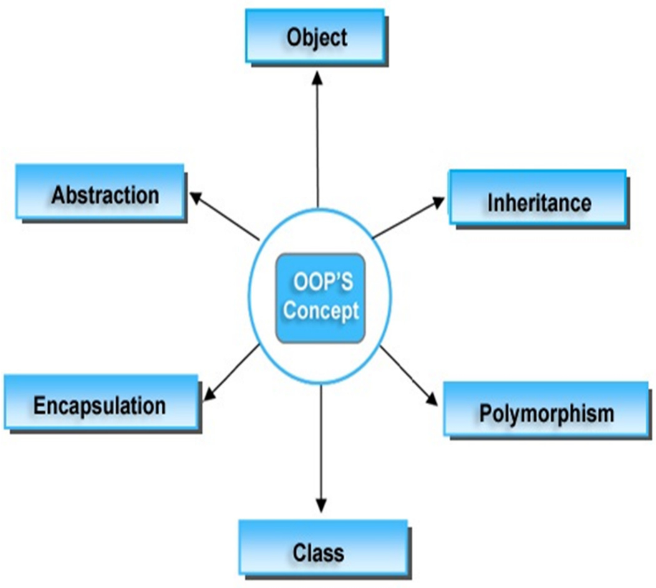
Chapter 1. Principles of Object Oriented Programming

Procedure Oriented Programming Vs Object Oriented Programming

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: C, VB, FORTRAN, Pascal.	Examples of OOP are: C++, JAVA, VB.NET, C#.NET.

Basic concepts of object oriented programming in C++

Object Oriented programming is associated with the concept of Class, Objects, Inheritance, Polymorphism, Abstraction, Encapsulation etc.



Objects

Objects are the basic unit of OOP. They are instances of class, in which data members and various member functions that operate on data are bundled as a unit called as object.

Class

It is similar to structures in C language. Class can also be defined as user defined data type that contains data members and member functions in it. So, class is a blueprint for object. that is, it defines what an object of the class will consist of and what operations can be performed on such an object.

For example, lets say we have a class **Car** which has data members (variables) such as speed, weight, price and functions such as gearChange(), slowDown(), brake() etc. Now lets say I create a object of this class named FordFigo which uses these data members and functions and give them its own values. Similarly we can create as many objects as we want using the blueprint (class).

```
class Car
{
   //Data members
   char name[20];
   int speed;
   int weight;

public:
   //Functions
   void brake(){
   }
   void slowDown(){
   }
};

int main()
{
  //ford is an object
  Car ford;
}
```

Abstraction

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes can provide methods to the outside world to access & use the data variables, keeping the variables hidden from direct access, or classes can even declare everything accessible to everyone, or maybe just to the classes inheriting it. This can be done using access specifiers.

For example, When you send an sms you just type the message, select the contact and click send, the phone shows you that the message has been sent, what actually happens in background when you click send is hidden from you as it is not relevant to you.

Encapsulation

It can also be said data binding. Encapsulation is all about combining the data variables and functions together in class. This is to avoid the access of private data members from outside the class.

Inheritance

Inheritance is a way to reuse once written code again and again. As the name suggests Inheritance is the process of forming a new class from an existing class, the existing class is called as base class, new class is called as derived class. They are also called parent and child class.

So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Polymorphism

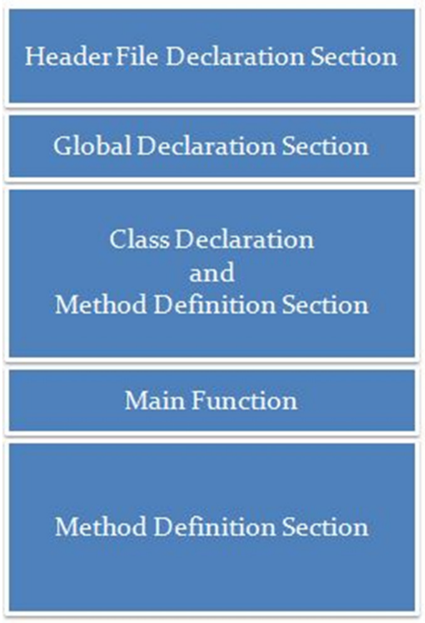
It is a feature, which lets us create functions with same name but different arguments, which will perform different actions. That means, functions with same name, but functioning in different ways. Or, it also allows us to redefine a function to provide it with a completely new definition. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Difference between C and C++		
Sr. No	**C**	**C++**
1	C was developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979 with C++'s predecessor "C with Classes".
2	When compared to C++, C is a	C++ is a superset of C. C++ can

	subset of C++.	run most of C code while C cannot run C++ code.
3	C supports procedural programming paradigm for code development.	C++ supports both procedural and object oriented programming paradigms; therefore C++ is also called a hybrid language.
4	C does not support object oriented programming; therefore it has no support for polymorphism, encapsulation, and inheritance.	Being an object oriented programming language C++ supports polymorphism, encapsulation, and inheritance.
5	In C, data and functions are separate and free entities.	In C++, data and functions are encapsulated together in form of an object. For creating objects class provides a blueprint of structure of the object.
6	In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding.	In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended.
7	C, being a procedural programming, it is a function driven language.	While, C++, being an object oriented programming, it is an object driven language.
8	C does not support function and operator overloading.	C++ supports both function and operator overloading.
9	C does not allow functions to be defined inside structures.	In C++, functions can be used inside a structure.
10	C uses functions for input/output. For example scanf and printf.	C++ uses objects for input output. For example cin and cout.
11	C does not support reference variables.	C++ supports reference variables.
12	C has no support for virtual and friend functions.	C++ supports virtual and friend functions.
13	C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.
14	C does not provide direct support for error handling (also called exception handling)	C++ provides support for exception handling. Exceptions are used for "hard" errors that make the code incorrect.

