Downloaded from: sarojpandey.com.np

INTRODUCTION TO C

1. History of C

An ancestor of C is BCPL – Basic Combined Programming Language. Ken Thompson, a Bell Laboratory scientist, developed a version of BCPL, which he called 'B'. Dennis Ritchie, another computer scientist, developed a version called 'C' in early 1970s that is modified and improved BCPL. Ritchie originally wrote the language for programming under UNIX operating system. Later Bell Laboratories rewrote UNIX entirely in C.

In C, I/O support is provided in the form of library of the object code that can be linked with the user's program. In the 1970s and 1980s, many organizations wrote and implemented C compilers, which differed from one another in their requirements and libraries. One type of machine might even have several different compilers. To establish uniformity and facilitate portability, in 1989 ANSI (American National Standards Institute) approved standards for the language as well as the required libraries. For e.g., ANSI specifies the standard I/O library, including stdio.h.

2. Characteristics of C

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language. We will be studying many of these aspects throughout the course.

- **Portability**
  One of the reasons of C's popularity is its portability. We can easily transform a program written in C from one computer to another with few or no changes and compile with appropriate compilers.

- **Faster and efficient**
  C is desirable because it is faster and more efficient than comparable programs in most other high level languages. For e.g., a program to increment a variable from 0 to 15000 takes about 50 seconds in BASIC while it takes 1 second in C.

- **Supports structured programming**
  It is well suited for structured programming, that means, the problem might be solved in terms of function modules or blocks. The modular structure makes program debugging, testing and maintenance easier.

- **Extendibility**
  Another property is extendibility. C is basically a collection of functions that are supported by C library. We can continuously add our own functions to C library.

- **Flexible**
  C is a flexible language. It permits us to write any complex programs with the help of its rich set of in-built functions and operators. In addition, it permits the use of low-level language. Hence it is also called "middle-level language" and therefore it is well suited for both system software and business package.

3. Basic structure of C programs

C is a group of building blocks called functions, which is subroutine that may include one or more statements designed to perform a specific task.

Documentation section consists set of comment lines giving the name of program, author, and other details to be used later by programmer. The compiler ignores any comment so they do not add to the file size during the time of execution. Comment lines which starts with // for single line comment OR /*….*/ for multiple line comment.

Link section provides instruction to compiler to link functions from system library.

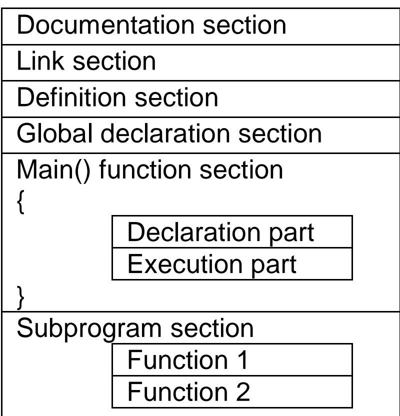
Downloaded from: sarojpandey.com.np

Definition section defines all symbolic constants.

Global declaration section declares all variables used in executable part globally.

Main function section is a must section and one program contains only one main. Main function section starts with opening brace '{' and ends with closing brace '}'. It consists declaration and execution section. **Declaration part** declares all variables used in executable part. There must be at least one statement in **executable part**. Each statement ends with semicolon except for function definitions, control statements and loops.

Subprogram section contains all user-defined functions that are called in main() function.



Every C program consists of one or more modules called *functions.* One- of the functions must be called main () .The program will always begin by executing the main() function, which may access other functions. Any other function definitions must be defined separately, either ahead of or after main ().

The main () function can be located somewhere in the program so that the computer can determine where to start the execution. This function can be allocated anywhere in the program but the general practice is to place it as the first function for better readability

Any function in C program consists of valid C statements and is linked together through function calls. A function is analogous to the subroutine or a procedure in other higher-level languages. Every function in a program has a unique name and is designed to perform a specific task. Each function is defined by of a block of statements, which are enclosed within a par and is treated as one single unit.

Every instruction in C program is written as a separate statement. These statements must appear in the same order in which we wish them to be executed; unless logic of the problem demands a deliberate jump or transfer of control to a statement, which is out of sequence.

Rules for statements:

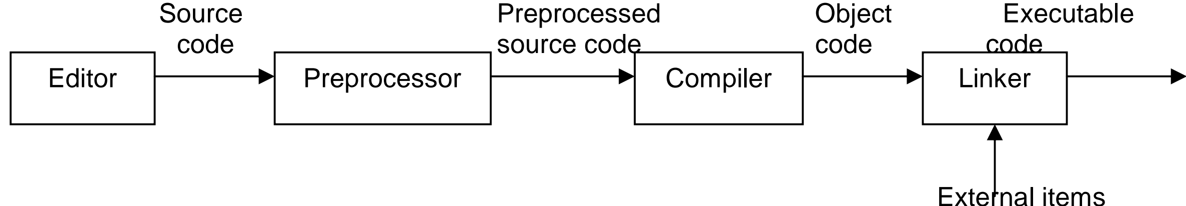
- Generally all C statements are entered in small cases letters.
- Any C statement always ends with a semicolon (;).
- C has no specific rules about the position at which different parts of a statement are to be written.

Example:-

```
#include <stdio.h>        //header files
#include <conio.h>
void main()       //main function
{
        clrscr();             // library functiion to Clear the screen
        printf("This is first lecture in C programming");// library function that prints the given string
        getch();
}
```

Downloaded from: sarojpandey.com.np

4. Steps to execution



Editor: It is a specialized word processor to create and edit source code and data. Source code is a program typed into the computer. You write a computer program with words and symbols that are understandable to human beings. This is the *editing* part of the development cycle. You type the program directly into a window on the screen and save the resulting text as a separate file. This is often referred to as the *source*. The custom is that the text of a C program is stored in a file with the extension .c for C programming language.

Preprocessor: It is a program that removes all comments and modifies source code according to directives supplied in the program. A preprocessor directive begins with '#' and it is an instruction to the program. For e.g., "#include<stdio.h>" instruct the preprocessor to replace the directive with the contents of the file stdio.h.

Compiler: You cannot directly execute the source file. To run on any computer system, the source file must be translated into binary numbers understandable to the computer's Central Processing Unit. Compiler translates the preprocessed source code into machine language that consists sequences of 0s and 1s. If the compiler finds any error, the compilation may continue in order to detect further error but computer won't produce any compiled program. If compiler doesn't detect any error, then it produces object code, which is machine language version of source code. This process produces an intermediate object file - with the extension .obj. The .obj stands for Object.

Linker: It combines all the object code of a program with necessary items (i.e., library files) to form an executable program. Often a program is so large that it is convenient to break it down into smaller units, with each part stored in a separate file. Many compiled languages come with library routines that can be added to your program. Theses routines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. In the case of C the standard input and output functions are contained in a library (stdio.h) so even the most basic program will require a library function. Moreover our program might use features like 'printf' that are defined elsewhere, perhaps in a C library. After compilation of our program files, the computer must somehow link these separate pieces to form a single executable program. This linking is done by linker. After linking the file extension is .exe which are executable files.

Executable files: Thus the text editor produces .c source files, which go to the compiler, which produces .obj object files, which go to the linker, which produces .exe executable file. You can then run .exe files as you can other applications, simply by typing their names at the DOS prompt or run using windows menu.

Downloaded from: sarojpandey.com.np

FUNDAMENTALS OF C

1. C character set

The C character sets are used to form words, numbers and expressions. Different categories of character sets are – letters, digits and special characters.

Letters – upper case: A…Z
Lower case: a…z
Digits – 0…9
Special characters: ‘ “ , . : ; { } [ ] ! # $ & ^ ( ) _ - + * < > ? / \ | !
White spaces: blank, tab, newline

2. Identifiers and keywords

Identifiers

Identifiers are user defined names of variables, functions and arrays. It may be a combination of character(letter and digits) set with the first character as letter.

Rules for identifiers:

- Legal characters are a-z, A-Z, 0-9, and _.
- Case is significant. Since C is case sensitive, the lower case letters are not equal to uppercase.
- The first character must be a letter or _.
- The blank space is not allowed but can include(_) between two identifiers.
- Identifiers can be of any length (although only the first 31 characters are ***guaranteed*** to be significant).

Here are some examples of legal identifiers:

i
count
NumberOfAardvarks
number_of_aardvarks
MAX_LENGTH

Keywords

Keywords are the reserved words having fixed meaning in C. They cannot be used as identifiers. C makes use of only 32 keywords which combine with the formal syntax to the form the C programming language. Note that all keywords are written in lower case – C uses upper and lowercase text to mean different things. If you are not sure what to use then always use lowercase text in writing your C programs. A keyword may not be used for any other purposes. For example, you cannot have a variable called **auto**. If we try to use keyword as variable name, then we’ll be trying to assign new meaning to the keyword, which is not allowed by the computer.

The following are reserved keywords, and may not be used as identifiers:

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

3. Header files

Header file defines certain values, symbols and operations which is included in the file to obtain access to its contents. The header file have the suffix ‘.h’. It contains only the prototype of the function in the corresponding source file. For e.g., stdio.h contains the prototype for ‘printf’ while the corresponding source file contains its definition. It saves time for writing and debugging the code.

Downloaded from: sarojpandey.com.np

4. Constants and variables

Constants

ANSI C allows you to declare ***constants***. Constant is an identifier that is always associated with the same data value nad is taken in its absolute terms. When you declare a constant it is a bit like a variable declaration except the value cannot be changed. There are two types of constants - literal constant and symbolic constant.

A *literal constant* is a value typed directly into your computer and its value must match with the corresponding data type. A *symbolic constant* is represented by a name just like as the variable is represented. Unlike a variable, however a constant is initialized. There are two ways to declare symbolic constant:

1. The *const* keyword is to declare a constant, as shown below:
   Syntax:
   <const><datatype><identifier> = <constant value>

   ```
    int const a = 1;
   const int a =2;
   ```
   Note:
   - You can declare the `const` before or after the type.
   - It is usual to initialize a `const` with a value, as it cannot get a value any other way.

2. The preprocessor *#define* is another more flexible method to define constants in a program. #define makes the preprocessor replace every occurrence of the constant. For e.g., the line *#define PI 3.14* appearing at the beginning of the program specifies that the identifier PI will be replaced by the 3.14 through out the program.

Variables

Variables are the most fundamental part of any programming language. Variable is a symbolic name which is used to store different types of data in the computer's memory. When the user gives the value to the variable, that value is stored at the same location occupied by the variable. Variables may be of integer, character, string, or floating type.

Variable declaration and initialization

Before you can use a variable you have to declare it. As we have seen above, to do this you state its *type* and then give its *name*. For example, **`int i;`** declares an integer variable. You can declare any number of variables of the same type with a single statement.

To declare a variable in C, do:
<Data type>< list variables >;

For example:
```
int a, b, c;
float x,y,z;
char ch;
```
declares three integers: a, b and c, three floating variables: x, y and z and one character: ch. You have to declare all the variables that you want to use at the start of the program.

Variables are defined in the following way:-
```
main()
{
        short number,sum;
        int bignumber,bigsum;
        char letter;
                }
```

Downloaded from: sarojpandey.com.np

It is also possible to initialize variables using the = operator for assignment.
For example:-

```
main()
{
float sum=0.0;
int bigsum=0;
char letter='A';
}
```

This is the same as:-

```
main()
{
        float sum;
        int bigsum;
        char letter;
                sum=0.0;
                bigsum=0;
                letter='A';
}
```

...but is more efficient.

C also allows multiple assignment statements using =, for example:

```
a=b=c=d=3;
```

...which is the same as, but more efficient than:

```
a=3;
b=3;
c=3;
d=3;
```

This kind of multiple assignment is only possible if all the variable types in the statement are the same.

Furthermore, you can assign an initial value to a variable when you declare it. For example:

```
int i=1;
```

sets the **int** variable to one as soon as it's created. This is just the same as:

```
int i;
i=1;
```

but the compiler may be able to speed up the operation if you initialize the variable as part of its declaration. Don't assume that an uninitialized variable has a sensible value stored in it. Some C compilers store 0 in newly created numeric variables but nothing in the C language compels them to do so.

5. Data types

The first thing we need to know is that we can create *variables* to store *values* in. A variable is just a named area of storage that can hold a single value (numeric or character). C is very fussy about how you create variables and what you store in them. It demands that you declare the name of each variable that you are going to use and its *type*, before you actually try to do anything with it. Hence, data type can be defined as the storage representation and machine instruction to handle constants and variables.

There are four basic data types associated with variables:

- Primary or fundamental data type
- User defined data type
- Derived data type
- Empty data set

Primary/fundamental data type

Downloaded from: sarojpandey.com.np

The data type that is used without any modifier is known as primary data type. The primary data type can be categorized as follows:

1. Integral type
    a. signed integer type
        i. int
        ii. short int
        iii. long int
    b. unsigned integer type
        i. unsigned int
        ii. unsigned short int
        iii. unsigned long int

2. Character type
    a. signed char
    b. unsigned char

3. Floating point type
    a. float
    b. double
    c. long double

- int - integer: a whole number. We can think of it as a positive or negative whole number. But, no fractional part is allowed
  To declare an `int` you use the instruction:
  `int variable name;`
  For example:
  `int a;`
  declares that you want to create an `int` variable called `a`.
- float - floating point or real value, i.e., a number with a fractional part.
- double - a double-precision floating point value.
- char - a single character.
  To declare a variable of type character we use the keyword **char**. - A single character stored in one byte.
  For example:
  `char c;`
  To assign, or store, a character value in a **char** data type is easy - a character variable is just a symbol enclosed by single quotes. For example, if `c` is a **char** variable you can store the letter **A** in it using the following C statement:
  c='A'
  Notice that you can only store a single character in a **char** variable. But, a string constant is written between double quotes. Remember that a **char** variable is **'A'** and not **"A"**.

Here are the list of data types with their corresponding required memory range of value it can take:

Data type	Size (bytes)	Range
int	2	-32,768 to 32,737
unsigned int	2	0 to 65,535
short int	1	-128 to +127
unsigned short int	1	0 to 255
long int	4	-2,147,483,648 to +2,147,483,647
unsigned long int	4	0 to 4,294,967,295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308

Downloaded from: sarojpandey.com.np

long double	10	3.4E-4932 to 1.1E+4932
signed char	1	128 to +127
unsigned char	1	0 to 255

User defined data type

The user defined data type can be later used to declare variable. We can define our own types using *typedef* and *enum*. As an example of a simple use, let us consider how we may define new type 'letter'. These new types can then be used in the same way as the pre-defined C types:

Syntax: <typedef> <datatype> <new type>;
<new type> <variable >;

```
typedef char letter;
  letter name, address;
```

here, a new type 'letter' is created whose data type is of char. Later, this 'letter' is used as data type to declare the variables name and address.

Derived data type

Different user defined data type can be created using fundamental data types which are called derived data type. Array, structure, union and functions are derived data types.

Empty data set/ Void

Void means valueless or empty special purpose type, which we will examine closely in later sections. It is used with function where a function doesn't return any value.

6. Operators

C provides rich set of operator environment. Operators are symbols that tell the computer to perform certain mathematical and logical manipulations. They are used to form a part of mathematical and logical expressions. The variable/quantity on which operation is to be performed is called operand. The operators can be categorized as:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Conditional operators
5. Unary operators
6. Assignment operators
7. Special operators

Arithmetic Operators

C provides arithmetic operators found in most languages. These operators work on all data types. Each operator require at least two operands. There are 5 different arithmetic operators- addition (+), subtraction (-), multiplication (*), division (/) and remainder or modulus (%). The % (modulus) operator only works with integers. Division / is for both integer and float division.

