

Unit 1
Algorithm and Programming Development
Steps in development of a program
Program Development Life Cycle

When we want to develop a program using any programming language, we follow a sequence of steps. These steps are called phases in program development. The program development life cycle is a set of steps that are used to develop a program in any language.

Generally, the program development life cycle contains 6 phases, they are as follows....

- Problem Definition
- Problem Analysis
- Algorithm Development
- Coding & Documentation
- Testing & Debugging
- Maintenance

1. Problem Definition

In this phase, we define the problem statement and we decide the boundaries of the problem. In this phase we need to understand the problem statement, what is our requirement, what should be the output of the problem solution. These are defined in this first phase of the program development life cycle.

2. Problem Analysis

In phase 2, we determine the requirements like variables, functions, etc. to solve the problem. That means we gather the required resources to solve the problem defined in the problem definition phase. We also determine the bounds of the solution.

3. Algorithm Development

During this phase, we develop a step by step procedure to solve the problem using the specification given in the previous phase. This phase is very important for program development. That means we write the solution in step by step statements.

4. Coding & Documentation

This phase uses a programming language to write or implement the actual programming instructions for the steps defined in the previous phase. In this phase, we construct the actual program. That means we write the program to solve the given problem using programming languages like C, C++, Java, etc.,

5. Testing & Debugging

During this phase, we check whether the code written in the previous step is solving the specified problem or not. That means we test the program whether it is solving the problem for various input data values or not. We also test whether it is providing the desired output or not.

6. Maintenance

During this phase, the program is actively used by the users. If any enhancements found in this phase, all the phases are to be repeated to make the enhancements. That means in this phase, the solution (program) is used by the end-user. If the user encounters any problem or wants any enhancement, then we need to repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

Algorithm and Flowchart development

ALGORITHM:

The word “algorithm” relates to the name of the mathematician Al-khowarizmi, which means a procedure or a technique. Software Engineer commonly uses an algorithm for planning and solving the problems. An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time

Algorithm has the following characteristics

- **Input:** An algorithm may or may not require input
- **Output:** Each algorithm is expected to produce at least one result
- **Definiteness:** Each instruction must be clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm are executed, the algorithm should terminate after finite number of steps

The algorithm and flowchart include following three types of control structures.

1. **Sequence:** In the sequence structure, statements are placed one after the other and the execution takes place starting from up to down.
2. **Branching (Selection):** In branch control, there is a condition and according to a condition, a decision of either TRUE or FALSE is achieved. In the case of TRUE, one of the two branches is explored; but in the case of FALSE condition, the other alternative is taken. Generally, the 'IF-THEN' is used to represent branch control.
3. **Loop (Repetition):** The Loop or Repetition allows a statement(s) to be executed repeatedly based on certain loop condition e.g. WHILE, FOR loops.

Advantages of algorithm

- It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
- An algorithm uses a definite procedure.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.

HOW TO WRITE ALGORITHMS

Step 1 Define your algorithms input: Many algorithms take in data to be processed, e.g. to calculate the area of rectangle input may be the rectangle height and rectangle width.

Step 2 Define the variables: Algorithm's variables allow you to use it for more than one place. We can define two variables for rectangle height and rectangle width as HEIGHT and WIDTH (or H & W). We should use meaningful variable name e.g. instead of using H & W use HEIGHT and WIDTH as variable name.

Step 3 Outline the algorithm's operations: Use input variable for computation purpose, e.g. to find area of rectangle multiply the HEIGHT and WIDTH variable and store the value in new variable (say) AREA. An algorithm's operations can take the form of multiple steps and even branch, depending on the value of the input variables.

Step 4 **Output the results of your algorithm's operations:** In case of area of rectangle output will be the value stored in variable AREA. if the input variables described a rectangle with a HEIGHT of 2 and a WIDTH of 3, the algorithm would output the value of 6.

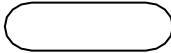
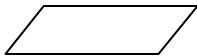
FLOWCHART


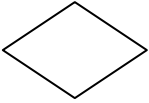
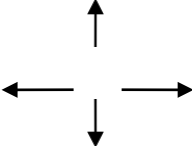


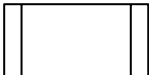

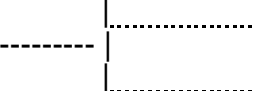
The first design of flowchart goes back to 1945 which was designed by John Von Neumann. Unlike an algorithm, Flowchart uses different symbols to design a solution to a problem. It is another commonly used programming tool. By looking at a Flowchart one can understand the operations and sequence of operations performed in a system. Flowchart is often considered as a blueprint of a design used for solving a specific problem.

Advantages of flowchart:

- Flowchart is an excellent way of communicating the logic of a program.
- Easy and efficient to analyze problem using flowchart.
- During program development cycle, the flowchart plays the role of a blueprint, which makes program development process easier.
- After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
- It is easy to convert the flowchart into any programming language code.