Structure of C++ Program

C++ is first Object oriented programming language.We have summarize structure of C++ Program in the following Picture –


Header File Declaration Section
Global Declaration Section
Class Declaration and Method Definition Section
Main Function
Method Definition Section

Structure of C++ Program

Section 1 : Header File Declaration Section

1. Header files used in the program are listed here.
2. Header File provides **Prototype declaration** for different library functions.
3. We can also include **user define header file**.
4. Basically all preprocessor directives are written in this section.

Section 2 : Global Declaration Section

1. Global Variables are declared here.
2. Global Declaration may include –
    - Declaring Structure
    - Declaring Class
    - Declaring Variable

Section 3 : Class Declaration Section

1. Actually this section can be considered as sub section for the global declaration section.
2. Class declaration and all methods of that class are defined here.

Section 4 : Main Function

1. Each and every C++ program always starts with main function.
2. This is entry point for all the function. Each and every method is called indirectly through main.

3. We can create class objects in the main.
4. Operating system call this function automatically.

Section 5 : Method Definition Section

1. This is optional section . Generally this method was used in C Programming.

Simple C++ Program

```
#include<iostream>      // Header Files
#include<conio.h>       // Header Files
int main()           //Main Function
{
    cout<<"Hello World";   //Standard Ouput Statement
    getch();               // Wait For Output Screen
    return 0;                          //Main Function return Statement
}
```

Simple Program for read user Input (Integer) Using cin

```
/* Add Two Integers Programs, Addition Programs,C++ Examples */

#include<iostream>      // Header Files
#include<conio.h>       // Header Files
//Main Function
int main()
{
  int a;      // Local Variable 'a' Declaration
  cout << "Simple Program for Read user Input (Integer) Using cin \n";
  cout << "Enter Number : ";
  cin >> a;
  cout << "Entered Input Is = " << a;
  getch();
  return 0;
}
```

C++ Basic Elements

Programming language is a set of rules, symbols, and special words used to construct programs. There are certain elements that are common to all programming languages. Now, we will discuss these elements in brief :

C++ Character Set

Character set is a set of valid characters that a language can recognize.

Letters	A-Z, a-z
Digits	0-9
Special Characters	Space + - * / ^ \ () [] {} = != <> ‘ “ $, ; : % ! & ? _ # <= >= @
Formatting characters	backspace, horizontal tab, vertical tab, form feed, and carriage return

Tokens

A token is a group of characters that logically belong together. The programmer can write a program by using tokens. C++ uses the following types of tokens.
Keywords, Variables, Literals, Punctuators, Operators.

1. Keywords

These are some reserved words in C++ which have predefined meaning to compiler called keywords. It is discussed in previous section.

2. Variables

Symbolic names can be used in C++ for various data items used by a programmer in his program. Variable are used in C++, where we need storage for any value, which will change in program. The variable is a sequence of characters taken from C++ character set. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable. The rule for the formation of an variable are:

- An variable can consist of alphabets, digits and/or underscores.
- It must not start with a digit
- C++ is case sensitive that is upper case and lower case letters are considered different from each other.
- It should not be a reserved word.

3. Constants / Literals

Literals (often referred to as constants) are data items that never change their value during the execution of the program. The following types of literals are available in C++.

- Integer-Constants
- Character-constants
- Floating-constants
- Strings-constants

Integer Constants

Integer constants are whole number without any fractional part. C++ allows three types of integer constants.
Decimal integer constants : It consists of sequence of digits and should not begin with 0 (zero). For example 124, - 179, +108.
Octal integer constants: It consists of sequence of digits starting with 0 (zero). For example. 014, 012.
Hexadecimal integer constant: It consists of sequence of digits preceded by ox or OX.

Character constants

A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks. For example 'A', '9', etc. C++ allows nongraphic characters which cannot be typed directly from keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character.

Floating constants

They are also called real constants. They are numbers having fractional parts. They may be written in fractional form or exponent form. A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits. For example 3.0, -17.0, -0.627 etc.

String Literals

A sequence of character enclosed within double quotes is called a string literal. String literal is by default (automatically) added with a special character '\0' which denotes the end of the string. Therefore the size of the string is increased by one character. For example "COMPUTER" will re represented as "COMPUTER\0" in the memory and its size is 9 characters.

4. Punctuators

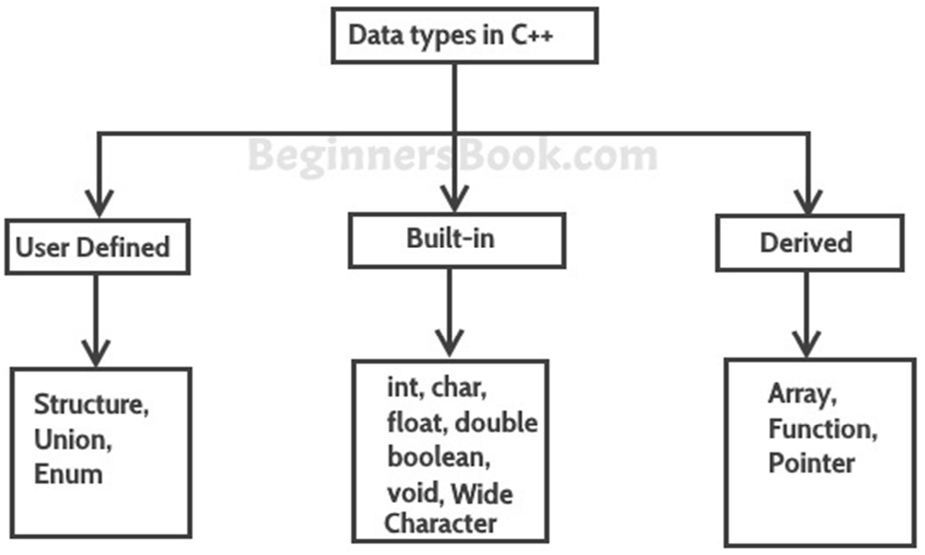
The following characters are used as punctuators in C++.

