolute classic and it's a bear to debug. Watch Out! And need I say: "Professional Programmer's Language."

Logical operators (!, &&, ||)

The Operator ! is the C operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand.

Example:

```
!(5 == 5)    // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)    // evaluates to true because (6 <= 4) would be false.
!true        // evaluates to false
!false       // evaluates to true.
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves The following table shows the result of operator && evaluating the expression a && b and operator with results of a || b

a || b	a && b	b	a
T	T	T	T
T	F	F	T
T	F	T	F
F	F	F	F

For example:

((5 == 5) && (3 > 6)) // evaluates to false (true && false).
((5 == 5) || (3 > 6)) // evaluates to true (true || false).

Ternary operator (? :)

This is called ternary or conditional operator, which evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

condition ? expression1 : expressiion2;

If condition is true the expression1 will return, if it is not it will return expression2.

7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.

Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:
a=(2,3,4,-5);
comma operator works left to right so right must value i.e. (-5) will assign to “a”.

a = (b=3, b+2);

First assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify the variables by the bit patterns that represent the values what they store.

Description	**Equivalent**	**Operator**
Bitwise AND	AND	&
Bitwise Inclusive OR	OR	|
Bitwise Exclusive OR	XOR	^
Unary complement (bit inversion)	NOT	~
Shift Left	SHL	<<
Shift Right	SHR	>>

Example

```
unsigned char x = '\011';
unsigned char y = '\027';
```

Table:

How the bits are calculated.

							Bit Sequence	**Octal Value**	**Example**
1	0	0	1	0	0	0	0	011	x
1	1	1	0	1	0	0	0	027	y
0	1	1	0	1	1	1	1	366	~x
1	0	0	0	0	0	0	0	001	x & y
1	1	1	1	1	0	0	0	037	x | y
0	1	1	1	1	0	0	0	036	x ^ y
0	0	1	0	0	1	0	0	044	x << 2
0	1	0	0	0	0	0	0	002	x >> 2

sizeof() operator

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

a = sizeof (char);

This will assign the value 1 to “a“ because char is a one-byte size.

//sizeof.c. Demonstrate the sozeof() operator

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

```
printf("\n the size of char data type is=%d bytes",sizeof (char));
printf("\n the size of short int data type is=%d bytes",sizeof (short int));
printf("\n the size of int data type is=%d bytes",sizeof (int));
printf("\n the size of long data type is=%d bytes",sizeof (long));
printf("\n the size of float data type is=%d bytes",sizeof (float));
printf("\n the size of double data type is=%d bytes",sizeof(double));
printf("\n the size of long double data type is=%d bytes",sizeof(long double));
printf("\n the size of long long data type is=%d bytes",sizeof(long long));

printf("\n the size of char* pointer is=%d bytes",sizeof (char*));
printf("\n the size of int* pointer is=%d bytes",sizeof (int*));

printf("\n the size of 'A' is=%d bytes",sizeof ('A'));
printf("\n the size of 23.45 is=%d bytes",sizeof (23.45));
printf("\n the size of 23.45f is=%d bytes",sizeof (23.45f));
printf("\n the size of 23.45L is=%d bytes",sizeof (23.45L));
printf("\n the size of \"Hello\" is=%d bytes",sizeof ("Hello"));

return 0;
}
```

When run, the program will produce the following output (on 16-bit compiler):

```
the size of char data type is=1 bytes
the size of short int data type is=2 bytes
the size of int data type is=2 bytes
the size of long data type is=4 bytes
the size of float data type is=4 bytes
the size of double data type is=8 bytes
the size of long double data type is=10 bytes
the size of long long data type is=4 bytes

the size of char* pointer is=2 bytes
the size of int* pointer is=2 bytes

the size of 'A' is=2 bytes
the size of 23.45 is=8 bytes
the size of 23.45f is=4 bytes
the size of 23.45L is=10 bytes
the size of "Hello" is=6 bytes
```

When run, the program will produce the following output (on 32-bit compiler):

```
the size of char data type is=1 bytes
the size of short int data type is=2 bytes
the size of int data type is=4 bytes
the size of long data type is=4 bytes
the size of float data type is=4 bytes
the size of double data type is=8 bytes
the size of long double data type is=12 bytes
the size of long long data type is=8 bytes

the size of char* pointer is=4 bytes
the size of int* pointer is=4 bytes

the size of 'A' is=1 bytes
the size of 23.45 is=8 bytes
the size of 23.45f is=4 bytes
the size of 23.45L is=12 bytes
the size of "Hello" is=6 bytes
```

NOTE: The size will vary from compiler to compiler and from PC to PC, my PC is 32-bits processor so the size of character pointer will be 4bytes(any type of pointer the size is fixed). But for 16-bit processor the size will be 2 byte. Some difference are highlighted please remember them,

- ***NOTE:*** *The value returned by sizeof is a constant, so it is always determined before program execution.*

Explicit type casting operator

Type casting operators allow you to convert a data type to another data type. There are several ways to do this in C. The simplest one, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int)

More examples:

```
(int) 3.14               // converts 3.14 to an int to give 3
(long) 3.14              // converts 3.14 to a long to give 3L
(double) 2               // converts 2 to a double to give 2.0
(char) 122               // converts 122 to a char whose code is 122
(unsigned short) 3.14       // gives 3 as an unsigned short
```

As shown by these examples, the built-in type identifiers can be used as **type operators**. Type operators are unary (i.e., take one operand) and appear inside brackets to the left of their operand.

This is called **explicit type conversion**. When the type name is just one word, an alternate notation may be used in which the brackets appear around the operand:

```
int(3.14)            // same as: (int) 3.14
```

In some cases, C also performs **implicit type conversion**. This happens when values of different types are mixed in an expression. For example:

```
double      d = 1;              // d receives 1.0
int         i = 10.5;           // i receives 10
i = i + d;                      // means: i = int(double(i) + d)
```

In the last example, i + d involves mismatching types, so i is first converted to double (*promoted*) and then added to d. The result is a double which does not match the type of i on the left side of the assignment, so it is converted to int (*demoted*) before being assigned to i.

The above rules represent some simple but common cases for type conversion. More complex cases will be examined later in the book after we have discussed other data types and classes.

5.5 Operator Precedence Table

Grouping	**Descriptions**	**Operators**	**Level**
Left-to-right	Scope resolution	::	1
Left-to-right	Postfix	() [] . -> ++ --	2
Right-to-left	Unary Prefix	++ -- ~ ! sizeof	3
	Indirection and reference (pointers)	* &	
	Unary sign operator	+ -	
Right-to-left	type casting	(type)	4
Left-to-right	pointer-to-member	.* ->*	5
Left-to-right	Multiplicative	* / %	6
Left-to-right	Additive	+ -	7
Left-to-right	Shift	<< >>	8
Left-to-right	Relational	< > <= >=	9
Left-to-right	Equality	== !=	10
Left-to-right	bitwise AND	&	11
Left-to-right	bitwise XOR	^	12
Left-to-right	bitwise OR	|	13
Left-to-right	logical AND	&&	14
Left-to-right	logical OR	||	15
Right-to-left	Conditional	?:	16
Right-to-left	Assignment	= *= /= %= += -= >>= <<= &= ^= !=	17
Left-to-right	Comma	,	18

Q&A

Q. Why use unnecessary parentheses when precedence will determine which operators are acted on first?
A. Although it is true that the compiler will know the precedence and that a programmer can look up the precedence order, code that is easy to understand is easier to maintain.

Q. If the relational operators always return 1 or 0, why are other values considered true?

A. The relational operators return 1 or 0, but every expression returns a value, and those values can also be evaluated in an if statement. Here's an example:

```
if ( (x = a + b) == 35 )
```

This is a perfectly legal C statement. It evaluates to a value even if the sum of a and b is not equal to 35. Also note that x is assigned the value that is the sum of a and b in any case.

Q. What effect do tabs, spaces, and new lines have on the program?

A. Tabs, spaces, and new lines (known as whitespace) have no effect on the program, although judicious use of whitespace can make the program easier to read.

Q. Are negative numbers true or false?

A. All nonzero numbers, positive and negative, are true.

Quiz

1. What is an expression?

2. Is x = 5 + 7 an expression? What is its value?

3. What is the value of 201 / 4?

4. What is the value of 201 % 4?

5. If myAge, a, and b are all int variables, what are their values after:

```
myAge = 39;
a = myAge++;
b = ++myAge;
```

6. What is the value of 8+2*3?

7. What is the difference between x = 3 and x == 3?

Home work Assignment

Q1. Write a program to convert a length into yard, feet and inches (file name: vi length2YFI.c) (Hints:- 1yard=3 feets and 1 feet=12 inches)

Q2. Write a program to convert a temperature in Fahrenheit to Celsius and Celsius to Fahrenheit. (Hints: - C/5= (F-32)/9, Declare C and F as float data type) (File name: vi Far2Ces.c)

Q3. Write a program to Add two fractions. (The result should be in fraction format i.e. a/b) (Hints: The numerator and denominator of the two fractions are given) (File name: vi Add2Fractions.c)

Q4. Write a program to calculate the value of the given polynomial: $Y= ax^3 + bx^2 + cx + d$. (Hints: the value of a, b, c, d and x are given)(file name: vi Poly.c)

Q5. Write a program to calculate the value of distance by using the given formula $S=ut+1/2at^2$ (file name: vi distance.c)

Q6. Write a program to calculate the roots of a quadratic equation.(Hints use the formula to find the roots and use sqrt() function to find the square root of a number and include math.h header file at the binging of the program.)(file name: vi roots.c)

Q7. Write a program to calculate the sum and average of five numbers. (Hints: declare five variables as num1, num2,num3,num4,num5 and sum as integer type. Declare avg as float type. Find the sum of all numbers and store them in sum, then find the average using formula.) (file name: vi sumaverage.c)

Q8. Write a program to swap two numbers using third variable(file name **vi swap.c**)

Q9. Write a program to swap two numbers without using third variable.
(file name vi **swap_without_third_varible.c**)

Q10. Write a program to find the Simple interest using the formula SI=PTR/100.

Chapter -4
Selection Statements with control logic

Selection statements are also called control logic statements. The logic means the output will be true or false there are several control logic statements given below.

if (expression) statement
if (expression) statement else statement
if (expression) statement else if (expression) statement else statement
nested if(expression)
switch (expression) case : statement

The if Statement

Normally, your program flows along line by line in the order in which it appears in your source code. The if statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.
The simplest form of an if statement is this:

```
if (expression)
    statement;
```

The expression in the parentheses can be any expression at all, but it usually contains one of the relational expressions. If the expression has the value 0(**zero**), it is considered false, and the statement is skipped. If it has any **nonzero** value, it is considered true, and the statement is executed. Consider the following example:

```
if (bigNumber > smallNumber)
    bigNumber = smallNumber;
```

This code compares bigNumber and smallNumber. If bigNumber is larger, the second line sets its value to the value of smallNumber.
Because a block of statements surrounded by braces is exactly equivalent to a single statement, the following type of branch can be quite large and powerful:

```
if (expression)
{
    statement1;
    statement2;
    statement3;
}
```

Relational_operators.c A demonstration of branching based on relational operators.

```
1:  // A demonstrates if statement with relational operators
2:  #include <stdio.h>
3:
4:  int main()
5:  {
6:        int tomScore, bobScore;
7:        printf("Enter the score for the Tom: ");
8:        scanf(“%d”,&tomScore);
9:
10:       printf("\nEnter the score for the Bob: ");
```

```
      scanf(“%d”,&bobScore);

      printf(“\n”);

      if (tomScore > bobScore)
          printf("Go Tom!\n");

      if (tomScore < bobScore)
      {
          printf("Go Bob!\n");
         printf("Happy days in I-Generic Lab!\n");
      }

      if (tomScore == bobScore)

          printf("Match is draw due to equal score.\n");

             return 0;
 }
```

Output: Enter the score for the tom: 10

```
Enter the score for the bob: 10
Match is draw due to equal score.
```

Program that demonstrates the use of the keyword else.

```
demonstrates if statement with else clause
#include <stdio.h>

 int main(void)
 {
    int firstNumber, secondNumber;
    printf("\nPlease enter a number: ");
    scanf(“%d”,&firstNumber);
    printf("\nPlease enter a second number: ");
     scanf(“%d”,&secondNumber);
     if (firstNumber > secondNumber)
          printf(\n%d is bigger!\n",firstNumber);
     else
          printf(\n%d is bigger!\n",secondNumber);

        return 0;
}
```

Output: Please enter a number: 10

```
Please enter a second number: 12

12 is bigger!
```

Nestedif.c. A complex, nested if statement.

```
// Nestedif.c - a complex nested if statement
#include <stdio.h>

int main(void)
{
    // Ask for two numbers
    // Assign the numbers to firstNumber and secondNumber
    // If firstNumber is bigger than secondNumbe,
    // see if they are evenly divisible
    // If they are, see if they are the same number

    int firstNumber, secondNumber;
    printf("Enter First number: ");
    scanf(“%d”,&firstNumber);
    printf("Enter the Second number: ");
    scanf(“%d”,&secondNumber);


    if (firstNumber >= secondNumber)
    {
      if ( (firstNumber % secondNumber) == 0) // evenly divisible?
      {
           if (firstNumber == secondNumber)
                printf("They are the same!\n");
           else
                printf(“They are divisible!\n");
      }
      else
           printf(“They are not divisible!");
    }
    else
      printf("Hey! The second one is larger!");
       return 0;
}
```

Output:

```
Enter First number: 10
Enter second number: 5
They are divisible!
```

Selection-statement (switch case)

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concate of several if and else if instructions

In this statement there is a selection case. Means at first it is getting a case value. Depending the case value it will do the work.

The syntax of the selection statement is given below.

```
switch(value){
            case  constant-value1:
                     job1;
                     break;
            case constant-value2:
                     job2;
                     break;
            case constant-value3:
                     job3;
                     break;
                      ……………………….
                      ……………………….
            default:
                   Error message;
            }
```

It works in the following way: switch evaluates value and checks if it is equivalent to constant-value1, if it is, it executes job1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant-value1 it will be checked against constant-value2. If it is equal to this, it will execute job2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of value did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).
Both of the following code fragments have the same behavior:

if-else equivalent

```
if (x == 1) {
   printf(”I am in case 1");
  }
else if (x == 2) {
  printf(”I am in case 2");

  }
else {
 printf(”value of x I don’t known");
  }
```

switch example

```
switch (x) {
  case 1:
   printf(”I am in case 1");
    break;
  case 2:
    printf(”I am in case 2");
    break;
  default:
    printf(”value of x I don’t known");
  }
```

The switch statement is a bit peculiar within the C language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated.

Example:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    printf(”x is 1, 2 or 3");
    break;
  default:
    printf(“x is not 1, 2 nor 3");
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we **cannot** put *variables* as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C constants.

If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.

Example: to print the day by entering the day number:

```
#include<stdio.h>
int main(void)
{
    int day;
    printf("\nEnter the day number(1-7):");
    scanf("%d",&day);
    switch(day)
     {
       case 1: printf("Sunday\n");
               break;
       case 2: printf("Monday\n");
               break;
       case 3: printf("Tuesday\n");
               break;
       case 4: printf("Wednesday\n");
```

```
            break;
        case 5: printf("Thursday\n");
            break;
        case 6: printf("Friday\n");
            break;
        case 7: printf("Saturday\n");
            break;
        default:
             printf("Wrong choice!!\n");
             }
return 0;
}
```

Input: Enter the day number(1-7):3
OutPut: Tuesday
Input: Enter the day number(1-7):-12
OutPut: Wrong choice!!

The default statement can be writing at **any** place of the switch case, if we are writing default statement at any place **we need a break** after default statement. The case order can be written any order,

```
#include<stdio.h>
int main(void){
    int day;
    printf("\nEnter the day number(1-7):");
    scanf("%d",&day);
    switch(day)    {
         case 5: printf("Thuesday\n");
            break;
        case 2: printf("Monday\n");
            break;
        case 1: printf("Sunday\n");
            break;
        case 7: printf("Saterday\n");
            break;
        case 3: printf("Tuesday\n");
            break;
        default:printf("Wrong choice!!\n");
            break;
        case 4: printf("Wendesday\n");
            break;
                /*case 2:  printf("Monday\n");break;      *///error
       case 6: printf("Friday\n");
            break;
     /*default: printf("Wrong choice!!");*///error
             }
return 0;
}
```

NOTE: Duplicate case values and default statements can not be present.

Exercise

1. Write a program to read a number from the user then find the absolute value of it and then print it
2. Write a program to calculate the roots of a quadratic equation.(Hints use the formula to find the roots and use sqrt() function to find the square root of a number and include math.h header file at the binging of the program.)
3. Write a program to read two numbers from the user and print the biggest among them.
4. Write a program to read two numbers from the user and print the smallest among them
5. Write a program to read two characters from the user and print the biggest among them
6. Write a program to read three numbers from the user and print the biggest among them.
7. Write a program to read three numbers from the user and print the smallest among them
8. Write a program to read four numbers from the user and print the smallest among them
9. Write a program to read a date from the user and test weather it a valid date or not

1. Write a program to read a character from the user and check whether it is a upper case alphabet or lower case alphabet (if it is alphabet check whether it is vowel or not) or digit or non-printable character or special character and print the ASCII value of that char
2. Write a program to read a number from the user and print that number is even or odd.
3. Write a program to read a point from the user and print that point belongs to 1st or 2nd or 3rd or 4th quadrant
4. Write a program to read three points of a triangle and test for collinearity.
5. Write a program to check the two lines L1 (formed by points p1 & p2) and L2 (formed by points p3 & p4) are parallel or perpendicular or collinear.
6. Write a program to read a point from the user and print that point whether it present above the x-axis or below the x-axis.
7. Write a program to read the three lengths of a triangle and test whether a triangle can construct or not.
8. Write a program to read a point from the user and print that point whether it presents left side of the y-axis or right side of the y-axis.
9. Write a program to read a year from the user and test it is a leap year or not.
10. Write a program to read the three lengths of a triangle and test what type of that triangle is it(equilateral or right hand or isosceles or scalene)
11. Write a program to read a mark x between $0 \leq x \leq 100$.

Print ‘O’ if $x \geq 90$
Print ‘E’ if $x \geq 80$ and $x < 90$
Print ‘A’ if $x \geq 70$ and $x < 80$
Print ‘B’ if $x \geq 60$ and $x < 70$
Print ‘C’ if $x \geq 50$ and $x < 60$
Print ‘D’ if $x \geq 35$ and $x < 50$

Print ‘F’ if x< 35

Let the Income Tax slab is given in the table (2007-08)

Rates of Income tax	**Slab (Taxable Income)**
Nil	Upto Rs 50000
10% of the income exceeding Rs 50000	Between Rs 50001 to 60000
Rs 1000 + 20% of the income exceeding Rs 60000	Between Rs 60001 to 150000
19000 + 30% of the income exceeding Rs 150000	Above Rs 150000

21. The following are the details of the income and investments of Mr. Smith for a particular year:

Annual Salary	Read from user (Let Rs 160000)
L.I.C. Premium	Read from user (Let Rs 18000 per anum)
Provident Fund	Read from user (Let Rs 1500 per month)
Tax deduct at source	Read from user (Let Rs 500 pre month)

Calculate the tax payable at the end of the year. You have to use the slab table for calculating the tax

The standard deduction: Rs 20000

Tax Rebate: 20% of all investments.

23. Mr. Alice San. is a executive engineer, the annual income statement for the financial year is given below.

1. Basic Pay	Read from the user (Let Rs 181540)
2. Dearness allowance(DA)	Read from the user (Let Rs 30258)
3. House-rent allowance(HRA)	Read from the user (Let Rs 18740)
4. Interim relief	Read from the user (Let Rs 1800)
5. City compensatory allowance(CCA)	Read from the user (Let Rs 14200)
6. Deduction :	
(a). Contribution to PF	Read from the user (Let Rs 10654)
(b). LIC premium	Read from the user (Let Rs 1300)
(c). Group Insurance premium	Read from the user (Let Rs 240)

If Mr. San lives in a rent-free house, and **Rs 1000** (Read from the user) are Tax deducted from his salary per month for **first 11** months, find the amount of tax he has to pay at the end of the 12th month.

Chapter- 5
Program Flow or Looping

Introduction

Programs accomplish most of their work by branching and looping. On chapter 4, "Expressions and Statements," you learned how to branch your program using the if statement. In this lesson you will learn

- What loops are and how they are used.
- How to build various loops.
- An alternative to deeply-nested if/else statements.

Looping

Many programming problems are solved by repeatedly acting on the same data. There are two ways to do this: **recursion** and **iteration**.
Iteration means doing the same thing again and again until condition is true, when condition is false loop will not work. The principal method of **iteration** is the loop.

While Loops

A while loop causes your program to repeat a sequence of statements as long as the starting condition remains true. In the example of goto, in Pch5_1.C, the counter was incremented until it was equal to 5. Pch5_2.C shows the same program rewritten to take advantage of a while loop.

Pch5_1.C. while loops.