There are three modes of arithmetic operations:

- Integer arithmetic: Both operands are integer. Hence the result will also be integer.
  For example, if a=11, b=4 then,
  a+ b = 15 ; a-b = 7 ; a/b = 2 ; a*b = 44 ; a%b = 3

- Real arithmetic: Both operands are real.
  For example, if a=1.0, b=3.0 then,
  a+ b = 4.000000 ; a-b = -2.000000 ; a/b = 0.333333 ; a*b = 3.000000

- Mixed mode arithmetic: If both real and integer operands are used. The result will be real.
  For example, if a=11.0, b=4 then,
  a+ b = 15.0 ; a-b = 7.0 ; a/b = 2.75 ; a*b = 44 .0

Downloaded from: sarojpandey.com.np

Note: The answer to: $x = 3 / 2$ is 1 even if x is declared a float. But x=3.0/2.0 will give the result 1.5. also note that x=3.0/2.0 is 1 if X is declared as integer.

Relational operators

Relational operators compare two quantities. The result of comparison is TRUE or FALSE. They are used for decision-making. The operators < (less than) , > (greater than), <= (less than or equals), >= (greater than or equals), == (equals to), != (not equal to) are relational operators. The operators == and != are also known as equality operators.

For example: in the expression x<y, it returns 'true' if x is greater than y otherwise returns 'false'.

warning: Beware of using ''=" instead of ''==", such as writing accidentally
if (i = j) .....
This is a perfectly **LEGAL** C statement (syntactically speaking), which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is non-zero. This is called **assignment by value** -- a key feature of C.

Assignment operators

The assignment operator us used to assign result of an expression to a variable. The assignment operator '=' assigns the value of right operand to left. For e.g., in an expression x=y, the value of y is assigned to x.

There is also a convenient shorthand way to express computations in C. It is very common to have expressions like: $i = i + 3$ or $x = x*(y + 2)$

In C (generally) in a ***shorthand*** form like this:
expr1 <op>= expr2
which is equivalent to (but more efficient than):
expr1 = expr1 <op> expr2

= assignment
+= addition assignment
-= subtraction assignment
*= multiplication assignment
/= division assignment
%= remainder/modulus assignment

So we can rewrite $i = i + 3$ as $i += 3$
and $x = x*(y + 2)$ as $x *= y + 2$.
NOTE: that $x *= y + 2$ means $x = x*(y + 2)$ and **NOT** $x = x*y + 2$.

Logical Operators

Logical operators are usually used with conditional statements. It performs test on multiple relations and Boolean operation. It also returns either true or false. The three basic logical operators are: && for logical AND, || for logical OR , ! for logical NOT. The truth table for AND and OR is as follows:

Operand1	Operand2	Operand1 && Operand2	Operand1 || Operand2
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

Downloaded from: sarojpandey.com.np

For example,

```
if(a>=1 && a<=6)
printf("a lies between 1 and 6"); // prints the message if both conditions are satisfied.
```

Beware & and | have a different meaning for bitwise AND and bitwise OR.

Unary operators

The two operators that are used frequently are ++ and --. ++ specifies increment, the -- specifies decrement. You can place these in front or on the back of variables. If the operator is placed in front, it is ***prefix*** if it is placed behind, it is ***postfix***. Prefix means, increment before any operations are performed, postfix is increment afterwards. These are important considerations when using these operators.

Increment ++, Decrement -- which are more efficient than their long hand equivalents, for example:- x++ is faster than x=x+1.

For e.g., if a=5, x=a++, y=++a and z= a--, then the values of x, y, and z will be 5, 7 and 7 respectively.

Conditional operators

C includes special ternary (3 way) operator that can replace certain if-else statement. The ?: operator is called ternary operator. The general form of this operator is:

Expression1 ? expression2 : expression3

That means if expression1 is true, then expression2 is executed else expression3 is executed.

For e.g., X= (a<2) ? (a+10) : (a+5);

This is equivalent to,

```
if (a<2)
  X= a+10;
else
  X= a+5;
```

Special operators

- **The comma operator**

  C allows you to put multiple expression in the same statement, separated by a comma. It is also used in loops. The expressions are evaluated in left-to-right order.

  For eg, value = (x=5, y=10, x+y); //comma operator in multiple expression
  for(m=0, n=10; m<n; m++) // comma operator in for loop

- **sizeof operator**

  It is a compile time operator that returns the number of bytes the operand occupies. For example, to determine the number of bytes occupied by the integer, we can use sizeof operator as:

  ```
  int x;
  printf(" Size of integer =%d", sizeof(x));
  ```

  which is equivalent to,

  ```
  printf(" Size of integer =%d", sizeof(int));
  ```

Associativity and Order of Precedence

The ***precedence*** of an operator gives the order in which operators are applied in expressions: the highest precedence operator is applied first, followed by the next highest, and so on. The ***associativity*** of an operator gives the order in which expressions involving operators of the same precedence are evaluated.

Downloaded from: sarojpandey.com.np

Operators on the same line have the same precedence, and are evaluated in the order given by the associativity. To specify a different order of evaluation you can use parentheses. In fact, it is often good practice to use parentheses to guard against making mistakes in difficult cases, or to make your meaning clear.

It is necessary to be careful of the meaning of such expressions as a + b * c. We may want the effect as either (a + b) * c or a + (b * c)
All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that
    a - b - c is evaluated as (a - b) - c
as you would expect.

Thus , a < 10 && 2 * b < c is interpreted as ,
  (a < 10) && ((2 * b) < c)

Consider the following calculation:
a=10.0 + 2.0 * 5.0 - 6.0 / 2.0
What is the answer? If you think its 27, then you are wrong! Perhaps you got that answer by following each instruction as if it was being typed into a calculator. A computer doesn't work like that and it has its own set of rules when performing an arithmetic calculation. All mathematical operations form a hierarchy that is shown below. In the above calculation the multiplication and division parts will be evaluated first and then the addition and subtraction parts. This gives an answer of 17.

Note: *To avoid confusion use brackets*. The following are two different calculations:
a=10.0 + (2.0 * 5.0) - (6.0 / 2.0)
a=(10.0 + 2.0) * (5.0 - 6.0) / 2.0

The following table lists all the operators, in order of precedence, with their associativity:

```
Operator                                    Associativity
--------                                    -------------
() [] ->> .                                 left-to-right
- ++ -- ! ~ * & sizeof (type)               right-to-left
* / %                                       left-to-right
+ -                                         left-to-right
<< >>                                       left-to-right
< <= > >=                                   left-to-right
== !=                                       left-to-right
&                                           left-to-right
^                                           left-to-right
|                                           left-to-right
&&                                          left-to-right
||                                          left-to-right
?:                                          right-to-left
= += -= *= /= %= &= ^= |= <<= >>=           right-to-left
,                                           left-to-right
```

Note: the - and * operators appear twice in the above table. The unary forms (on the second line) have higher precedence that the binary forms.

Downloaded from: sarojpandey.com.np

CONTROL STATEMENTS

A program consists of a number of statements, which are usually executed in sequence. A Statement is an instruction given to the computer to perform any kind of action such as manipulation of data, reading/writing of data, making decision, repeating action and so on. Statements fall into three general types;

- Assignment, where values, usually the results of calculations, are stored in variables.
- Input / Output, data is read in or printed out.
- Control, the program makes a decision about what to do next.

Programs can be much more powerful if we can control the order in which statements are run. Control statements determine the flow of control of a program or an algorithm. The flow of control of a program is the order in which the computer executes the statements. The sequential control statements execute one after another. Generally, the statements are executed sequentially which is known as normal flow of program. To over-ride the sequential flow of program statements, a programming language uses control statements to advance and branch based on changes to state of a program. The control statements in C can be used to write powerful programs by;

- Selecting between optional sections of a program – conditional execution/selection.
- Repeating important sections of the program – looping.

Selection or decision making statements:

With the help of selection control structure, the computer makes decision by evaluating the logical expression. It allows our program to choose different path of execution based upon the outcome of an expression or the state of the variable. The C supports following type of selection or decision making statements :

- If statement
- If…else statement
- Switch statement

The segment of the code that is executed repeatedly is known as loop. The loop control structures are:

- while loop
- do…while loop
- for loop

The if Statement

The **if** statement is a conditional branch statement. If the condition is true, then the statement after condition is executed otherwise execution will skip to next statement.

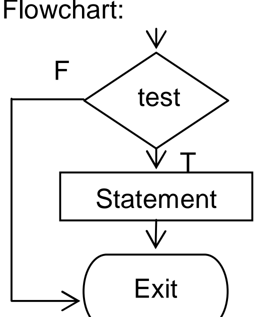
Downloaded from: sarojpandey.com.np

Syntax:

```
if (condition/ test expression)
     statement;

OR
if (condition/ test expression)
     {
       block of  statements;
      }
```

Flowchart:



For example:

```
#include<stdio.h>
void main()
{
            int a;
            printf("Enter the value of a:",);
            scanf("%d", &a);
            if(a>=0)
                 printf("The number is positive");
}
```

The if else Statement

The **if…else** statement consist of an **if** statement followed by statement or block of statement, followed by **else** keyword which is again followed by another statement or block of statement. In an if…else statement, the condition is evaluated first. If the condition is true, the statement in the immediate block is executed. If the condition is false, the statement in the else block is executed. This is used to decide whether to do something at a special point, or to decide between two courses of action.

Syntax:

```
if (condition/ test expression)
     statement;
else
     statement;

OR

if (condition/ test expression)
     {
       block of  statements;
      }
else
    {
       block of  statements;
      }
```

Flowchart:

test
F
Statement
T
Statement
Exit

The following test decides whether a student has passed an exam with a pass mark of 45 :

```
if (result >= 45)
       printf("Pass\n");
else
       printf("Fail\n");
```

Downloaded from: sarojpandey.com.np

It is possible to use the if part without the else.

```
if (temperature < 0)
        print("Frozen\n");
```

Each version consists of a test, (this is the bracketed statement following the if). If the test is true then the next statement is obeyed. If it is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.

If we wish to have more than one statement following the if or the else, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```
if (result >= 45)
{       printf("Passed\n");
        printf("Congratulations\n")
}
else
{       printf("Failed\n");
        printf("Good luck in the resits\n");
}
```

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 75)
        printf("Passed: Grade A\n");
else if (result >= 60)
        printf("Passed: Grade B\n");
else if (result >= 45)
        printf("Passed: Grade C\n");
else
        printf("Failed\n");
```

In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final one else may be left out. It is up to the programmer to devise the correct structure for each programming problem.

<u>The nested if Statement</u>

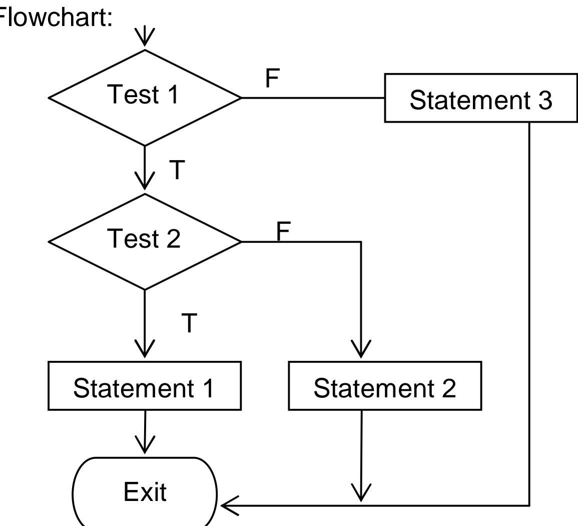
Nested **if**s are very common in programming. Nested if is a structure which has another **if…else** body within its body of structure. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block.

Downloaded from: sarojpandey.com.np

Syntax:

```
if (condition 1)
  { if (condition 2)
      {   statement 1; }
    else
      {   statement 2; }
  }
else
  {   statement 3; }
```

Flowchart:



The switch Statement

This is another form of the multi way decision. It checks the value of an expression to the list of constant values. If the condition is matched, the statement/statements associated with it will be executed. If the expression does not match any of he case statement, and if there is a default statement, execution switches to default statement otherwise the switch statement ends. It is well structured, but can only be used in certain cases where;

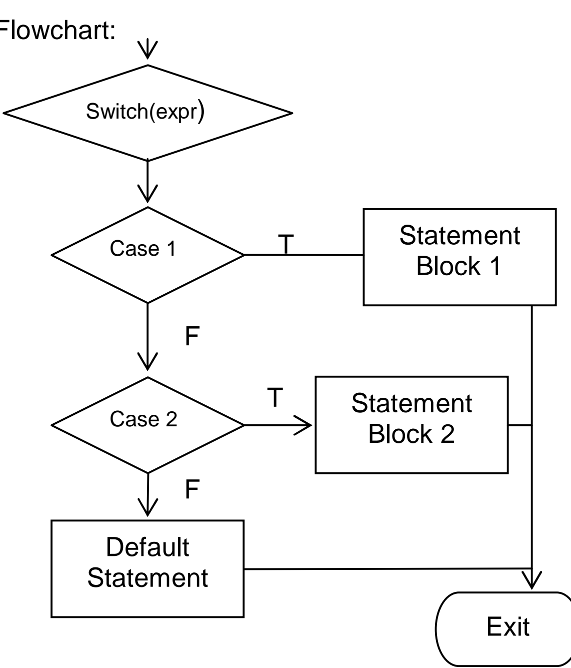
- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- You cannot use ranges as an expression i.e. the expression must give as absolute value.
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Syntax:

```
switch(expression)
  { case value1:  {   statement block 1; }
                  break;

    case value2:  {   statement block 2; }
                  break;
    ………….
    ………….
   default:   {   statement block; }
 }
```

Flowchart:



Downloaded from: sarojpandey.com.np

```
/* Estimate a number as none, one, two, several, many */
void main()
{
int number;
{       switch(number) {
        case 0 :
                printf("None\n");
                break;
        case 1 :
                printf("One\n");
                break;
        case 2 :
                printf("Two\n");
                break;
        case 3 :
        case 4 :
        case 5 :
                printf("Several\n");
                break;
        default :
                printf("Many\n");
                break;
        }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

Loops

The other main type of control statement is the loop. Computers are very good at repeating simple tasks many times; the loop is C's way of achieving this. Loops allow a statement, or block of statements, to be repeated a certain number of times. The loop repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the

Downloaded from: sarojpandey.com.np

statement following the loop. Loops can be classified into entry-controlled loops and exit-controlled loops. In the entry-controlled loops, the user knows the times of repetition before entering the loop. Whereas in exit-controlled loops, the number of repetition can be known only after the loop.

C gives you a choice of three types of loop, while, do while and for.

- The while loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The do while loops is similar, but the test occurs after the loop body is executed. This ensures that the loop body is run at least once.
- The for loop is frequently used, usually where the loop will be traversed a fixed number of times. It is very flexible, and novice programmers should take care not to abuse the power it offers.

Every loop constitutes three main parts:

- Initialization: Every loop must have a starting point called initialization
- Test expression: It determines how many times a loop must execute. The loop continues as long as the test expression is true.
- Update: A loop must be updated after the execution of certain statements to reach its final destination. We may update the loop by increment, decrement operators or by using any other arithmetic operators.

Downloaded from: sarojpandey.com.np

The while Loop

The while loop repeats a statement until the test at the top proves false.

Syntax:

```
Initialization;
while (test expression)
{
    body of loop;
    update;
}
```

Flowchart:

initialization → test → (F) Exit; (T) Body of loop → update → back to test

/* a program to print 10 numbers using while loop */

```
#include<stdio.h>
void main()
{
        int digit=0;
        while(digit <= 10)
        {
                printf("%d\t", digit);
                digit++;
        }
}
```

The do while Loop

This is very similar to the while loops except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

Syntax:

```
Initialization;
do
{
    body of loop;
    update;
}
while (test expression);
```

Flowchart:

initialization → Body of loop → update → test → (F) Exit; (T) back to Body of loop

```
do
{       printf("Enter 1 for yes, 0 for no :");
        scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

Downloaded from: sarojpandey.com.np

```
/* a program to print 10 numbers using do…while loop */

#include<stdio.h>
void main()
{
          int digit=0;
          do
          {         printf("%d\t", digit);
                    digit++;
          }
          while(digit <= 10);

}
```

The for Loop

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- The first is run before the loop is entered. This is usually the initialisation of the loop variable.
- The second is a test, the loop is exited when this returns false.
- The third is a statement to be run every time the loop body is completed. This is usually an increment of the loop counter.