Brackets [ ]	Opening and closing brackets indicate single and multidimensional array subscript.
Parentheses ( )	Opening and closing brackets indicate functions calls,; function parameters for grouping expressions etc.
Braces { }	Opening and closing braces indicate the start and end of a compound statement.
Comma ,	It is used as a separator in a function argument list.
Semicolon ;	It is used as a statement terminator.
Colon :	It indicates a labeled statement or conditional operator symbol.
Asterisk *	It is used in pointer declaration or as multiplication operator.
Equal sign =	It is used as an assignment operator.
Pound sign #	It is used as pre-processor directive.

5. Operators

Operators are special symbols used for specific purposes. C++ provides six types of operators. Arithmetical operators, Relational operators, Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator

Data Types in C++



Basic C++ data types –

Type	Keyword	Typical Bit Width
Character	char	1byte
Integer	int	2bytes
Floating point	float	4bytes
Double floating point	double	8bytes

Example :

```
char a = 'A';          // character type

int a = 1;             // integer type

float a = 3.14159;     // floating point type

double a = 10098.98899;      // double type
```

Modifiers
Specifiers modify the meanings of the predefined built-in data types and expand them to a much larger set. There are four data type modifiers in C++, they are :

1. long
2. short
3. signed
4. unsigned

Below mentioned are some important points you must know about the modifiers,

- **long** and **short** modify the maximum and minimum values that a data type will hold.
- A plain int must have a minimum size of **short**.
- Size hierarchy : short int < int < long int
- Size hierarchy for floating point numbers is : float < double < long double
- **long float** is not a legal type and there are no **short floating point** numbers.

- **Signed** types includes both positive and negative numbers and is the default type.
- **Unsigned**, numbers are always without any sign, that is always positive.

User-defined data types
We have three types of user-defined data types in C++
1. struct
2. enum

Structures in C++

Structure is a compound data type that contains different variables of different types. For example, you want to store Student details like student name, student roll num, student age.

For example,

```
struct Student
{
   char stuName[30];
   int stuRollNo;
   int stuAge;
};
```

Now these three members combined will act like a separate variable and you can create structure variable like this:

```
        structure_name variable_name;
```

So if you want to hold the information of two students using this structure then you can do it like this:

```
        Student s1, s2;
```

Then we can access the members of Student structure like this:

```
s1.stuName = "Aksha";          //Assigning name to first student
s2.stuAge = 8;                 //Assigning age to the second student
```

Example in C++

```
#include <iostream>
struct Student
{
   char stuName[30];
   int stuRollNo;
   int stuAge;
};
int main()
{
   Student s;
   cout<<"Enter Student Name: ";
   cin.getline(s.stuName, 30);
   cout<<"ENter Student Roll No: ";
   cin>>s.stuRollNo;
   cout<<"Enter Student Age: ";
   cin>>s.stuAge;
   cout<<"Student Record:"<<endl;
   cout<<"Name: "<<s.stuName<<endl;
   cout<<"Roll No: "<<s.stuRollNo<<endl;
   cout<<"Age: "<<s.stuAge;
   return 0;
}
```

Output:

```
Enter Student Name: Aksha
ENter Student Roll No: 006
Enter Student Age: 8
Student Record:
Name: Aksha
Roll No: 006
Age: 8
```

Enumeration in C++

Enum is a user defined data type where we specify a set of values for a variable and the variable can only take one out of a small set of possible values. We use enum keyword to define a Enumeration.

```
          enum direction {East, West, North, South}dir;
```

Here Enumeration name is direction which can only take one of the four specified values, the dir at the end of the declaration is an enum variable.

Simple enum Example

```
#include <iostream>
enum direction {East, West, North, South};
int main()
{
  direction dir;
  dir = South;
  cout<<dir;
  return 0;
}
```

Output:

```
3
```

How to change default values of Enum

```
#include <iostream>
enum direction {East=11, West=22, North=33, South=44};
int main()
{
  direction dir;
  dir = South;
  cout<<dir;
  return 0;
}
```

Output:

```
44
```

Typecasting in C++

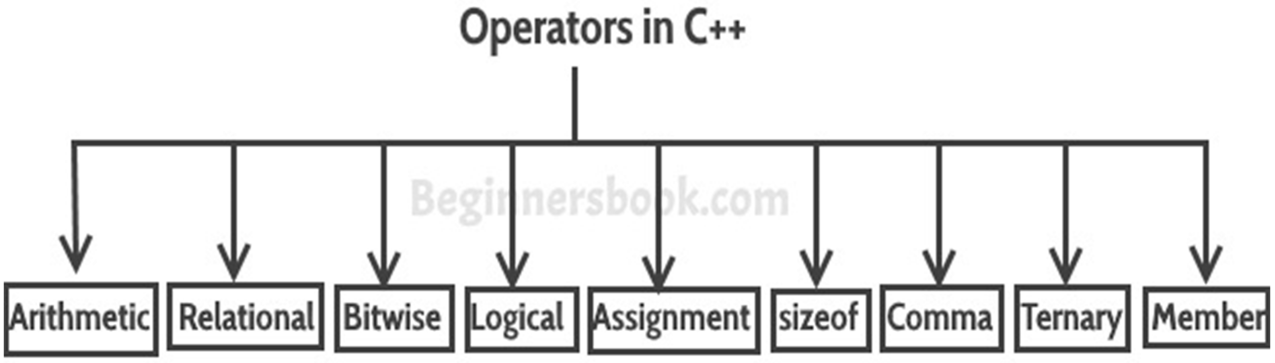
Typecasting is making a variable of one type, such as an int, act like another type, a char, for one single operation. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. (char)a will make 'a' function as a char.

```
#include <iostream>
int main()
{
        for ( int x = 0; x < 128; x++ )
         {
                cout<< x <<". "<< (char)x <<" ";
         }
         return 0;
}
```

Operators in C++

Operator represents an action. For example + is an operator that represents addition. An operator works on two or more operands and produce an output. For example 3+4+5 here + operator works on three operands and produce 12 as output.