```
// Looping with while
#include <stdio.h>
int main(void)
{
  int counter = 0;           // initialize the condition

    while(counter < 5)     // test condition still true
     {
        counter++;              // body of the loop
        printf("counter: %d\n",counter);
    }

    printf ("Complete. Counter: %d",counter);
     return 0;
}
```

Output:

```
counter: 1
counter: 2

counter: 3
```

```
counter: 4
counter: 5
Complete. Counter: 5
```

do...while Loops

It is possible that the body of a while loop will never execute. The while statement checks its condition before executing any of its statements, and if the condition evaluates false, the entire body of the while loop is skipped. Pch5_5.C illustrates this.

Pch5_5.C. Skipping the body of the while Loop.

```
//Demonstrates skipping the body of the while loop when the condition is false.

  #include <stdio.h>



    int main(void)
    {
       int counter;
        printf ("Enter How many counter you want?: ");
        scanf(“%d”,&counter);
        while (counter > 0)
        {
           printf ("Hello Counter value is:%d\n",counter);
           counter--;
        }
        printf ("Final Counter value is : ",counter);
         return 0;
}
```

Input: Enter How many counter you want?: 2

Output:

```
Hello Counter value is:2
Hello Counter value is:1
Final Counter value is :0
```

Input: Enter How many counter you want?: 0

Output:

```
Final Counter value is :0
```

Pch5_6.C. Demonstrates do...while loop.

```
// Demonstrates do while

#include <stdio.h>


```

```
int main(void)
{
   int counter;
   printf(" Enter How many counter you want?:");
    scanf("%d",&counter);
    do
    {
       printf(" Hello Counter value is:%d\n",counter);
       counter--;
    }  while (counter >0 );
    printf (" Final Counter value is:%d\n",counter);
     return 0;
}
```

Input: Enter How many counter you want?: 2
Output:

```
Hello Counter value is:2
Hello Counter value is:1
Final Counter value is:0
```

Input: Enter How many counter you want?: 0
Output:

```
Hello Counter value is:0
Final Counter value is:0
```

An example to find the greatest common factor (GCD) between two numbers

Pch5_7.C. Complex while loops.

```

// Complex while statements

#include <stdio.h>

int main(void)
{
           unsigned int num1,num2;
int reminder;
 printf("Enter Two unsigned integers To find the GCD/HCF:");
 scanf("%d%d",&num1,&num2);
 do
 {
       reminder = num1%num2;
       if(reminder  == 0)
           break; /* gcd computed, i.e. num2 is gcd*/

       num1=num2;
       num2 = reminder;
 }while(num1>0);
21:   printf("The GCD is:%d\n",num2);
22: return 0;
23: }
```

Analysis: This program is used to find the gcd of two numbers. In the line 10 two numbers are asked to enter. They are stored in num1 and num2.In line 12 the do loop starts. In line 14 we are finding the reminder among num1 and num2.In the line 15 we are checking if the reminder is 0 then we are breaking he loop because we got the gcd which is num2.If the reminder is not 0 the break will not work. Then line 18 will work, in the line 18 num2 value will assign to num1. In the line 19, the reminder value is assign to num2. In line 20 the while condition is checking for it, a time will cone the line 15 will be true and then loop will break. When the break will work us at that time the GCD will be num2. So at the end the line 21 will work and it will print modified value in num2.

For Loops

When programming while loops, you'll often find yourself setting up a starting condition, testing to see if the condition is true, and incrementing or otherwise changing a variable each time through the loop. Pch5_8.C demonstrates this.

Pch5_8.C. While reexamined.

```
//  Looping with while

          #include <stdio.h>


int main(void)
{
  int counter = 0;

    while(counter < 5)
    {
         counter++;
         printf("Looping!  ");
    }

    printf ("\nCounter:%d\n ",counter);
     return 0;
}
```

Output: Looping! Looping! Looping! Looping! Looping!
Counter: 5.

Pch5_9.C . Demonstrating the for loop.

```
// Looping with for

        #include <stdio.h>


int main(void)
{
  int counter;
  for (counter = 0; counter < 5; counter++)
      printf("Looping! ");

    printf (“\nCounter:%d\n ",counter);
     return 0;
}
```

Output: Looping! Looping! Looping! Looping! Looping!
Counter: 5.

Nested Loops

Loops may be nested, with one loop sitting in the body of another. The inner loop will be executed in full for every execution of the outer loop. Pch5_14.C illustrates writing marks into a matrix using nested for loops.

Pch5_14.C. Illustrates nested for loops.

```
// Illustrates nested for loops

           #include <stdio.h>


int main(void)
{
    int rows, columns;
    char theChar;
      printf ( "How many rows? ");
      scanf(“%d”,&rows);
      printf ("How many columns? ");
      scanf(“%d”,columns);
      printf ("What character? ");
      scanf(“%c”,&theChar);
      for (int i = 0; i<rows; i++)
      {
         for (int j = 0; j<columns; j++)
             printf”%c”,theChar);
         printf("\n");
      }
    return 0;
}
```

Output:
```
How many rows? 4
How many columns? 10
What character? x
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
```

Exercises

Q. How do you choose between if/else and switch?

A. If there are more than just one or two else clauses, and all are testing the same value, consider using a switch statement.

Q. How do you choose between while and do...while?

A. If the body of the loop should always execute at least once, consider a do...while loop; otherwise, try to use the while loop.

Q. How do you choose between while and for?

A If you are initializing a counting variable, testing that variable, and incrementing it each time through the loop, consider the for loop. If your variable is already initialized and is not incremented on each loop, a while loop may be the better choice.

Q. How do you choose between recursion and iteration?

A. Some problems cry out for recursion, but most problems will yield to iteration as well. Put recursion in your back pocket;

Program Exercises:

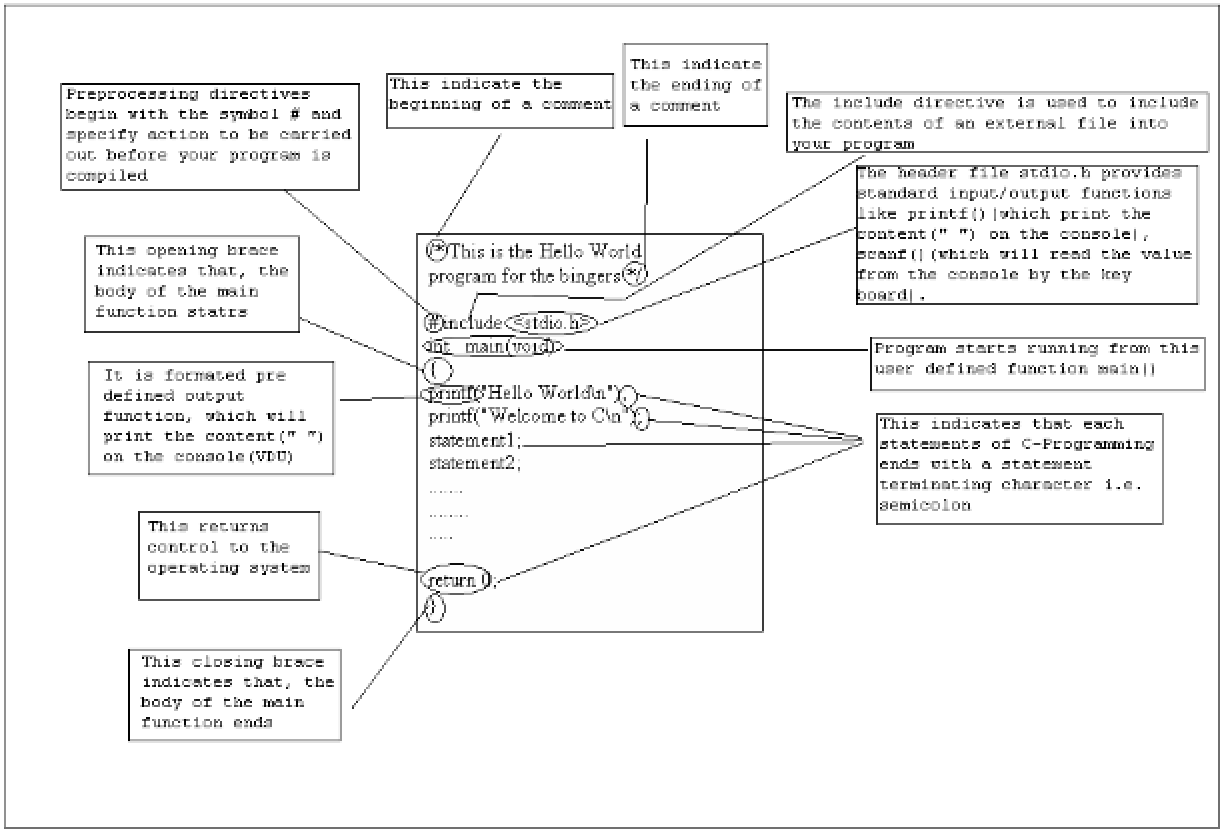
1. Write a program to find the sum of the series .(sum_series.c)
   Sum = 1 – 2 + 3 – 4 + 5 - ………………+n
2. Write a program to print the series .(print_series.c)
   if user will input> g
   Output will be a – b + c – d + e – f + g (it will print upto g)
3. Write a program to find the sum of the series .(sum_series.c).
4. Sum= 1 + 2 – 3 – 4 + 5 + 6 – 7 – 8 + ………..+ n
5. Write a program to find the biggest among 10 elements with out using array.(big_from_10.c)
6. write a program to find the Factorial of a number(factorial_of_a_number.c)
7. Write a program to find the sum of digits of a number (sum_digit_of_a_number.c)

Example: 9864=9+8+6+4=27

8. Write a program to print first 15 Fibonacci numbers. (print_fibonacci.c)
9. Write a program is to check the given number is prime or not (prime_mumber.c)
10. Write a program to convert a decimal number into its corresponding binary number.(decimal_2_binary.c)
11. Write a program to find the gcd of two numbers.(gcd_of_two_numbers.c)
12. Write a program(WAP) to find the biggest among three number and print the in descending order
13. WAP to print the all unsigned characters using any loop
14. WAP is to print the prime numbers from 2 to 100
15. WAP to print the Twin Prime numbers upto a range
16. WAP to check a number Armstrong or not
17. WAP to check a number is a strong number or not
18. WAP to find the factors of a given number.
19. WAP to check a number is a perfect number or not,
20. WAP to check a number is a Palindrome number or not
21. WAP to convert the binary number into decimal number.
22. WAP to convert a binary number to corresponding octal number
23. WAP to convert a decimal number to corresponding octal number.
24. WAP to convert a octal number to corresponding binary number.
25. WAP to convert a octal number to corresponding decimal number
26. WAP to to find the LCM of two numbers.
27. WAP to check a given number is odd or even, if it is odd then check it is prime or not
28. WAP to find the sum of digits of a number and reduce it into single digit.
29. WAP to print the day using switch case.
30. WAP to print the month using switch case
31. WAP using do while and switch to do the arithmetic operations like 1: ADD, 2:SUB, 3:MUL,   4:DIV, 5:REM, 6:EXIT.

Program solutions

<u>This program is to print the "Hello World"</u>


Preprocessing directives begin with the symbol # and specify action to be carried out before your program is compiled
This indicate the beginning of a comment
This indicate the ending of a comment
The include directive is used to include the contents of an external file into your program
The header file stdio.h provides standard input/output functions like printf()|which print the content(" ") on the console|, scanf()|(which will read the value from the console by the key board|.
This opening brace indicates that, the body of the main function statrs
Program starts running from this user defined function main()
It is formated pre defined output function, which will print the content(" ") on the console(VDU)
This indicates that each statements of C-Programming ends with a statement terminating character i.e. semicolon
This returns control to the operating system
This closing brace indicates that, the body of the main function ends

```
/*This is the Hello World
program for the bingers*/
#include<stdio.h>
int  main(void)
{
printf("Hello World\n");
printf("Welcome to C\n");
statement1;
statement2;
......
.......
....
return 0;
}
```

<u>Program is to print the name, branch, college, roll no, reg No. using printf function.</u>

```
#include<stdio.h>
int main(void)
{
printf("\n Name      : Biswa");
printf("\n Branch    :CSE");
printf("\n College   :UU");
printf("\n Age       :23");
printf("\n Roll No   :200810163");
printf("\n Reg No    :1233445");
return 0;
}
```

Output:

```
Name     : Alice
Branch   :CSE
College  :NIST
Age      :23
Roll No  :200810163
Reg No   :1233445
```

Program to demonstrate the sizeof() operator

Program to demonstrate the Size of data types.

```
#include <stdio.h>
int main(void)
{
 printf("\n the size of char data type is=%d bytes",sizeof (char));
 printf("\n the size of short int data type is=%d bytes",sizeof (short int));
 printf("\n the size of int data type is=%d bytes",sizeof (int));
 printf("\n the size of long data type is=%d bytes",sizeof (long));
 printf("\n the size of float data type is=%d bytes",sizeof (float));
 printf("\n the size of double data type is=%d bytes",sizeof(double));
 printf("\n the size of long double data type is=%d bytes",sizeof(long double));
 printf("\n the size of long long data type is=%d bytes",sizeof(long long));
 printf("\n the size of char* pointer is=%d bytes",sizeof (char*));
 printf("\n the size of int* pointer is=%d bytes",sizeof (int*));
 printf("\n the size of 'A' is=%d bytes",sizeof ('A'));
 printf("\n the size of 23.45 is=%d bytes",sizeof (23.45));
 printf("\n the size of 23.45f is=%d bytes",sizeof (23.45f));
 printf("\n the size of 23.45L is=%d bytes",sizeof (23.45L));
 printf("\n the size of \"Hello\" is=%d bytes",sizeof ("Hello"));
 return 0;
}
```

Program to print the upper and lower range of a data type

```
#include <stdio.h>
#include<limits.h>
#include<float.h>
int main(void)
{
  printf("Variables of type char store values from %d to %d",CHAR_MIN, CHAR_MAX);
printf("\nVariables of type unsigned char store values from 0 to %u",UCHAR_MAX);
printf("\nVariables of type short store values from %d to %d",SHRT_MIN, SHRT_MAX);
printf("\nVariables of type unsigned short store values from 0 to %u",USHRT_MAX);
printf("\nVariables of type int store values from %d to %d", INT_MIN,INT_MAX);
printf("\nVariables of type unsigned int store values from 0 to %u",UINT_MAX);
printf("\nVariables of type long store values from %ld to %ld",LONG_MIN, LONG_MAX);
printf("\nVariables of type unsigned long store values from 0 to %lu",ULONG_MAX);
printf("\nVariables of type long long store values from %lld to %lld",LLONG_MIN,
LLONG_MAX);
printf("\nVariables of type unsigned long long store values from 0 to %llu",ULLONG_MAX);
printf("\n\nThe size of the smallest non-zero value of type float is %.3e",FLT_MIN);
printf("\nThe size of the largest value of type float is %.3e", FLT_MAX);
printf("\nThe size of the smallest non-zero value of type double is %.3e",DBL_MIN);
printf("\nThe size of the largest value of type double is %.3e", DBL_MAX);
printf("\nThe size of the smallest non-zero value of type long double is %.3Le", LDBL_MIN);
printf("\nThe size of the largest value of type long double is %.3Le\n",LDBL_MAX);
printf("\nVariables of type float provide %u decimal digits precision.",FLT_DIG);
printf("\nVariables of type double provide %u decimal digits precision.",DBL_DIG);
printf("\nVariables of type long double provide %u decimal digits precision.",LDBL_DIG);
return 0;
}
```

Program to print the different data type with proper output format specification.