Flowchart is diagrammatic /Graphical representation of sequence of steps to solve a problem. To draw a flowchart following standard symbols are use

Symbol Name	Symbol	function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation

Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
		Off Page Connector
		Predefined Process /Function Used to represent a group of statements performing one processing task.
		Preprocessor
		Comments

Algorithm & Flowchart to find the sum of two numbers

Algorithm

Step-1 Start

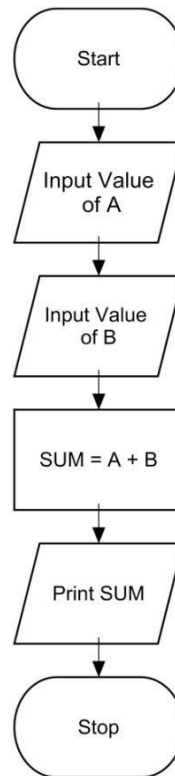
Step-2 Input first numbers say A

Step-3 Input second number

say B Step-4 $SUM = A + B$

Step-5 Display

SUM Step-6 Stop



Algorithm & Flowchart to find the largest of three numbers

Algorithm

Step-1 Start

Step-2 Read three numbers say num1,num2,

num3 Step-3 if num1>num2 then go to

step-5

Step-4 IF num2>num3 THEN

print num2 is

largest ELSE

print num3 is

largest ENDIF

GO TO Step-6

Step-5 IF num1>num3 THEN

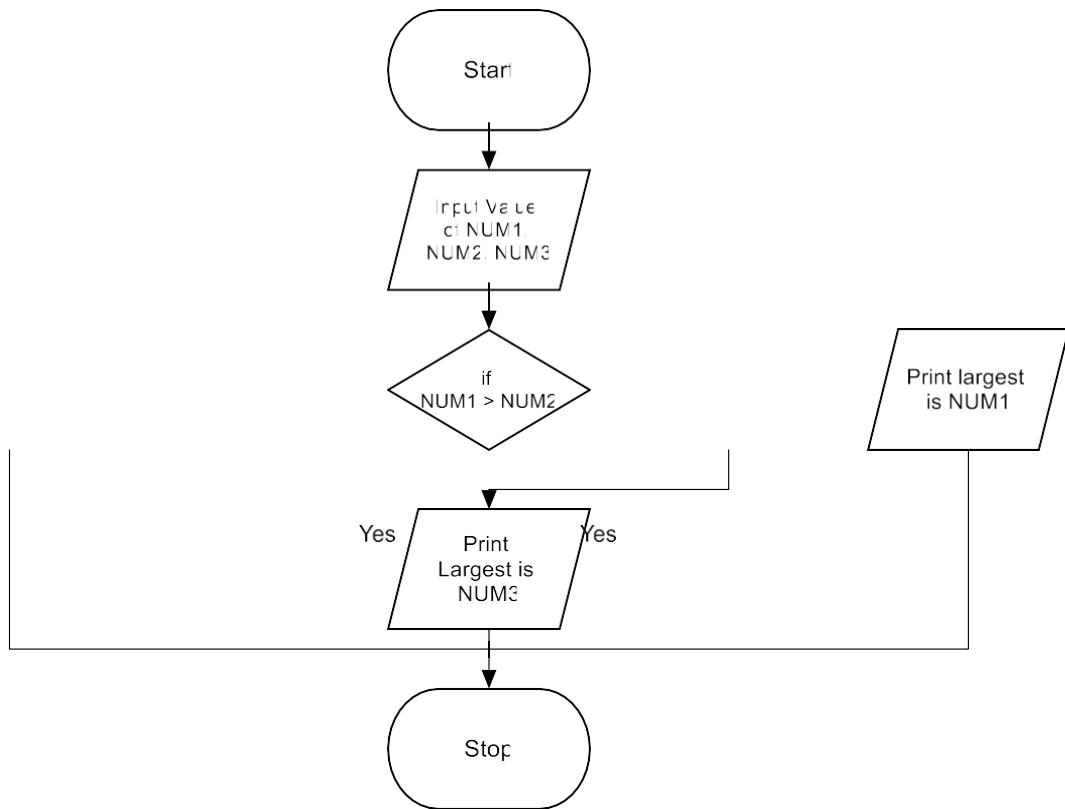
print num1 is largest

ELSE

print num3 is

largest ENDIF

step-6 Stop



Programme Debugging

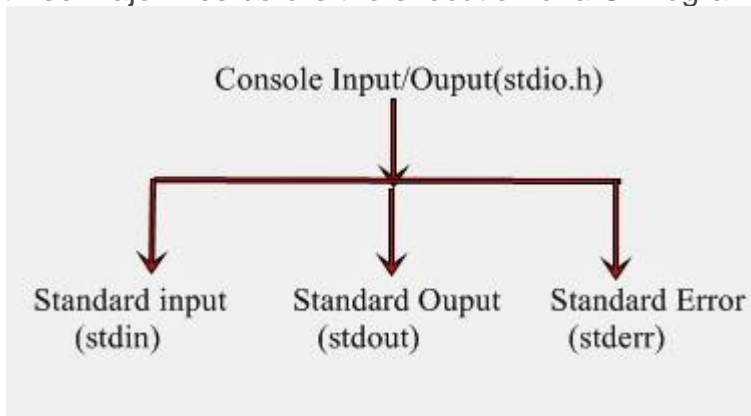
Definition: Debugging is the process of detecting and removing of existing and potential errors (also called as 'bugs') in a software code that can cause it to behave unexpectedly or crash. To prevent incorrect operation of a software or system, debugging is used to find and resolve bugs or defects. When various subsystems or modules are tightly coupled, debugging becomes harder as any change in one module may cause more bugs to appear in another. Sometimes it takes more time to debug a program than to code it. To debug a program, user has to start with a problem, isolate the source code of the problem, and then fix it. A user of a program must know how to fix the problem as knowledge about problem analysis is expected. When the bug is fixed, then the software is ready to use. Debugging tools (called debuggers) are used to identify coding errors at various development stages. They are used to reproduce the conditions in which error has occurred, then examine the program state at that time and locate the cause. Programmers can trace the program execution step-by-step by evaluating the value of variables and stop the execution wherever required to get the value of variables or reset the program variables. Some programming language packages provide a debugger for checking the code for errors while it is being written at run time.

Unit 2

Program Structure

I/O statements, assign statements Unformatted and Formatted IOS

In C Language input and output function are available as C compiler function or C library provided with each C compiler implementation. These all functions are collectively known as Standard I/O Library function. Here I/O stands for Input and Output used for different inputting and outputting statements. These I/O functions are categorized into three processing functions. Console input/output function (deals with keyboard and monitor), disk input/output function (deals with floppy or hard disk) and port input/output function (deals with serial or parallel port). As all the input/output statements deal with the console, so these are also Console Input/Output functions. Console Input/Output function access the three major files before the execution of a C Program. These are as:



stdin: This file is used to receive the input (usually is keyboard file, but can also take input from the disk file).

stdout: This file is used to send or direct the output (usually is a monitor file, but can also send the output to a disk file or any other device).