Types of Operators in C++



1) Basic Arithmetic Operators
2) Assignment Operators
3) Auto-increment and Auto-decrement Operators
4) Logical Operators
5) Comparison (relational) operators
6) Bitwise Operators
7) Ternary Operator

1) Basic Arithmetic Operators

Basic arithmetic operators are: +, -, *, /, %
Consider that we have A = 20 and B = 10

Operator	Description	Example
+	Adds two operands or variables	A + B = 30
-	Subtracts second operand from the first	A – B = 10
*	Multiplies both operands	A * B = 200
/	Divides numerator by denominator	A / B = 2
%	After dividing the numberator by denominator remainder will be returned after division	A % B = 0

Note: Modulo operator returns remainder, for example 20 % 5 would return 0

Example of Arithmetic Operators

```
#include <iostream>

int main()
{
   int num1 = 240;
   int num2 = 40;
   cout<<"num1 + num2: "<<(num1 + num2)<<endl;
   cout<<"num1 - num2: "<<(num1 - num2)<<endl;
   cout<<"num1 * num2: "<<(num1 * num2)<<endl;
   cout<<"num1 / num2: "<<(num1 / num2)<<endl;
   cout<<"num1 % num2: "<<(num1 % num2)<<endl;
   return 0;
}
```

Output:

```

num1 + num2: 280
num1 - num2: 200
num1 * num2: 9600
num1 / num2: 6
num1 % num2: 0
```

2) Assignment Operators

Assignments operators in C++ are: =, +=, -=, *=, /=, %=

num2 = num1 would assign value of variable num1 to the variable.

num2+=num1 is equal to num2 = num2+num1

num2-=num1 is equal to num2 = num2-num1

num2*=num1 is equal to num2 = num2*num1

num2/=num1 is equal to num2 = num2/num1

num2%=num1 is equal to num2 = num2%num1

Example of Assignment Operators

```
#include <iostream>

int main()
{
        int num1 = 240;
        int num2 = 40;
        num2 = num1;
        cout<<"= Output: "<<num2<<endl;
        num2 += num1;
        cout<<"+= Output: "<<num2<<endl;
        num2 -= num1;
        cout<<"-= Output: "<<num2<<endl;
        num2 *= num1;
        cout<<"*= Output: "<<num2<<endl;
        num2 /= num1;
        cout<<"/= Output: "<<num2<<endl;
        num2 %= num1;
        cout<<"%= Output: "<<num2<<endl;
        return 0;
}
```

Output:

```
= Output: 240
+= Output: 480
-= Output: 240
*= Output: 57600
/= Output: 240
%= Output: 0
```

3) Auto-increment and Auto-decrement Operators

++ and --

Operator	Description	Example
++	Increment operator will increases integer value by one	A++ = 21
--	Decrement operator will decreases integer value by one	A-- = 20

Example of Auto-increment and Auto-decrement Operators

```
#include <iostream>

int main()
{
   int num1 = 240;
   int num2 = 40;
   num1++; num2--;
   cout<<"num1++ is: "<<num1<<endl;
   cout<<"num2-- is: "<<num2;
   return 0;
}
```

Output:

```
num1++ is: 241
num2-- is: 39
```

4) Logical Operators

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition.

Logical operators in C++ are: &&, ||, !

Consider that A = 0 and B = 0

Operator	Description	Example
Logical AND (&&)	If both the operands are non-zero then only condition becomes true	(A && B) is false.
Logical OR (||)	If both the operands are zero then only condition becomes false	(A || B) is true.
Logical NOT (!)	It will reverses the state of its operand i.e true will become false	(!A) is true.

Example of Logical Operators

```
#include <iostream>

int main()
{
   bool b1 = true;
   bool b2 = false;
   cout<<"b1 && b2: "<<(b1&&b2)<<endl;
   cout<<"b1 || b2: "<<(b1||b2)<<endl;
   cout<<"!(b1 && b2): "<<!(b1&&b2);
   return 0;
}
```

Output:

```
b1 && b2: 0
b1 || b2: 1
!(b1 && b2): 1
```

5) Relational operators

We have six relational operators in C++: ==, !=, >, <, >=, <=

Consider that A = 40 and B = 20

Symbol	Meaning	Example
>	Greater than	A > B returns true
<	Less than	A < B returns false
>=	Greater than equal to	A >= B returns false
<=	Less than equal to	A <= B returns false
==	Equal to	A == B returns false
!=	Not equal to	A != B returns true

Example of Relational operators

```
#include <iostream>

int main()
{
   int num1 = 240;
   int num2 =40;
   if (num1==num2)
   {
      cout<<"num1 and num2 are equal"<<endl;
   }
   else
   {
      cout<<"num1 and num2 are not equal"<<endl;
    }
   if( num1 != num2 )
   {
      cout<<"num1 and num2 are not equal"<<endl;
   }
   else
  {
      cout<<"num1 and num2 are equal"<<endl;
   }
   if( num1 > num2 )
   {
      cout<<"num1 is greater than num2"<<endl;
   }
   else
   {
      cout<<"num1 is not greater than num2"<<endl;
   }
   if( num1 >= num2 )
   {
      cout<<"num1 is greater than or equal to num2"<<endl;
   }
   else
   {
      cout<<"num1 is less than num2"<<endl;
   }
   if( num1 < num2 )
   {
      cout<<"num1 is less than num2"<<endl;
   }
   else
   {
      cout<<"num1 is not less than num2"<<endl;
   }
   if( num1 <= num2)
  {
      cout<<"num1 is less than or equal to num2"<<endl;
   }
   else
   {
      cout<<"num1 is greater than num2"<<endl;
   }
   return 0;
}
```