```
#include <stdio.h>
int main(void)
{
unsigned short int usint=34;
short int sint=12;
unsigned int age=23;
int roomNo=123;
unsigned long ulg=234124;
long roll=200810163;
unsigned char unchar='#';
char gender='M';
float price=12.23f;
double distance=2233.5456;
long double far_distance=4567.67E-45;
char name[20]="Alice San";
printf("\nTo print unsigned short int(usint)=%hu",usint);
printf("\nTo print signed short int(sint)=%hd",sint);
printf("\nTo print signed short int(sint)=%hi",sint);
printf("\nTo print unsigned int(age)=%u",age);
printf("\nTo print singed int(roomNo)=%d",roomNo);
printf("\nTo print (default signed) int(roomNo)=%i",roomNo);
printf("\nTo print unsigned long(ulg)=%Lu",ulg);
printf("\nTo print long(roll)=%ld",roll);
printf("\nTo print long(roll)=%li",roll);
printf("\nTo print unsigned char(unchar)=%c",unchar);
printf("\nTo print char(gender)=%c",gender);
printf("\nTo print float(price)=%.2f",price);
printf("\nTo print double(distance)=%g",distance);
printf("\nTo print double(distance)=%e",distance);
printf("\nTo print long double(far_diatnce)=%Le",far_distance);
printf("\nTo print the string(Name=%s",name);
  return 0;
}
```

<u>This program is to print the ASCII value of the character</u>

```
#include <stdio.h>
int main(void)
{
char first = 'N';
char second = 92;
printf("\nThe first example as a letter looks like this = %c", first);
printf("\nThe first example as a number looks like this = %d", first);
printf("\nThe second example as a letter looks like this = %c", second);
printf("\nThe second example as a number looks like this = %d\n", second);
return 0;
}
```

Output:

```
The first example as a letter looks like this = N
The first example as a number looks like this = 78
The second example as a letter looks like this = a
The second example as a number looks like this = 97
```

<u>Write a program to convert kilometer into meter.</u>

```
#include <stdio.h>
int main(void)
{
int km,m;
km=23; /*input for kilometer*/
m=km*1000; /*finding the meter */
printf("The meter is=%d",m); /*printing the meter*/
return 0;
}
```

<u>Write a program to convert kilometer into meter.(using reading data from console(scanf))</u>

```
#include <stdio.h>
int main(void)
{
float km,meter;
printf("\n Enter the length in kilometer :");
scanf("%f",&km);/*reading a floating number*/
meter=km*1000;
printf("\n\t The length in meter is :%f",meter);
return 0;
}
```

Program to find the area and perimeter of a circle.

```
#include<stdio.h>
#define PI 3.141
int main(void)
{
float area, perimeter, radius;
printf("\n Enter the Radius of a circle :");
scanf("%f",&radius);
perimeter=(PI+PI)*radius;
area=PI*radius*radius;
printf("\nThe perimeter and area of a circle is given bellow");
printf("\n\n\t\t Radious \t Perimeter \t Area ");
printf("\n\n\t\t %f     \t  %f\t%f  ",radius,perimeter,area);
return 0;
}
```

This program is to find the surface area and volume of a sphere

```
#include<stdio.h>
#define PI 3.141
int main(void)
{
float surface_area, volume, radius;
printf("\n Enter the Radius of a sphere :");
scanf("%f",&radius);
volume=((PI+PI+PI)*radius*radius)/4;
surface_area=(PI+PI+PI+PI)*radius*radius;
printf("\nThe surface area and volume of a sphere is given bellow");
printf("\n\n\t Radious \t Surface area \t Volume ");
printf("\n\n\t %f     \t  %f\t%f  ",radious,surface_area,volume);
return 0;
}
```

Program to convert a length into yard, feet and inches

```
#include <stdio.h>
int main(void)
{
    int length,yard,feet,inches;
    printf("\nENtre the length in inches:");
    scanf("%d",&length);
    yard=length/36;
    length=length%36;
    feet=length/12;
    inches=length%12;
    printf("\nThe %d yard, %d feet and %d inches",yard,feet,inches);
   return 0;
}
```

This program is to convert Fahrenheit to Celsius

```
#include<stdio.h>
int main(void)
{
float farn,cel;
printf("\n Enter the temp in Fahrenheit :");
scanf("%f",&farn);
cel=(farn-32)*5.0/9;
printf("\n The temp in Celsius is :%.2f",cel);
return 0;
}
```

Output:

```
Enter the temp in Fahrenheit :98.7
The temp in Celsius is :37.06
```

Write a program to Add two fractions

```
#include <stdio.h>
int main(void)
{
                int n1,d1,n2,d2,n,d;
        printf("\n Enter the nuenator and denominator of 1st firction:");
        scanf("%d%d",&n1,&d1);
        printf("\n Enter the nuenator and denominator of 2st firction:");
        scanf("%d%d",&n2,&d2);
        d=d1*d2;
        n=n1*d2+n2*d1;
        printf("\nThe sum of two friction is %d/%d",n,d);
   return 0;
}
```

Output:

```
Enter the numerator and denominator of 1st fraction:2 3
Enter the numerator and denominator of 2st fraction:5 7
The sum of two fraction is 29/21
```

Program to calculate the value of the given polynomial: $Y= ax^3 + bx^2 + cx + d$.

```
#include <stdio.h>
int main(void){
   int a,b,c,d,x,Y;
   printf("\nEnter  the value of a,b,c,d and x:");
   scanf("%d%d%d%d%d",&a,&b,&c,&d,&x);
   Y=((a*x+b)*x+c)*x+d;
   printf("\nThe value of Y is: %d",Y);
  return 0;
}
```

Output:

```
Enter  the value of a,b,c,d and x:2 3 4 5 2
The value of Y is: 41
```

Program to calculate the value of distance by using the given formula $S=ut+1/2at^2$

```
#include <stdio.h>
int main(void){
    float s,u,a,t;
    printf("\nEnter  the value of u,t and a :");
    scanf("%f%f%f",&u,&t,&a);
    s=u*t+a*t*t*1/2;
    printf("\nThe value of s is: %.2f",s);
  return 0;
}
```

Program is to find the roots of a quadratic equation

```
//header files
#include<stdio.h>
#include<math.h>
int main(void){
int a,b,c,D;
float root1,root2;
printf("\n Enter the value of a,b and c:");
scanf("%d%d%d",&a,&b,&c);
printf("\nRoots are given bellow");
D=(b*b)-(4*a*c);
if(D<0)
  {
    printf("\n \t The roots are Imginary because D have -ve value");
  }
else
if(D==0)
    {
     printf("\n\t The roots are equal and the value is given bellow");
     root1=root2=(-b)/(a+a);
     printf("\n \n\t\troot1=root2=%f",root1);
    }
  else
    {
    printf("\n\t The roos are real and unequal biven bellow:");
    root1=((-b)+(sqrt(D)))/(a+a);
    root2=((-b)-(sqrt(D)))/(a+a);
    printf("\n\n\t root1  \t root2\n");
    printf("\n\t %f\t%f",root1,root2);
    }
return 0;
}
```

Program to calculate the sum and average of five numbers

```
#include <stdio.h>
int main(void){
    int n1,n2,n3,n4,n5,sum;
    float avg;
    printf("\nEnter  the 5 numbers :");
    scanf("%d%d%d%d%d",&n1,&n2,&n3,&n4,&n5);
    sum=n1+n2+n3+n4+n5;
    avg=(float)sum/5;
    printf("\nThe sum is=%d",sum);
    printf("\nThe avg is=%.3f",avg);
  return 0;
}
```

Output:

```
Enter  the 5 numbers :4 12 23 45 67
The sum is=151
The avg is=30.20
```

Program to swap two numbers using third variable

```
#include <stdio.h>
int main(void){
    int a,b,temp;
    printf("\nEnter  the of a and b:");
    scanf("%d%d",&a,&b);
    printf("\nBefore swap the value of a=%d and b=%d",a,b);
    temp=a;
    a=b;
    b=temp;
    printf("\nAfter swap the value of a=%d and b=%d",a,b);
  return 0;
}
```

Program to swap two numbers without using third variable

```
#include <stdio.h>
int main(void){
    int a,b,temp;
    printf("\nEnter  the of a and b:");
    scanf("%d%d",&a,&b);
    printf("\nBefore swap the value of a=%d and b=%d",a,b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf("\nAfter swap the value of a=%d and b=%d",a,b);
  return 0;
}
```

Program to find the simple interest

```
#include<stdio.h>
int main(void){
float time,rate,principal,interest,amount;
printf("\n Enter the principal  :");
scanf("%f",&principal);
printf("\n Enter the time in years :");
scanf("%f",&time);
printf("\n Enter the rate :");
scanf("%f",&rate);
interest=principal*time*rate/100;
amount=principal+interest;
printf("\n\t Principal \tTime \tRate \tInterest \tAmount ");
printf("\n\n\t %.2f  \t%.2f \t %.2f  \t%.2f  \t%.2f  ",principal,time,rate,interest,amount);
return 0;
}
```

Output:

```
Enter the principal  :200
Enter the time in years :1.5
Enter the rate :12.25
        Principal     Time    Rate    Interest       Amount
        200.00        1.50    12.25   36.75          236.75
```

Program to find the Area of triangle using hero's formulahero's formula s2=s*(s-a)*(s-b)*(s-c)*/

```
#include <stdio.h>
#include <math.h>
int main(void){
  float s,a,b,c,A;
  printf("\nEnter the three sides of a Triangle a, b & c:");
  scanf("%f%f%f",&a,&b,&c);
  s=(a+b+c)/2;/* to find the semi perimeter (s) of the triangle */
  A=sqrt( s*(s-a)*(s-b)*(s-c));
  printf("\nThe area of the triangle is: %.2f",A);
  return 0;
}
```

Output:

```
Enter the three sides of a Triangle a, b & c:3 4 2
The area of the triangle is: 2.90
```

<u>Program to do operations over Operators</u>

```
#include<stdio.h>
int main(void){
int a,b,c;
a = 2 + (b = 5);
printf("a=%d\n",a);
printf("b=%d\n",b);
a = b = c = 5;
printf("a=%d\nb=%d\nc=%d\n",a,b,c);
c=a+b;
printf("c=%d\n",c);
c    =       a    +     b     ;
printf("c=%d\n",c);
{
a=10;
b=20;
c=a+b;
printf("value ic c in side the block is=%d\n",c);
}
printf("value ic c out side the block is=%d\n",c);
a=2;
b=4;
c=(a,b);
printf("c=%d\n",c);
c=(b=7,a=5,a+b);
printf("c=%d\n",c);
return 0;
}
```

Output:

```
a=7
b=5
a=5
b=5
c=5
c=10
c=10
value ic c in side the block is=30
value ic c out side the block is=30
c=4
c=12
```

<u>Program to do operations over conditional Operators</u>

```
#include<stdio.h>
int main(void){
int a,b,c;
a=12;
b=19;
c=(a>b);
printf("c=%d\n",c);
c=(a>=b);
printf("c=%d\n",c);
c=(a<b);
printf("c=%d\n",c);
c=(a<=b);
printf("c=%d\n",c);
c=(a==b);
printf("c=%d\n",c);
c=(a!=b);
printf("c=%d\n",c);
c=(a>4 && b>a);
printf("c=%d\n",c);
c=(a>4 || b>a);
printf("c=%d\n",c);
return 0;
}
```

Program to do operations over bit wise logical Operators

```
#include<stdio.h>
int main(void)
{
int a,b,c;
a=12;
b=19;
c=(a&b);
printf("c=%d\n",c);
c=(a|b);
printf("c=%d\n",c);
c=(a^b);
printf("c=%d\n",c);
c=~a;
printf("c=%d\n",c);
c=~b;
printf("c=%d\n",c);
c=(a>>3);
printf("c=%d\n",c);
c=(a<<4);
printf("c=%d\n",c);
c=(a>b) ? a : b;
printf("c=%d\n",c);
return 0;
}
```

Program to do operations over increment and decrement Operators

```
#include<stdio.h>
int main(void){
int a=7;
printf("\n%d",a++);
printf("\n%d",a);
printf("\n%d",a++);
printf("\n%d",a);
printf("\n%d",++a);
printf("\n%d",a);
printf("\n%d",a++);
printf("\n%d",a);
printf("\n%d",++a);
printf("\n%d",a);
printf("\n%d",--a);
printf("\n%d",a);
printf("\n%d",a--);
printf("\n%d",a);
printf("\n%d",a++ + a);
printf("\n%d",a);
printf("\n%d",a + ++a);
printf("\n%d",a);
printf("\n%d",a+++a);
printf("\n%d",a);
printf("\n%d",a++ + ++a);
printf("\n%d",a);
printf("\n%d",++a + ++a);
printf("\n%d",a);
printf("\n%d",a++ + a++);
```

Program to swap two numbers without using third variable(using bit wise XOR(^) operator)

```
#include<stdio.h>
int main(void){
int a,b;
printf("\nEnter  the of a and b:");
scanf("%d%d",&a,&b);
printf("Before swap a=%d and b=%d\n",a,b);
a=a^b;
b=a^b;
a=a^b;
printf("After swap a=%d and b=%d\n",a,b);
return 0;
}
```

Program to swap two numbers without using third variable(using ^= operator)

```
#include<stdio.h>
int main(void){
int a,b;
printf("\nEnter  the of a and b:");
scanf("%d%d",&a,&b);
printf("Before swap a=%d and b=%d\n",a,b);
a ^=b ^=a ^=b;
printf("After swap a=%d and b=%d\n",a,b);
return 0;
}
```

```
#include <stdio.h>
int main(void){
    int a,b,c,temp;
    printf("\nEnter  the of a , b and c:");
    scanf("%d%d%d",&a,&b,&c);
    printf("\nBefore swap the value of a=%d ,b=%d and c=%d",a,b,c);
    temp=c;
    c=b;
    b=a;
    a=temp;
    printf("\nAfter swap the value of a=%d ,b=%d and c=%d",a,b,c);
    return 0;
}
```

Output:

```
Enter  the of a , b and c:10 20 30
Before swap the value of a=10 ,b=20 and c=30
After swap the value of a=30 ,b=10 and c=20
```

Program to swap three numbers without using fourth variable

```
#include <stdio.h>
int main(void){
    int a,b,c,temp;
    printf("\nEnter  the of a , b and c:");
    scanf("%d%d%d",&a,&b,&c);
    printf("\nBefore swap the value of a=%d ,b=%d and c=%d",a,b,c);
    a=a+b+c;
    b=a-(b+c);
    c=a-(b+c);
    a=a-(b+c);
    printf("\nAfter swap the value of a=%d ,b=%d and c=%d",a,b,c);
    return 0;
}
```

Program to swap three numbers without using fourth variable(using XOR(^) operator)

```
#include <stdio.h>
int main(void){
    int a,b,c,temp;
    printf("\nEnter  the of a , b and c:");
    scanf("%d%d%d",&a,&b,&c);
    printf("\nBefore swap the value of a=%d ,b=%d and c=%d",a,b,c);
    a=a^b^c;
    b=a^b^c;
    c=a^b^c;
    a=a^b^c;
    printf("\nAfter swap the value of a=%d ,b=%d and c=%d",a,b,c);
    return 0;
}
```

Program to find the absolute value of an integer.

```
#include<stdio.h>

int main(void)
{
int number;
printf("\nEnter a number to find the absolute value :");
  scanf("%d",number);
if(number<0)
   {
        number = - number;
   }
printf("\nThe absolute value of the given number is:%d",number);
  return 0;
}
```

Program to read two numbers from the user and print the biggest among them

```
#include <stdio.h>
int main(void){
   int a,b;
    printf("\nEnter  the of a and b:" );
    scanf("%d%d",&a,&b);
    if(a>b)
      printf("\nThe number %d is bigger ",a);
    else
      printf("\nThe number %d is bigger ",b);
  return 0;
}
```

Program to find the biggest among 3 numbers

```
#include<stdio.h>
int main(void){
int first,second,third,big;
printf("\n Enter the first number :");
scanf("%d",&first);
printf("\n Enter the second number :");
scanf("%d",&second);
printf("\n Enter the third number :");
scanf("%d",&third);
if(first>=second&&first>=third)
   big=first;
else if(second>=third)
              big=second;
     else
              big=third;
printf("\nThe bigest amoung three values is ");
printf("%d , %d and %d  \tBig=%d  ",first,second,third,big);
return 0;
}
```

Output:

```
Enter the first number :4 2 6
Enter the second number :
Enter the third number :
The bigest amoung three values is        4 , 2 and 6    Big=6
```

Program to find the biggest among 4 numbers

```
#include <stdio.h>
int main(void){
        int a,b,c,d,big;
printf("\n Enter 4 numbers :");
scanf("%d%d%d%d",&a,&b,&c,&d);
if(a>b && a>c && a>d)
   big=a;
else if(b>c && b>d)
              big=b;
   else if(c>d)
      big=c;
        else
          big=d;
printf("\nThe big is=%d",big);
  return 0;
}
```

Program to a number is even or odd

```
#include <stdio.h>
int main(void){
        int a;
printf("\n Enter a number :");
scanf("%d",&a);
if(a%2==0)
printf("\nThe %d is even",a);
else
printf("\nThe %d is odd",a);
  return 0;
}
```

Output:

```
Enter a number:34
The 34 is even
```

Program to read three points of a triangle and test for co linearity

```
#include <stdio.h>
int main(void){
        int x1,y1,x2,y2,x3,y3;
printf("\n Enter x and y coorinate of 1st point :");
scanf("%d%d",&x1,&y1);
printf("\n Enter x and y coorinate of 2st point :");
scanf("%d%d",&x2,&y2);
printf("\n Enter x and y coorinate of 3st point :");
scanf("%d%d",&x3,&y3);
if(((x1*y2+x2*y3+x3*y1)-(y1*x2+y2*x3+y3*x1))==0)
  printf("\nThe points are colliner");
else
    printf("\nThe points are not colliner");
  return 0;
}
```

Program to read a point from the user and print that point belongs to 1st or 2nd or 3rd or 4th quadrant

```
#include <stdio.h>
int main(void){
        int x,y;
printf("\n Enter x and y coorinate of point :");
scanf("%d%d",&x,&y);
if(x==0 && y==0)
  printf("point P(%d,%d) is at origin",x,y);
if(x==0 && y!=0)
  printf("point P(%d,%d) is at y-axis",x,y);
if(x!=0 && y==0)
  printf("point P(%d,%d) is at x-axis",x,y);
if(x>0 && y>0)
  printf("point P(%d,%d) is at 1st Quadrant ",x,y);
if(x<0 && y>0)
  printf("point P(%d,%d) is at 2nd Quadrant ",x,y);
if(x<0 && y<0)
  printf("point P(%d,%d) is at 3rd Quadrant ",x,y);
if(x>0 && y<0)
  printf("point P(%d,%d) is at 4th Quadrant ",x,y);
  return 0;
}
```

Output:

```
Enter x and y coordinate of point :3 0
point P(3,0) is at x-axis
Enter x and y coordinate of point :-3 4
point P(-3,4) is at 2nd Quadrant
```

Program to read a point from the user and print that point whether it present above the x-axis or below the x-axis.