Syntax:

```
for (Initialization; test expression; update)
{
    body of loop;
}
```

Flowchart:

initialization
test
F
Exit
T
Body of loop
update

```
/* a program to print 10 numbers using for loop */
#include<stdio.h>
void main()
{
          int digit;
          for(digit = 0; digit <= 10; digit++ )
              printf("%d\t", digit);
}
```

The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialization or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely unless interrupted by some other means. This might be a return or a break statement.

It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below

Downloaded from: sarojpandey.com.np

illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```
for (hi = 100, lo = 0; hi >= lo; hi--, lo++)
```

The for loop is extremely flexible and allows many types of program behavior to be specified simply and quickly.

The goto Statement

The **goto** statement transfers control anywhere in the program. Destination of **goto** statement is marked by a label or the user defined label. The label is always terminated by a colon. The **goto** statement is used for condition and unconditional branching from one location to another location. Unconditional branching is an unhealthy practice of a program. The **goto** statement can be used for the forward jump or backward jump in the program.

Syntax:

```
goto<label>;
………..
……….
label:
   statement;
```

Forward jump

```
label:
   statement;
………..
……….
goto<label>;
```

Backward jump

Example:

```
#include<stdio.h>
void main()
{
          int a;
          top:  //label name
          printf("Enter the positive value of a:");
          scanf("%d", &a);
          if(a<0)
             goto top;
          else
              printf("The entered value = %d", a)
}
```

The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The following program calculates the sum of entered number only if the value is positive otherwise it exits from the loop. Here the break statement causes to terminate the loop when we enter the negative value.

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

void main()
{
```

Downloaded from: sarojpandey.com.np

```
        int i, sum = 0, num;
        clrscr();

        for(i= 0; i<=5; i++ )
        {
                printf("Enter number %d :", i);
                scanf("%d", &num);
                if(num<0)
                        break;
                sum+= num;
        }
        printf("The sum = %d", sum);
        getch();
}
```

The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

The following program calculates the sum of entered number only if the value is positive. Unlike the break statement, the continue statement does not terminate the loop when we enter the negative value, but it skips the following statement and continues the loop.

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

void main()
{
        int i, sum = 0, num;
        clrscr();

        for(i= 0; i<=5; i++ )
        {
                printf("Enter number %d :", i);
                scanf("%d", &num);
                if(num<0)
                        continue;
                sum+= num;
        }
        printf("The sum = %d", sum);
        getch();
}
```

Downloaded from: sarojpandey.com.np

INPUT OUTPUT STATEMENTS

The C language comprises of several library functions that carry out various commonly used operation or calculations. For example, there are library functions that carry out standard input / output operations, functions that perform operations on the characters, functions that operate on strings, and functions that carry out various mathematical calculations.

Functionally similar library functions are usually grouped together as object programs in separate library files. In order to use a library function it may be necessary to include certain specific information within the main portion of the program. This is accomplished with the following preprocessor directive statement:

#include<Header filename>

The I/O statements can be categorized as unformatted and formatted I/O. The unformatted I/O statements do not specify how the input and output are carried out. But the formatted I/O statements determine the formats in which the input and output are executed. The functions such as getc(), putc(), getchar(), putchar() are considered as unformatted I/O because they do not contain any information about the format specifiers, field width and any escape sequences. They simply take variable as parameter. Formatted I/O generates output under the control of a *format string* (its first argument) which consists of literal characters to be printed and also special character sequences--*format specifiers*--which request that other arguments be fetched, formatted, and inserted into the string.

CONVERSION SPECIFICATION

Conversion specification specifies to what notation the computer should covert a value for input or output operations. Conversion specification is also known as format specifier. It uses conversion character '%' and type specifier. The type conversion specifier does what you ask of it - it convert a given bit pattern into a sequence of characters that a human can read.

The more frequently used format specifiers for scanf () and printf() functions are listed below:

Format specifiers	Meaning
%c	data item is displayed as a single character.
%d	data item is displayed as a signed decimal integer
%u	data item is displayed as an unsigned decimal integer
%e	data item is displayed as a floating-point value with an exponent.
%f	data item is displayed as a floating-point value without an exponent
%o	data item is an octal integer without a leading zero
%x	data item is a hexadecimal integer without the leading 0's
%s	data item is displayed as a string

ESCAPE SEQUENCES

The escape sequences comprises of escape character backslash symbol(\) followed by a character with special meaning. The escape sequences cause an escape form normal interpretation of a string so that the next string is recognized as having a special meaning. The escape sequences are considered as single character rather than a string. Here are some of the mostly used escape sequences:

```
\b      backspace
\f      formfeed
\n      new line
\r      carriage return
\t      horizontal tab
\'      single quote
\"      double quote
\\      back slash
\0      null
```

Downloaded from: sarojpandey.com.np

If you include any of these in the control string then the corresponding ASCII control code is sent to the screen, or output device, which should produce the effect listed.

<u>SINGLE CHARACTER INPUT-- THE getchar FUNCTION</u>

Single characters can be entered into the computer using the C library function getchar. The function does not require any arguments, though a pair of empty parentheses must follow the word getchar.

```
Syntax: character variable=getchar();
Usage:          char ch;
                ch = getchar();
```

<u>SINGLE CHARACTER OUTPUT-- THE putchar FUNCTION</u>

Single character output can be displayed using the C library function putchar. The character being transmitted will normally be represented as a character- type variable. It must be expressed as an argument to the function, following the word putchar.

```
Syntax: putchar(character variable)
Usage:          char ch;
                ………
                Putchar(c)
```

Following program accepts a single character from the keyboard and displays it on the VDU.

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

/* function to accept and display a character*/
void main()
{
        char ch;
        clrscr();
        printf("\n Enter any character of your choice:-");
        ch = getchar();
        printf("\n The character u entered was ");
        putchar(ch);
        getch();
}
```

Output:
Enter any character of your choice: - p
The character u entered was p

The above program can also be written as,

```
#include<stdio.h>
#include<conio.h>
void main()
{       char ch;
        clrscr();
        printf("\n Enter any character of your choice:- ");
        ch = getc(stdin);
        printf("\n The character u entered was ");
        putc(ch,stdout);
        getch();
}
```

Downloaded from: sarojpandey.com.np

In the above program getc(stdin) is equivalent to getchar() and putc(ch,stdout) is equivalent to putchar(ch);

Downloaded from: sarojpandey.com.np

More Example of getchar() and putchar():
Following program reads a line of text in lowercase letters and displays them in the upper case.

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

void main()
{       char t, text[80];      // to store the text up to 80 characters long
        int i = 0, tag;
        printf(" Enter any text below:\n");
        do
        {       t =getchar();
                text[i]=t;
                i++;
        }
        while(t! = '\n');
        tag = i;                /* to store the maximun number
                                        of characters entered by users*/
        for(i=0;i<tag;i++)
        putchar ( toupper (text[i]));     /* the library function toupper() is used
                                                to convert the text to upper case */
        getch();
}
```

Output:
Enter any text below:
my name is harry
MY NAME IS HARRY

<u>THE gets AND puts FUNCTION</u>

C provides the functions gets() and puts() for string input-output. Though these operations can be done using the scanf and printf functions with %s conversion character, but the limitation is that with scanf, a string which has a blank space within it can never be accepted. The function gets() is the solution then.

The function gets() accepts the string variable into the location where the input string is to be stored. The function puts() accepts as a parameter, a string variable or a string constant for displayed on the standard output.

Syntax: gets(string variable);
puts(string constant/string variable);

The following program accepts a name and displays it with a message.

```
#include<stdio.h>
void main()
{
    char name[20];
    puts(" Enter your name: ");        gets(name);
    puts("\nHello , How are you? ");
    puts(name);
}
```

output:
Enter your name: ram
Hello, How are you? Ram

Downloaded from: sarojpandey.com.np

ENTERING INPUT DATA – THE scanf FUNCTION

Input data can be entered into the computer from a standard input device by means of the C library function scanf. This function can be used to enter any combination of numerical values, single characters and strings.

Syntax: scanf(control string, argl, arg2, …, argn)

Where control string refers to a string containing certain required formatting information, and argl, arg2… argn are argument list that represent individual data items. The control string comprises individual groups of characters, with one character group for each data item. Each character group must begin with a percent sign (%). In this case the *control string* specifies how strings of characters, usually typed on the keyboard, should be converted into values and stored in the listed variables. The most obvious is that `scanf` has to change the values stored in the parts of computers memory that is associated with parameters (variables). The `scanf` function has to have the *addresses* of the variables rather than just their values. This means that simple variables have to be passed with a proceeding &.

The arguments are written as variables or arrays, whose types match the corresponding character groups in control strings. Each variable name must be preceded by an ampersand (&). However, array names should not begin with an ampersand because the array name is already a pointer.

Usage:
```
#include <stdio.h>
void main ()
{
char item [20] ;
int partno ;
float cost ;
…
scanf ("%s %d %f", item, &partno, &cost) ;
…
}
```

When the program reaches the `scanf` statement it pauses to give the user time to type something on the keyboard and continues only when users press `<Enter>`, to signal that we have finished entering the value. The `scanf` processes the control string from left to right and each time it reaches a specifier it tries to interpret what has been typed as a value. If you input multiple values then these are assumed to be separated by white space - i.e. spaces, newline or tabs. This means you can type:

```
3 4 5
```

or

3
4
5

and it doesn't matter how many spaces are included between items.

For example:

```
scanf("%d %d",&i,&j);
```

will read in two integer values into `i` and `j`. The integer values can be typed on the same line or on different lines as long as there is at least one white space character between them. The only exception to this rule is the `%c` specifier which always reads in the next character typed no matter what it is. You can also use a *width* modifier in `scanf`. In this case its effect is to limit the number of characters accepted to the *width*.

For example:

```
scanf("%lOd",&i)
```

would use at most the first ten digits typed as the new value for `i`.

Downloaded from: sarojpandey.com.np

There is one main problem with `scanf` function which can make it unreliable in certain cases. The reason being is that `scanf` tends to ignore white spaces, i.e. the space character. If you require your input to contain spaces this can cause a problem. Therefore for *string* data input the function `gets()` may well be more reliable as it records spaces in the input text and treats them as an ordinary characters.

WRITING OUTPUT DATA -– THE printf FUNCTION

Output data can be written from the computer into a standard output device using the library function printf (). This function can be used to output any combination of numerical values, single characters and strings.

Syntax: printf(control string, argl, arg2, …, argn);

Where control string refers to a string that contains formatting information, and argl, arg2 are arguments that represent the individual output data items. The argument can be written as constants, single variable or array names, or more complex expressions. A format specifier is used to control what format will be used by the printf() to print the particular variable.

Usage:
```
#include <stdio.h>
main ()
{
      char item [20] ;
      int partno ;
      float cost ;
      …
      printf (“%s %d %f”, item, partno, cost) ;
}
```

The *control string* is all-important because it specifies the type of each *variable* in the list and how you want it printed. The way that this works is that `printf` scans the string from left to right and prints on the screen, or any suitable output device, any characters it encounters - except when it reaches a `%` character. The `%` character is a signal that what follows it is a specification for how the next variable in the list of variables should be printed. `printf` uses this information to convert and format the value that was passed to the function by the variable and then moves on to process the rest of the control string and anymore variables it might specify.

For example:
```
printf("Hello World");
```
only has a control string and, as this contains no `%` characters it results in `Hello World` being displayed and doesn’t need to display any variable values.

The specifier `%d` means *convert the next value to a signed decimal integer* and so:
```
printf("Total = %d",total);
```
will print `Total =` and then the value passed by total as a decimal integer.

The `%d` isn’t just a format specifier, it is a *conversion specifier*. It indicates the data type of the variable to be printed and how that data type should be converted to the characters that appear on the screen. That is `%d` says that the next value to be printed is a signed integer value (i.e. a value that would be stored in a standard `int` variable) and this should be converted into a sequence of characters (i.e. digits) representing the value in decimal. If by some accident the variable that you are trying to display happens to be a `float` or a `double` then you will still see a value displayed - but it will not correspond to the actual value of the `float` or `double`.

FIELD WIDTH SPECIFIERS

Each *specifier* can be preceded by a *modifier* which determines how the value will be printed. The most general modifier is of the form:

`flag width.precision`

The *`flag`* can be any of:

Downloaded from: sarojpandey.com.np

```
flag          meaning
   -          left justify
   +          always display sign
   space      display space if there is no sign
   0          pad with leading zeros
```

The *width* specifies the number of characters used in total to display the value and *precision* indicates the number of characters used after the decimal point.