stderr: This file is used to display or store error messages.

Input Output Statement

Input and Output statement are used to read and write the data in C programming. These are embedded in stdio.h (standard Input/Output header file). There are mainly two of Input/Output functions are used for this purpose. These are discussed as:

A) Unformatted I/O functions

There are mainly six unformatted I/O functions discussed as follows:

- a) getchar()
- b) putchar()
- c) gets()
- d) puts()
- e) getch()
- f) getche()

a) getchar()

This function is an Input function. It is used for reading a single character from the keyboard. It is buffered function. Buffered functions get the input from the keyboard and store it in the memory buffer temporarily until you press the Enter key.

The general syntax is as:

```
v = getchar();
```


where v is the variable of character type.

A simple C-program to read a single character from the keyboard is as:

```
/*To read a single character from the keyboard using the getchar() function*/
#include
main()
{
char n;
n = getchar();
}
```

b) putchar()

This function is an output function. It is used to display a single character on the screen. The general syntax is as: putchar(v);
where v is the variable of character type.

A simple program is written as below, which will read a single character using getchar() function and display inputted data using putchar() function:

```
/*Program illustrate the use of getchar() and putchar() functions*/
#include
main()
{
char n;
n = getchar();
putchar(n);
}
```

c) gets()

This function is an input function. It is used to read a string from the keyboard
gets(v);

where v is the variable of character type

A simple C program to illustrate the use of gets() function:

```
/*Program to explain the use of gets() function*/
#include
main()
{
charn[20];
gets(n);
}
```

d) puts()

This is an output function. It is used to display a string inputted by gets() function

The general syntax is as:

puts(v);

where v is the variable of character type.

A simple C program to illustrate the use of puts() function:

e) getch()

This is also an input function. This is used to read a single character from the keyboard like getchar() function. But getch() function is a buffered is function, getch() function is

a non-buffered function. The character data read by this function is directly assign to a variable rather it goes to the memory buffer, the character data directly assign to a variable without the need to press the Enter key.

Another use of this function is to maintain the output on the screen till you have not press the Enter Key. The general syntax is as:

```
v = getch();
```

where v is the variable of character type.

A simple C program to illustrate the use of getch() function:

f) getche()

All are same as getch() function except it is an echoed function. It means when you type the character data from the keyboard it will visible on the screen. The general syntax is as:

```
v = getche();
```

where v is the variable of character type.

A simple C program to illustrate the use of getch() function:

B)Formatted I/O functions

Formatted I/O functions which refers to an Input or Ouput data that has been arranged in a particular format. There are mainly two formatted I/O functions discussed as follows:

a) scanf()

b) printf()

a) scanf()

The scanf() function is an input function. It used to read the mixed type of data from keyboard. You can read integer, float and character data by using its control codes or format codes.