Output:

```
num1 and num2 are not equal
num1 and num2 are not equal
num1 is greater than num2
num1 is greater than or equal to num2
num1 is not less than num2
num1 is greater than num2
```

6) Bitwise Operators

There are six bitwise Operators: &, |, ^, ~, <<, >>

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

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

A = 0011 1100

B = 0000 1101

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

~num1 is a complement operator that just changes the bit from 0 to 1 and 1 to 0. In our example it would return -12 which is signed 8 bit equivalent to 11110100

num1 << 2 is left shift operator that moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. In our case output is 44 which is equivalent to 00101100

Note: In the example below we are providing 2 at the right side of this shift operator that is the reason bits are moving two places to the left side. We can change this number and bits would be moved by the number of bits specified on the right side of the operator. Same applies to the right side operator.

num1 >> 2 is right shift operator that moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. In our case output is 2 which is equivalent to 00000010

Example of Bitwise Operators

```
#include <iostream>

int main()
{
   int num1 = 11;  /* 11 = 00001011 */
   int num2 = 22;  /* 22 = 00010110 */
   int result = 0;
   result = num1 & num2;
   cout<<"num1 & num2: "<<result<<endl;
   result = num1 | num2;
   cout<<"num1 | num2: "<<result<<endl;
   result = num1 ^ num2;
   cout<<"num1 ^ num2: "<<result<<endl;
   result = ~num1;
   cout<<"~num1: "<<result<<endl;
   result = num1 << 2;
   cout<<"num1 << 2: "<<result<<endl;
   result = num1 >> 2;
   cout<<"num1 >> 2: "<<result;
   return 0;
}
```

Output:

```
num1 & num2: 2
num1 | num2: 31
```

```
num1 ^ num2: 29
~num1: -12
num1 << 2: 44 num1 >> 2: 2
```

7) Ternary Operator

This operator evaluates a boolean expression and assign the value based on the result.
Syntax:

```
variable num1 = (expression) ? value if true : value if false
```

If the expression results true then the first value before the colon (:) is assigned to the variable num1 else the second value is assigned to the num1.

Example of Ternary Operator

```
#include <iostream>

int main()
{
  int num1, num2; num1 = 99;
  /* num1 is not equal to 10 that's why
   * the second value after colon is assigned
   * to the variable num2
   */
  num2 = (num1 == 10) ? 100: 200;
  cout<<"num2: "<<num2<<endl;
  /* num1 is equal to 99 that's why
   * the first value is assigned
   * to the variable num2
   */
  num2 = (num1 == 99) ? 100: 200;
  cout<<"num2: "<<num2;
  return 0;
}
```

Output:

```
num2: 200
num2: 100
```

Miscellaneous Operators

sizeof operator.

- sizeof is a compile-time operator used to calculate the size of data type or variable.
- sizeof operator will return the size in integer format.
- sizeof is a keyword.
- sizeof operator can be nested.

Syntax of sizeof Operator:

```
sizeof(data type)
```

Data type include variables, constants, classes, structures, unions, or any other user defined data type.

Example of sizeof operator:

```
#include <iostream>
int main()
{
   int i;
   char c;
   cout << "Size of variable i : " << sizeof(i) << endl;
   cout << "Size of variable c : " << sizeof(c) << endl;
   return 0;
}
```

Output:

```
Size of variable i : 4
Size of variable c : 1
```

Operator Precedence in C++

This determines which operator needs to be evaluated first if an expression has more than one operator. Operator with higher precedence at the top and lower precedence at the bottom.

Unary Operators
++ − − ! ~

Multiplicative
* / %

Additive
+ –

Shift
<< >> >>>

Relational
> >= < <=

Equality
== !=

Bitwise AND
&
Bitwise XOR
^

Bitwise OR
|

Logical AND
&&

Logical OR
||

Ternary
?:

Assignment
= += -= *= /= %= > >= < <= &= ^= |=

C++ Decision Making Statements

Decision-making is an important concept in any programming language and to accomplish this, C++ uses the following decision making statements:

- if statement
- if..else statement
- switch statement
- conditional operator

if statement

An if statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an if statement in C++ is −

```
if(boolean_expression)
{
  // statement(s) will execute if the boolean expression is true
}
```

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

For example,

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

int main ()
{
  int a = 10;              // local variable declaration:

  if( a < 20 )   // check the boolean condition
  {
    cout << "a is less than 20;" << endl;
   }
  cout << "value of a is : " << a << endl;

  getch();
  return 0;
}
```

Output:

```
a is less than 20;
value of a is : 10
```

if...else statement

An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

Syntax

The syntax of an if...else statement in C++ is −

```
if(boolean_expression)
{
        // statement(s) will execute if the boolean expression is true
}
else
{
        // statement(s) will execute if the boolean expression is false
}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

For example,

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

int main ()
{
  int a = 100;
  if( a < 20 )
  {
    cout << "a is less than 20;" << endl;
  }
  else
  {
    cout << "a is not less than 20;" << endl;
  }
  cout << "value of a is : " << a << endl;

  getch();
  return 0;
}
```

Output:

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

if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very usefull to test various conditions using single if...else if statement.

When using if, else if, else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in C++ is −

```
if(boolean_expression 1)
{
  // Executes when the boolean expression 1 is true
}
else if( boolean_expression 2)
{
  // Executes when the boolean expression 2 is true
}
else if( boolean_expression 3)
{
  // Executes when the boolean expression 3 is true
}
else
{
  // executes when the none of the above condition is true.
}
```

For example,

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

int main ()
{
  int a = 100;

  if( a == 10 )
  {
    cout << "Value of a is 10" << endl;
  }
  else if( a == 20 )
  {
    cout << "Value of a is 20" << endl;
  }
  else if( a == 30 )
  {
    cout << "Value of a is 30" << endl;
  }
  else
  {
    // if none of the conditions is true
    cout << "Value of a is not matching" << endl;
  }
  cout << "Exact value of a is : " << a << endl;

  getch();
  return 0;
}
```