```
#include <stdio.h>
int main(void){
        int x,y;
printf("\n Enter x and y coorinate of point :");
scanf("%d%d",&x,&y);
if(x==0 && y==0)
  printf("point P(%d,%d) is at origin",x,y);
if(x==0 && y!=0)
  printf("point P(%d,%d) is at y-axis",x,y);
if(x!=0 && y==0)
  printf("point P(%d,%d) is at x-axis",x,y);
if(y>0)
  printf("point P(%d,%d) is above  x-axis ",x,y);
if(y<0)
  printf("point P(%d,%d) is at below x-axis ",x,y);
if(x>0)
```

```
  printf("point P(%d,%d) is at right side of y-axis ",x,y);
if(x<0)
  printf("point P(%d,%d) is at left side of y-axis ",x,y);
  return 0;
}
```

Output:

```
Enter x and y coordinate of point :-3 4
point P(3,0) is at left side of y-axis
Enter x and y coordinate of point :3 -4
point P(-3,4) is at below x-axis Quadrant
```

<u>Program to read the three lengths of a triangle and test whether a triangle can construct or not.</u>

```
#include <stdio.h>
int main(void){
        int a,b,c;
printf("\n Enter the lengths of the triangle :");
scanf("%d%d%d",&a,&b,&c);
if((a+b)>c && (b+c)>a && (a+c)>b)
  printf("\nThe triangle is possible");
else
    printf("\nThe triangle is not possible");
  return 0;
}
```

Output:

```
Enter the lengths of the triangle :3 4 5
The triangle is possible
Enter the lengths of the triangle :3 3 8
The triangle is not possible
```

<u>Program to read a year from the user and test it is a leap year or not.</u>

```
#include <stdio.h>
int main(void){
        int y;
printf("\n Enter a year:");
scanf("%d",&y);
if((y%4==0&& y%100!=0)||y%400==0)
printf("%d year is leap year",y);
else
printf("%d year is not leap year",y);
  return 0;
}
```

Output

```
Enter a year:1900
1900 year is not leap year
Enter a year:2000
```

```
2000 year is leap year
Enter a year:2008
2008 year is  leap year
```

Write a program to check the two lines L1 (formed by points p1 & p2) and L2 (formed by points p3 & p4) are parallel or perpendicular or collinear.

Program to read the three lengths of a triangle and test what type of that triangle is it

```
#include<stdio.h>
int main(void)
{
        int a,b,c,x,y,z;
        int TRIANGLE=0,OBTUSE=0,ACUTE=0,RIGHTANGLE=0,EQLTRL=0,ISOCLS=0;
        printf("\n Enter three sides of a triangle ");
        scanf("%d %d %d",&a,&b,&c);
        if(a>b && a>c)
        {
              x=a;
              y=b;
              z=c;
        }
        else          if(b>c)
              {
                     x=b;
                     y=c;
                     z=a;
              }
              else
              {
                     x=c;
                     y=a;
                     z=b;
              }
TRIANGLE= x < y + z;                    /*  Triangle is formed                      */
OBTUSE= x*x > y*y > z*z;         /*  A Obtuse-angled triangle is formed   */
ACUTE= x*x < y*y + z*z;                 /*  A Acute-angled triangle is formed    */
RIGHTANGLE= x*x==y*y+z*z;          /*  A Right-angled triangle is formed    */
EQLTRL= x==y && y==z;                   /*  An Equilateral triangle is formed    */
ISOCLS= x==y || y==z || z==x;          /*  An Isosceles triangle is formed      */

if(TRIANGLE)
  {
        printf("\n Triangle is possiable");
        if(RIGHTANGLE)
              printf("\n Right angle triangle");
```

```
            else
            if(OBTUSE)
                    printf("\n Obtuse angle triangle");
            else
            if(ACUTE)
            {
                    printf("\n Acute angle triangle");
                    if(EQLTRL)
                            printf("\nEquilateral triangle");
                    else
                    if(ISOCLS)
                            printf("\nIsosceles triangle");
                    else
                            printf("\nScalene triangle");
            }
  }
  else
            printf("\nNot a triangle");
return 0;
}
```

<u>This program is to find the biggest among three number and print the in descending order</u>

```
#include<stdio.h>
#define AND &&
int main(void){
int a,b,c;
printf("\n Enter the three numbers a,b and c:");
scanf("%d%d%d",&a,&b,&c);
printf("\nThe biggest and descending order is given bellow");
printf("\n Big  Descending Order");
if(a>b AND a>c)
   {
  printf("\nBig=%d",a);
   if(b>c)
      printf("\t %d %d %d",a,b,c);
   else
      printf("\t %d %d %d",a,c,b);
   }
else
  if(b>c)
      {
            printf("\n\n \t %d",b);
            if(a>c)
               printf("\t %d %d %d",b,a,c);
            else
```

```
                printf("\t %d %d %d",b,c,a);
      }
   else
    {
       printf("\n\n \t %d",c);
              if(a>b)
              printf("\t %d %d %d",c,a,b);
      else
              printf("\t %d %d %d",c,b,a);
    }
return 0;
}
```

This check a character whether it is vowel or not

```
#include<stdio.h>
int main(void){
char c;
printf("\nEnter Character to check it is vowel or not:");
scanf("%c",&c);
if(c=='a'||c=='A'||c=='e'||c=='E'||c=='i'||c=='I'||c=='o'||c=='O'||c=='u'||c=='U')
{
  printf("\nThe Entered character '%c' is a vowel ",c);
}
else
{
  printf("\nThe Entered character '%c' is not a vowel",c);
}
return 0;
}
```

Program is to check a character is a Capital Or Small alphabets Or Digit or Special or nonprintable character *

```
#include<stdio.h>
#define AND &&
#define OR ||
int main(void){
char c;
printf("\nEnter Character to check it is Capital Or Small alphabets Or Digit or Special or
nonprintable character:");
scanf("%c",&c);
if(c>=65 AND c<=90)
   printf("\nThe Entered character '%c' is a Capita Alphabet ",c);
else if(c>=97 AND c<=122)
   printf("\nThe Entered character '%c' is a Small Alphabets",c);
else if(c>=48 AND c<=57)
  printf("\nThe Entered character '%c' is a Digit ",c);
else if(c>=0 AND c<=31)
```

```
  printf("\nThe Entered character '%c' is a non printable character",c);
else
printf("\nThe Entered character '%c' is a Special Character ",c);
return 0;
}
```

program to find the largest and second largest of three numbers

```
# include <stdio.h>

# define BIG(a,b) ((a>b)?(a):(b))
# define SAM(a,b) ((a<b)?(a):(b))
int main(void){
            int n,m,o,lar,sma,seclar;
            printf("Enter the three numbers :");
            scanf("%d %d %d",&n,&m,&o);
            lar = BIG((BIG(n,m)),o);
            sma = SAM((SAM(n,m)),o);
            seclar = (n+m+o)-(lar+sma);
            printf("Largest of three numbers is  %d\n",lar);
            printf("Second Largest of three numbers is %d",seclar);
return 0;
}
```

program to find the largest and second largest of three numbers

```
# include <stdio.h>
# define BIG(a,b) ((a>b)?(a):(b))
# define SAM(a,b) ((a<b)?(a):(b))
int main(void)
{
            int n,m,o,lar,sma,seclar;
            printf("Enter the three numbers :");
            scanf("%d %d %d",&n,&m,&o);
            lar = BIG((BIG(n,m)),o);
            sma = SAM((SAM(n,m)),o);
            seclar = (n+m+o)-(lar+sma);
            printf("Largest of three numbers is  %d\n",lar);
            printf("Second Largest of three numbers is %d",seclar);
return 0;
}
```

Program to find the smallest among N elements with out using array.

```
#include<stdio.h>
int main(void){
int n,number,i,small;
printf("\n Enter the value of  N:");
scanf("%d",&n);
printf("Enter the elements:");
for(i=1;i<=n;i++)
{
  scanf("%d",&number);
  if(small>number)
     small=number;
}
printf("\n The smalest element is %d",small);
return 0;
}
```

Program is to print the table of a given number

```
#include<stdio.h>
int main(void){
int number,i;
printf("\n Enter a number to the Table :");
scanf("%d",&number);
printf("\nThe %dth Table is",number);
for(i=1;i<=10;i++)
   printf("\n\t %d * %d  =%d",number,i,number*i);
return 0;
}
```

Program to print odd and even numbers up to a range

```
#include<stdio.h>
int main(void){
int number,i=0;
printf("\n Enter a number to print the odd and even numbers upto that range :");
scanf("%d",&number);
printf("\n\t ODD \t\t EVEN");
while(i<=number) {
  if(i%2==0)
  printf("\n\t  %d",i);
  else
  printf("\t\t  %d",i);
i++;
   }
return 0;
}
```

Programs to print even numbers upto a range and find the average

```
#include<stdio.h>
int main(void){
int number,i=0,sum=0,n=0;
float avg=0;
printf("\n Enter a number to print even numbers and find the average upto range :");
scanf("%d",&number);
printf("\n\t EVEN");
while(i<=number)
  {
```

```
  if(i%2==0)
  {
    sum +=i;
    printf("\n\t  %d",i);
    n++;
  }
  i++;
 }
  avg=sum/n;
printf("\n\t The sum of %d even number is:%d",n,sum);
printf("\n\t The avg of %d even number is: %f",n,avg);
return 0;
}
```

program is to find the factorial of a given number

```
#include<stdio.h>
int main(void){
int number,i;
long factorial=1;
printf("\n\t Enter a number to the factorial :");
scanf("%d",&number);
if(number==0 || number <0)
  printf("\n\t the factorial of %d is: %d",number,1);
else
  for(i=1;i<=number;i++)
    factorial *=i;

printf("\n\t the factorial of the number %d is %ld",number,factorial);
return 0;
}
```

Program is to a given number is prime or not

```
#include<stdio.h>
#include<math.h>
int main(void){
          int n,i,flag;
          printf("Enter a number");
          scanf("%d",&n);
          printf("\n Enter a numbers:");
          flag=0;
                  for(i=2;i<=sqrt(n);i++)
                  {
                  if(n%i==0)
                  {
                           flag=1;
                           break;
                  }
```

```
                }
        if(flag)
                printf("\nThe Number %d is prime",n);
        else
                printf("\nThe Number %d is not prime",n);
return 0;
}
```

Program is to print the prime numbers from 2 to N

```
#include<stdio.h>
int main(void){
int j,i,flag,n;
printf("\nEnter Value of N:");
scanf("%d",&n);
for(i=2;i<=n;i++)
{
flag=0;
for(j=2;j<=i/2;j++)
{
if(i%j==0)
{
flag=1;
break;
}
}
if(flag==0)
printf(" %d ",i);
}
return 0;
}
```

Program to print the Twin Prime numbers upto a range (twin prime means if the difference of two prime number is 2
/* example: 3-5, 5-7, 11-13, 17-19 etc. */

```
#include<stdio.h>
int main(void){
int j,i,flag,p=0;
int a[100];
printf("\nEnter the value of N:");
scanf("“%d”,&n);
for(i=2;i<=n;i++)
{
flag=0;
for(j=2;j<=i/2;j++)
{
if(i%j==0)
{
flag=1;
break;
}
}/* end of jth loop*/
if(flag==0)
a[p++]=i;
} /*end of ith loop */
for(j=0;j<=p-1;j++)/* check for twin prime*/
{
if(a[j+1]-a[j]==2)
{
printf("\n\t The %d and %d are Twin Prime numbers ",a[j],a[j+1]);
}
}
return 0;
}
```

Program is to fine the nCr

```
#include<stdio.h>
int main(void)
{
        int n,r,i,ncr,nfact,n_rfact,rfact;
        printf("Enter n and r values");
        scanf("%d %d",&n,&r);
        if(n<r)
              printf("\nInvalid input");
        else
        {
              if(n==r)
                     ncr=1;
              else
              {
                     nfact=1;
                     n_rfact=1;
                     rfact=1;
                     for(i=1;i<=n;i++)
                        nfact*=i;

                     for(i=1;i<=n-r;i++)
                        n_rfact*=i;

                     for(i=1;i<=r;i++)
                     rfact*=i;

                     ncr=nfact/(n_rfact*rfact);
              }
              printf("\n Factorial of %d = %d",n,nfact);
```

```
                printf("\n Factorial of %d = %d",r,rfact);
                printf(" \n%dC%d=%d",n,r,ncr);
         }
return 0;
}
```

<u>Program to find the reverse of a given number</u>

```
#include<stdio.h>
int main(void) {
  int num,dig;
  int rev=0;
  printf("\nEnter the number here: ");
  scanf("%d",&num);
  while(num>0)
  {
                dig=num%10;
                rev=rev*10+dig;
               num=num/10 ;

  }
  printf("\nThe reversed number is: %d ",rev);
return 0;
}
```

<u>Program to find the factors of a given number.</u>

```
#include<stdio.h>
int main(void){
int i,n;
printf("\n ENter the number to find the factor:");
scanf("%d",&n);
printf("\n The factors of the number %d is:",n);
for(i=1;i<=n;i++)
{
if(n%i==0)
printf("%d ",i);
}
return 0;
}
```

Program to check number is a perfect number or not
A number is called perfect number if the sum of the factors(not that value) of a number is equal to the given number
/*example: 6= factors are 1,2,3 so 1+2+3=6*/
/*28= factors are 1,2,4,7,14 so 1+2+4+7+14=28*/

```
#include<stdio.h>
int main(void){
int number,sum=0,i=1,p;
printf("\n Enter a number to check whether it is a perfect number or not :");
scanf("%d",&number);
p=number;
while(i<number)  {
   if(number%i==0)
          sum +=i;
   i++;
  }
if(sum==p)
       printf("\n\tThe number %d is perfect number",p);
   else
      printf("\n\t The number  %d is not perfect number:",p);
return 0;
}
```

Program to check number is a strong number or not
a number is called strong number if the factorial sum of the digits of a number is equal to the given number
145=1!+4!+5!=1+24+120=145

```
#include<stdio.h>
int main(void){
int number,temp,sum=0,fact,rem;
printf("\n Enter a number to check whether it is a Strong number or not :");
scanf("%d",&number);
temp=number;
while(number>0)  {
   rem=number%10;
   fact=1;
   while(rem>0)
     {
       fact *=rem--;
     }
   sum +=fact;
   number /=10;
  }
if(sum==temp)
```

```
    printf("\n\tThe number %d is Strong number",temp);
else
   printf("\n\t The number  %d is not a Strong number:",temp);
return 0;
}
```

Program to check number is a Armstrong number or not
a number is Armstrong if cube sum of the digit is equal to the number
example: 153=1*3+5*3+3*3= 1+125+27=153

```
#include<stdio.h>
#include<math.h>
int main(void){
int number,temp,sum=0,rem;
printf("\n Enter a number to check whether it is a Armstrong number or not :");
scanf("%d",&number);
temp=number;
while(number>0)  {
   rem=number%10;
   sum +=pow(rem,3);
   number /=10;
  }
if(sum==temp)
    printf("\n\tThe number %d is Armstrong number",temp);
else
   printf("\n\t The number  %d is not a Armstrong number:",temp);
return 0;
}
```

Program to check a number is Palindrome number or not

```
#include<stdio.h>
int main(void){
int number,temp,newnumber=0,rem;
printf("\n Enter a number to check whether it is a Palindrome number or not :");
scanf("%d",&number);
temp=number;
while(number>0)  {
   rem=number%10;
   newnumber =newnumber*10+rem;
   number /=10;
  }
printf("\n\t The Reverse of the given %d number is: %d",temp,newnumber);
if(newnumber==temp)
    printf("\n\t The number %d is palindrome number",temp);
else
   printf("\n\t The number  %d is not a palindrome number",temp);
return 0;
}
```

Program is to convert a number into a single digit

```
#include<stdio.h>
int main(void){
long number;
int rem,sum=0;
printf("\n Enter a number to convert it in to single numbers :");
scanf("%ld",&number);
  while(number>0)
    {
     rem=number%10;
      sum +=rem;
      number /=10;
    if(number==0 && sum>9)
    {
      number=sum;
      sum=0;
    }
}
printf("\nThe sum of digits of number to single digit is:%d",sum);
return 0;
}
```

<u>Program is to find the sum or sub or mul or div or mod of two numbers using switch case</u>

```
#include<stdio.h>
int main(void){
int a,b,result;
int c;
printf("\n Enter a two numbers to do the arithmetic operations :");
scanf("%d%d",&a,&b);
printf("\n\t1 .(+)Addition");
printf("\n\t2 .(-)substraction");
printf("\n\t3 .(*)Multipication");
printf("\n\t4 .(/)Division");
printf("\n\t5 .(%)Remender");
printf("\n\t6 .Exit");
printf("\nSelect case for operation:");
scanf("%d",&c);
switch(c){
case 1 :  result=a+b; break;
case 2 :  result=a-b;  break;
case 3 : result=a*b;  break;
case 4 :  result=a/b;  break;
case 5 :  result=a%b;  break;
case 6 :  break;
default :  printf("\n not a valid operator");
}
if(c<6)
  printf("\n\t the result is %d",result);
return 0;
}
```

program to find the sum of the given series using loop.
sum=1 + 2 + 3 + ...+ n.