Format Code	Meaning
%c	To read a single character
%d	To read a signed decimal integer (short)
%ld	To read a signed long decimal integer

```
/*Program to illustrate the use of formatted code by using the formatted scanf() function */
```

```
#include
```

```
main()
```

```
{
```

```
int abc;
```

```
printf("Enter the integer value");
```

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

```
getch();
```

```
}
```

b) printf()

This is an output function.

```
/*Below the program which show the use of printf() function*/
```

```
main()
```

```
{
```

```
int a;
```

```
printf("Enter the mixed type of data");
```

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

```
getch();  
}
```

Variables, Constant and data types

Variable

A variable is an identifier which is used to store some value. Constants can never change at the time of execution. Variables can change during the execution of a program and update the value stored inside it.

A single variable can be used at multiple locations in a program. A variable name must be meaningful. It should represent the purpose of the variable.

Example: Height, age, are the meaningful variables that represent the purpose it is being used for. Height variable can be used to store a height value. Age variable can be used to store the age of a person

A variable must be declared first before it is used somewhere inside the program. A variable name is formed using characters, digits and an underscore.

Following are the rules that must be followed while creating a variable:

A variable name should consist of only characters, digits and an underscore.

A variable name should not begin with a number.

A variable name should not consist of whitespace.

A variable name should not consist of a keyword.

'C' is a case sensitive language that means a variable named 'age' and 'AGE' are different.

For example, we declare an integer variable my_variable and assign it the value 48:

```
int my_variable;  
my_variable = 48;
```

By the way, we can both declare and initialize (assign an initial value) a variable in a single statement:

```
int my_variable = 48;
```

Data types

'C' provides various data types to make it easy for a programmer to select a suitable data type as per the requirements of an application. Following are the three data types:

Primitive data types

Derived data types

User-defined data types

There are five primary fundamental data types,

int for integer data

char for character data

float for floating point numbers

double for double precision floating point numbers

void

Array, functions, pointers, structures are derived data types. 'C' language provides more extended versions of the above mentioned primary data types. Each data type differs from one another in size and range. Following table displays the size and range of each data type.

Integer data type

Integer is nothing but a whole number. The range for an integer data type varies from machine to machine. The standard range for an integer data type is -32768 to 32767.

Whenever we want to use an integer data type, we have place int before the identifier such as,

```
int age;
```

Here, age is a variable of an integer data type which can be used to store integer values.

Constants

Constants are the fixed values that never change during the execution of a program.

Following are the various types of constants:

Integer constants

An integer constant is nothing but a value consisting of digits or numbers. These values never change during the execution of a program. Integer constants can be octal, decimal and hexadecimal.

1. Decimal constant contains digits from 0-9 such as,

Example, 111, 1234

Above are the valid decimal constants.

Above are the valid hexadecimal constants.

2.Character constants

A character constant contains only a single character enclosed within a single quote (' ').

We can also represent character constant by providing ASCII value of it.

Example, 'A', '9'

Above are the examples of valid character constants.

3.String constants

A string constant contains a sequence of characters enclosed within double quotes (" ").

Example, "Hello", "Programming"

These are the examples of valid string constants.

Operators and Expressions

Mathematical Operators in C:

Operator	Meaning	Example
+	Addition	A + B
-	Subtraction	A - B
*	Multiplication	A * B
/	Division	A / B
^	Power	A^3 for A ³
%	Reminder	A % B

Relational Operators

Operator	Meaning	Example
<	Less than	A < B
<=	Less than or equal to	A <= B
= or ==	Equal to	A = B
# or !=	Not equal to	A # B or A !=B
>	Greater than	A > B
>=	Greater the or equal to	A >= B

Logical Operators

Operator	Example	Meaning
----------	---------	---------

AND	A < B AND B < C	Result is True if both A<B and B<C are true else false
OR	A< B OR B < C	Result is True if either A<B or B<C are true else false
NOT	NOT (A >B)	Result is True if A>B is false else true

Selection control Statements

Selection Control	Example	Meaning
IF (Condition) Then ... ENDIF	IF (X > 10) THEN Y=Y+5 ENDIF	If condition X>10 is True execute the statement between THEN and ENDIF
IF (Condition) Then ... ELSE ENDIF	IF (X > 10) THEN Y=Y+5 ELSE Y=Y+8 Z=Z+3 ENDIF	If condition X>10 is True execute the statement between THEN and ELSE otherwise execute the statements between ELSE and ENDIF

Loop control Statements

Selection Control	Example	Meaning
WHILE (Condition) DO ENDDO	WHILE (X < 10) DO print xx=x+1 ENDDO	Execute the loop as long as the condition is TRUE
DO UNTILL (Condition)	DO print x x=x+1 UNTILL (X >10)	Execute the loop as long as the condition is false

Data type casting

Converting one datatype into another is known as type casting or, type-conversion. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the cast operator as follows –

(type_name) expression

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation –

```
main() {
```

```
    int sum = 17, count = 5;  
    double mean;
```

```
mean = (double) sum / count;  
printf("Value of mean : %f\n", mean );  
}
```

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

Value of mean : 3.400000

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value.