Output:

```
Value of a is not matching
Exact value of a is : 100
```

switch statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The syntax for a **switch** statement in C++ is as follows −

```
switch(expression)
{
  case constant-expression  :
    statement(s);
    break; //optional
  case constant-expression  :
    statement(s);
    break; //optional

  // you can have any number of case statements.
  default : //Optional
    statement(s);
}
```

For example,

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

int main ()
{
  char grade = 'D';

  switch(grade)
  {
    case 'A' :
      cout << "Excellent!" << endl;
      break;
    case 'B' :
        cout << "Very Good!" << endl;
      break;

    case 'C' :
      cout << "Good" << endl;
      break;
    case 'D' :
      cout << "You passed" << endl;
      break;
    case 'F' :
      cout << "Better try again" << endl;
      break;
    default :
      cout << "Invalid grade" << endl;
  }
  cout << "Your grade is " << grade << endl;

  getch();
  return 0;
}
```

Output:

```
You passed
Your grade is D
```

The conditional Operator- ? :

Conditional operator “? :” can be used to replace if...else statements. It has the following general form −

```
Exp1 ? Exp2 : Exp3;
```

Exp1, Exp2, and Exp3 are expressions. The value of a ‘?’ expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ‘?’ expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

For example,

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

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

  x = (y < 10) ? 30 : 40;
  cout << "value of x: " << x << endl;

  getch();
  return 0;
}
```

Output:

```
value of x: 40
```

Looping Statements

There may be a situation, when you need to execute a block of code several numbers of times. Programming languages provide various loop statements that allows us to execute a statement or group of statements multiple times.

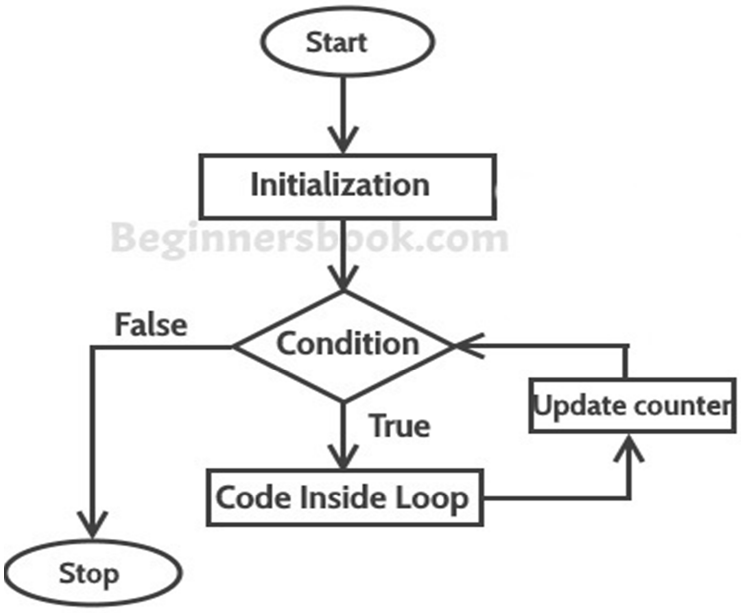
In C++ we have three types of basic loops:
1. For
2. while
3. do-while

For loop

Syntax of for loop

```
for(initialization; condition ; increment/decrement)
{
  C++ statement(s);
}
```

Flow of Execution of the for Loop



For example,

Display elements of array using for loop

```
#include <iostream.h>
#include<conio.h>
int main()
{
  int arr[]={21,9,56,99, 202};
  for(int i=0; i<5; i++)
  {
    cout<<arr[i]<<endl;
  }

  getch();
  return 0;
}
```

Output:

```
21
9
56
99
202
```

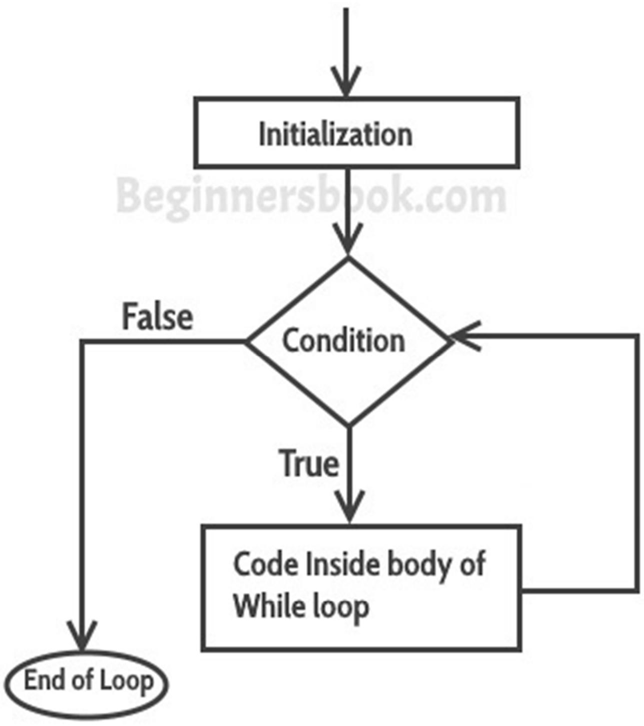
While loop

Syntax of while loop

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

In while loop, condition is evaluated first and if it returns true then the statements inside while loop execute, this happens repeatedly until the condition returns false. When condition returns false, the control comes out of loop and jumps to the next statement in the program after while loop.