```
#include<stdio.h>
int main(void){
int n,sum=0,i;
printf("\n Enter a number :");
scanf("%d",&n);
for(i=1;i<=n;i++)
sum = sum+i;

printf("\nThe sum 1+2+3+....+%d=%d",n,sum);
return 0;
}
```

program to find the sum of the series . Sum =   1 – 2 + 3 – 4 + 5 - ………………+n

```
#include<stdio.h>
int main(void)
{
int num,sum=0,i;
printf("ENter the value of n:");
scanf("%d",&num);
for(i=1;i<=n;i++){
   ifi%2==0)
          sum=sum+i;
   else
          sum=sum-i;
}
printf("\n The sum of the series is:%d",sum);
return 0;
}
```

program to find the sum of the given series using loop.
sum=$1^2 + 2^2 + 3^2 + ...+ N^2$.

```
#include<stdio.h>
int main(void){
int n,sum=0,i;
printf("\n Enter a number :");
scanf("%d",&n);
for(i=1;i<=n;i++)
sum = sum+i*i;

printf("\nThe sum 12 + 22 + 32 + ……..+ n2+%d=%d",n,sum);
return 0;
}
```

program to find the sum of the given series using loop.
sum=$1^3 + 2^3 + 3^3 + ...+ N^3$.

```
#include<stdio.h>
int main(void){
int n,sum=0;
printf("\n Enter a number :");
scanf("%d",&n);
for(i=1;i<=n;i=i+2)
{
sum = sum+i*i*i;
}
printf("\nThe sum =%d",sum);
return 0;
}
```

program to find the sum of the given series using loop.
sum=1! + 2! + . . . + n!.

```
#include<stdio.h>
int main(void){
int n,sum=0,i,f;
printf("\n Enter a number :");
scanf("%d",&n);
for(i=1,f=1;i<=n;i++)
{
f=f*i;
sum = sum+f;
}
printf("\nThe sum 1! + 2! + 3! + ………..+ !%d=%d",n,sum);
return 0;
}
```

program to find the sum of the given series using loop.
sum=1/1! + 2/2! + . . . + n/n!.

```
#include<stdio.h>
int main(void){
int n,i,f;
float sum;
printf("\n Enter a number :");
scanf("%d",&n);
for(i=1,f=1;i<=n;i++)
{
f=f*i;
sum = sum+i/(float)f;
}
printf("\nThe sum 1/1! + 2/2! + ……….. + %d/%d!=%f",n,n,sum);
return 0;
}
```

program to find the sum of the given series using loop.

sum=X - X^3/!3 + X^5/!5- X^7/!7 + ….

```
#include<stdio.h>
#include<math.h>
int main(void){
int x,n,i,j,f,p=1;
float sum=0;
printf("\n Enter a value of 'n' :");
scanf("%d",&n);
printf("\nEnter the value of 'X':");
scanf("%d",&x);
for(i=1;i<=n;i=i+2)
{
for(j=1,f=1;j<=i;j++)/*to find the factorial of 'i'*/
f=f*j;
if(p%2==0)
sum = sum + pow(x,i)/(float)f;
else
sum = sum - pow(x,i)/(float)f;
p=p+1;
}
printf("\nThe sum X – X3/!3 + X5/!5 – X7/!7 + …………+%d/%d!=%f",n,n,sum);
return 0;
}
```

program to find the sum of the given series using loop.

sum=1 - X2/!2 + X4/!4 - X6/!6 + . . .

```
#include<stdio.h>
#include<math.c>
int main(void){
int x,n,i;
float sum=0,p=0;
printf("\n Enter a value of 'n' :");
scanf("%d",&n);
printf("\nNEtre the value of 'X':");
scanf("%d",&x);
for(i=0;i<=n;i=i+2)
{
for(j=1,f=1;j<=i;j++)/*to find the factorial of 'i'*/
{
if(i!=0)
f=f*j;
}
if(p%2==0)
sum = sum + pow(x,i)/(flaot)f;
else
sum = sum - pow(x,i)/(flaot)f;
p=p+1;
}
printf("\nThe sum=1 – X2/!2 + X4/!4 – X6/!6 + %d/%d!=%f",n,n,sum);
return 0;
}
```

<u>Program is to convert the decimal number into binary number</u>

```
#include<stdio.h>
int main(void){
int number,i,p;
long binary=0;
i=0,k=1;
printf("\n Enter a decimal number to find the binary value :");
scanf("%ld",&number);
while(number!=0){
   p=number%2;
   p=p*k;
   binary +=p;
   k=k*10;
   number /=2;
}
printf("\n The binary value of the given decimal number is:%ld",binary);
return 0;
}
```

<u>Program is to convert the binary number into decimal number</u>

```
#include<stdio.h>
int main(void){
long number;
int sum=0,r,i=1,flag=0;
printf("\n Enter a binary number to find the decimal
value :");
scanf("%ld",&number);
while(number>0){
r=number%10;
                                        if(r==0 || r==1)/*if reminder is either 1 or 0 the it is a valid binay
                                        number*/
{
sum=sum+r*i;
i=i*2;
number=number/10;
}
else
{
flag++;
break;
}
}
if(flag==0)
printf("\n The decimal value of the given binary number is:%d",sum );
else
printf("It is not a valid binary number !!!!!");
return 0;
}
```

program to convert a binary number to corresponding octal number

```
#include <stdio.h>
#include<math.h>
int main(void){
int oct=0,i,j,b=1,r,r1,a;
long bin;
printf("Enter the binary value:");
scanf("%ld",&bin);
while(bin>0){
r=bin%1000;
i=0,a=0;
while(r>0){
r1=r%10;
a=a+r1*pow(2,i);
r /=10;
i++;
}
oct=oct+a*b;
b=b*10;
bin /=1000;
}
printf("\n the octal value of the given number is:%d ",oct);
return 0;
}
```

program is to convert the decimal number into binary number

/* THIS PROGRAM IS USED TO FIND THE BINARY VALUE OF ANY LONG NUMBER*/

/* I DID THIS PROGRAM USEING ARRAY*/

```
#include<stdio.h>
int main(void){
int number,i,p,a[50];
i=0;
printf("\n Enter a decimal number to find the binary value :");
scanf("%ld",&number);
while(number>0){
p=number%2;
a[i++]=p;
number /=2;
}
printf("\n The binary value of the given decimal number is:");
for(i=i-1;i>=0;i--)
{
printf("%d",a[i]);
}
return 0;
}
```

program to convert a decimal number to corresponding octal number.

```
#include<stdio.h>
int main(){
long int S=0,q=1;
int n,r;
printf("Enter a decimal number:");
scanf("%d",&n);
while(n>0){
r=n%8;
S=S+r*q;
q=q*10;
n=n/8;
}
printf("The Octal equivalent is %ld",S);
return 0;
}
```

Program to convert a octal number to corresponding binary number

```
 #include<stdio.h>
int main(void){
long sum=0,j=1;
int r,flag=0,S,r1,i,number;
printf("\n Enter a Octal number to find the binary value :" );
scanf("%d",&number);
while(number>0){
r=number%10;
if(r !=8 && r !=9)/*if reminder is either 8 or 9 the
it is not a valid octal number*/
{
S=0,i=1;
while(r>0){ /* to find the binary value of corresponding octal digit 'r' */
r1=r%2;
S=S+r1*i;
i=i*10;
r=r/2;
}
sum= sum + S*j;
j =j*1000;
number=number/10;
}
else
{
flag++;
break;
}
}
if(flag==0)
printf("\n The binary value of the given Octal number is:%ld",sum );
else
printf("It is not a valid Octal number !!!!!");
return 0;
}
```

Program to find the gcd of two numbers.

```
#include<stdio.h>
int main(void)
{
int a,b,r,c,d;
printf("Enter two numbers:");
scanf("%d %d",&a,&b);
do{
r=a%b;
if(r==0)
  break;
a=b;
b=r;
}while(a>0);
printf("The GCD of two numbers is %d\n",b);
return 0;
}
```

program to find the LCM of two numbers.

```
#include<stdio.h>
int main(void){
int a,b,m,n,r;
printf("\nEnter the two number :");
scanf("%d%d",&a,&b);
m=a;
n=b;
do
{
r=a%b; /* find the remende*/
if(r==0)/* if remender is zero(0) then break it because
gcd is 'b'*/
break;
a=b;
b=r;
}while(r>0);
printf("\nthe GCD of two number is :%d",b);
printf("\nthe LCM of two number is :%d",m*n/b);
return 0;
}
```

Program printing day using switch case

```
#include<stdio.h>
int main(void)
{
int day;
printf("Enter a day number from 1to 7:");
scanf("%d",&day);
switch(day){
case 1:printf("Sunday");break;
case 2:printf("Monday");break;
case 3:printf("Tuesday");break;
case 4:printf("Wednesday");break;
case 5:printf("Thursday");break;
case 6:printf("Friday");break;
case 7:printf("Saturday");break;
default:printf("It is not a valid day ");
}
return 0;
}
```

Program to printing month using switch case

```
#include<stdio.h>
int main()
{
int month;
printf("Enter a month number from 1 to 12:");
scanf("%d",&month);
switch(month){
case 1:printf("Jan");break;
case 2:printf("Feb"); break;
case 3:printf("March"); break;
case 4:printf("April"); break;
case 5:printf("May"); break;
case 6:printf("June"); break;
case 7:printf("July"); break;
case 8:printf("Aug"); break;
case 9:printf("Sept"); break;
case 10:printf("oct"); break;
case 11:printf("Nov"); break;
case 12:printf("Dec"); break;
default:printf("\nIt is not a valid month number!!! ");
}
return 0;
}
```

program to check a given number is odd or even,if it is odd then check it is prime or not

```
#include<stdio.h>
int main(void){
int n,i,j,flag=0;
printf("\nEnter a number :");
scanf("%d",&n);
if(n%2==1)
{
for(i=2;i<=n/2;i++)
{
if(n%i==0)
{
flag++;
break;
}
}
if(flag==0)
printf("\nThe number %d is prime and Odd",n);
else
printf("\nThe number %d is not prime and ODD",n);
}
else
printf("\n The number %d is even",n);
return 0;
}
```

program to find the prime factors of a given number.

```
#include<stdio.h>
int main(void){
int i,n,j,flag;
printf("\n ENter the number to find the prime factor:");
scanf("%d",&n);
for(i=2;i<=n;i++)
{
if(n%i==0) /* check for factor*/
{
for(flag=0,j=2;j<=i/2;j++) /* check for prime factor*/
{
if(i%j==0)
{
flag++;
break;
}
}
if(flag==0)
printf("%d ",i);

18
}
}
return 0;
}
```

Program is to print the Fibonacci series upto a range

```
#include<stdio.h>
int main(void){
int number,first=0,second=1,third;
printf("\n Enter a number to print the Fibonacci Series :");
scanf("%d",&number);
printf("\nThe Fibonacci Series up to the given %d number is:",number);
  while(first<=number)
  {
   printf(" %d",first);
   third=first+second;
   first=second;
   second=third;
  }
return 0;
}
```

program to print out put if Input -> g Output -> a – b + c – d + e – f + g

```
#include<stdio.h>
int main(void){
char ch,i;
printf("Enter the character:");
scanf("%c",&ch);
for(i='a';i<=ch;i++)
  {
   printf("%c",i);
  if(i==ch)
    break;
  if(i%2==0)
              printf("-");
   else
         printf("+");
  }
return 0;
}
```

Program to find the power of a number i.e. (M^N)

```
#include<stdio.h>
int main(void){
float base,result=1.0f;
int exp;
printf("Enter the value of base:");
scanf("%f",&base);
printf("Enter the value of expont:");
scanf("%d",&exp);
if(exp>=0)
{
      for(;exp!=0;exp--)
        result=result*base;
}
else
{
     for(;exp!=0;exp++)
        result=result*1/base;
}
printf("The result is:%.2f",result);
return 0;
}
```

program to find e^x using the series

```
#include<stdio.h>
#include<math.h>
int main(void){
          int i,j;
          float t,x,sum=1,expo;
          long f;
          printf("Enter 'x' for e power x:");
          scanf("%f",&x);
          /*            using library function                    */
                  expo=exp(x);
          printf("The value of e^x is %.5f",expo);
          /*            using exponential series          */
          t=x;i=12;
          do{
      for(j=f=1;j<=i;j++)
        f=f*j;
                  t=pow(x,i)/(float)f;
                  sum+=t;
                  i--;
          }while(i>=1);
          printf("\n Value of e^x = %.6f",sum);
return 0;
}
```

We have two functions y1=3.x+3,y2=5.x/2+4 Plot the graph for the two curve when x varies between 0-5

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

```
float i,y1,y2,x;
printf("'0','*','#' are the  co-ordinates for y1,y2 & for intersection of y1&y2 respectively\n");
for(x=0;x<=5.0;x=x+0.25){
y1=x*2;
y2=x*3;
for(i=0.25;i<=y1;i+=0.25)
{
printf(" ");
}
if(y1!=y2)
printf("0");
else
printf("#");
for(i=0.25;i<=y2-y1;i+=0.25)
{
printf(" ");
}
if(y1!=y2)
printf("*");
printf("\n");
}
return 0;
}
```

<u>Program to find all the Pythagorean triplets a,b,c which satisfies the condition (a*a)+(b*b)=(c*c)</u>

```
# include <stdio.h>
# include <conio.h>
int main(void)
{
        int a,b,c,d;
        clrscr();
        for(a=1;a<50;a++)
        {
              for(b=1;b<50;b++)
              {
                     d=(a*a)+(b*b);
                     for(c=1;c<50;c++)
                     {
                            if(d==(c*c))
                            printf("%d,%d,%d \t",a,b,c);
                     }
              }
        }
return 0;
}
```

Chapter - 6
Arrays

Introduction

In fourth lesson we discuss how to store the data in a variable. One integer variable can store one value at a time. I want to store more than one data with single variable which is impossible. So I need a special type data type which is called array. That Array can store a similar data type which continuous memory allocation. It also called a derived data type because array is neither primitive nor user defined but derived from either int, char, float .. etc..
You can create arrays of any data type. You can define an integer array, a floating-point array, and a double floating-point array. Any time you need lists of values, it's a lot easier to define the list as an array instead of as differently named variables. You can step through all the elements of an array with a for loop, and you'll often need to process and print elements throughout an array.

Declaration Syntax

Syntax:

data-Type array_Name[size-of-array];

Datatype:

data-Type refers to what type of data the array can contains(i.e. int, floats, double, chars, etc.)

array_Name:
array-Name same as the identifier name. It should be a valid identifier.

Size of array;
size-of-array means how many elements array can contain, which must be a constant..

A simple array of 5 int would look like:

int   roll[5];

This would effectively make an area in memory (if availble) for roll, which is 5 * sizeof(int). Basically sizeof() returns the size of what is being passed. On a typical 16-bit machine, sizeof(int) returns 2 bytes, so we will get 10 bytes of memory for our array. But in case of 32-bit machine, sizeof(int) returns 4 bytes, so we will get 20 bytes of memory for our array.

How do we reference areas of memory within the array? By using the [ ] we can effectively "dereference" those areas of the array to return values.

```
printf("%d",roll[3]);
```

This would print the fourth index element from the array on the screen. Why the fourth? This is because array elements are numbered from **0 as index** value.

Note: You **cannot** initialize an array **using a variable**. ANSI C does not allow this. For example:

```
int x = 5;
int   roll[x];
```

This above example is **illegal** in Turbo C or Borland C ,ANSI C restricts the array initialization size to be constant. So is this **legal** in Dev CPP, LINUX, UNIX?

```
int ia[];
```

Because the size of array is not known at compile time. So it is **illegal.**

How can we get around this? By using macros(read later) we can also make our program more readable!

```
#define SIZE  5
            int roll[SIZE];
```

Now if we wanted to change the array size, all we'd have to do is change the define statement!

But what if we don't know the size of our array at compile time? That's why we have Dynamic Memory Allocation. We will read later chapter
Here I am going to show a character array named cArray with five values. Each individual value in the array is an *element*. The subscripts range from 0 to 4 for a five-character array. No matter what kind of array you define, each value in the array is called an *element*, and the number that *references* each element is also a ***subscript.***

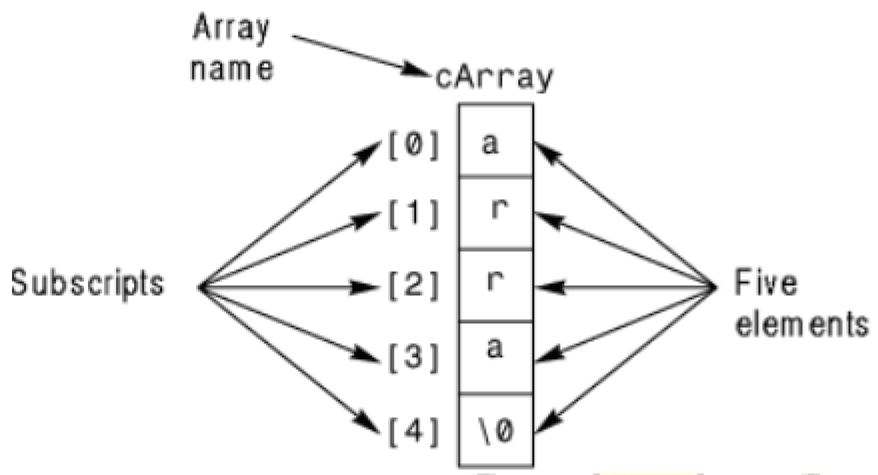


Figure.A five-element character array holding a string.

When a character array holds a string value, treating the array as a single but aggregate string value with the individual elements holding individual characters makes a lot of sense. (Other programming languages store strings in string variables.)
In reality, you work with lists of numbers all the time. When your program must process several numeric values, array storage is perfect for holding and stepping through those values. Although you don't refer to the array aggregately in the same way that you do a character array holding a string, numeric arrays are extremely important in computing.
To help with your understanding of numeric arrays, let's review the basics of character arrays. The array in ***Figure*** might be initialized like this:

```
char cArray[5] = "arra";
```

C's syntax for initializing strings in character arrays, as shown here, doesn't apply to numeric arrays. There's no need to worry about a null zero at the end of numeric arrays because only strings need null zero values at their termination. If you define and initialize a numeric array at the same time, you must initialize the array's individual elements one at a time. Use braces to initialize individual elements in the array.
Here's how you would initialize cArray one element at a time:

```
char cArray[5] = {'a', 'r', 'r', 'a', '\0'};
```

This definition is identical to the previous one, except that it's a little more tedious for you to type. Nevertheless, this character-by-character initialization will prepare you for the syntax required for all numeric array initialization. C ***automatically*** adds the terminator zero to the end of the string when you assign an initial string value.
When you define and initialize an array at the same time, you don't need the initial subscript. The following definition is identical to the preceding one:

char **cArray[]** = {'a', 'r', 'r', 'a', '\0'};

When you reserve five array elements, don't try to store six or more elements in the array! If you do, you will overwrite memory that might be holding other kinds of variables. If you need to define a large array, you can always define the array with extra elements. The following array definition holds the four-character string “arra” with a terminator, but it also has extra reserved room in case you need to store a longer string in the array later:

```
char cArray[35] = {'a', 'r', 'r', 'a', '\0'};
```

Character arrays might hold individual characters and not strings. If that's the case, **don't** worry about the terminator. It exists only to terminate strings.

Unlike most programming languages, C doesn't complain if you store a value in an undefined array element. For example, the statement

```
cArray[3200] = 'X';
```

puts an X approximately 3,200 memory locations after the start of the 35-character array named cArray. Who knows what the X will overwrite? Whatever happens, you'll realize something has gone wrong when your computer freezes or reboots because the X overwrote an important area of memory or changed **a critical internal** program value!

How to initialize a numeric array?