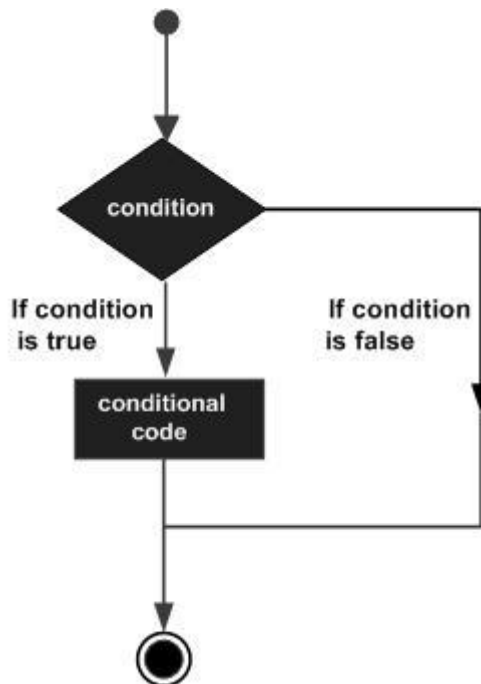
Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary

Unit 3 Control Structures

Introduction

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

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



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

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

1 <u>if statement</u>	An if statement consists of a boolean expression followed by one or more statements.
2 <u>if...else statement</u>	An if statement can be followed by an optional else statement , which executes when the Boolean expression is false.
3 <u>nested if statements</u>	You can use one if or else if statement inside another if or else if statement(s).
4 <u>switch statement</u>	

A **switch** statement allows a variable to be tested for equality against a list of values.

5 nested switch statements

You can use one **switch** statement inside another **switch** statement(s).

C provides two styles of flow control:

- Branching
- Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching is so called because the program chooses to follow one branch or another.

1.if statement

This is the most simple form of the branching statements.

It takes an expression in parenthesis and an statement or block of statements. if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be true if its evaluated values is non-zero.

if statements take the following form:

Show Example

```
if (expression)
    statement;

or

if (expression)
{
    Block of statements;
}

or

if (expression)
{
    Block of statements;
}
else
{
    Block of statements;
}
```



```

or
if (expression)
{
    Block of statements;
}
else if(expression)
{
    Block of statements;
}
else
{
    Block of statements;
}

```

2.switch statement:

The switch statement is much like a nested if .. else statement. Its mostly a matter of preference which you use, switch statement can be slightly more efficient and easier to read.

Show Example

```

switch( expression )
{
    case constant-expression1: statements1;
    [case constant-expression2: statements2;]
    [case constant-expression3: statements3;]
    [default : statements4;]
}

```

If none of the listed conditions is met then default condition executed.

Looping

Loops provide a way to repeat commands and control how many times they are repeated. C provides a number of looping way.

1)while loop

The most basic loop in C is the while loop.A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statments get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again.This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:

Show Example

```

while ( expression )
{
    Single statement
}

```

```
or  
Block of statements;  
}
```

2)for loop

for loop is similar to while, it's just written differently. for statements are often used to process lists such a range of numbers:

Basic syntax of for loop is as follows:

Show Example

```
for( expression1; expression2; expression3)  
{  
    Single statement  
    or  
    Block of statements;  
}
```

In the above syntax:

- expression1 - Initialises variables.
- expression2 - Conditional expression, as long as this condition is true, loop will keep executing.
- expression3 - expression3 is the modifier which may be simple increment of a variable.

3)do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

Show Example

```
do  
{  
    Single statement  
    or  
    Block of statements;  
}while(expression);
```

Break, continue and goto statements

C provides two commands to control loop:

- break -- exit form loop or switch.
- continue -- skip 1 iteration of loop.