```
   n.m        (n)  a number specifying minimum field width
               .   to separate n from m
              (m)  significant fractional digits for a float
```

For example, `%10.3f` will display the `float` using ten characters with three digits after the decimal point. Notice that the ten characters includes the decimal point, and a `-` sign if there is one. If the value needs more space than the *width* specifies then the additional space is used - *width* specifies the smallest space that will be used to display the value. The specifier `%-10d` will display an `int` left justified in a ten character space. The specifier `%+5d` will display an `int` using the next five character locations and will add a `+` or `-` sign to the value.

The scanf() and printf() function gives the programmer considerable power to format the printed output.
Let's explain this by following example:

```
#include<stdio.h>
void main()
{
            int rollno=12;
            printf(" The roll no is %f", rollno);
}
```

the output for this program is:
The roll no is 12.000000

Now in above example it would be nice to suppress the extra zeros in the output, and the printf function includes a way to do just that.

Let's rewrite the above program using the field width specifier:

```
#include<stdio.h>
void main()
{
            int rollno=12;
            (" The roll no is %.2f", roll);
}
```

the output for this program is:
The roll no is 12.00

Thus we see that a number following the decimal point in the field width specifier controls how many characters will be printed following the decimal point. Also the number preceding the decimal point in field width specifier controls the width of the space to be used to contain the number when it is printed.

Following example demonstrates this:

```
            num = 23;
            printf(" A no is %2d.",num);
```

A		n	o		i	s		2	3	.

Downloaded from: sarojpandey.com.np

printf(" A no is %4d.",num);

A		n	o		i	s				2	3	.

printf(" A no is %5d.",num);

A		n	o		i	s					2	3	.

Downloaded from: sarojpandey.com.np

ARRAYS AND STRINGS

ARRAYS

In many of the programming situations, we may require the processing of data items that have common characteristics. Now in such case, it would be easier if we place these data items into one single variable called **array** which is capable of storing number of data, sharing common name.

The individual data items can be characters, integers, and floating-point numbers and so on. They must all, however be of the same type and the same storage class. The individual data items in an array are called **array elements.**

Each array element is referred to by specifying the array name followed by one or more subscript enclosed in square brackets. Each **subscript or index** must be expressed as non – negative integer.

Thus, we represent array containing n elements as:

x[n]

where, x is array name

n is subscript. &

x[n] has its array element as x[0], x[1], x[2],…......,x[n-1].

The value of each subscript can be expressed as an integer constant, integer variable or a more complex integer expression.The number of subscript determines the dimensionality of the array.

Example,

x[i] ⟶ refers to an element in the one dimensional array x
y[i] [j] ⟶ refers to an element in a two – dimensional array y.

Other higher dimensional can be formed by adding additional subscripts in the same manner like, z[i] [j] [k].

Definition of Array

An array can be defined as a group of homogeneous elements sharing a common name.

Each array element is identified by the array name followed by its index enclosed in square bracket. These elements are stored in consecutive memory locations.

ARRAY DECLARATION

In general terms, a one – dimensional array definition can be expressed as follows:

Syntax:- **data – type variable – name [expression]**

Where, data - type refers to variable type variable – name is a valid name of array.
expression refers to the valid positive – valued integer expression and this indicates the number of array elements.

Examples of one dimensional array are:

int x[10]; ⟶ 10 – element integer array
char name[20]; ⟶ 20 – element character array
float list [35]; ⟶ 35 – element float array

INITIALIZING AN ARRAY:

Downloaded from: sarojpandey.com.np

Array can also be initialized as other variables during its declaration. This means array declarations can include the initial values as per requirement. But the initial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas.

The general form is:

data_type array_name[expression] = {val1,val2…..valn};

Where, val1 refers to the value of first array element, val2 refers to value of second element and so on.

The appearance of the expression, which includes the number of array elements, is optional when initial values are present.

Example of array initialization:

```
int num[5] = { 1,2,3,4,9,12};
```

Now the result of this assignment, in terms of individual array elements is as follows:

```
num[0] = 1
num[1] = 2
num[2] = 4
num[3] = 9
num[4] = 12
```

The array size need not be specified explicitly when initial values are included as part of array declaration. The array size will automatically be set equal to the number of initial values included within the declaration.

Thus the array num can be define and initialized as:

```
int num[ ] = {1,2,4,9,12};
```

Other examples:-

```
float x[4] = {10.5, 15.2, 2.0, -4.5};
or
float x[ ] = {10.5, 15.2, 2.0, -4.5};
```

<u>ARRAY ELEMENTS IN MEMORY</u>

Let's consider following array declaration:

```
int y[8];        // One dimensional array
```

When we declare this array, the complier reserves 16 bytes of memory in the computer 16, because each of the 8 integers occupy 2 bytes [8×2 = 16]

And since the array is not initialized, all eight values present in it would be garbage values. Whatever be the initial values all the array elements would always be present in the contiguous memory locations. The arrangement of array elements would be a shown below:

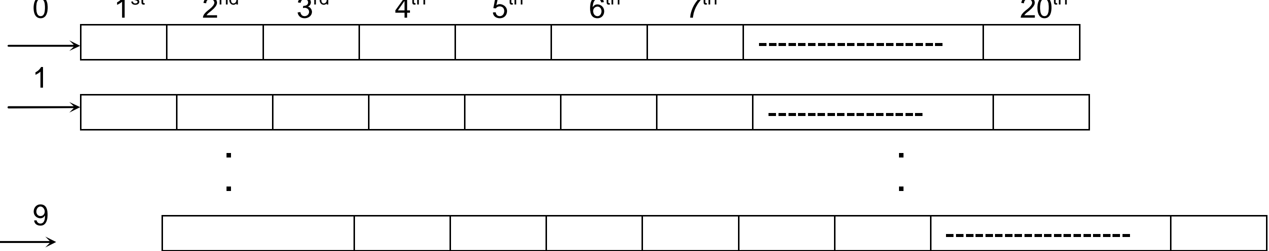
10	24	28	35	40	-2	5	-77
5001	5003	5005	5007	5009	5011	5013	5015

Let's have little knowledge about 2 – dimensional array:

Downloaded from: sarojpandey.com.np

int x[10] [20]

In above example x is two – dimensional array having 10 rows and 20 columns and this will be capable of holding 10×20 = 200 elements. The two dimensional array can be represented as:



PROCESSING AN ARRAY:

C does not permit the use of single operation which involves entire arrays. Thus if a and b are two similar arrays, assignment operations, comparison operations etc must be carried out on an **element – by – element** basis. This can be achieved by using loop where each pass or iteration through the loop is used to process one array element. Thus the no. of iterations through the loop will be equal to the no. of array elements to be processed.

a) Following program inputs the numeric array and prints the entered numbers

```
#define SIZE 4
void main()
{
          int num[SIZE}, i;
          printf("please enter %d integers\n", SIZE);
          for(i=0;i<SIZE;i++)
          scanf("%d",&num[i]);
          printf("\n");
          printf("the entered array elements are:\n");
          for(i=0;i<SIZE;i++)
          {
               printf("%d",num[i]);
               printf("\n");
          }
          getch();
}
```

Output:

```
Please enter 4 integers
1
12
75
60
the entered array elements are:
1
12
75
60
```

b) Following program reverses the entered numbers

Downloaded from: sarojpandey.com.np

```
#define SIZE 4
void main()
{
            int num[SIZE], i;
            printf("please enter %d integers\n", SIZE);
            for(i=0;i<SIZE;i++)
            scanf("%d",&num[i]);
            printf("\n");
            printf("the reversed array elements are:\n");
            for( I = SIZE-1; i>=0; i--)
            {
                  printf("%d",num[i]);
                  printf("\n");
            }
            getch();
}
```

Output:

```
Please enter 4 integers
1
12
75
60
the reversed array elements are:
60
75
12
1
```

c) Following program reads a one – dimensional character array, converts the elements to uppercase and displays the converted array.

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
            char t, text[80];  //to store the text up to 80 characters long
            int i=0, tag;
            printf(" Enter any text below:\n");
            do
            {
                  t =getchar();
                  text[i]=t;
                  i++;
            }
            while(t!='\n');
            tag=i;                /* to store the maximun number
                                of characters entered by users*/
            for(i=0;i<tag;i++)
            putchar(toupper(text[i]));  /*  the  library  function  toupper()  is
            used to convert the text to upper case */
            getch();
}
```

Downloaded from: sarojpandey.com.np

PASSING ARRAYS TO FUNCTIONS

Before, when an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. For example if a[10] is an array and sort_a(a) is the function for sorting array, then we are passing the address of a[0] to the function sort. Thus the function uses this address for manipulating the array elements.

To pass an array to a function, the array name must appear by itself with or without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written in same, manner but it must be declared as array within formal argument declarations. When one-dimensional array appears as formal argument in the function the array name is written with a pair of empty square brackets. The size pf array need not be written within formal argument declaration.

Example:

```
void diagonal(int a[10][10], int m);
void main()
{          int i,j,n,a[10][10];
           clrscr();
           printf("enter degree of matrix n=");scanf("%d",&n);
           printf("Enter matrix elements below\n");
           for(i=0;i<n;i++)
              for(j=0;j<n;j++)
                 scanf("%d",&a[i][j]);
           diagonal(a,n);
           getch();
}
void diagonal(int a[10][10], int m)
{          int i;
           float sum=0;
           for(i=0;i<m;i++)
              sum+=a[i][i];
           printf("\nSum of diagonal elements is %.2f",sum);
           getch();
}
```

Output

```
enter degree of matrix n=3
Enter matrix elements below
1 2 3
1 1 1
4 5 6
Sum of diagonal elements is 8.00
```

MULTIDIMENSIONAL ARRAY

Multidimensional array consists of more than one subscript or pair of square brackets in their declarations. That means a two –dimensional array will require two pair of square brackets, and a 3-dimensional array will require three pairs of such square brackets and so on.

Thus in general a multidimensional array definition can be written as,

data_type array [exp1][exp2]…….[expn];

Where data _type is the data type of the array
array is the array name &
exp1…..expn are positive valued integers.

Examples

Downloaded from: sarojpandey.com.np

1) float table[20][30];
2) double record[100][20][30];
3) double record[x][y][z];

We can initialize the three – dimensional array as following :

```
int a[3][2[2] =   {
                      {
                            {2,4},
                            {2,3},
                      },

                      {
                            {3,5},
                            {5,7},
                      },
                      {
                            {7,8},
                            {9, 10},
                      }
                  };
```

In above example one-dimensional array is constructed first. Then two such one-dimensional arrays are placed one below another to give a 2D array containing two rows. Then three such 2D arrays are placed one below another to give a 3D array containing three 2D arrays.

PROCESSING OF MULTIDIMENSIONAL ARRAYS

Multidimensional arrays are also processed in the same manner as one dimensional arrays on the element by element basis.Suppose we have a 2 matrices as given below:

```
   2  2             3  1
A          &     B
   2  2             3  2
```

now to add above matrix (which is a 2D array) we need to access every element of matrix A and B and perform addition by element by element basis and place the sum in third matrix C.

General form of matrix for above case is :

```
a11   a12        b11   b12        c11   c12

a21   a22        b21   b22        c21   c22
```

Besides these, we can also perform various arithmetic operation and comparisons of elements of multidimensional matrices

Two Dimensional Array

The number of subscripts in an array determines its dimensionality. Thus it is possible for arrays to have two or more dimensions. A two dimensional array is an array having 2 subscripts. It is also known as a **matrix.**

Example of 2D array is:

int student[10][20];   This means a 2D array which can hold up to 20*10=200 elements.

Downloaded from: sarojpandey.com.np

Initializing a 2D Array

Example

```
int  a [2] [2]=  {
                    {11, 22 } ,
                    {99, 20 }
                 };
```

OR

```
int  a [ ] [2]=  { 11, 22, 99, 20 };
```

STRINGS

Strings are arrays of characters i.e. they are characters arranged one after another I memory.
To mark the end of string, C uses the null character'\0'. Strings in C are enclosed with double quotes

For e.g. "My name is Peter"

Strings are stored as ASCII (American Standard Codes for Information Interchange) codes of the characters that make up the string, appended with zero (which is ASCII value for NULL)

For e.g.	Alphabets	Ascii value
	a	97
	A	65

Initializing string

'String initialization must have following form:

e.g. char month1[ ] = {'J', 'a', 'n', 'u', 'a', 'r', 'y'};

OR

char month1[ ] = {"January"};

J
a
n
u
a
r
y
\0

Indicating end of string → NULL character

Array Of strings

We often use lists of character strings such as names of students in a class, list of names of employees in an organization, list of places etc. A list of names can be treated as array of strings and a two dimension character array **name[5][6]** can be used to store a list of 5 names, each of length not more than 6 characters which is shown below:

	0	1	2	3	4	5
0	r	a	m	\0		
1	f	a	r	I	\0	
2	s	a	r	a	d	\0
3	p	e	t	e	r	
4	r	i	t	a	\0	

Thus in above example name [5][6] id an array of strings.

String handling functions

C library supports large number of string handing functions that can be used to carry out many of the string manipulations .the definitions for the string handing functions are available in the header files **string.h.**

Following are the most commonly used string handling functions:

Downloaded from: sarojpandey.com.np

Function	Action
a) strcat()	concatenates two strings
b) strcmp()	compares two strings
c) strcpy()	copies one string to another
d) strlen()	finds the length of string

strcat() Function

This function concatenates two strings i.e. appends one string at the end of another. It accepts 2 strings as parameters and stores the content second string at the end of the first. The first string should be capable of holding the second string after concatenation.

```
#include<string.h>
void  main()
{
            char string1[20]="Flash";
            char string2[ ]="Light";
            strcat(string1,string2);
            printf(string1);
            getch();
}
output:     FlashLight
```

strcmp() function

It compares two strings. It is useful while writing programs for constructing and searching strings arranged in a dictionary. This function compares the strings on character by character basis.The function accepts two strings and returns an integer whose value is:

LESS THAN ZERO	if string1< string2
EQUAL TO ZERO	if string1= string2
GREATER THAN ZERO	if string1>string2

```
void main()
{           char str1[10],str2[10];
            int result;
            scanf("%s %s ",str1, str2);
            if(result<0)
                  printf("str1<str2");
            else if(result==0)
                  printf("str1=str2");
            else
                  printf("str1>str2");
            getch();
}
output
Hello  hello
str1< str2
```

strcpy() Function

This function copies one string to another. It accepts two strings as parameters and copies the second string to the first string, character by character into first one, up to the last and including the null character of second string. Here also the size of first character array should be greater than that of second one.

```
void main()
{           char str1[10]= "Ram";
```

Downloaded from: sarojpandey.com.np

```
            char str2[ ]= "Shyam";
            printf("before strcpy function\n");
            printf("%s\n",str1);
            strcpy(str1,str2);
            printf("after strcpy function\n");
            printf("%s\n",str1);
}
```

Output

```
before strcpy function
Ram
after strcpy function
Shyam
```

strlen() Function

It returns an integer which denotes the length of string passed. The length of string is the number of characters present in it, excluding the terminating null character.

```
void main()
{           char str[ ]= "Hello";
            printf("Length=%d \n",strlen(str));
            getch();
}
```

Output

```
Length=5
```

Downloaded from: sarojpandey.com.np

FUNCTIONS

INTRODUCTION

A function is a self-contained program segment or the block of statements that perform some specific, well- defined task. Every C program comprises of one or more such functions. Among these functions one must be called main (). The execution of any program always begins by the executing the instructions in the main () function.

In a program there may be any number of functions and their definition can appear in any order but they should be independent of one another. We need to **access** the function (which is known as **calling a function)** in order to carry out the intended task.

A single function can be accessed or called from different places within a program. After execution of the intended task, the control will be returned to the point from which the function was accessed or called.

Simple C function program:

```
void message()
{       printf("\n This is the statement inside the function message");
}

void main()
{       printf("\n This is the statement  in main before a function call");
        message();
        printf("\n This is the statement in main after a function call");
}
```

Output: -

This is the statement in main before function call
This is the statement inside the function message
This is the statement in main after a function call

In above e.g., we are calling the function message ( ) (which is user defined function) from the main ( ) function. This means the program control is passed to the function message ( ). The task of main ( ) function is stopped for a while, and the message ( ) function starts to execute it's statements. After the message ( ) function completes its task the control returns back to the main ( ) function and continues its normal job again by executing its statements at the exact point where it left off before. Thus here main ( ) becomes a **calling function** and message ( ) becomes a **called function**.

Next Example:-

```
message1()
{       printf("\n I  am in  message1");
}
message2()
{       printf("\n I am in  message2");
}
main()
{       message1();
        message2();
}
```

Output: -

I am in message1

Downloaded from: sarojpandey.com.np

I am in message2
C supports the use of **library functions**, which are used to carry out a number of commonly used operations or calculations. Besides this, C also allows the programmers to define their own functions to carry out different types of individual tasks called **user-defined functions**.

LIBRARY FUNCTIONS

C- programming language comprise of no. of library functions that perform some specific task or operations.

Functions may:

- Return a data item to their access point, or
- Indicate whether the function is true or false by returning 1 or 0, or
- Perform specific operations on data items but return nothing.

There are numerous library functions in C like,

1. library functions that carryout standard input/output (like read write characters, numbers, etc).
   e.g. printf( ), scanf( ), putchar(c)
2. function that carryout various mathematical calculations
   e.g. log(d) - function that return natural logarithm
   sin(d)
   cos(d)
   cosh(d)
   sqrt( )
   pow (x1, x2) return x1, raised to x2 power.
3. function that operats on strings
   e.g. strcmp (strings, strings) ,
   tolower (c)
   toupper(c)

There may be several other types of such functions. Library functions that are functionary similar are usually grouped together as object programs in separate library files, which are supplied as a part of each c compiler.