```
int values[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

Despite the fact that a 0 is in value's last element, that 0 is considered an integer zero, because there is no string in the array. In other words, it is considered a valid part of the array data.

If you don't initialize an array when you define it, you must fill the array one value at a time. For example, this is not possible in the body of the program:

```
values[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

And neither is this:

```
values = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

In the body of the program, you must assign each value to each array element one at a time. Later in this unit, you will learn some techniques that most programs use to initialize arrays.

The following two array definitions reserve two numeric arrays. The first array is a double array and the second is a long int array:

```
double factors[25];      // Defines an array of 25 doubles
long int weights[100];   // Defines an array of 100 long integers
```

At the time you define these arrays, you could initialize them with a few values, like this:

```
double factors[25] = {4244.56, 78409.21, 2930.55433, 569401.34};
long int weights[100] = {43567, 70935, 32945, 102, 49059};
```

If you define an array of a specific size but initialize fewer elements as shown here, C fills **zeros** every remaining element in the array. In other words, the first five elements of weights contain the values inside the braces (from 43567 to 49059), and the **remaining elements hold zero.**

If you don't initialize any elements, don't rely on C to initialize them for you! C doesn't automatically initialize values to zeros given this definition, all blocks are filled with **garbage** value.:

```
int a[100];   // Defines but doesn't initialize "a"
```

If you want to initialize an array to all zeros, initialize it with a single zero:

```
int vals[100] = {0};   // The entire array now holds 0
```

we can initialize integer array with double value, but the decimal part of the double number will vanish. Only integer part will store.

```
int a[]={2.3, 4, 5.23, 67.453, 8.08, 12, 87.98};
```

It will store {2, 4, 5, 67, 8, 12, 87} in a[] only not the whole number.
But while doing program compiler will give some warning message as “initialization to int from double”. But practically no use from examination point of view you have to know it.

```
int a[5];
a[2]=34;
a[1]=45;
```

rest of the blocks(i.e a[0],a[3] and a[4]) are garbage value

```
int a[5];
a[5]={2,1,3,4,5);   //error
```

Printing the value present in a given array

We can access the array elements by there subscript value. If I want to print, what value present at 5th block. then we have to write printf(“%d”,a[4]), here 4 because array index start with 0. let a[9] array contain some elements as

Pch6_1.c

```
/*let a[9] array contain some elements as*/
#include<stdio.h>


int main(void)
{

        int a[9]={2,12,3,14,7,23,34};
        printf("value present at a[0]:%d\n",a[0]);
        printf("value present at a[2]: %d\n",a[2]);
        printf("value present at a[5]: %d\n",a[5]);
        printf("value present at a[8]: %d\n",a[8]);
        printf("value present at a[9]: %d\n",a[9]);
        printf("value present at a[12]: %d\n",a[12]);
        printf("value present at a[202]: %d\n",a[202]);
        printf("value present at a[-1]: %d\n",a[-1]);
        printf("value present at a[-0]: %d\n",a[-0]);
18:             printf("value present at a[-2]: %d\n",a[-2]);
19:             printf("value present at a[-342]: %d\n",a[-342]);
20:
21:  return 0;
22: }
```

out put:

```
value present at a[0]:2
value present at a[2]:3
value present at a[5]:23
value present at a[8]:0
value present at a[9]:8
value present at a[12]:2088809675
value present at a[202]:0
value present at a[-1]:0
value present at a[-0]:2
value present at a[-2]:2009252814
value present at a[-342]:2089932117
```

Analysis:

In the above Pch6_1.c used to print the content of an array. I need to print the content of a block. So I have to pass the subscript value with array name.

In line 8 I declared an array of integer type with size 9 and initialized the array with seven elements. So first 7 blocks are filled with initialize value from a[0] to a[6]. But the block a[7] and a[8] are initialize with 0.

From line 9-12 the content of the array will print. Form line 13-15, I want print the value from a[9], a[12] and a[202]. There are no such blocks, but there will no error in the program. What is happening, they are giving me as garbage value. The task is given to user, find it?. Same form lines 16-20, I want to print the value present at a[-0], a[-1], a[-2] and a[-342]. I am passing the negative subscript value which will never possible, but I am getting some out put. The task also given to user. Please find is!

Pch6_1.c is used to print the all elements of the array using loop.

```
1: /*let a[9] array contain some elements as*/
2: #include<stdio.h>
3:
4:
5: int main(void)
6:  {
7:         int i;
8:          int a[9]={2,12,3,14,7,23,34};
9:          printf("Printing the values of the array by using loop:");
10:         for(i=0;i<9;i++)
```

```
        printf("value present at a[%d]:%d",i,a[i]);

        return 0;
}
```

Out Put:

```
Printing the values of the array by using loop:
value present at a[0]:2
value present at a[1]:12
value present at a[2]:3
value present at a[3]:14
value present at a[4]:7
value present at a[5]:23
value present at a[6]:34
value present at a[7]:0
value present at a[8]:0
```

Example to find the biggest and smallest element present in array

```
#include<stdio.h>
int main(void){
     int a[50],i,len,big,small;
     printf("ENter the length of the array:");
     scanf("%d",&len);
     printf("\nEnter %d elements ",len);
       for(i=0;i<len;i++)
         scanf("%d",&a[i]);
      /*finding sall and big element*/
      big=small=a[0];//assume a[0] element be small and big initially
        for(i=1;i<len;i++)
        {
          if(a[i]>big)
             big=a[i];
          if(a[i]<small)
            small=a[i];
        }
            printf("\nThe big element is %d",big);
            printf("\nThe small element is %d",small);


        return 0;
          }
```

Pch6_5.c

```
#include<stdio.h>

int sequntialSearch(int a[], int length, int value);
int main(void)
{
        int match;
        int userKey;
        int a[10]={3, 2, 5, 4, 7, 8, 6, 1, 0, 9};
          printf("What is your customer code? ");
          scanf("%d",&userKey);
          match = sequntialSearch(a, 10, userKey);/*length of the array is 10*/
          if (match!=-1)
          printf("\nYou are already in the list with %d Index",match);
          else
          printf( "You are not in the list");

   return 0;
}
          /*function definition*/
  int sequntialSearch(int a[], int length, int value)
    { int i;
              for(i=0; i< length; i++)
              {
                    if(a[i]==value)
                      return i; /*value present at ith index*/
              }
      return -1;/*value is not present in the array*/
  }

```

Pch6_6.c

```
#include<stdio.h>

int binarySearch(int a[], int findex, int lindex, int value);

int main(void)
{
        int match;
        int userKey;
          int a[10]={3, 5, 7, 8, 10, 12, 15, 23, 30, 39};
          printf("What is your customer code? ");
          scanf("%d",&userKey);
          match = binarySearch(a, 0, 9, userKey);/*length of the array is 10*/
          if (match!=-1)
          printf("\nYou are already in the list with %d Index",match);
```

```
            else
            printf("You are not in the list");

            return 0;
}
 /*function defination*/
                int binarySearch(int a[], int findex, int lindex, int value) /*a=sorted array
        findex=first index of the array
        lindex=last index of the array
        value= value to be search*/
{
   int mid;
   mid=(findex+lindex)/2;
            if (findex>lindex)
    return -1; /*function termination and value is not present*/
            else
              if(a[mid]==value)
                return mid; /*value present at mid index position*/
              else
                if(a[mid]>value)
                        /*value must be present left to mid position*/
                  return binarySearch(a, findex, mid-1,value);
                else
                        /*value must be present right to mid postion*/
        return binarySearch(a, mid+1, lindex,value);
   }


```

Q&A

Q. What happens if I write to element 25 in a 24-member array?
A. You will write to other memory, with potentially disastrous effects on your program.
Q. What is in an uninitialized array element?
A. Whatever happens to be in memory at a given time. The results of using this member without assigning a value are unpredictable.
Q. Can I combine arrays?
A. Yes. With simple arrays you can use pointers to combine them into a new, larger array. With strings you can use some of the built-in functions, such as strcat, to combine strings.
Q. Why should I create a linked list if an array will work?
A. An array must have a fixed size, whereas a linked list can be sized dynamically at runtime.
Q. Why would I ever use built-in arrays if I can make a better array class?
A. Built-in arrays are quick and easy to use.
Q. Must a string class use a char * to hold the contents of the string?
A. No. It can use any memory storage the designer thinks is best.

Chapter -7
Functions

7.1. Introduction

Doing a big work, for a person is difficult to complete. If we divide that work into different parts distribute among different persons and then combine them at the is easier as compare to the bigger one. Dividing into smaller part is called *module*. Now that we have a understanding of the very basics of C, it is time now to turn our focus over to making our programs not only run correctly but more efficiently and are more understandable.
Although object-oriented programming has shifted attention from functions and toward objects, functions nonetheless remain a central component of any program. In this chapter you will learn

- What a function is and what its parts are.
- Why we need functions.
- How to declare and define functions.
- How to pass parameters into functions.
- Call by value and Call by reference.
- How to return a value from a function.
- Recursion

7.2. What Is a Function?

A function is, in effect, a subprogram or a subroutine that can act on data and return a value. Every C program has at least one function, main(void). When your program starts, main(void) is called automatically. main(void) might call other functions, some of which might call still others. Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function. When the function returns, execution resumes on the next line of the calling function. This flow is illustrated in **Figure 7.1**.

When a program calls a function, execution switches to the function and then resumes at the line after the function call. Well-designed functions perform a specific and easily understood task. Complicated tasks should be broken down into multiple functions, and then each can be called in turn.
Functions come in two varieties: **user-defined** and **built-in**. **Built-in** functions are part of your compiler package (i.e. Header files)--they are supplied by the manufacturer for your use.

Why need a Function

Why should we make functions in our programs when we can just do it all under main? Has a very good analogy that I'll borrow: Think for a minute about high-end stereo systems. These stereo systems do not come in an all-in-one package, but rather come in separate components: *pre-amplifier, amplifier, equalizer, receiver, cd player, tape deck*, and *speakers*. The same concept applies to programming. Your programs become **modularized** and much more readable if they are broken down into components.

This type of programming is known as **top-down** programming, because we first analyze what needs to be broken down into components. Functions allow us to create top-down modular programs.

Declaring and Defining Functions

Using functions in your program requires that you first declare the function and that you then define the function. The declaration tells the compiler the *name*, *return type*, and *parameters* of the function. The definition tells the compiler *how* the function works. No function can be called from any other function that hasn't first been declared. The declaration of a function is called its **prototype**.

Declaring the Function

There are three ways to declare a function:

- Write your prototype into a file, and then use the **#include** directive to include it in your program.
- Write the prototype into the file in which your function is used.
- Define the function before it is called by any other function. When you do this, the definition acts as its own declaration.

Although you can define the function before using it, and thus avoid the necessity of creating a function prototype, this is not good programming practice for three reasons.
First, it is a bad idea to require that functions appear in a file in a particular order. Doing so makes it hard to maintain the program as requirements change.
Second, it is possible that function A() needs to be able to call function B(), but function B() also needs to be able to call function A() under some circumstances. It is not possible to define function A() before you define function B() and also to define function B() before you define function A(), so at least one of them **must** be declared in any case.

```
B( ); / * declaration of function B( ) must needed because function B( ) is used by function A( )*/
A( )
{
//Body
::::::::::
B( );
return;
}

B( )
{
// body;
::::::::::
return;
}
```

Third, function prototypes are a good and powerful debugging technique. If your prototype declares that your function takes a particular set of parameters, or that it returns a particular type of value, and then your function does not match the prototype, the compiler can flag your error instead of waiting for it to show itself when you run the program.

Function Prototypes

Many of the built-in functions you use will have their function prototypes already written in the

files you include in your program by using #include. For functions you write yourself, you must include the prototype.
The function prototype is a statement, which means it ends with a *semicolon*. It consists of the function's *return type*, *name*, and *parameter list*.
A function declaration and the definition and use of that function.

```
/*demonstrates the use of function prototypes */
#include <stdio.h>


long areaOfRectangle(int length, int width); //function prototype

int main(void)
{
  int length;
  int width;
  int area;

  printf( “Enter the length in centimeter:”);
  scanf(“%d”,&length);
  printf("Enter the width in centimeter:");
  scanf(“%d”,&width);

  area= areaOfRectangle(length,width);//function calling

  printf("The area of the rectangle is: %d\n",area);


 return 0;
 }

 int areaOfRectangle(int len, int wid)//function definition
 {
     return len * wid;
}
```

Defining the Function

The definition of a function consists of the function header and its body. The header is exactly like the function prototype, except that the parameters must be named, and there is **no** terminating semicolon.
The body of the function is a set of statements enclosed in braces. Here is the header and body of a function.

```
returnType functionName(Parameter1, Parameter2……..ParemeterN)
{
:::::::::::::::::::
// body of the function
:::::::::::::::::::
```

```
return  value;
}
```

In our example

```

int areaOfRectangle(int len, int wid)
{
     return (len * wid);
}
```

Function Definition Examples

```
int areaOfRectangle(int len, int wid)
  {
     return (len * wid);
}

void checkEvenOdd(int number)
{
    if (number%2 == 0)
        printf("The number is Even”) ;
    if (number%2 == 1)
        printf("The number is ODD”);

}

Void display()
  {
            printf(“I am Alice San”);
            printf(“I love C”);
  }

addTwoNumber(int a, int b)
{
    return a+b;
}
```

Execution of Functions (function calling)

When you call a function, execution begins with the first statement after the opening brace ({). Means the execution of the program will shift to the function definition because the actual work written in the function definition, at tt time of function calling we will pass the actual value of the function.

Syntax:

```
FunctionName(actual-value1, actual-value2 ……. Actual-valueN);
```

Example:

```
int p= areaOfRectangle(34,45);
checkEvenOdd(34);
display();
```

int p = addTwoNumber(34,23);
Functions can also call other functions and can even call themselves (see the section "Recursion," later in this chapter).

Local Variables

Not only can you pass in variables to the function, but you also can declare variables within the body of the function. This is done using local variables, so named because they exist only locally within the function itself. When the function returns, the local variables are no longer available.
Local variables are defined like any other variables. The parameters passed in to the function are also considered local variables and can be used exactly as if they had been defined within the body of the function. Pch7_2.C is an example of using parameters and locally defined variables within a function.

The use of local variables and parameters.

```
#include <stdio.h>

float Convert(float);
int main(void)
{
  float Fahrenheit;
  float Celsius;

  printf( "Please enter the temperature in Fahrenheit: ");
   scanf("%f",&Fahrenheit);
   Celsius = Convert(Fahrenheit);
   printf("Here's the temperature in Celsius:%f ", Celsius);

        return 0;
 }

 float Convert(float Fahrenheit)
 {
   float Celsius;
   Celsius = ((Fahrenheit - 32) * 5) / 9;
   return Celsius;
}
```

Pch7_4.C. Demonstrating global and local variables.

```
#include <stdio.h>

void abc();          // function declaration
int x = 5, y = 7;          // global variables
int main(void)
{

     printf( "x in side main: %d\n",x);
     printf("y in side main: %d\n",y);
      abc();    /*function calling*/
      printf("Back from abc() function !");
      printf("value of x after function calling: %d",x);
      printf(" value of y after function calling:%d ",y);
      return 0;
 }

 void abc()
 {
      int y = 10;

      printf( "x in side the abc function: %d",x);
      printf( "y in side the abc function: %d",y);
}
```

Output:

```
x in side main: 5
y in side main: 7
x in side the abc function: 5
y in side the abc function: 10
Back from abc() function !
value of x after function calling: 5
value of y after function calling: 7
```

Function Statements

There is virtually no limit to the number or types of statements that can be in a function body. Although you can't define another function from within a function, you can call a function, and of course main(void) does just that in nearly every C program. Functions can even call themselves, which is discussed soon, in the section on recursion.
Although there is no limit to the size of a function in C, well-designed functions tend to be **small**. Many programmers advise keeping your functions short enough to fit on a single screen so that you can see the entire function at one time. This is a rule of thumb, often broken by very good programmers, but a smaller function is easier to understand and maintain.
Each function should carry out a single, easily understood task. If your functions start getting large, look for places where you can divide them into component tasks.

Function Arguments

Function arguments do not have to all be of the same type. It is perfectly reasonable to write a function that takes an integer, two longs, and a character as its arguments.
Any valid C expression can be a function argument, including constants, mathematical and logical expressions, and other functions that return a value.

Using Functions as Parameters to Functions

Although it is legal for one function to take as a parameter a second function that returns a value, it can make for code that is hard to read and hard to debug.
As an example, say you have the functions double(), triple(), square(), and cube(), each of which returns a value. You could write

```
Answer = (triple(square(cube(myValue))));
```

This statement takes a variable, myValue, and passes it as an argument to the function cube(), whose return value is passed as an argument to the function square(), whose return value is in turn passed to triple().The return value of this tripled, squared, and cubed number is now passed to Answer.
It is difficult to be certain what this code does (was the value tripled before or after it was squared?), and if the answer is wrong it will be hard to figure out which function failed.
An alternative is to assign each step to its own intermediate variable:

```
long myValue = 2;
long cubed  =   cube(myValue);       // cubed = 8
long squared = square(cubed);        // squared = 64
long tripled = triple(squared);      // tripled = 196
```

Now each intermediate result can be examined, and the order of execution is explicit.