This will produce following output:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10
```

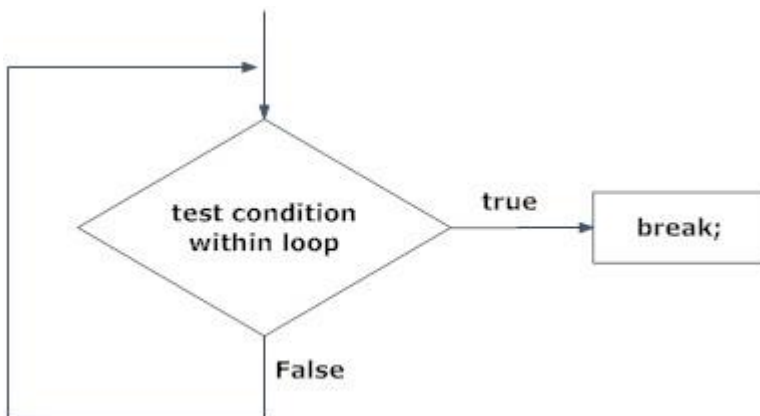
The **break;**, **continue;** and **goto;** statements are used to alter the normal flow of a program. Loops perform a set of repetitive task until test expression becomes false but it is sometimes desirable to skip some statement/s inside loop or terminate the loop immediately without checking the test expression. In such cases, break and continue statements are used.

break statement

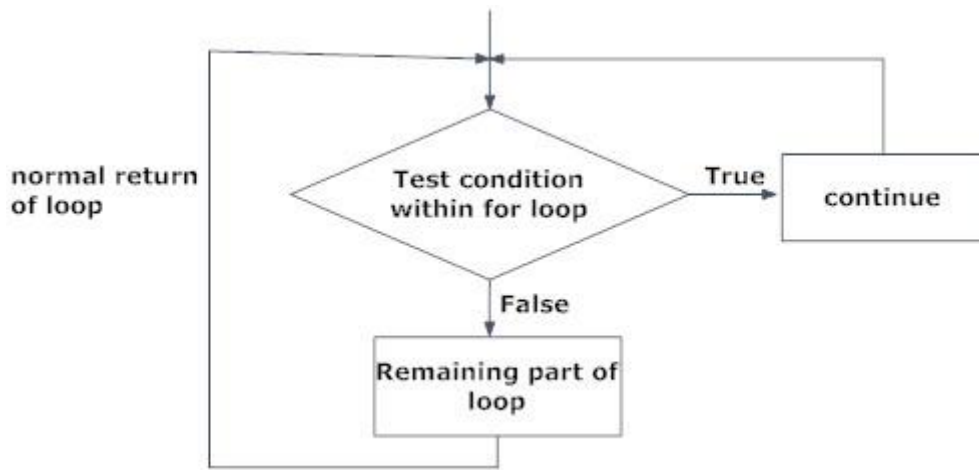
In C programming, break statement is used with conditional if statement. The break is used in terminating the loop immediately after it is encountered.

break;

The break statement can be used in terminating loops like **for**, **while** and **do...while**



Just like break, continue is also used with conditional if statement.



go to statement

- Though, using goto statement give power to jump to any part of program, using goto statement makes the logic of the program complex and tangled.
- In modern programming, goto statement is considered a harmful construct and a bad programming practice.
- The goto statement can be replaced in most of C program with the use of break and continue statements.
- In fact, any program in C programming can be perfectly written without the use of goto statement.

Unit 4 Pointers

Pointers

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

Consider the following example to define a pointer which stores the address of an integer.

1. `int n = 10;`
2. `int* p = &n;` // Variable p of type pointer is pointing to the address of the variable n of type integer.

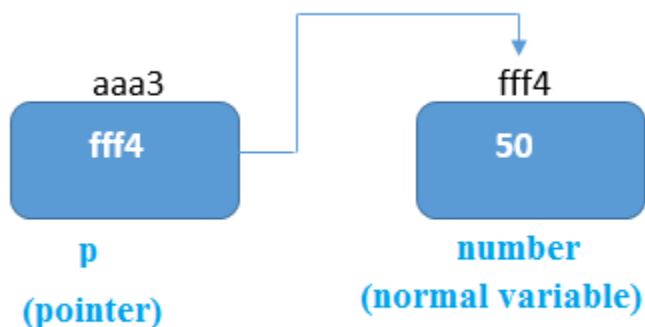
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. `int *a;`//pointer to int
2. `char *c;`//pointer to char

Pointer Example

An example of using pointers to print the address and value is given below.



javatpoint.com

As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`

4. `int *p;`
5. `p=&number;//stores the address of number variable`
6. `printf("Address of p variable is %x \n",p);` // p contains the address of the number therefore printing p gives the address of number.
7. `printf("Value of p variable is %d \n",*p);` // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
8. `return 0;`
9. `}`

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

1. `int arr[10];`
2. `int *p[10]=&arr;` // Variable p of type pointer is pointing to the address of an integer array arr.

Pointer to a function

1. `void show (int);`
2. `void(*p)(int) = &display;` // Pointer p is pointing to the address of a function

Advantage of pointer

- 1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
- 2) We can **return multiple values from a function** using the pointer.
- 3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using `malloc()` and `calloc()` functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. printf("value of number is %d, address of number is %u",number,&number);
5. return 0;
6. }
```