Flow Diagram of While loop



For example,

Displaying the elements of array using while loop

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

int main()
{
  int arr[]={21,87,15,99, -12};
  int i=0;
  while(i<5)
  {
    cout<<arr[i]<<endl;
    i++;
  }

  getch();
  return 0;
}
```

Output:

```
21
87
15
99
-12
```

do-while loop

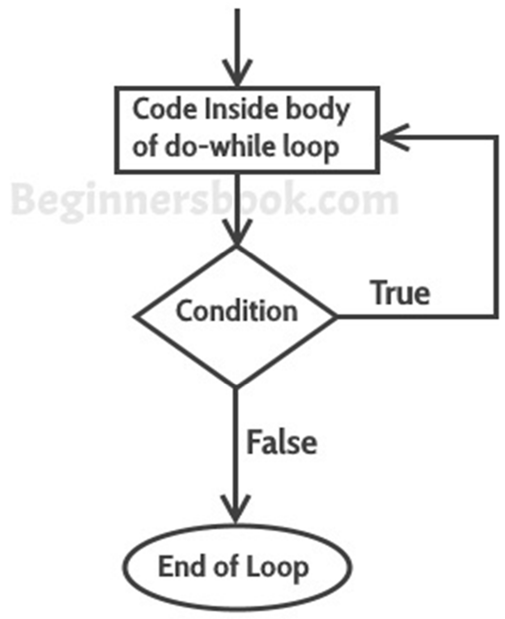
do-while loop is similar to while loop, however there is a difference between them: In while loop, condition is evaluated first and then the statements inside loop body gets executed, on the other hand in do-while loop, statements inside do-while gets executed first and then the condition is evaluated.

Syntax of do-while loop

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

First, the statements inside loop execute and then the condition gets evaluated, if the condition returns true then the control jumps to the "do" for further repeated execution of it, this happens repeatedly until the condition returns false. Once condition returns false control jumps to the next statement in the program after do-while.

Flow Diagram of do-While loop



For example,

Displaying array elements using do-while loop

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

int main()
{
  int arr[]={21,99,15,109};
  int i=0;
  do
  {
    cout<<arr[i]<<endl;
    i++;
  }while(i<4);

  getch();
  return 0;
}
```

Output:

```
21
99
15
109
```

Continue Statement

Continue statement is used inside loops. Whenever a continue statement is encountered inside a loop, control directly jumps to the beginning of the loop for next iteration, skipping the execution of statements inside loop’s body for the current iteration.

Syntax of continue statement

```
continue;
```

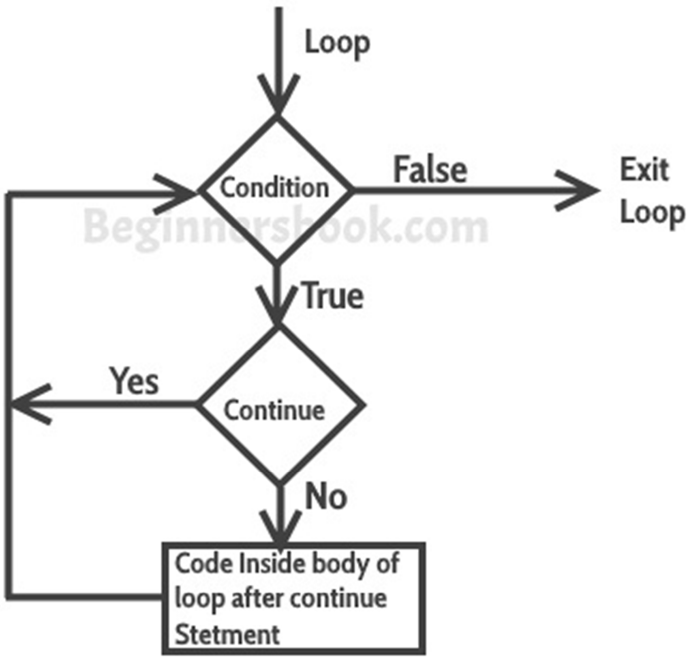
For example,

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

int main()
{
  for (int num=0; num<=6; num++)
  {
    /* This means that when the value of num is equal to 3 this continue
     * statement would be encountered, which would make the control to
     * jump to the beginning of loop for next iteration, skipping the current
     * iteration
     */

       if (num==3)
       {
              continue;
       }
       cout<<num<<" ";
  }

  getch();
  return 0;
}
```

Output:

```
0 1 2 4 5 6
```

Flow Diagram of Continue Statement



Break statement

The **break statement** is used in following two scenarios:

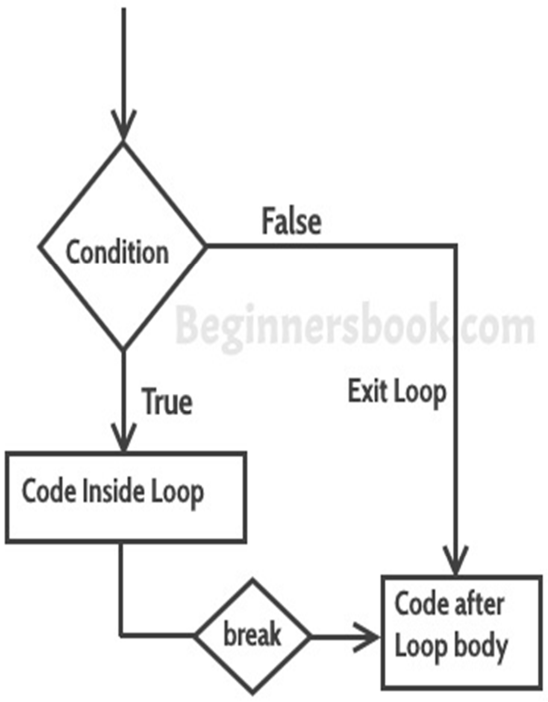
a) Use break statement to come out of the loop instantly. Whenever a break statement is encountered inside a loop, the control directly comes out of loop terminating it. It is used along with if statement, whenever used inside loop(see the example below) so that it occurs only for a particular condition.

b) It is used in switch case control structure after the case blocks. Generally all cases in switch case are followed by a break statement to avoid the subsequent cases (see the example below) execution. Whenever it is encountered in switch-case block, the control comes out of the switch-case body.

Syntax of break statement

break;

break statement flow diagram



For example,

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

int main()
{
  int num =10;
  while(num<=200)
  {
    cout<<"Value of num is: "<<num<<endl;
    if (num==12)
    {
      break;
    }
    num++;
  }
  cout<<"Hey, I'm out of the loop";

  getch();
  return 0;
}
```