How to access a library function?
A library function is accessed by writing the function name, followed by a list of arguments that represent information being passed to the function. The arguments must be enclosed in parentheses and separated by comma. The arguments can be constants, variables names, or other complex expressions. There must be parentheses even if there are no arguments.

USER DEFINED FUNCTIONS:

Those functions that are created by user, where the user has the freedom to choose the function name, return data type and arguments are called user-defined functions.

Need for user defined functions:-
We already know that, main ( ) function is specially recognized by function in c. Every program must have main function to indicate where the program has begin its execution. We can write program utilizing only main function. But this may result in too complex or too large program. Due to this program difficulty, now if program is divided into functional parts, then each part may be independently coded and later on combined into single unit. Thus such subprograms are known as functions.

Downloaded from: sarojpandey.com.np

Advantages of Function

1) Saves time and resources

Use of function avoids the writing of same code again and again. For example if we want to calculate the area of the rectangle in your program, we write a code for it. Again later on, if we need to calculate the area of some other rectangle we may not want to repeat the same code or instructions all over again. Inst4ead we may be willing to move to some section in the program that calculates area and return back to the place from where we left off. Thus this section of code is the *function.* Thus it saves time and resource.

2) Easy and efficient program writing

Functions help to keep the track of what is being done. A program can be divided into several activities and each of these activities can be placed in a different function.

3) Faster testing or debugging

Since separate activities form separate functions, error correction becomes much easier.

4) Functions also make the program easy to understand and use.

5) They also help in easy modification and design of program.

ACCESSING/CALLING A FUNCTION

We can access or invoke or call a function by specifying function name, followed by a list of arguments (also called function parameters) enclosed within the parentheses and separated by commas.

If we do not need to pass any arguments to functions then empty pair of parentheses must follow the function name.

When function call is encountered, the control is passed to the respective function.

FUNCTION ARGUMENTS OR FUNCTION PARAMETERS

A calling function can send a package of values for the called function to operate on, or to control how the function is to operate. Each value passed in this manner is called argument or parameter. There are two types of arguments:

a) Actual argument
b) Formal argument

a) Actual argument

These are arguments that appear in the function call in the calling function. In other words, when a function is called, the parameter specified within the parentheses of the calling statement is known as actual arguments or parameters. They are called so because they are the values that are actually transmitted to the function.

b) Formal or dummy arguments

They appear in the first line of the function definition (or function declarator). There will be one actual argument for each formal argument. Each of the actual arguments and its corresponding formal argument should have same data type i.e. if the actual argument is integer then its corresponding formal argument also should be an integer. The value of each actual argument will be passed to the function and that will be assigned to or stored int the corresponding formal argument.

Downloaded from: sarojpandey.com.np

FUNCTION PROTOTYPE AND DEFINITION

Function Prototype

Function prototype is also termed as **function declaration**. The function prototypes provides following information:

- It provides the name of the function
- It also specifies the type of value returned by the function (this is optional and default is integer)
- It also provides the number and type of arguments that must be supplied in the call to a function.
- Facilitate error checking between calls to a function and the corresponding function definition.

When we write the user -defined function ahead of the main () function, it is not necessary to use the function prototype because the user- defined function will have been defined before the first function access. However, many programmers prefer a "top-down "approach, in which the main () function appears ahead of the user-defined function. Thus in such case the function call within main () will precede the function definition. This can be confusing to the compiler, unless the compiler is first alerted to the fact that the function being called will be defined later in the program. Thus a function prototype is used for this purpose

Syntax for function prototype:

```
return_type function_name( type, type, ………,type);
```

Function definition

The function definition is similar to the function declaration but there is no semicolon. The first line of the function definition is known as the function declarator that is followed by the function body. The declarator and the prototype must use the same function name, number of arguments, argument types and return type.

Syntax for function definition:

```
return_type function_name(  type 1 arg1, type 2 arg 2, type 3 arg 3,... type n arg n)
{
        Body of function;
}
```

where, return_type represents the data -type of value being returned,
function_name represents the name of function, and
type 1, type 2,…type n represent the data types of the arguments arg1, arg2, …arg n.

Function example:

```
#include<stdio.h>
#include<conio.h>
int  rectangle(int length, int breadth);   //function prototype
/* OR   int  rectangle( int, int);  */
void main()
{       int a, l, b;
        printf("\nlength =");      scanf("%d ",&l);
        printf("\nbreadth =");   scanf("%d ",&b);
        a=rectangle(l,b);        // function call in which we are passing values of l and b
        printf("\narea of rectangle=%d",a);
        getch();
```

Downloaded from: sarojpandey.com.np

```
}
int rectangle(int length, int breadth)
{       int area;
        area=length * breadth;
        return (area);
}
```

program output:
length =2
breadth =3
area of rectangle =6

RETURN STATEMENT (RETURNING A VALUE FROM A FUNCTION)

One of the main feature of the function is that it can return a value to the calling function. In order for a function to return a value to the calling function, the function should have a **return** statement. This statement comprises of the keyword called ***return,*** followed by the value to be returned. This value may be an expression, variable, constant, or even another function call. The returned expression may be enclosed within a parentheses; this is often done even though it is not necessary.

When a return statement is executed, it immediately transfers the control back to the calling program. A function call can return only a single value each time it is called. As soon as a return statement is executed, the function terminates.

Like, in the above example, the rectangle () function takes values of length and breadth from the main() function, calculates the area of the rectangle and returns the integer value of the rectangle to the main () function which finally prints the value of area.

If the function does not return any value then it is not necessary to include the return statement in the function.

CALL BY VALUE AND CALL BY REFERENCE

Call by value:

- We pass values of variables to the function from the calling section.
- Each argument is evaluated and its value is used locally in place of the corresponding formal parameter.
- If a variable is passed into the function, the stored value of that variable in the calling environment will not be changed.
- Can return only one value

Example:

```
void change(int a);
void main()
{
        int a = 10;
        printf("Before calling function, a = %d", a);
        change(a);
        printf("After calling function, a = %d", a);
}

void change(int a)
```

Downloaded from: sarojpandey.com.np

```
{       a = a + 10;
}
```

The output will be:
Before calling function, a =10
After calling function, a = 10

Call by reference:

- We pass the address or location number of a variable to the function
- Pointers are used
- If a variable is passed into the function, the stored value of that variable in the calling environment will be changed.
- Can return more than one value at a time

Example:

```
void change(int *a);
void main()
{
        int a = 10;
        printf("Before calling function, a = %d", a);
        change(&a);
        printf("After calling function, a = %d", a);
}

void change(int *a)
{
        *a = *a + 10;
}
```

The output will be:
Before calling function, a =10
After calling function, a = 20

<u>RECURSIVE FUNCTIONS</u>

A recursive function is one which calls itself. This is another complicated idea which we are unlikely to meet frequently. We shall provide some examples to illustrate recursive functions. Recursive functions are useful in evaluating certain types of mathematical function. We may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function.

```
int fib(int num)
/* Fibonacci value of a number */
{     switch(num) {
      case 0:
            return(0);
            break;
      case 1:
            return(1);
            break;
      default:  /* Including recursive calls */
```

Downloaded from: sarojpandey.com.np

```
            return(fib(num - 1) + fib(num - 2));
            break;
      }     }
```

We met another function earlier called power. Here is an alternative recursive version.

```
double power(double val, unsigned pow)
{       if(pow == 0)  /* pow(x, 0) returns 1 */
            return(1.0);
      else
            return(power(val, pow - 1) * val);
}
```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly, otherwise the function calls itself, passing a changed version of the input values. Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the fibonacci function of a moderate size number.

Input Value	Number of times fib is called
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177

CATEGORIES OF FUNCTION

Depending on the presence or absence of arguments and whether the value is returned or not, the function can be categorized as:

- Function without any arguments and return values
- Function with arguments and but no return values
- Function with both arguments and return values
- Function without arguments and but with return values

Function without any arguments and return values

If we do not want to return a value we must use the return type void and miss out the return statement. Here we simply call a function without passing arguments and the called function doesn't have to return any values to the calling function.

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

```
void rect_area()          // function definition
{
int l, b, area;
```

Downloaded from: sarojpandey.com.np

```
void rect_area();    // function declaration
rect_area();         // function call
getch();
}
```

```
printf("Enter l and b:");  scanf("%d %d", &l, &b);
area = l * b;
printf("The area of rectangle is %d", area);
}
```

Function with arguments and but no return values

This type of function has one-way communication. Here, an argument is passed from the calling function to the called function, but there is no need of return statement in the called function.

```
#include <stdio.h>
#include<conio.h>
void main ()
{
void rect_area(int, int);    // function declaration
int l, b;
printf("Enter l and b:");  scanf("%d %d", &l, &b);
rect_area(l, b);          // function call
getch();
}
```

```
void rect_area(int l, int b)          // function
definition
{
int area;
area = l * b;
printf("The area of rectangle is %d", area);
}
```

Function with both arguments and return values

This type of function has two-way communication. Here, an argument is passed from the calling function to the called function, and there will also be return statement in the called function.

```
#include <stdio.h>
#include<conio.h>
void main ()
{
int rect_area(int, int);    // function declaration
int l, b, a;
printf("Enter l and b:");  scanf("%d %d", &l, &b);
a = rect_area(l, b);          // function call
printf("The area of rectangle is %d", a);
getch();
}
```

```
int rect_area(int l, int b)           // function
definition
{
int area;
area = l * b;
return area;
}
```

Function without arguments and but with return values

This type of function also has one-way communication. Here, an argument is not passed from the calling function to the called function, but there is need of return statement in the called function.

```
#include <stdio.h>
#include<conio.h>
void main ( )
{
int rect_area(void);    // function declaration
int a;
printf("Enter l and b:");  scanf("%d %d", &l, &b);
a = rect_area( );          // function call
printf("The area of rectangle is %d", a);
getch();
}
```

```
void rect_area( )          // function definition
{
int area;
printf("Enter l and b:");
scanf("%d %d", &l, &b);
area = l * b;
return(area);
}
```

Downloaded from: sarojpandey.com.np

Downloaded from: sarojpandey.com.np

LOCAL, GLOBAL AND STATIC VARIABLES

Local variables

Local variables are those variables, which are declared inside a function in which they are to be utilized. They are created when we call a function and destroyed automatically when the function is exited. Thus such variables are private or local to the function in which they are declared. The values of such variables cannot be changed by what happens inside the other functions in the program i.e. the same named variable can be used in some other functions without creating any confusion.

Following program illustrates this concept:

```
#include<stdio,h>
modify_n();

void main()
{       int n ;         //local variable
        n=10;
        printf("the value of n is %d  \n", n);
        modify_n( );
        printf("the value of n after function call is %d  \n", n);
        getch();   }
modify_n()
{
        int n;          //local variable
        n = 20;
        return n;
}
```

Output:
the value of n is 10
the value of n after function call is 10

In above example, we have declared a variable n in the function main () and assigned it the value 10. The program then calls the function modify_n() which creates its own local variable n, and assigns the value 20 to it. This assignment has no effect, however, on the variable named n that was created in the main function.

When the control returns to the main function, the variable n- the one local to main function – still contains the value 10. A function does not terminate its execution when calling another function; it suspends it. Therefore the variable n local to the main function remains in existence while the function modify_n is executing.

Global variables

Global variables are those variables which are alive and active throughout the entire program. Such variables can be accessed and used by any function in the program and their values can be altered. Thus the global variables help to establish a two- way communication among different functions. In contrast to the local variables, these variables are not declared within a specific function, instead they are declared outside of all the functions in the program.

The previous program is modified as follows using n as the global variable

```
int n;          //global variable
```

Downloaded from: sarojpandey.com.np

```
modify_n();
void main()
{       n=10;
        printf("the value of n is %d  \n", n);
        modify_n( );
        printf("the value of n after function call is %d  \n", n);
        getch();
}

modify_n()
{
        n = 20;
        return n;
}
```

Output:
the value of n is 10
the value of n after function call is 20

Here n is declared as the global variable so both main() and modify_n() functions are able to change its content.

Here is another example to illustrate the use of global variable:

```
int length,breadth,area;
get_number();
void calc_area();

void main()
{
        clrscr();
        printf(" \nEnter length of rectangle:");
        length=get_number();
        printf(" \nEnter breadth of rectangle:");
        breadth=get_number();
        calc_area();
        printf("\n The area is %d",area);
        getch();
}

get_number()
{
        int a;
        scanf("%d",&a);
        return(a);
}

void calc_area()
{
        area=length*breadth;
}
```

Output:

Downloaded from: sarojpandey.com.np

Enter length of rectangle: 3
Enter breadth of rectangle: 4
The area is 12
In the above program, three global variables are declared- length, breadth, and area. The values of length and breadth are read through the get_number() function. The get_number() function does not refer to any global variables, because it is called twice. During the execution, there is no way the function can know for which variable a particular value is being read. There fore, it uses the one way communication power of the return statement.

The area is calculated by the calling the calc_area() function. This function uses the values that already have been placed in the length and breadth in order to calculate area. The result is placed in the global variable area. This variable is accessible by the main function, so its value can be printed by the printf statement located in the main function.

Static variables
Like the local variables Static variables are also local to the block in which they are declared. The basic difference is that they don't disappear when the function is no longer active. Their values persist. if the control comes back to the same function again the static variables have the same values they had last time around.Let us illustrate this by following example:

```
increase();
void main()
{       clrscr();
        increase();
        increase();
        increase();
        getch();
}
increase()
{       int i=1;    //local variable
        printf(" \n i = %d",i);
        i++;
}
output:
i=1
i=1
i=1
```

```
increase();
void main()
{       clrscr();
        increase();
        increase();
        increase();
        getch();
}
increase()
{       static int i=1;  // static variable
        printf("\n i = %d",i);
        i++;
}
output:
i=1
i=2
i=3
```

In above example when i is declared as local variable, each time the increase () function is called it is re- initialized to 1.when the function terminates, I vanishes and it new value i.e. 2 is lost. So the result remains 1 no matter how many times a function increase is called.

But when the variable i is declared as the static variable, it is initialized to 1 only once. It is never initialized again. During the first call of increase(), i is incremented to 2. Since i is **static**, this value persists. Now when the increase() is called next time, i is not re-initialized to 1, but its old value 2 is retained. Thus the current value of i (i.e. 2) is printed and i is incremented to 3. When the increase() is called for the third time , the current value of i i.e. 3 is printed and once again i is increased to 4. Hence the statement static int i=1 is executed only once irrespective of how many times the same function is accessed.

Basic Rules associated with static variables are:-

1) Initial values must be constants not expressions.
   Like, static float i=3.0;

Downloaded from: sarojpandey.com.np

2) The initial value is assigned to the variable at the beginning of the program execution. The variables retain these values throughout the life of the program, unless different value is assigned during the computation process.

3) The static variables will have there default values equal to zero if their declarations do not include explicit initial values.

Downloaded from: sarojpandey.com.np

STRUCTURES AND UNIONS

INTRODUCTION TO STRUCTURES

A structure is a collection of one or more variables, possibly of different data types, grouped together under a single name for convenient handling. Structures help to organize data, particularly in large programs, because they allow a group of related variables to be treated as a single unit.