Output

```
value of number is 50, address of number is fff4
```

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
1. #include<stdio.h>
2. int main(){
3. int a=10,b=20,*p1=&a,*p2=&b;
4.
5. printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
6. *p1=*p1+*p2;
7. *p2=*p1-*p2;
8. *p1=*p1-*p2;
9. printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
10.
11. return 0;
12. }
```

Output

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

A **pointer** is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of a integer variable. In this guide, we will discuss pointers in [C programming](#) with the help of examples.

Before we discuss about **pointers in C**, lets take a simple example to understand what do we mean by the address of a variable.

A simple example to understand how to access the address of a variable without pointers?

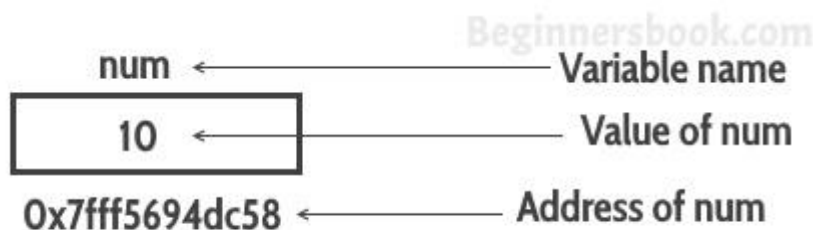
In this program, we have a variable `num` of `int` type. The value of `num` is 10 and this value must be stored somewhere in the memory, right? A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. For example we live in a house and our house has an address, which helps other people to find our house. The same way the value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

So let's say the address assigned to variable `num` is `0x7fff5694dc58`, which means whatever value we would be assigning to `num`

```
#include <stdio.h>
int main()
{
    int num = 10;
    printf("Value of variable num is: %d", num);
    /* To print the address of a variable we use %p
     * format specifier and ampersand (&) sign just
     * before the variable name like &num.
     */
    printf("\nAddress of variable num is: %p", &num);
    return 0;
}
```

Output:

```
Value of variable num is: 10
Address of variable num is: 0x7fff5694dc58
```



A Simple Example of Pointers in C

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we have discussed them later in this guide. For now, we just need to know how to link a pointer to the address of a variable.

Important point to note is: The data type of pointer and the variable must match, an `int` pointer can hold the address of `int` variable, similarly a pointer declared with `float` data type can hold the address of a `float` variable. In the example below, the pointer and the variable both are of `int` type.

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



```
{  
  //Variable declaration  
  int num = 10;  
  
  //Pointer declaration  
  int *p;  
  
  //Assigning address of num to the pointer p  
  p = #  
  
  printf("Address of variable num is: %p", p);  
  return 0;  
}
```

Output:

Address of variable num is: 0x7fff5694dc58

..
..

Unit 5 Functions

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

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

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

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

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

1. Defining a Function

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

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

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

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

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

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

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

return result;
}
```

2.Function Declarations

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

A function declaration has the following parts –

```
return_type function_name( parameter list );
```

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

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

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

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

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

3.Calling a Function

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

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

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

```
#include <stdio.h>

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

int main () {

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

```

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

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

return 0;
}

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

/* local variable declaration */
int result;

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

return result;
}

```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result –

Max value is : 200

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function –

Sr.No.	Call Type & Description
1	<p>Call by value</p> <p>This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.</p>
2	<p>Call by reference</p> <p>This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that</p>

changes made to the parameter affect the argument.

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>

int main () {

    /* local variable declaration */
    int a, b;
    int c;

    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;

    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

    return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>

/* global variable declaration */
int g;
```

```
..  
int main () {  
  
    /* local variable declaration */  
    int a, b;  
  
    /* actual initialization */  
    a = 10;  
    b = 20;  
    g = a + b;  
  
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);  
  
    return 0;  
}
```

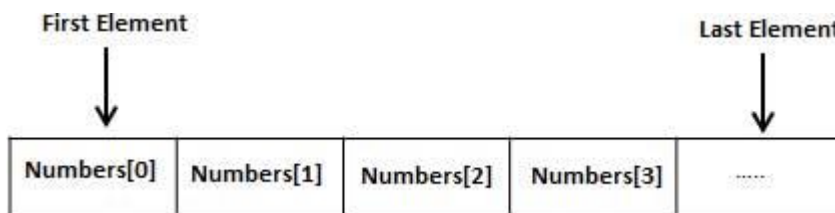
Arrays and Strings

Arrays

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

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

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



Declaring Arrays

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

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type `double`, use this statement –

```
double balance[10];
```

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

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

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

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

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

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

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

```
balance[4] = 50.0;
```

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

..
representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements

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

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```
#include <stdio.h>

int main () {

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

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

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

    return 0;
}
```

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

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

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration –

..
type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional integer array –

```
int threedim[5][10][4];
```

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows –

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows –

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3}, /* initializers for row indexed by 0 */  
    {4, 5, 6, 7}, /* initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example –

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example –

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array –

```

#include <stdio.h>

int main () {

    /* an array with 5 rows and 2 columns*/
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
    int i, j;