Parameters Are Local Variables

The arguments passed in to the function are local to the function. Changes made to the arguments **do not** affect the values in the calling function. This is known as ***passing by value***, which means a local copy of each argument is made in the function. These local copies are treated just like any other local variables. Pch7_5.C illustrates this point.

A demonstration of passing by value.

```
/* demonstrates passing by value*/
 #include <stdio.h>


 void swap(int x, int y);/* function declaration*/

 int main(void)
 {
   int x = 5, y = 10;

     printf("Main. Before swap, x:%d  y: %d\n",x,y) ;
12:       swap(x,y);
13:       printf("Main. After swap, x:%d  y: %d\n",x,y) ;
14:          return 0;
15:     }
16:
17:     void swap (int x, int y)
18:     {
19:       int temp;
20:
21:       printf("Swap. Before swap, x:%d  y: %d\n",x,y) ;
22:
23:       temp = x;
24:       x = y;
25:       y = temp;
26:
27:       printf("Swap. After swap, x:%d  y: %d\n",x,y) ;
28:
29: }
```

```
Output: Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

Analysis: This program initializes two variables in main(void) and then passes them to the swap() function, which appears to swap them. When they are examined again in main(void), however, they are unchanged! The variables are initialized on line 9, and their values are displayed on line 11. swap() is called, and the variables are passed in. Execution of the program switches to the swap() function, where on line 21 the values are printed again. They are in the same order as they were in main(void), as expected. On lines 23 to 25 the values are swapped, and this action is confirmed by the printout on line 27. Indeed, while in the swap() function, the values are swapped. Execution then returns to line 13, back in main(void), where the values are no longer swapped.

As you've figured out, the values passed in to the swap() function are passed by value, meaning that copies of the values are made that are local to swap(). These local variables are swapped in lines 23 to 25, but the variables back in main(void) are unaffected.

Return Values

Functions return a value or return void. Void is a signal to the compiler that no value will be returned.

To return a value from a function, write the keyword return followed by the value you want to return. The value might itself be an expression that returns a value. For example:

```
return 5;
return (x > 5);
return (Abc());
```

Pch7_6.C. A demonstration of multiple return statements.

```

#include <stdio.h>

int abc(int x)/*function declaration */

 int main(void)
   {
    int  value, rvalue;
    printf(“ Enter an integer value:”);
                 scanf(“%d”,&value);
     rvalue=abc(value);
  printf(“The Returned value of the abc() function is:%d”,rvalue);
    return 0;
 }
 /*function definition */

     int abc(int n)
       {
      if(n<0)
         return -1;
      else
         if(n>0)
           return 1;
         else
           return 0;

   }
```

Recursion

A function that calls itself (is an important feature of C and C) such functions are called recursive functions and recursion can be **direct** or **indirect**. It is direct when a function calls itself; it is indirect recursion when a function calls another function that then calls the first function.
The term “**recursion**” was derived from the Latin word **recursus**, which means, “**to run back**”. “**Recursive**” thinking may be tough for beginners. I am going to present some interesting recursive programs

Factorial of number.

This is the most famous program on recursion. Many versions of this program are available. All programs differ only in checking conditions. I prefer to write like the following one.

```
long Factorial( int n )
{
if ( n>0 )
return( n * Factorial(n-1) );
else
return( 1 ); }
```

Fibonacci series

The following program returns the nth Fibonacci number. Fibonacci series is:
1, 1, 2, 3, 5, 8, 13, 21…

```
int Fibonacci( int n )
{
if ( n==1 || n==2 )
return( 1 );
else
return( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

GCD of two numbers

Here is the program to find the Greatest Common Divisors (GCD or HCF) of two numbers a & b.

```
int GCD( int a, int b )
{
if (a%b == 0 )
  return( b );
else
  return( GCD( b, a%b ) );
}
```

Power of number

I haven’t yet come across user defined power function, which could handle negative n (say, 4.5-5). Here is the program I tried…it could handle negative n too!

```
double Power( double x, int n)
{
if ( n==0 )
return( 1 );
else if ( n>0 )
```

```
    return( x * Power( x, n-1 ) );
    else
    return( (1/x) * Power( x, n+1 ) );
}
```

Reverse Printing a string.

This is a wonderful program to understand the behavior of recursion.

```
    void ReverseChar( void )
    {
    char ch;
    if ( (ch=getchar( ))!='\n' )
    ReverseChar( );
    putchar( ch );
    }
```

Decimal to Binary Conversion

The following recursive function gets decimal value as input and prints binary value. It prints each bit value (0 or 1) one by one.

```
    void ToBin( int n )
    {
    if (n>1)
    ToBin( n/2 );
    printf( << n%2;
    }
```

Decimal to Hexadecimal Conversion

```
    void ToHex( int n )
    {
    char *htab[ ] = { "0", "1", "2", "3", "4", "5", "6", "7", "8","9", "A", "B", "C", "D", "E", "F" };
    if (n>15)
    ToHex( n/16 );
    printf( << htab[n%16] ;
    }
```

Chapter – 8
Structure and Union

Introduction

The array is an example of a data structure. It takes simple data types like int, char or double and organizes them into a linear array of elements. The array serves most but not all of the needs of the typical C program. The restriction is that an array is composed of elements all of the *same type*. At first this seems perfectly reasonable. After all why would you want an array to be composed of twenty chars and two integers? Well this sort of mixture of data types working together is one of the most familiar of data structures. Consider for a moment a record card which records name, age and salary. The name would have to be stored as a string, i.e. an array of chars terminated with an ASCII null character, and the age and salary could be integer.

At the moment the only way we can work with this collection of data is as separate variables. This isn't as convenient as a single data structure using a single name and so the C language provides *struct* (it is a key word). At first it is easier to think of this as a record - although it's a little more flexible than this suggests.

Structure

A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures are declared in C using the following syntax:

Syntax:

```
        struct  structure_name
           {
        member_type1 member_name1;
                member_type2 member_name2;
                member_type3 member_name3;
                ……………………………..
                …………………………….
           } object_names;
```

struct is a key word used to declare a structure data type, which is a user defined data type. where structure_name is a name for the structure type, object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name. The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure_name is created and can be used in the rest of the program as if it was any other type.

Example:

```
        struct product
          {
                int weight;
                 float price;
        } ;
          struct product apple;
          struct   product banana, melon;
```

//example about structures

```
#include<stdio.h>
int main(void)
{
struct student
    {
            int roll;
            char gender;
            char name[20];
     };
struct student s1={163,’M’,”Alice”};
printf(‘\n The value initialized are :”);
printf(“\n The roll of s1 is %d”,s1.roll);
printf(“\n the gender is  %c”,s1.sec);
printf(“\n The name of the s1 is  %s”,s1.name);
return 0;
}
```

Structures, or structs, are very useful in creating data structures larger and more complex than the ones we have discussed so far. We will take a cursory look at some more complex ones in the pointer chapter.

Array of structure

Structure can also store more than one data members of a structure variable. For that we have array of structure, which is given below for array of student.

```
struct student
   {
         int roll;
         char gender;
         char name[20];
    };
struct student info[100];
```

Here it can store 100 students information. The initialization of structure array is given below.

```
struct student info[2]={{163,’M’,”Alice”},{164,’F’,”Tom”}};
```

Array within structure

As we know array of structure. Now can we declar array within a structure. Yes we can. Here is the format given below:

```
struct student
   {
```

```
                int roll[10];
                char gender;
                char name[20];
        };
```

In this structure we have name it is a array of 20 characters, means a name of a student can be up to 20 characters.

Structures and Functions:

For example, C has no facilities for manipulating complex numbers but this is easy enough to put right using a struct and a few functions. A complex number is composed of two parts – a real and imaginary part - which can be implemented as single or double precision values. This suggests defining a new struct type:

```
struct complex
{
  float real;
  float imag;
};
```

After this you can declare new complex variables using something like:

```
struct complex a,b;
```

The new complex variables cannot be used as if they were simple variables because they are not. Most versions, of the C language do allow you to assign structures so you could write:

```
a=b;
```

as shorthand for

```
a.real=b.real;
a.imag=b.imag;
```

Being able to assign structures is even more useful when they are bigger. However you can't expect C to sort out what you mean by c = a + b - for this you have to write out the rule for

addition as:

```
c.real=a.real+b.real;
c.imag=a.imag+b.imag;
```

Of course a sensible alternative to writing out the addition each time is to define a function to do the same job - but this raises the question of passing structures as parameters. Fortunately this isn't a big problem. Most C compilers, will allow you to pass entire structures as parameters and return entire structures. As with all C parameters structures are passed by value and so if you want to allow a function to alter a parameter you have to remember to pass a pointer to a struct. Given that you can pass and return structs the function is fairly easy:

```
struct comp add(struct comp a , struct comp b)
{
struct comp c;
c.real=a.real+b.real;
c.imag=a.imag+ b.imag;
return c;
}
```

After you have defined the add function you can write a complex addition as:

```
x=add(y,z)
```

which isn't too far from the x=y+z that you would really like to use. Finally notice that passing a struct by value might use up rather a lot of memory as a complete copy of the structure is made for the function.

Nested structure

Structure within structure is called nested structure. As we learn nested in nested-if . so a structure can present within a structure.

```
struct teacher
   {
    int tid;
    char tgender;
    char tname[20];
     struct student{
            int sid;
            char sgender;
            char sname[20];
            }s1;

}t1;
```

OR

```
struct student{
            int sid;
            char sgender;
            char sname[20];
         };

struct teacher
```

```
    {
     int tid;
     char tgender;
     char   tname [20];
     struct student s1;
    }
  struct teacher t1;
```

Here I an declaring a structre of teacher and within a teacher I am declaring student structure., means a teacher must the information of a student.

The teacher wants to access the roll of student s1, so the format is

```
  t1.s1.sid;l,
so  the prints format is
printf(" the roll is %d",t1.s1.sid);
```

In this way structure can be done.

Sizeof of a structure object or structure

sizeof a structure depends on the total sum of size of variables present inside it.

In case sizeof(s1) = 4 or 2 + 1 + 20 = 25 or 23 (depending on the compiler)
In case sizeof(t1) = 4 or 2 + 1 + 20 + 25 or 23 = 50 or 46

Union

Unions are declared in the same fashion as structs, but have a fundamental difference. Only one item within the union can be used at any time, because the memory allocated for each item inside the union is in a shared memory location. Why you ask? An example first:

```
union student
   {
          int roll[10];
          char gender;
          char name[20];
    };
```

The size of union will be 20 byte, but in case of structure the size of structure will be sum of all data members. So the size will be 41 bytes.

We can also declare union within structure as follows:

```
  struct conditions
{
          float temp;
                 union feels_like
```

```
        {
                  float wind_chill;
                  float heat_index;
                 };
          } today;
```

As you know, wind_chill is only calculated when it is "cold" and heat_index when it is "hot". There is no need for both. So when you specify the temp in today, feels_like only has one value, either a float for wind_chill or a float for heat_index.

Types inside of unions are unrestricted; you can even use structs within unions.

Sizeof of a union object or union

sizeof a union depends on the maximum size of variables present inside it.

```
union student{
        int sid;
        char sgender;
        char sname[20];
      };

union teacher
          {
           int tid;
           char tgender;
           char   tname [20];
           union student s1;
          }
        union teacher t1;
```

In case sizeof(s1) = 20 bytes
In case sizeof(t1) = 25 or 23 (because the size of s1 is high)

Chapter - 9
Pointer

Introduction

There's a lot of nice, tidy code you can write without knowing about pointers. But once you learn to use the power of pointers, you can never go back. There are too many things that can only be done with pointers. But with increased power comes increased responsibility. Pointers allow new and more ugly types of bugs, and pointer bugs can crash in random ways which makes them more difficult to debug. Though even with their problems, pointers are an irresistibly powerful programming construct.

Definition of pointer

To understand pointers, you must know a little about computer memory. Computer memory is divided into **sequentially** numbered memory locations. Each variable is located at a unique location in memory, known as its address. .
A *pointer* is a variable that holds a memory address of another variable or constant of **same type**.
Pointers solve two common software problems. *First*, pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better. *Second*, pointers enable complex "linked" data structures like linked lists and binary trees.
This unit takes you into the world of advanced C programming by teaching you about pointer variables. Be warned: At first, pointer variables seem to add more work without offering any advantages. Be patient! Before this unit ends, you'll see how pointers let you write programs that manipulate string data in arrays. Until you learn about pointers, you can't keep track of string data using an array.
Different computers number this memory using different, complex schemes. Usually programmers don't need to know the particular address of any given variable, because the compiler handles the details.
The *unary* or *monadic* operator & gives the ``address of a variable''.
The *indirection* or dereference operator * gives the ``contents of an object *pointed to* by a pointer''.
To declare a pointer to a variable do:
data-Type * pointerVaribleName;

```
  int*pointer;
```

NOTE: We must associate a pointer to a particular type: You can't assign the address of a int to a long, for instance.
Consider the effect of the following code:

```
  int x = 1, y = 2;
  int *ip;
  ip = &x;
  y = *ip;
  x = ip;
  *ip = 3;
```

Assume the address of x is 2000, address of y is 3000 and the address of ip is 4000.
The assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to

be a *pointer to an integer* and is assigned to the address of x (&x). So ip gets loaded with the value 2000.
Next y gets assigned to the *contents of* ip. In this example ip currently *points* to memory location 2000, the location of x. So y gets assigned to the values of x, which is 1.
We have already seen that C is not too fussy about assigning values of different type. Thus it is perfectly legal (although not all that common) to assign the current value of ip to x. The value of ip at this instant is 2000.
Finally we can assign a value to the contents of a pointer (*ip).
NOTE: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it. Other wise memory will crash.
So ...

```
  int *ip;
*ip = 2000;
```

will generate an error (program crash!!).
The correct use is:

```
  int *ip;
  int x;
 ip = &x;
*ip = 2000;
```

Call by Value vs. Call by Reference

C passes parameters "by value" which means that the actual parameter values are copied into local storage. The caller and callee functions do not share any memory -- they each have their own copy. This scheme is fine for many purposes, but it has two disadvantages.

Because the callee has its own copy, modifications to that memory are not communicated back to the caller. Therefore, value parameters do not allow the callee to communicate back to the caller. The function's return value can communicate some information back to the caller, but not all problems can be solved with the single return value.

Sometimes it is undesirable to copy the value from the caller to the callee because the value is large and so copying it is expensive, or because at a conceptual level copying the value is undesirable.

The alternative is to pass the arguments "by reference". Instead of passing a copy of a value from the caller to the callee, pass a pointer to the value. In this way there is only one copy of the value at any time, and the caller and callee both access that one value through pointers.

Some languages support reference parameters automatically. C does not do this – the programmer must implement reference parameters manually using the existing pointer

The classic example of wanting to modify the caller's memory is a swap() function which exchanges two values. Because C uses call by value, the following version of Swap will not work...

```
void Swap(int x, int y)
{
	int temp;
	temp = x;
	x = y; // these operations just change the local x,y,temp
	y = temp; // nothing connects them back to the caller's a,b
}
	// Some caller code which calls Swap()...
int a = 1;		int b = 2;
Swap(a, b);
```

Swap() does not affect the arguments a and b in the caller. The function above only operates on the copies of a and b local to Swap() itself. This is a good example of how "local" memory such as (x, y, temp) behaves -- it exists independent of everything else only while its owning function is running. When the owning function exits, its local memory disappears.

Reference Parameter Technique

To pass an object X as a reference parameter, the programmer must pass a pointer to X instead of X itself. The formal parameter will be a pointer to the value of interest. The caller will need to use & or other operators to compute the correct pointer actual parameter. The callee will need to dereference the pointer with * where appropriate to access the value of interest. Here is an example of a correct Swap() function.

```
void Swap(int* x, int* y) { // params are int* instead of int
int temp;
temp = *x; // use * to follow the pointer back to the caller's memory
*x = *y;
*y = temp; }
// Some caller code which calls Swap()...
int a = 1;		int b = 2;
Swap(&a, &b);
```

Things to notice...
The formal parameters are int* instead of int.
The caller uses & to compute pointers to its local memory (a,b).
The callee uses * to dereference the formal parameter pointers back to get the caller's memory.

Since the operator & produces the address of a variable &a is a pointer to a. In Swap() itself, the formal parameters are declared to be pointers, and the values of swap (a,b) are accessed through them. There is no special relationship between the names used for the actual and formal parameters. The function call matches up the actual and formal parameters by their order the first actual parameter is assigned to the first formal parameter, and so on. I deliberately used different names (a,b vs x,y) to emphasize that the names do not matter.

The NULL Pointer

The constant NULL is a special pointer value which encodes the idea of "points to nothing." It turns out to be convenient to have a well defined pointer value which represents the idea that a pointer does not have a pointee. It is a runtime error to dereference a NULL pointer. NULL is equal to the integer constant 0, so NULL can play the role of a boolean false

Demonstrating address of variables.

```
//Pch9_1.C
#include <stdio.h>


 int main(void)
 {

  int intVar=5;
  unsigned long  longVar=65535;
   long sVar = -65535;

   printf("intVar:%d\n",intVar);
   printf(" Address of intVar: %u",&intVar);

   printf("longVar: %ld\n",longVar);
   printf(" Address of unsigned longVar:%u\n",&longVar);

   printf("sVar:uld",sVar);
   printf(" Address of sVar:%u\n",&sVar);

 return 0;
 }
```

<u>Storing and printing data in an array of character pointers.</u>

```
// Stores and prints a list of city names

 #include <stdio.h>


int main(void)
{
int i;
 char * stationCode[5] = {"BBSR", "BAM", "CTC", "BLS", "SBC"};
/*Print the stationCode Anywhere a character array can appear, so can the elements from
the cities array of pointers*/
printf("Here are the stored stationCode:\n");
 for (i = 0; i < 5; i++)
   {
      printf(“%s\n”,stationCode[i]);
   }
printf(“\n”);
/* Change the stationCode with literals, These assignments store the  address of the string literals in the elements*/

 stationCode[0] = "Bhubaneswar";
 stationCode[1] = "Berhampur";
 stationCode[2] = "Cuttack";
 stationCode[3] = "Balasore";
 stationCode[4] = "Bangalore";
 // Print the station Code again using pointer notation
printf("\nAfter changing the pointers:\n");
 for (i = 0; i < 5; i++)
  {
        printf(“%s\n”,*(stationCode + i));
  }
 printf(“\n”);

 return 0;
}
```

Chapter - 10
Dynamic Memory Allocation

Tying up loose ends
Remember that:

- You can declare multiple pointers on a line, but you need a * for each one
- There is a difference between passing by value and passing by reference. How is this difference manifest in C?
- Pointers in C are very flexible. We can even have pointers to functions.
E.g.: int f(char); // a function that takes a char and returns an int int (*pf) (int) = &f; // a pointer to f

Types of Memory: Stack vs Heap

All local variables and parameters of a function are declared in a part of memory called the stack. Can you guess why it is called the stack?
On the other hand, global variables and dynamically allocated memory comes from a different part of memory called the heap.