Consider an example . In an organization, an employee's details (i.e., his name , address, designation, salary etc.) have to be maintained. One potential method would be to create a two-dimensional array, which would contain all the details. But in this case, this is not possible because the parameters are of different types, i.e., name is of type char, salary is of type float , and so on. There is a simple solution to this problem- structures.

STRUCTURE DECLARATION

Let us create a structure to illustrate the above-mentioned example

```
struct employee  {
                        char *name;
                        char *address;
                        char *desig;
                        float salary;
                } emp1, emp2;
```

The keyword struct introduces a structure declaration, which is a list of declarations enclosed in braces. The variables declared within the structure declaration are called its members. The struct declaration defines a datatype, and the variables emp1 and emp2, which follows the struct declaration, is defined as variables of this type and storage space is set-aside for them. This is equivalent to declaring variables of type int or char.

A structure declaration that is not followed by a list of variables reserves no storage space, it merely describes the template. In other words, the structure is only defined, not declared.

The word following the keyword struct (employee in this case) is called a structures tag, and is optional. It is optional because it only names this type that has been defined by the structure declaration, and can be used as a substitute later. For example, at a later stage, if a new variable emp3 of type struct employee has to be declared, it could be declared as

```
struct employee emp3;
```

There is no need to define the entire structure all over again.

The structure below is also equivalent to above-example:

```
struct  {
                        char *name;
                        char *address;
                        char *desig;
                        float salary;
                } emp1, emp2, emp3;
```

Downloaded from: sarojpandey.com.np

INITIALIZATION OF STRUCTURE

Like other variables and arrays, structure variables can also be initialized where they are declared. The format is quite similar to that used to initiate arrays. Let us initialize a structure variable emp3 in the above-mentioned example:

```
        struct employee  {      char *name;
                                char *address;
                                char *desig;
                                float salary;
                        } emp1, emp2;
struct employee emp3 = {"George", "Massachusetts", "15000", };
```

ACCESSING STRUCTURE ELEMENTS

After the declaration of structure name and structure variables, let us see how the elements of the structure can be accessed. A member of a structure is always referred to and accessed using the structure name.

Structure name. member name

This structure member access operator ".", connects the structure name and the member name. For example, to print the name of the employee,

```
printf("The employee's name is %s",emp1.name);
```

NESTED STRUCTURES

Nested structures are nothing but structures within structures. Let us take the same example discussed above- the address in the employee structure can be a structure by itself, which stores the house number, the street name, the area name and the pincode.

```
        struct emp_add {        int no;
                                char *street;
                                char *area;
                                int pincode;
                        };
```

The above definition could be used as a template to declare a variable address in the employee structure

```
struct  employee {              char *name;
                                struct emp_add address;
                                char *desig;
                                float salary;
                        } emp1, emp2;
```

or, the two definitions could even be consolidated as:

```
        struct employee {       char *name;
                                struct emp_add {        int no;
                                                        char *street;
                                                        char *area;
                                                        int pincode;
                                                } address;
                                char *desig;
                                float salary;
                        }emp1,emp2;
```

Downloaded from: sarojpandey.com.np

ARRAYS OF STRUCTURE

Let us consider the above-mentioned example (of employees in an organization) to illustrate this concept. Assume there are five employees in an organization whose records are to be maintained. The declaration goes like this

```
struct employee {    char *name;
                     char *roll_no;
                     int salary;
                } emp1[5];
```

This declaration defines five instances of the variable emp1, of type struct employee. They could be initialized as:

```
struct employee {    char *name;
                     char *roll_no;
                     int salary;
           } emp1[5] =
                     {
                          {“James”,   “e01”,   10000},
                          {“Mark”,   “e02”,    9000},
                          {“David”,  “e03”,    11000},
                          {“Victoria”,  “e04”,   15000},
                          {“Jeffery”,   “e05”,   13000}
                     };
```

As in the case of other datatypes, the index 5 does not have to be specified, as the initialization by itself will determine its size.

Note that in the initialization part of the above declaration, every particular employee’s details have been enclosed in braces. This is not necessary, but it improves clarity of code. In case a member is not to be initialized , the value for it can be omitted. For example, assume the name and salary of an employee are not known, the declaration will be

```
{  ,  e06,    }
```

Any member of the structure can be accessed using the index number along with the structure name. For example, if one wants to print the name of the first employee,
Printf(“The name of the first employee is %s”, emp1[0].name);

We can declare as many structures variables as we want to fulfill our requirement. Now when we need to store huge number of information like storing information of 100 or more students, employees etc, we can use structure variables of type array.

For example,

```
struct   student
         {
              char name[20];
              int rollno;
              int grade;
         }
struct  student  s[50];
```

In above example, the structure variable s is an array of structure that may contain as many as 50 elements. Each element is a structure of type student.

Thus if we want to access the name of 1st student i.e. s[0] , we would write s[0].name.
And to access rollno and grade we would write:

```
s[0].rollno
s[0].grade
```

Downloaded from: sarojpandey.com.np

Similarly to access the information of 2nd student we would write

s[1].name
s[1].rollno
s[1].grade

The syntax used to reference each element of array s is similar to the syntax used for normal arrays. The only difference is that every element of array s is a structure.

In an array of structure all elements of the array are stored in adjacent memory locations. Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations we can easily guess the arrangement of array of structures in memory.

For example, s[0]' s name, rollno and grade in the memory would be immediately followed by s[1]'s name, rollno and grade and so on.

POINTERS TO STRUCTURES

Pointers to structures are just like pointersto other ordinary variables. The declaration

```
struct employee  {
                        ……
                        ……
                        ……
               } *emp1;
```

declares a pointer of type struct employee. If emp1 points to a structure, then emp1 is the pointer to the structure, and either (emp1).name or emp1->name might be used to access the structure members. Here, -> is the member access operator for pointers, which is nothing but a hyphen '-' followed by the greater than symbol '>'.

The arithmetic for a pointer of this type is same as for any other datatype, and any manipulation that can be done through pointers to variables of other types is also possible for structures.

STRUCTURES AND FUNCTIONS

Structures may be passed to functions either by value or by reference. However , they are usually passed by reference, i.e., the address of the structure is passed to the function. Let us consider the employee example for instance. Given the structure declaration below

```
struct employee   {           char *name;
                              float basic;
                              float bonus;
            } emp1;
```

assume that the gross salary has to be calculated by adding basic with bonus. The following function takes the address of the structure as its parameter, adds up the two (basic and bonus) and sends the gross salary as output.

Example:

```
#include<stdio.h>

struct employee {
              char name[20];
              float basic;
              float bonus;
         }emp1;

     main()
```

Downloaded from: sarojpandey.com.np

```
{       float gross, grosscalc();
        printf("\n Enter the employee's name ");
        scanf("%s",emp1.name);
        fflush(stdin);
        printf("\n Enter the employee's basic");
        scanf("%f",&emp1.basic);
        fflush(stdin);
        printf("\n Enter the employee's bonus");
        scanf("%f",&emp1.bonus);
        fflush(stdin);
gross=grosscalc(&emp1);     /*the address of the array is passed to the function, call by reference */
printf("The employee's %s's gross is Rs. %0.0f",emp1.name,gross);
}

float grosscalc(struct employee *emp)
                /*emp is a pointer of type struct employee */
{
        return(emp->basic +emp->bonus);
}
```

The following function reads input from the user , initializes each of the members and returns the entire structure.

```
struct employee init()
 /* returns a value of type struct employee */
{       struct employee emp;
        printf("\n Enter the employee's name");
        scanf("%s",emp.name);
        fflush(stdin);
        printf("\n Enter the employee's basic");
        scanf("%f",&emp.basic);
        fflush(stdin);
        printf("\n Enter the employee's bonus");
        scanf("%f",&emp.bonus);
        fflush(stdin);
        return emp;
}
```

USER- DEFINED DATATYPES

C allows programmers to define new datatypes equivalent to the existing system datatypes using the typedef statement. Let us take the employee structure for example. Assume that the employee's date of birth has to be included in the existing structure. The declaration would be

```
Typedef struct {
                int dd;         /* date  */
                int mm;/* month  */
                int yy;         /* year    */
        } dob;
```

Given this declaration, dob can be used in the structure just as if it were another datatype, like int, char and so on. For example,

```
Dob  emp_date_of_birth;
```

Downloaded from: sarojpandey.com.np

UNIONS

Consider a situation. "Silicon Info Solution" is a computer related company offering many courses to students. It offers two major courses, and a lot of minor ones. The major courses are classified as majors 1 and 2, and the minors are classified into 'others'. When a student enrolls for a particular course, his details are entered into a file and maintained till he completes the courses. The student structure looks like this

```
struct student {  char  *name;
                        int rollno;
                  };
```

Apart from this, in the students' register, one more detail has to be included – the course the particular student is enrolled for. But this poses a problem, since the two major courses are represented by numbers(1 and 2) and the minor courses by a string. The solution to this problem is-union

A union is a variable which may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. A union declaration is similar to a structure declaration.

```
Union courses {         int major;
                  char *minor;
                  } course1;
```

Here , course1 will be large enough to hold the largest of the two types, int or char. Any one of these variables might be stored in course1 and then used in expressions. Only condition is that the usage must be consistent, the type retrieved must be the type stored last.

In the students' register, the structure declaration would be

```
struct student {  char *name;
                  int rollno;
                  Union course course_no;
              } student1;
```

This would represent no storage problems as the member course_no has been declared as a union. Course_no can now contain any value – char or int. the following program illustrates the usage of unions.

```
#include<stdio.h>
union course    {         int major;
                          char minor[10];
                };
struct student  { char name[20];
                          int rollno;
                          union course course_no;
                } student1;
main()
{       char c_name;
        printf("\n Enter the name of  the student");          scanf("%s",student1.name);
        printf("\n Enter the roll no . of the student");      scanf("%d",&student1.rollno);
        printf("\n Enter the course ('M' for major or 'm' for minor)");     scanf("%c",&c_name);
        if(c_name=='M')
        {       printf("\n Enter the course ('1' or '2')");
        scanf("%d", &student1.course_no.major);
        }
        else
                strcpy(student1.course_no.minor, "others");
}
```

SUMMARY

Downloaded from: sarojpandey.com.np

A structure is a collection of one or more variables, possibly of different types, grouped together for convenient handling.
Structures help to organize data because they allow a group of variables to be treated as a single unit.
A structure declaration starts with the keyword struct, which introduces the declaration.
The structure tag, which follows the keyword, is optional. It is only a label and not an actual structure in memory.
The declaration then follows with a list of variables enclosed within curly braces. These variables are called members.
Members can be accessed using the member access operator '.' .
The struct declaration defines a new datatype.
Nested structures are nothing but structures within structures.
The 'typedef' statement is used to define a new datatype.
Pointers to structures are just like pointers to other variables.
Members can be accessed through the pointer variable using the member access operator '->'
Structures may be passed to functions either by value or by reference.
A union is a variable that may hold objects of different types at different times.
A union declaration is similar to a structure declaration.

Downloaded from: sarojpandey.com.np

POINTERS

Pointers are basically the same as any other variable. However, what is different about them is that instead of containing actual data, they contain a pointer to the memory location where information can be found. This is a very important concept, and many programs and ideas rely on pointers as the basis of their design.

Variable names are not sufficient to provide the manipulations C requires. Local variables are meaningful only within their declaring functions. However, memory addresses are global to all functions. One function can pass the address of local variable to another function, and the second function can use this address to access the contents of first function's local variable. Hence, we use pointer variable, which can store the address or memory location and points to whatever that memory location contains.

USES OF POINTER

- Pointers enable us to access a variable that is defined outside the function.
- Pointers reduce length and complexity of program and increase execution speed.
- Pointers are more efficient in handling data tables.
- The use of pointer array to character string results in saving data storage space in memory.

POINTER DECLARATION AND OPEARATIONS

In C we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. Imagine that we have an int called i. Its address could be represented by the symbol &i. If the pointer is to be stored as a variable, it should be stored like this.

*int *ip = &i;*

int * is the notation for a pointer to an int. Address or ampersand (&) is the operator which returns the address of its argument. When it is used, as in &i we say it is referencing i.

The opposite operator, which gives the value at the end of the pointer, is indirection operator (*). An example of use, known as de-referencing ip, would be

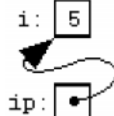
*i = *ip;*

We may think of setting a pointer variable to point to another variable as a two-step process: first we generate a pointer to that other variable, then we assign this new pointer to the pointer variable. We can say (but we have to be careful when we're saying it) that a pointer variable has a value, and that its value is ''pointer to that other variable". This will make more sense when we see how to generate pointer values.

Pointers (that is, pointer values) are generated with the ''address-of" operator &, which we can also think of as the ''pointer-to" operator. We demonstrate this by declaring (and initializing) an int variable i, and then setting ip to point to it:

int i = 5;
ip = &i;

The assignment expression ip = &i; contains both parts of the ''two-step process": &i generates a pointer to i, and the assignment operator assigns the new pointer to (that is, places it ''in") the variable ip. Now ip ''points to" i, which we can illustrate with this picture:



i is a variable of type int, so the value in its box is a number, 5. ip is a variable of type pointer-to-int, so the ''value" in its box is an arrow pointing at another box. Referring once again back to the ''two-step process" for setting a pointer variable: the & operator draws us the arrowhead pointing at i's box, and the assignment operator =, with the pointer variable ip on its left, anchors the other end of the arrow in ip's box.

Downloaded from: sarojpandey.com.np

We discover the value pointed to by a pointer using the operator, *. Placed in front of a pointer, the * operator accesses the value pointed to by that pointer. In other words, if ip is a pointer, then the expression *ip gives us whatever it is that's in the variable or location pointed to by ip.
For example, we could write something like

*printf("%d\n", *ip);*

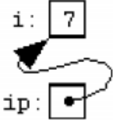
which would print 5, since ip points to i, and i is (at the moment) 5.

The contents-of operator * does not merely fetch values through pointers; it can also *set* values through pointers. We can write something like

**ip = 7;*

which means ''set whatever ip points to to 7." Again, the * tells us to go to the location pointed to by ip, but this time, the location isn't the one to fetch from--we're on the left-hand sign of an assignment operator, so *ip tells us the location to store *to*.

The result of the assignment *ip = 7 is that i's value is changed to 7, and the picture changes to:



If we called printf("%d\n", *ip) again, it would now print 7.

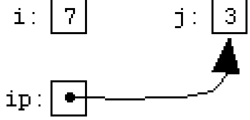
Let's notice the difference between changing a pointer (that is, changing what variable it points to) and changing the value at the location it points to. When we wrote *ip = 7, we changed the value pointed to by ip, but if we declare another variable j:

int j = 3;

and write

ip = &j;

We've changed ip itself. The picture now looks like this:



We have to be careful when we say that a pointer assignment changes ''what the pointer points to." Our earlier assignment

*ip = 7;

changed the value pointed to by ip, but this more recent assignment

ip = &j;

has changed what *variable* ip points to. It's true that ''what ip points to" has changed, but this time, it has changed for a different reason. Neither i (which is still 7) nor j (which is still 3) has changed. (What has changed is ip's value.) If we again call

printf("%d\n", *ip);

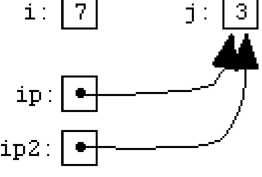
this time it will print 3.