    /* output each array element's value */
    for ( i = 0; i < 5; i++ ) {

        for ( j = 0; j < 2; j++ ) {
            printf("a[%d][%d] = %d\n", i,j, a[i][j] );
        }
    }

    return 0;
}

```

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

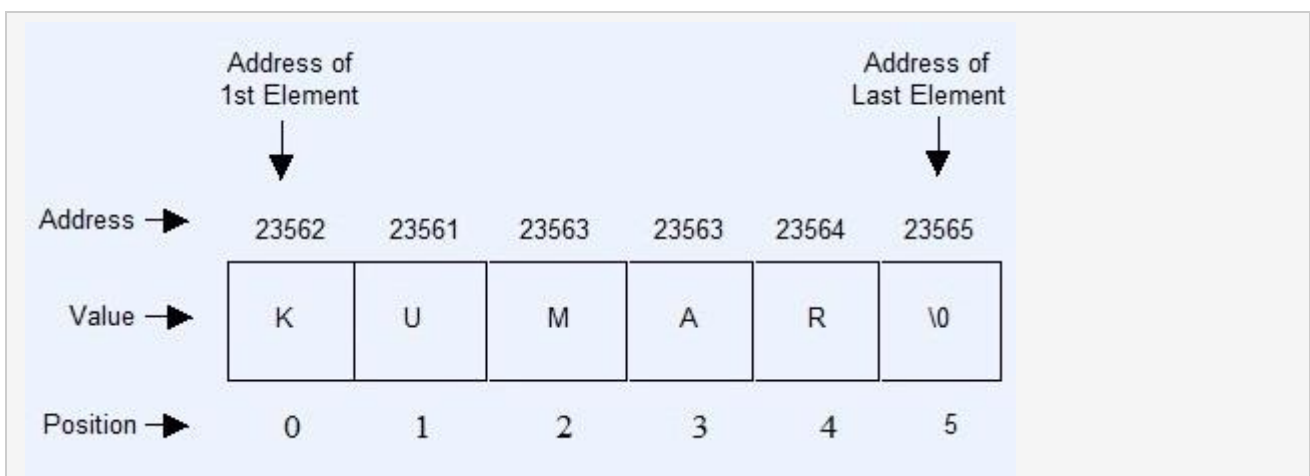
```

a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8

```

Array of character

A string is a collection of characters, stored in an array followed by null ('\0') character. Null character represents the end of string.



Address of first element is random, address of next element depend upon the type of array. Here, the type is character and character takes one byte in memory, therefore every next address will increment by one.

..
Index of array will always starts with zero.

Declaration of String or Character array

Declaration of array means creating sequential blocks of memory to hold fixed number of values.

Syntax for string declaration :

```
char String-name[size of String ];
```

Example for string declaration :

```
char String [25]; //Statement 1
```

In the above example we have declared a character array which can hold twenty-five characters at a time.

Initialization of String

Initialization of string means assigning value to declared character array or string.

Examples 1 :Using sequence of characters.

```
char String [ ] = {'H','e','l','l','o',' ','W','o','r','l','d','\0'};
```

In the above example, we are declaring and initializing a string at same time. When we declare and initialize a string at same time, giving the size of array is optional and **it is programmer's job to specify the null character at the end to terminate the string.**

Example 2 : Using assignment operator

```
char String [ ] = "Hello World";
```

In this approach, programmer doesn't need to provide null character, compiler will automatically add null character at the end of string.

Examples for input string with scanf() function

```
#include<stdio.h>

void main()
{
    char String [50];
```

```

printf("\n\n\tEnter your name : ");
scanf("%s", String );

printf("\n\n\tHello %s", String );

}

```

Output :

```

Enter your name :    Kumar Wadhwa
Hello Kumar

```

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

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

If you follow the rule of array initialization then you can write the above statement as follows –

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

Following is the memory presentation of the above defined string in C/C++ –

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

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```

#include <stdio.h>

int main () {

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

```

```

..
printf("Greeting message: %s\n", greeting );
return 0;
}

```

When the above code is compiled and executed, it produces the following result –
Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings –

Sr.No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –

```

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

int main () {

    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    /* copy str1 into str3 */

```

```

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

/* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2): %s\n", str1 );

/* total length of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1) : %d\n", len );

return 0;
}

```

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

```

strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10

```

Unit 7 Structures and Unions

A **union** is a special data type available in C that allows to store different data types in

..
the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

```
union [union tag] {  
    member definition;  
    member definition;  
    ...  
    member definition;  
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named `Data` having three members `i`, `f`, and `str` –

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, `Data` type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

```
#include <stdio.h>  
#include <string.h>  
  
union Data {  
    int i;  
    float f;  
    char str[20];  
};  
  
int main( ) {  
  
    union Data data;  
  
    printf( "Memory size occupied by data : %d\n", sizeof(data));  
}
```

```
..  
return 0;  
}
```

When the above code is compiled and executed, it produces the following result –
Memory size occupied by data : 20

Structures in C?

Structure stores the different types of elements i.e heterogeneous elements. The struct keyword is used to define structure.

Syntax

```
struct structure_name  
{  
data_type member1;  
  
data_type memberN;  
};
```

Declaring structure variable

You can declare structure variable in two ways:-

1. By struct keyword within main() function
2. By declaring variable at the time of defining structure.

Unions in C?

Union also stores the different types of elements i.e heterogeneous elements. The union keyword is used to define structure. Union takes the memory of largest member only so occupies less memory than structures.

Syntax

```
union union_name  
{  
data_type member1;  
  
data_type memeberN;  
};
```