Dynamic Memory Allocation
So far, we have only looked at static memory allocation. In static memory allocation, variables are allocated on the stack and they go away when the function returns.
In C these four functions are used for dynamic allocation.

Task	**Function**
Allocates memory requests size of bytes and returns a pointer to the 1st byte of allocated space	malloc
Allocates space for an array of elements initializes them to zero and returns a pointer to the memory	calloc
Frees previously allocated space	free
Modifies the size of previously allocated space.	realloc

Allocating a block of memory:
When we allocate memory dynamically, memory is allocated from the heap. In C, it is allocated using the *malloc()* function.
In Java, the garbage collector automatically gets rid of dynamically allocated memory that is no longer in use. In C **you are required to manually free dynamically allocated memory that is no longer in use.** In C, this is done using the *free()* function. If you do not free up dynamically allocated memory, you have what is called a *memory leak*, and your computer may run out of memory will running your program.

A block mf memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:
ptr=(cast-type*)malloc(byte-size);
ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with size byte-size.

Example:
x=(int*)malloc(100*sizeof(int));
On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int

For example, suppose we want to create an array to hold the grades of the students in a class, but we don't know how many students are in the class until we are told.

```
int specifyGrades(int n)
{
int i,mark;
    // allocate the grades array
    int* grades = (int*) malloc(n*sizeof(int));
    for (i=0; i<n;i++)
        {
        scanf("%d",&mark);
            grades[i] = mark;
            }
    for (i=0; i<n;i++)
            printf("Student %d's got: %d% \n", i, grades[i]);
    // free the grades array
    free(grades);
}
```

Chapter -11
I/O File Handling

printf( )

printf() is one of the most frequently used functions in C for output. The prototype for printf() is:

```
  int printf(const char *format, ...);
```

printf takes in a formatting string and the actual variables to print. An example of printf is:

```
int x = 5;
char str[] = "abc";
char c = 'z';
float pi = 3.14;
  printf("\t%d %s %f %s %c\n", x, str, pi, "Hi", c);
```

The output of the above would be:

```
5 abc 3.140000 Hi z
```

Let's see what's happening. The \t line signifies an escape sequence, specifically, a tab. Then the %d specifies a conversion specification as given by the variable x. The %s matches with the string and the %f matches with the float. The default precision for %f is 6 places after the decimal point. %f works for both floats and doubles. For long doubles, use %Lf. The %s matches with the "Hi", and the %c tells printf to output the char c. The \n signifies a newline. For a listing of escape sequences.

You can format the output through the formatting line.. By modifying the conversion specification, you can change how the particular variable is placed in output. For example:

```
printf("%10.4d", x);
```

Would print this:

```
0005
```

The . allows for precision. This can be applied to floats as well. The number 10 puts 0005 over 10 spaces so that the number 5 is on the tenth spacing. You can also add + and - right after % to make the number explicitly output as +0005. Note that this does not actually change the value of x. In other words, using %-10.4d will not output -0005. %e is useful for outputting floats using scientific notation. %le for doubles and %Le for long doubles

```
A=printf("NIST");
printf("%d",A);
```

Out put

A=4

Because printf returns number of char present in its body.

C=printf(“hi h r u”);
C will be 8

The printf() function returns the number of characters written or a negative value if an error occurs.

Code	**Format**
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%lf	Double numbers
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer

I/O :: scanf()

scanf() is useful for grabbing things from input. Beware though; scanf isn't the greatest function that C has to offer. Some people brush off scanf as a broken function that shouldn't be used often. The prototype for scanf is:

```
int scanf( const char *format, ...);
```

Looks similar to printf, but doesn't completely behave like printf does. Take for example:

```
scanf("%d", x);
```

You'd expect scanf to read in an int into x. But scanf requires that you specify the address to where the int should be stored. Thus you specify the address-of operator. Therefore,

```
scanf("%d", &x);
```
will put an int into x correctly.

scanf's major "flaw" is it's inability to process incorrect input. If scanf is expecting an int and your standard in keeps giving it a string, scanf will keep trying at the same location. If you looped scanf, this would create an infinite loop. Take this example code:

```
int x, args;

  for ( ; ; ) {
    printf("Enter an integer bub: ");
    if (( args = scanf("%d", &x)) == 0) {
      printf("Error: not an integer\n");
      continue;
    } else {
      if (args == 1)
        printf("Read in %d\n", x);
      else
        break;
      }
  }
```

The code above will fail. Why? It's because scanf isn't discarding bad input. So instead of using just continue;, we have to add a line before it to process input. We can use a function called

digestline().

```
  void digestline(void) {
    scanf("%*[^\n]");   /* Skip to the End of the Line */
    scanf("%*1[\n]");   /* Skip One Newline */  }
```

Using assignment suppression, we can use * to suppress anything contained in the set [^\n]. This skips all characters until the newline. The next scanf allows one newline character read. Thus we can digest bad input!

The scanf() function returns the number of data items successfully assigned a value. If an error occurs, scanf() returns EOF(end of file).

Code	**Meaning**
%c	Read a single character.
%d	Read a decimal integer.
%i	Read an integer in either decimal, hexadecimal format.
%e	Read a floating-point number.
%f	Read a floating-point number.
%g	Read a floating-point number.
%lf	Reading a double number
%o	Read an octal number.
%s	Read a string.
%x	Read a hexadecimal number.
%p	Read a pointer.

%u	Read an unsigned decimal integer.

NOTE:- The scanf() function stops reading a number when the first nonnumeric character is encountered.

Doing working with file we have to follow
Having read this section you should be able to know How to:

1. Open a file for reading or writing
2. Read/write the contents of a file
3. Close the file

The Stream File:

Although C does not have any built-in method of performing file I/O, the C standard library contains a very rich set of I/O functions providing an efficient, powerful and flexible approach. We will cover the ANSI file system but it must be mentioned that a second file system based upon the original UNIX system is also used but not covered on this course.

A very important concept in C is the stream. In C, the stream is a common, logical interface to the various devices that comprise the computer. In its most common form, a stream is a logical interface to a file. As C defines the term "file", it can refer to a disk file, the screen, the keyboard, a port, a file on tape, and so on. Although files differ in form and capabilities, all streams are the same. The stream provides a consistent interface and to the programmer one hardware device will look much like another.

A stream is linked to a file using an open operation. A stream is disassociated from a file using a close operation. The current location, also refered to as the current position, is the location in a file where the next file access will occur. There are two types of streams: text (used with ASCII characters some character translation takes place, may not be one-to-one correspondence between stream and whats in the file) and binary (used with any type of data, no character translation, one-to-one between stream and file).

To open a file and associate it with a stream, use fopen(). Its prototype is shown here:

FILE *fopen(char *fname, char *mode);

The fopen() function, like all the file-system functions, uses the header stdio.h . The name of the file to open is pointed to by fname (must be a valid name). The string pointed at for mode determines how the file may be accesed as shown:

Mode	Meaning
r	Open a text file for reading

w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Open a binary file for writing
ab	Append to a binary file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append a binary file for read/write

If the open operation is successful, fopen() returns a valid file pointer. The type FILE is defined in stdio.h. It is a structure that holds various kinds of information about the file, such as size.The file pointer will be used with all other functions that operate on the file and it must never be altered or the object it points to. If fopen() fails it returns a NULL pointer so this must always be checked for when opening a file. For example:

```
FILE *fp;

if ((fp = fopen("myfile", "r")) ==NULL){
  printf("Error opening file\n");
  exit(1);
}
```

To close a file, use fclose(), whose prototype is

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

The fclose() function closes the file associated with fp, which must be a valid file pointer previously obtained using fopen(), and disassociates the stream from the file. The fclose() function returns 0 if successful and EOF (end of file) if an error occurs.Once a file has been opened, depending upon its mode, you may read and/or write bytes to or from it using these two functions.

```
int fgetc(FILE *fp);
int fputc(int ch, FILE *fp);
```

The getc() function reads the next byte from the file and returns its as an integer and if error occurs returns EOF. The getc() function also returns EOF when the end of file is reached. Your routine can assign fget()'s return value to a char you dont't have to assign it to an integer.

The fput() function writes the bytes contained in ch to the file associated with fp as an unsigned char. Although ch is defined as an int, you may call it using simply a char. The fput() function returns the character written if successful or EOF if an error occurs.

Text File Functions:

When working with text files, C provides four functions which make file operations easier. The first two are called fputs() and fgets(), which write or read a string from a file, respectively. Their prototypes are:

```
int fputs(char *str,FILE *fp);
char *fgets(char *str, int num, FILE *fp);
```

The fputs() function writes the string pointed to by str to the file associated with fp. It returns EOF if an error occurs and a non-negative value if successful. The null that terminates str is not written and it does not automatically append a carriage return/linefeed sequence.

The fget() function reads characters from the file associated with fp into a string pointed to by str until num-1 characters have been read, a newline character is encountered, or the end of the file is reached. The string is null-terminated and the newline character is retained. The function returns str if successful and a null pointer if an error occurs.

The other two file handling functions to be covered are fprintf() and fscanf(). These functions operate exactly like printf() and scanf() except that they work with files. Their prototypes are:

```
int fprintf(FILE *fp, char *control-string, ...);
int fscanf(FILE *fp, char *control-string ...);
```

Instead of directing their I/O operations to the console, these functions operate on the file specified by fp. Otherwise their operations are the same as their console-based relatives. The advantages to fprintf() and fscanf() is that they make it very easy to write a wide variety of data to a file using a text format.

Binary File Functions:

The C file system includes two important functions: fread() and fwrite(). These functions can read and write any type of data, using any kind of representation. Their prototypes are:

```
size_t fread(void *buffer, size_t size, size_t num,FILE *fp);
size_t fwrite(void *buffer, size_t size, size_t num, FILE *fp);
```

The fread() function reads from the file associated with fp, num number of objects, each object size bytes long, into buffer pointed to by buffer. It returns the number of objects actually read. If this value is 0, no objects have been read, and either end of file has been encountered or an error has occurred. You can use feof() or ferror() to find out which. Their prototypes are:

```
int feof(FILE *fp);
int ferror(FILE *fp);
```

The feof() function returns non-0 if the file associated with fp has reached the end of file, otherwise it returns 0. This function works for both binary files and text files. The ferror() function returns non-0 if the file associated with fp has experienced an error, otherwise it returns 0.

The fwrite() function is the opposite of fread(). It writes to file associated with fp, num number of objects, each object size bytes long, from the buffer pointed to by buffer. It returns the number of objects written. This value will be less than num only if an output error as occurred.

The void pointer is a pointer that can point to any type of data without the use of a TYPE cast (known as a generic pointer). The type size_t is a variable that is able to hold a value equal to the size of the largest object surported by the compiler. As a simple example, this program write an integer value to a file called MYFILE using its internal, binary representation.

```
#include <stdio.h>  /* header file  */
#include <stdlib.h>
void main(void)
{
FILE *fp;   /* file pointer */
int i;
/* open file for output */
if ((fp = fopen("myfile", "w"))==NULL){
  printf("Cannot open file \n");
  exit(1);
}
i=100;

if (fwrite(&i, 2, 1, fp) !=1){
printf("Write error occurred");
  exit(1);
}
fclose(fp);

/* open file for input */
if ((fp =fopen("myfile", "r"))==NULL){
  printf("Read error occurred");
  exit(1);
}
printf("i is %d",i);
fclose(fp);
}
```

File System Functions:

You can erase a file using remove(). Its prototype is

int remove(char *file-name);

You can position a file's current location to the start of the file using rewind(). Its prototype is

void rewind(FILE *fp);

Hopefully I have given you enough information to at least get you started with files. Its really rather easy once you get started.

Command Line Parameters:

Many programs allow command-line arguments to be specified when they are run. A command-line argument is the information that follows the program's name on the command line of the operating system. Command-line arguments are used to pass information to the program. For example, when you use a text editor, you probably specify the name of the file you want to edit after the name of the word processing program. For example, if you use a word processor called WP, then this line causes the file TEST to be edited.

WP TEST

Here, TEST is a command-line argument. Your C programs may also utilize command-line arguments. These are passed to a C program through two arguments to the main() function. The parameters are called argc and argv. These parameters are optional and are not used when no command-line arguments are being used.

The argc parameter holds the number of arguments on the command-line and is an integer. It will always be at least 1 because the name of the program qualifies as the first argument. The argv parameter is an array of string pointers. The most common method for declaring argv is shown here.

char *argv[];

The empty brackets indicate that it is an array of undetermined length. All command-line arguments are passed to main() as strings. To access an indivual string, index argv. For example, argv[0] points to the program's name and argv[1] points to the first argument. This program displays all the command-line arguments that it is called with.

```
#include <stdio.h>

void main(int argc, char *argv[])
{
int i;
for (i=1; i&ltargc; i++)
  printf("%s",argv[i]);
}
```

The ANSI C standard does not specify what constitutes a command-line argument, because operatoring systems vary considerably on this point. However, the most common convention is as follows:

Each command-line argument must be separated by a space or a tab character. Commas, semicolons, and the like are not considered separators. For example:

This is a test is made up of four strings, but this,that,and,another is one string. If you need to pass a command-line argument that does, in fact contain spaces, you must place it between quotes, as shown in this example:

"this is a test"

A further example of the use of argc and argv now follows:

```
void main(int argc, char *argv[])
{
  if (argc !=2)  {
     printf("Specify a password");
     exit(1);
  }
  if (!strcmp(argv[1], "password"))
     printf("Access Permitted");
  else
    {
       printf("Access denied");

       exit(1);
    }
program code here ......
}
```

This program only allows access to its code if the correct password is entered as a command-line argument. There are many uses for command-line arguments and they can be a powerful tool.

My final example program takes two command-line arguments. The first is the name of a file, the second is a character. The program searches the specified file, looking for the character. If the file contains at least one of these characters, it reports this fact. This program uses argv to access the file name and the character for which to search.

```
/*Search specified file for specified character. */

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

```
void main(int argc, char *argv[])
{
  FILE *fp;   /* file pointer */
  char ch;

  /* see if correct number of command line arguments */
   if(argc !=3)  {
     printf("Usage: find <filename> <ch>\n");
     exit(1);
   }

   /* open file for input */
   if ((fp = fopen(argv[1], "r"))==NULL)  {
     printf("Cannot open file \n");
     exit(1);
   }

  /* look for character */

  while ((ch = getc(fp)) !=EOF)  /* where getc() is a */
    if (ch== *argv[2]) {         /*function to get one char*/
      printf("%c found",ch);     /* from the file */
      break;
    }
   fclose(fp);
}
```

The names of argv and argc are arbitrary - you can use any names you like. However, argc and argv have traditionally been used since C's origin. It is a good idea to use these names so hat anyone reading your program can quickly identify them as command-line parameters.

Chapter 12
Preprocessor

Recall that preprocessing is the first step in the C program compilation stage this feature is unique to C compilers.
The preprocessor more or less provides its own language which can be a very powerful tool to the programmer. Recall that all preprocessor directives or commands begin with a #.
Use of the preprocessor is advantageous since it makes:

- programs easier to develop,
- easier to read,
- easier to modify
- C code more transportable between different machine architectures.

The preprocessing step happens to the C source before it is fed to the compiler. The two most common preprocessor directives are #define and #include.

#define

The #define directive can be used to set up symbolic replacements in the source. As with all preprocessor operations, #define is extremely unintelligent, it just does textual replacement without understanding. #define statements are used as a crude way of establishing symbolic constants.

```
#define MAX 100
```

Later code can use the symbols MAX which will be replaced by the text to the right of each symbol in its #define.

#include

The "#include" directive brings in text from different files during compilation. #include is a very unintelligent and unstructured -- it just pastes in the text from the given file and continues compiling. The #include directive is used in the .h/.c file convention below which is used to satisfy the various constraints necessary to get prototypes correct.

```
#include "poo.h" // refers to a "user" poo.h file
// in the originating directory for the compile
#include <poo.h> // refers to a "system" poo.h file
// in the compiler's directory somewhere
```

poo.h vs poo.c

The universally followed convention for C is that for a file named "poo.c" containing a bunch of functions...

- A separate file named poo.h will contain the prototypes for the functions in poo.c which clients may want to call. Functions in poo.c which are for "internal use only"
- Near the top of poo.c will be the following line which ensures that the function definitions in poo.c, which ensures the "prototype before definition" rule above.

```
#include "poo.h" // show the contents of "poo.h"
// to the compiler at this point
```

- Any moo.c file which wishes to call a function defined in poo.c must include the following line to see the prototypes, ensuring the "clients must see prototypes" rule above.

```
#include "poo.h"
```

#if

At compile time, there is some space of names defined by the #defines. The #if test can be used at compile-time to look at those symbols and turn on and off which lines the compiler uses. The following example depends on the value of the FOO #define symbol. If it is true, then the "aaa" lines (whatever they are) are compiled, and the "bbb" lines are ignored. If FOO were 0, then the reverse would be true.

```
#define FOO 1
...
#if FOO
aaa
aaa
#else
bbb
bbb
#endif
```

You can use #if 0 ...#endif to effectively comment out areas of code you don't want to compile, but which you want to keeep in the source file.

The unconditional directives are:

#include - Inserts a particular header from another file
#define - Defines a preprocessor macro
#undef - Undefines a preprocessor macro

The conditional directives are:

#ifdef - If this macro is defined
#ifndef - If this macro is not defined
#if - Test if a compile time condition is true
#else - The alternative for #if
#elif - #else an #if in one statement
#endif - End preprocessor conditional

Other directives include:

# - Stringization, replaces a macro parameter with a string constant
## - Token merge, creates a single token from two adjacent ones