We can also assign pointer values to other pointer variables. If we declare a second pointer variable:

int *ip2;

then we can say ip2 = ip;

Now ip2 points where ip does; we've essentially made a ''copy" of the arrow:

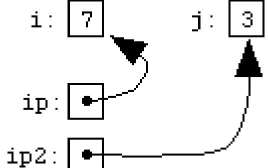


Now, if we set ip to point back to i again:

ip = &i;

Downloaded from: sarojpandey.com.np

the two arrows point to different places:



We can now see that the two assignments

```
ip2 = ip;
```

and

```
*ip2 = *ip;
```

do two very different things. The first would make ip2 again point to where ip points (in other words, back to i again). The second would store, at the location pointed to by ip2, a copy of the value pointed to by ip; in other words (if ip and ip2 still point to i and j respectively) it would set j to i's value, or 7.

It's important to keep very clear in your mind the distinction between *a pointer* and *what it points to*. You can't ''set ip to 5'' by writing something like

```
ip = 5;                /* WRONG */
```

5 is an integer, but ip is a pointer. You probably wanted to ''set *the value pointed to by* ip to 5,'' which you express by writing

```
*ip = 5;
```

Similarly, you can't ''see what ip is'' by writing

```
printf("%d\n", ip);      /* WRONG */
```

Again, ip is a pointer-to-int, but %d expects an int. To print *what* ip *points to*, use

```
printf("%d\n", *ip);
```

Finally, a few more notes about pointer declarations. The * in a pointer declaration is related to, but different from, the contents-of operator *. After we declare a pointer variable

```
int *ip;
```

the expression

```
ip = &i
```

sets what ip points to (that is, which location it points to), while the expression

```
*ip = 5
```

sets the value of the location pointed to by ip.

If you have a pointer declaration containing an initialization, and you ever have occasion to break it up into a simple declaration and a conventional assignment, do it like this:

```
int *ip3;
ip3 = &i;
```

Don't write

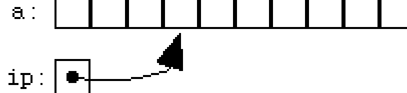
```
int *ip3;
*ip3 = &i;
```

<u>POINTER ARITHMATIC</u>

Pointers do not have to point to single variables. They can also point at the cells of an array. For example, we can write

```
int *ip;
int a[10];
ip = &a[3];
```

and we would end up with ip pointing at the fourth cell of the array a (remember, arrays are 0-based, so a[0] is the first cell). We could illustrate the situation like this:



Downloaded from: sarojpandey.com.np

We'd use this ip just like the one in the previous section: *ip gives us what ip points to, which in this case will be the value in a[3].

Once we have a pointer pointing into an array, we can start doing *pointer arithmetic*. Given that ip is a pointer to a[3], we can add 1 to ip:

```
ip + 1
```

What does it mean to add one to a pointer? In C, it gives a pointer to the cell one farther on, which in this case is a[4]. To make this clear, let's assign this new pointer to another pointer variable:

```
ip2 = ip + 1;
```

Now the picture looks like this:

a:

ip:

ip2:

If we now do

```
*ip2 = 4;
```

We've set a[4] to 4. But it's not necessary to assign a new pointer value to a pointer variable in order to use it; we could also compute a new pointer value and use it immediately:

```
*(ip + 1) = 5;
```

In this last example, we've changed a[4] again, setting it to 5. The parentheses are needed because the unary ''contents of" operator * has higher *precedence* (i.e., binds more tightly than) the addition operator. If we wrote *ip + 1, without the parentheses, we'd be fetching the value pointed to by ip, and adding 1 to that value. The expression *(ip + 1), on the other hand, accesses the value one past the one pointed to by ip.

Given that we can add 1 to a pointer, it's not surprising that we can add and subtract other numbers as well. If ip still points to a[3], then

```
*(ip + 3) = 7;
```

sets a[6] to 7, and

```
*(ip - 2) = 4;
```

sets a[1] to 4.

Up above, we added 1 to ip and assigned the new pointer to ip2, but there's no reason we can't add one to a pointer, and change the same pointer:

```
ip = ip + 1;
```

Now ip points one past where it used to (to a[4], if we hadn't changed it in the meantime). The shortcuts we learned in a previous chapter all work for pointers, too: we could also increment a pointer using

```
ip += 1;
```

or

```
ip++;
```

Of course, pointers are not limited to ints. It's quite common to use pointers to other types, especially char. Here is the innards of the mystrcmp function we saw in a previous chapter, rewritten to use pointers. (mystrcmp, you may recall, compares two strings, character by character.)

```
char *p1 = &str1[0], *p2 = &str2[0];
while(1)
        {
        if(*p1 != *p2)
                return *p1 - *p2;
        if(*p1 = = '\0' || *p2 = = '\0')
                return 0;
        p1++;   p2++;
```

Downloaded from: sarojpandey.com.np

}

The auto-increment operator ++ (like its companion, --) makes it easy to do two things at once. We've seen idioms like a[i++] which accesses a[i] and simultaneously increments i, leaving it referencing the next cell of the array a. We can do the same thing with pointers: an expression like *ip++ lets us access what ip points to, while simultaneously incrementing ip so that it points to the next element. The preincrement form works, too: *++ip increments ip, then accesses what it points to. Similarly, we can use notations like *ip-- and *--ip.

Hence we can summarize that:

- A pointer variable can be assigned address of any ordinary variable. (ip = &i)
- A pointer variable can be assigned value of another pointer variable provided both point the same data type. (ip = ix)
- A pointer variable can be assigned NULL value. (ip = NULL where NULL is symbolic constant having value 0)
- An integer quantity can be added or subtracted from pointer variable.(ip + 3, ++ip, ip--)
- Two pointer variables can be compared provided both point to object of same data type.
- Two pointer variables can be subtracted provided both point to elements of same array.

POINTERS AND FUNCTIONS

Up to this point we have been discussing pointers to data objects. C also permits the declaration of pointers to functions. Access to address in C allows a called function to communicate more than one piece of information back to the calling function. With pass by value, the value of an argument is copied into the parameter. Any change in parameter does not affect the corresponding argument. When we pass the address to a called function, the called function can alter the contents at that location. Pointers to functions have a variety of uses and some of them will be discussed here.

Consider the following real problem. You want to write a function that is capable of sorting virtually any collection of data that can be stored in an array. This might be an array of strings, or integers, or floats, or even structures. The sorting algorithm can be the same for all. For example, it could be a simple bubble sort algorithm, or the more complex shell or quick sort algorithm. We'll use a simple bubble sort for demonstration purposes.

```
/* Program bubble.c */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};
void bubble(int a[ ], int N);

int main(void)
{
   int i;
   putchar('\n');
   for (i = 0; i < 10; i++)
   {
      printf("%d ", arr[i]);
   }
   bubble(arr,10);
   putchar('\n');

   for (i = 0; i < 10; i++)
   {
      printf("%d ", arr[i]);
   }
   return 0;
}

void bubble(int a[], int N)
{
```

Downloaded from: sarojpandey.com.np

```
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (a[j-1] > a[j])
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}
```

The bubble sort is one of the simpler sorts. The algorithm scans the array from the second to the last element comparing each element with the one that precedes it. If the one that precedes it is larger than the current element, the two are swapped so the larger one is closer to the end of the array. On the first pass, this results in the largest element ending up at the end of the array. The array is now limited to all elements except the last and the process repeated. This puts the next largest element at a point preceding the largest element. The process is repeated for a number of times equal to the number of elements minus 1. The end result is a sorted array.

Here our function is designed to sort an array of integers. Thus in line 1 we are comparing integers and in lines 2 through 4 we are using temporary integer storage to store integers. What we want to do now is see if we can convert this code so we can use any data type, i.e. not be restricted to integers.

If our goal is to make our sort routine data type independent, one way of doing this is to use pointers to type void to point to the data instead of using the integer data type. As a start in that direction let's modify a few things in the above so that pointers can be used. To begin with, we'll stick with pointers to type integer.

```
/* Program bubble.c using pointers */

#include <stdio.h>
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};
void bubble(int *p, int N);
int compare(int *m, int *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
void bubble(int *p, int N)
{
```

Downloaded from: sarojpandey.com.np

```
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(&p[j-1], &p[j]))
            {
                t = p[j-1];
                p[j-1] = p[j];
                p[j] = t;
            }
        }
    }
}

int compare(int *m, int *n)
{
    return (*m > *n);
}
```

Note the changes. We are now passing a pointer to an integer (or array of integers) to **bubble()**. And from within bubble we are passing pointers to the elements of the array that we want to compare to our comparison function. And, of course we are dereferencing these pointer in our **compare()** function in order to make the actual comparison. Our next step will be to convert the pointers in **bubble()** to pointers to type void so that that function will become more type insensitive.

POINTER AND ARRAY

There are a number of similarities between arrays and pointers in C. If you have an array

```
int a[10];
```

you can refer to a[0], a[1], a[2], etc., or to a[i] where i is an int. If you declare a pointer variable ip and set it to point to the beginning of an array:

```
int *ip = &a[0];
```

you can refer to *ip, *(ip+1), *(ip+2), etc., or to *(ip+i) where i is an int.

There are also differences, of course. You cannot assign two arrays; the code

```
int a[10], b[10];
a = b;                          /* WRONG */
```

is illegal. As we've seen, though, you *can* assign two pointer variables:

```
int *ip1, *ip2;
ip1 = &a[0];
ip2 = ip1;
```

Pointer assignment is straightforward; the pointer on the left is simply made to point wherever the pointer on the right does. We haven't copied the data pointed to (there's still just one copy, in the same place); we've just made two pointers point to that one place.

The first such operation is that it is possible to (apparently) assign an array to a pointer:

```
int a[10];
int *ip;
ip = a;
```

What can this mean? C defines the result of this assignment to be that ip receives a pointer to the first element of a. In other words, it is as if you had written

```
ip = &a[0];
```

The second facet of the equivalence is that you can use the ‘‘array subscripting” notation [i] on pointers, too. If you write

Downloaded from: sarojpandey.com.np

```
ip[3]
```
it is just as if you had written
```
*(ip + 3)
```
So when you have a pointer that points to a block of memory, such as an array or a part of an array, you can treat that pointer ''as if" it *were* an array, using the convenient [i] notation. In other words, at the beginning of this section when we talked about *ip, *(ip+1), *(ip+2), and *(ip+i), we could have written ip[0], ip[1], ip[2], and ip[i].

The third facet of the equivalence (which is actually a more general version of the first one we mentioned) is that *whenever* you mention the name of an array in a context where the ''value" of the array would be needed, C automatically generates a pointer to the first element of the array, as if you had written &array[0]. When you write something like
```
int a[10];
int *ip;
ip = a + 3;
```
it is as if you had written
```
ip = &a[0] + 3;
```
which (and you might like to convince yourself of this) gives the same result as if you had written
```
ip = &a[3];
```

DYNAMIC MEMORY ALLOCATION

In many programs, we have dimensioned arrays to some maximum size. This technique can waste memory if amount of data is much less than the maximum. Unfortunately, we cannot always predict what the array size will be. A similar situation occurs with character strings. In certain implementations of other structures that hold data, we should be able to activate and free memory locations as needed, or employ dynamic memory allocation.

There are times when it is convenient to allocate memory at run time using **malloc()**, **calloc()**, or other allocation functions. Using this approach permits postponing the decision on the size of the memory block need to store an array, for example, until run time. Or it permits using a section of memory for the storage of an array of integers at one point in time, and then when that memory is no longer needed it can be freed up for other uses, such as the storage of an array of structures.

Allocating memory

When memory is allocated, the allocating function (such as **malloc()**, **calloc()**, etc.) returns a pointer. The type of this pointer depends on whether you are using an older compiler or the newer ANSI type compiler. With the older compiler the type of the returned pointer is **char**, with the ANSI compiler it is **void**.

If you are using an older compiler, and you want to allocate memory for an array of integers you will have to cast the char pointer returned to an integer pointer. For example, to allocate space for 10 integers we might write:
```
int *iptr;
iptr = (int *)malloc(10 * sizeof(int));
if (iptr == NULL)

{ .. ERROR ROUTINE GOES HERE .. }
```

If you are using an ANSI compliant compiler, **malloc()** returns a **void** pointer and since a void pointer can be assigned to a pointer variable of any object type, the **(int *)** cast shown above is not needed. The array dimension can be determined at run time and is not needed at compile time. That is, the **10** above could be a variable read in from a data file or keyboard, or calculated based on some need, at run time.

Because of the equivalence between array and pointer notation, once **iptr** has been assigned as above, one can use the array notation. For example, one could write:

```
int k;
  for (k = 0; k < 10; k++)
    iptr[k] = 2;
```
to set the values of all elements to 2.

Downloaded from: sarojpandey.com.np

Even with a reasonably good understanding of pointers and arrays, one place the newcomer to C is likely to stumble at first is in the dynamic allocation of multi-dimensional arrays. In general, we would like to be able to access elements of such arrays using array notation, not pointer notation, wherever possible. Depending on the application we may or may not know both dimensions at compile time. This leads to a variety of ways to go about our task.

Downloaded from: sarojpandey.com.np

Freeing Memory

Memory allocated with malloc lasts as long as you want it to. It does not automatically disappear when a function returns, as automatic-duration variables do, but it does not have to remain for the entire duration of your program, either. Just as you can use malloc to control exactly when and how much memory you allocate, you can also control exactly when you deallocate it.

In fact, many programs allocate some memory, use it for a while, but then reach a point where they don't need that particular piece any more. Because memory is not infinite, it's a good idea to deallocate (that is, release or *free*) memory you're no longer using. Dynamically allocated memory is deallocated with the free function. If p contains a pointer previously returned by malloc, you can call

```
free(p);
```

which will ‘‘give the memory back’’ to the stock of memory from which malloc requests are satisfied. Calling free is sort of the ultimate in recycling: it costs you almost nothing, and the memory you give back is immediately usable by other parts of your program.

(Freeing unused memory is a good idea, but it's not mandatory. When your program exits, any memory which it has allocated but not freed should be automatically released. If your computer were to somehow ‘‘lose’’ memory just because your program forgot to free it, that would indicate a problem or deficiency in your operating system.)

Naturally, once you've freed some memory you must remember not to use it any more. After calling

```
free(p);
```

it is probably the case that p still points at the same memory. However, since we've given it back, it's now ‘‘available,’’ and a later call to malloc might give that memory to some other part of your program. If the variable p is a global variable or will otherwise stick around for a while, one good way to record the fact that it's not to be used any more would be to set it to a null pointer:

```
free(p);
p = NULL;
```

Now we don't even have the pointer to the freed memory any more, and (as long as we check to see that p is non-NULL before using it), we won't misuse any memory via the pointer p.

Downloaded from: sarojpandey.com.np

Data Files

Introduction

Data stored on a medium, such as disk, are called a file. A computer file is analogous to the customary office file, which stores related information according to title or some other convenient label. In many programming situations, it is convenient to access a file than to enter a succession of individual data items from the keyboard.

Many applications require that information be written to or read from an auxiliary memory device in the firm of a data files. Thus, data files allow us to store information permanently, and to access and after that information whenever necessary.

There are two different types of data files, called stream–oriented (or standard) data files and system–oriented (or low–level) data files. Stream oriented data files are easier to work with and are therefore more commonly used. The basic difference between High level and Low-level disk I/O functions is that High level disk I/O functions do their own buffer management, whereas in low level disk I/O functions, buffer management has to be done explicitly by the programmer. Low-level disk I/O is more efficient both in term of operation and the amount of memory used by the program but high level disk I/O functions are more commonly used in C programs, since they are easier to use than low level disk I/O functions. The high-level file I/O functions are further categorized into text and binary file.