Output:

```
Value of num is: 10
Value of num is: 11
Value of num is: 12
Hey, I'm out of the loop
```

Scope resolution operator

- In C++ language the scope resolution operator is written "::".
- C++ supports to the global variable from a function, Local variable is to define the same function name.
- Identify variables with use of scope resolution operator when we use the same name for local variable and global variable (or in class or namespace)
- Resolution operator is placed between the in front of the variable name then the global variable is affected. If no resolution operator is placed between the local variable is affected.

Scope Resolution Operator Syntax

```
:: identifier   // for Global Scope
class-name :: identifier  // for Class Scope
namespace :: identifier   // for Namespace Scope

//simple syntax
:: global variable name
```

For example,

```
// Example program for Scope Resolution Operator Global Variable

#include<iostream.h>
#include<conio.h>

int n = 12; //global variable

int main()
{
   int n = 13; //local variable
   cout << ::n << endl; //print global variable:12
   cout << n << endl; //print the local variable:13

   getch();
   return 0;
}
```

Output:

```
12
13
```

Scope Resolution Operator Example Program: For Function

```
/* Simple scope resolution operator Example for Function defined in C++*/
#include <iostream.h>
#include<conio.h>


class ScopeFn
{
        public:
                void print();  //function declaration
};

void ScopeFn::print()
{
  cout << "Function defined using scope resolution operator.\n";
}

int main()
{
   ScopeFn obj;
   obj.print();

   getch();
   return 0;

}
```

Output:

```
Function defined using scope resolution operator
```

Memory Management Operators in C++

C++ has new operator to allocate memory and delete operator to release the memory.

new operator

The **new operator** allocates memory dynamically to a variable and returns the base address of that memory to a pointer.

Syntax

pointer_name = new datatype;

Delete operator

Delete operator is used to release the memory allocated in the heap using new operator.

Syntax

delete pointer_name;

Example,

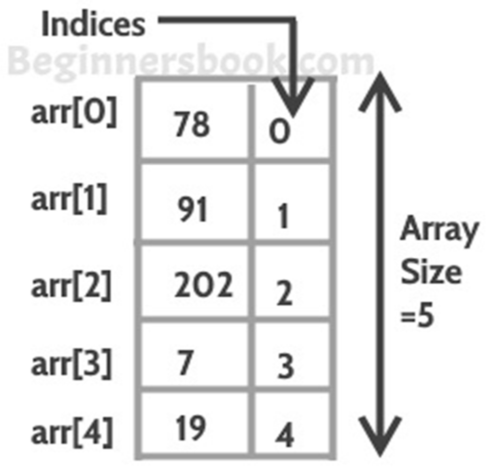
```
#include<iostream.h>
#include<conio.h>

void main()
{

          int size,i;
          int *ptr;
          cout<<"\n\tEnter size of Array : ";
          cin>>size;

          ptr = new int[size];

          for(i=0;i<5;i++)
          {
                     cout<<"\nEnter any number : ";
                     cin>>ptr[i];
          }

          for(i=0;i<5;i++)
          {
                    cout<<ptr[i]<<", ";
          }

          delete[] ptr;
          getch();
          return 0;

}
```

Output :

```
Enter size of Array : 5

Enter any number : 78
Enter any number : 45
Enter any number : 12
Enter any number : 89
Enter any number : 56

78, 45, 12, 89, 56,
```

Arrays

Array is a group of similar types of elements that have contiguous memory location. Array index starts from 0. We can store only fixed set of elements in array.



Declaring an array in C++

There are couple of ways to declare an array.

Method 1:

```
int arr[5];
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
```

Method 2:

```
int arr[] = {10, 20, 30, 40, 50};
```

Method 3:

```
int arr[5] = {10, 20, 30, 40, 50};
```

For example,

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

int main()
{
  int arr[] = {11, 22, 33, 44, 55};
  int n=0;

  while(n<=4)
  {
    cout<<arr[n]<<endl;
    n++;
  }
  getch();
  return 0;
}
```

Strings

Strings are known as sequence of characters. In C++ we have two ways to create and use strings: 1) By creating char arrays and treat them as string 2) By creating string object

Getting user input as string

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

int main()
{
  char book[50];
  cout<<"Enter your favorite book name:";
  //reading user input
  cin>>book;
  cout<<"You entered: "<<book;

  getch();
  return 0;
}
```

Output:

```
Enter your favorite book name:The Murder of Roger Ackroyd
You entered: The
```

Correct way of capturing user input string using cin.get

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

int main()
{
  char book[50];
  cout<<"Enter your favorite book name:";
```

```
   //reading user input
   cin.get(book, 50);
   cout<<"You entered: "<<book;

   getch();
   return 0;
}
```

Output:

```
Enter your favorite book name:The Murder of Roger Ackroyd
You entered: The Murder of Roger Ackroyd
```

Drawback of this method

1) Size of the char array is fixed, which means the size of the string created through it is fixed in size, more memory cannot be allocated to it during runtime.

String object in C++

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

int main()
{
   string str;
   cout<<"Enter a String:";
   getline(cin,str);
   cout<<"You entered: ";
   cout<<str<<endl;

   getch();
   return 0;
}
```

Output:

```
Enter a String:XYZ
You entered: XYZ
```