End-Of-File:

It is important to understand that EOF is not a character. It is actually an integer value sent to the program by the operating system and is defined in a header file stdio.h to have a value of 1. No character with this value is stored in a file in a disk. While creating a file, when the operating system finds the last character to the file has been sent, it transmits the EOF signal. Both in text as well as in binary mode, the system keeps track of the total length of the file and will signal an EOF when this length has been reached.

File –Modes explanation

File type Meaning

"r" read only mode specifies that the file can only be read from not written into.

"w" the file is opened in the write only mode. This means that the file can only be written into. Nothing can be read from it. If the file is already certain data in if, the data is erased, and if no file in the specified name exists, it is created and opened in the read only mode.

"a" append mode. The file is opened in the write only mode, the only difference being that the data that is being written into the file gets appended at the end of the file (if the file certain data, it is not truncated or deleted)

"r+" the file is opened in the read + write mode. The file must first be read form, and then it can be written into.

"w+" the file is opened in the write + read mode. The file can first be written into and then read from.

"a+" allows existing data to be read, and new data to be added at the end of file.

A Null value is returned, if the file cannot be opened where fopen() function is used. Eg. When an existing data file cannot be found.

File Handling Functions

Function	Description	Syntax
fopen()	create or opens a file	fopen("filename", "mode")
fclose()	closes a file	fclose(fileptr)
getc()	reads a character from a file	c = getc(fileptr)

Downloaded from: sarojpandey.com.np

putc()	writes a character to a file	putc(c,fileptr)
fprintf()	writes a set of data values to a file	fprintf(fileptr,“ctrl str”, list)
fscanf()	reads a set of data from a file	fscanf(fileptr,“ctrl str”, list)
getw()	reads an integer from a file	getw(fileptr)
putw()	writes an integer to a file	putw(integer,fileptr)
fseek()	sets the position to a desired point in a file	fseek(fileptr, offset, position)
ftell()	gives the current position in the file	ftell(fileptr)
rewind()	sets the position to the beginning of a file	rewind(fileptr)

where, c is a character. fileptr is a pointer to a data file. ctrl str is a conversion specification for the items in the list. offset specifies number of bytes to be moved from the location specified by the position. mode specifies manner in which the data file will be utilized.

Processing a data files

Most data files application requires that a data file be altered as it is being processed. For e.g. in an application involving the processing of customer records, it may be desirable to add new records to the file, to delete existing records, to modify the contents of existing records or to rearrange the records.

The processing of a file involves following steps.

Opening a file

Reading from / writing on to a file and

Closing the opened file

Defining and opening a file

We must specify certain things about a file, if we want to store the file in the secondary memory. They include – filename, data structure and purpose. Filename is a string of characters that make up valid filename for the operating system. Data structure is a framework for storing data and algorithms that implement and perform operations on the structure. Data structure of a file is defined as FILE in the library of standard I/O function definition. All files should be declared as FILE before they are used. Finally we must also specify what we want to do with the file.

The temporary space (buffer) allows information to be read from or written to the data files, ore rapidly them would otherwise be possible.

```
FILE * fp;
```

Where FILE is a special structure type that establishes buffer area and fp is a pointer variable that indicates the beginning of the buffer area (storage area). This file pointer fp contains all the information about the file and is used as communication link between the system and the program.

A data file must be opened before it can be created or processed. This gives the file name with the buffer area. It also specifies how the data file will be utilized i.e. read only, write only or read/write file. The library function fopen() is used to open a file. This is written as:

```
fp = fopen (“file name”, “file mode”);
```

Here, file name and file modes are strings that represent the name of the data file and the manner in which the data file will be utilized.

Closing a Data file

Although you can open multiple files, there's a limit to how many you can have open at once. If your program will open many files in succession, you'll want to close each one as you're done with it; otherwise the standard I/O library could run out of the resources it uses to keep track of open files. Closing a file simply involves calling library function fclose() with the file pointer as its argument:

```
fclose(fp);
```

Downloaded from: sarojpandey.com.np

Calling fclose() arranges that (if the file was open for output) any last, buffered output is finally written to the file, and that those resources used by the operating system (and the C library) for this file are released. If you forget to close a file, it will be closed automatically when the program exits. But, it is good programming practice to close a data file explicitly using the fclose() function, though most 'c' compilers will automatically close a data file at the end of program execution if a call to fclose() is not present.

Example:

```
void main()
{           char c;
            FILE  *fileptr;
            fileptr=fopen("myfile.dat","w+");
            -----------
            -----------
            fclose(fileptr);     }
```

I/O with File Pointers

For each of the I/O library functions we've been using so far, there's a companion function that accepts an additional file pointer argument telling it where to read from or write to. The companion function to printf is fprintf, and the file pointer argument comes first. To print a string to the myfile.dat file we opened in the previous example, we might call

```
fprintf(fileptr, "Hello, world!\n");
```

The companion function to getchar is getc, and the file pointer is its only argument. To read a character from the myfile.dat file we opened in the previous section, we might call

```
int c;
c = getc(fileptr);
```

The companion function to putchar is putc, and the file pointer argument comes last. To write a character to myfile.dat, we could call

```
putc(c, fileptr);
```

Basic file manipulation

When a C program begins execution, it has access to three predefined streams or standard files – stdin, stdout, and stderr. stdin is a constant file pointer corresponding to standard input, and stdout is a constant file pointer corresponding to standard output. Both of these can be used anywhere a file pointer is called for; for example, getchar() is the same as getc(stdin) and putchar(c) is the same as putc(c, stdout). The third predefined stream is stderr. Like stdout, stderr is typically connected to the screen by default. The difference is that stderr is not redirected when the standard output is redirected. Anything printed to stdout is redirected to the file filename, but anything printed to stderr still goes to the screen. The intent behind stderr is that it is the ``standard error output''; error messages printed to it will not disappear into an output file.

For example, a more realistic way to print an error message when a file can't be opened would be

```
if((fp = fopen(filename, "r")) = = NULL)
        {       printf("can't open file %s\n", filename);
                exit or return          }
```

where filename is a string variable indicating the file name to be opened. Not only is the error message printed to stderr, but it is also more informative in that it mentions the name of the file that couldn't be opened.

Creating a file

A data file must be created before it can be processed. A stream oriented data file can be created in two ways; one is to create the file directly, using a text editor or a word processor. The other is to write a program that enters information into the computer and then writes it out to the data file. When creating a new data file with a specially written program the usual approach is to enter the information from the keyboard and then write it out to the data file. If the data file consists of individual characters, the library function getchar()/getc() and putchar()/putc() can be used to enter the data form the keyboard and to write it out to the data file.

Downloaded from: sarojpandey.com.np

Example : creating a data file (to convert lowercase to upper case)

```
#include<stdio.h>
#include<ctype.h>
void main()
{               FILE  *fp1;
                Char c;
                fp1=fopen("sample.dat", "w");
                do
                    putc(toupper(c=getchar()),fp1);
                while("c!='\n');
                fclose(fp1);      }
```

1. Character Input/Output:
Using character I/O data can be read or written one character at a time. This is analog to the way functions putchar (), and getch (); write data to screen and read data from the keyboard.

Writing to a file:
Once the program has established the line of communication with a particular file by opening it, then it can write to the file. The syntax of function that writes one character at a time is

```
fputc( ch, fptr );
```

Where ch is character variable or constant and 'fptr' is a file pointer.
The fputc () is similar to the putchar () function the only difference is that a putchar function always writes to screen unless redirection employed.

Reading from a file:
If the program can write to a file, it should also be able to read from a file. The syntax of the function that reads and returns one character at a time is

```
Ch= fgetc ( fptr );
```

Where ch is character variable and fptr is file pointer. The fgetc ( ) funtion is similar to getchar ( ), getche( ), and getch ( ); functions, the only difference is that the later functions always read from the keyboard unless redirection is employed.

```
// Writing to a file
#include<stdio.h>
#include<string.h>
void main()
{
FILE *fp;
char ch;
fp=fopen("ex2.dat", "w");
if(fp==NULL)
   {   printf("Unable to create file");
       getch();
      exit(1);
   }
printf("Enter set of characters\n");
while((ch=getchar())!='\n')
   {
   fputc(ch, fp);
   }
fclose(fp);
}
```

```
// reading from a file
#include<stdio.h>
#include<string.h>
void main()
{
FILE *fp;
char ch;
fp=fopen("ex2.dat", "r");
if(fp==NULL)
   {   printf("Unable to read file");
       getch();
      exit(1);
   }
printf("\n Characters from file\n");
while((ch=fgetc(fp))!=EOF)
   {
   putchar(ch);
   }
getch();
fclose(fp);
}
```

Downloaded from: sarojpandey.com.np

2. String Input/ Output:
Using string I/O data can be read or written in the form of string of characters. Reading and writing strings of character is as easy as individual characters. This is analogous to puts (),and gets() functions that write data to screen and read data from the keyboard.

Writing to a file:
The syntax of the function that writes a string of characters at a time is

fputs (str, fptr);

Where str is an array of characters or a string constant, and fptr is a file pointer.
The fputs() functon is similar to puts() function, the only difference is that the put() function always writes to the screen unless redirection is employed.

Reading from a file:
The syntax of the function that reads strings from a file is

fgets (str, n, fptr);

where str is an array of characters and specifies the address where the string is to be stord , n is the maximum length of the input string, and fptr is a file pointer. The fgets ( ) function is similar to gets ( ) function, the only difference is that the gets ( ) function always reads from keyboard unless redirection is employed. The fgets ( ) function returns NULL value when it reads EOF.

```
// Writing to a file
#include<stdio.h>
#include<string.h>
void main()
{
FILE *fp;
char str[80];
fp=fopen("ex1.dat", "w");
if(fp==NULL)
   {
   printf("Unable to create file");
   getch();
   exit(1);
   }
printf("Enter set of strings\n");
while(strlen(gets(str))>0)
   {
   fputs(str, fp);
   }
fclose(fp);
}
```

```
// reading from a file
#include<stdio.h>
#include<string.h>
void main()
{
FILE *fp;
char str[80];
fp=fopen("ex1.dat", "r");
if(fp==NULL)
   {
   printf("Unable to open file");
   getch();
   exit(1);
   }
printf("Set of strings from file\n");
while(fgets(str, 80,fp)!=NULL)
   {
   puts(str);
   }
   getch();
fclose(fp);
}
```

Downloaded from: sarojpandey.com.np

3. Formatted Input/Output:

So far we have considered reading and writing of characters and strings. What about numbers? Let us suppose that we want to store information about an agent comprising his name (a string) code number (an integer number), and height (a real number). We want to create a data file for a given list of agents.

Writing to a file:

The syntax of the function that writes formatted data to a file is

fprintf (fptr, " format-string", ditems);

Where fptr is a file pointer and ditems is a lists of variables to be written to a file. The fprintf ( ), is similar to printf ( ) function, the only difference is that printf ( ) function writes formatted data on to the screen instead of file.

Reading from a file:

The syntax of the function that reads formatted data from a file is

fscanf (fptr, " format-string", ditems);

Where fptr is a file pointer and ditems is a lists of variables to be written to a file. The fscanf ( ), is similar to scanf ( ) function, the only difference is that scanf ( ) function reads formatted data from the keyboard.

```
// Writing to a file
#include<stdio.h>
#include<string.h>
void main()
{
FILE *fp;
char name[80];
int roll;
fp=fopen("ex3.dat", "w");
if(fp==NULL)
   {
   printf("Unable to create file");
   getch();
   exit(1);
   }
printf("Enter name and roll\n");
scanf("%s%d", name, &roll);
fprintf(fp," %s %d", name, roll);
fclose(fp);
}
```

```
//reading from a file
#include<stdio.h>
#include<string.h>
void main()
{
FILE *fp;
char name[80];
int roll;
if((fp=fopen("ex3.dat", "r"))= =NULL)
   {
   printf("Unable to read file");
   getch();
   exit(1);
   }
printf("\n Name and roll\n");
while(fscanf(fp, "%s%d",name,&roll)!=EOF)
printf("%s %d", name, roll);
getch();
fclose(fp);
}
```

4. Record Input/Output:

We have seen that character I/O and string I/O permits reading and writing of character data only. Whereas the formatted I/O permit reading and writing of character data as well as numeric data. However, the arrays and structures are stored as sequence of values. So these are handled as block of data instead of processing one data at a time.

Writing to a file:

The syntax of the function that writes block of data at a time is

fwrite(ptr, m, n, fptr)

where ptr is an address of array or structure to be written, m is the size of an array or a structure, n is the number of such arrays or structures to be written and fptr is a file pointer.

Reading from a file:

The syntax of the function that writes block of data at a time is

fread(ptr, m, n, fptr)

Downloaded from: sarojpandey.com.np

where ptr is an address of array or structure where block will be stored after reading, m is the size of an array or a structure, n is the number of such arrays or structures to be read and fptr is a file pointer opened for reading.

5. Array Input/Output:

```
// Writing to a file
#include<stdio.h>
void main()
{
FILE *fp;
int a[10], i;
if((fp=fopen("ex4.dat", "w"))==NULL)
   {
   printf("Unable to create file");
   getch();
   exit(1);
   }
printf("\n Enter array elements\n");
for(i=0; i<10; i++)
scanf("%d",&a[i]);
fwrite(&a,sizeof(a),1,fp);
fclose(fp);
}
```

```
//reading from a file
#include<stdio.h>
void main()
{
FILE *fp;
int a[10], i;
if((fp=fopen("ex4.dat", "r"))==NULL)
   {
   printf("Unable to open file");
   getch();
   exit(1);
   }
printf("\n Array elements are\n");
fread(&a,sizeof(a),1,fp);
for(i=0; i<10; i++)
printf("%d\t",a[i]);
fclose(fp);
getch();
}
```

6. Structure Input/Output:

```
// Writing to a file
#include<stdio.h>
void main()
{
FILE *fp;
struct student
        {       char name[20];
                int roll;
        }s;
if((fp=fopen("ex5.dat", "w"))==NULL)
   {
   printf("Unable to create file");
   getch();
   exit(1);
   }
printf("\n Enter name,roll of student\n");
scanf("%s%d",s.name,&s.roll);
fwrite(&s,sizeof(s),1,fp);
fclose(fp);
}
```

```
//reading from a file
#include<stdio.h>
void main()
{
FILE *fp;
struct student
        {       char name[20];
                int roll;
        }s;
if((fp=fopen("ex5.dat", "r"))==NULL)
   {
   printf("Unable to open file");
   exit(1);
   }
printf("\n structure members are\n");
fread(&s,sizeof(s),1,fp);
printf("%s%d\t",s.name, s.roll);
fclose(fp);
getch();
}
```

Downloaded from: sarojpandey.com.np

Program that appends one file to another

```
#include<stdio.h>
void main()
{       char c;
        FILE *fp1, *fp2;

        /*assume that two files exam1.dat and exam2.dat exist already*/
        fp1=fopen("exam1.dat","a");
        fp2=fopen("exam2.dat","r");

        c=fgetc(fp2);
        while(c!=EOF)
        {
                fputc(c,fp1);
                c=fgetc(fp2);
        }
        fclose(fp1);
        fclose(fp2);
}
```

Program that displays the contents of a file in reverse order

```
#include<stdio.h>
void main()
{       char c;
        long int pos;

        FILE *fp;
        /*assume that two files exam.dat exist already*/

        fp=fopen("exam.dat", "r");
        pos=ftell(fp);
        fseek(fp,-1,2);
        while(ftell(fp)>pos)
          {
                c=fgets(fp);
                fseek(fp,-2,1);
                printf("%c",c);
          }
        c=fgets(fp);
        printf("%c",c);
        fclose(fp);
}
```