

# **Programming Using C++**

**MILAN KUMAR NAYAK**

(LECT.IN COMPUTER SCIENCE)

CONT.6370158954/9556472420

# History of C++

---

The C++ programming language has a history going back to 1979, when Bjarne Stroustrup was doing work for his Ph.D. thesis. He began work on "C with Classes", which as the name implies was meant to be a superset of the C language. His goal was to add object-oriented programming into the C language, which was and still is a language well-respected for its portability without sacrificing speed or low-level functionality.

His language included classes, basic inheritance, inlining, default function arguments, and strong type checking in addition to all the features of the C language. The first C with Classes compiler was called Cfront, which was derived from a C compiler called CPre. It was a program designed to translate C with Classes code to ordinary C.

In 1983, the name of the language was changed from C with Classes to C++. The ++ operator in the C language is an operator for incrementing a variable, which gives some insight into how Stroustrup regarded the language. Many new features were added around this time, the most notable of which are virtual functions, function overloading, references with the & symbol, the const keyword, and single-line comments using two forward slashes.

In 1985, C++ was implemented as a commercial product. The language was not officially standardized yet. The language was updated again in 1989 to include protected and static members, as well as an inheritance from several classes.

In 1990, Turbo C++ was released as a commercial product. Turbo C++ added a lot of additional libraries which have had a considerable impact on C++'s development.

In 1998, the C++ standards committee published the first international standard for C++ ISO/IEC 14882:1998, which is informally known as C++98. The Standard Template Library, which began its conceptual development in 1979, was also included. In 2003, the committee responded to multiple problems that were reported with their 1998 standard and revised it accordingly. The changed language was named C++03.

In mid-2011, the new C++ standard (C++11) was finished. The new features included Regex support, a randomization library, a new C++ time library, atomics support, a standard threading library, a new for loop syntax providing functionality similar to for each loops in certain other languages, the auto keyword, new container classes, better support for unions and array-initialization lists and variadic templates.

## **Object-oriented programming (OOP)**

Object-oriented programming (OOP) is a programming language model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

Simply put, OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. Due to the organization of an object-oriented program, this method is also conducive to collaborative development where projects can be divided into groups. Additional benefits of OOP include code reusability, scalability and efficiency.

The first step in OOP is to identify all of the objects a programmer wants to manipulate and how they relate to each other, an exercise often known as data modeling. Examples of an object can range from physical entities, such as a human being that is described by properties like name and address, down to small computer programs, such as widgets. Once an object is known, it is generalized as a class of objects that defines the kind of data it contains and any logic sequences that can manipulate it.

Each distinct logic sequence is known as a method and objects can communicate with well-defined interfaces called messages.

## **Principle of Object Oriented Programming Language**

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

## **Concepts in OOP's**

### **- Object**

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

### **- Class**

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

- A Language is called as OOP Language if a Language Provides us the Following Concepts Are Five Concepts in OOP's:-

**1) Data Abstraction:** - Data Abstraction is that in which A User Can use any of the data and Method from the Class Without knowing about how this is created So in other words we can say that A user can use all the Functions without Knowing about its detail For Example When a User gives Race to Car The Car will be Moved but a User doesn't know how its Engine Will Work.

**2) Inheritance:-** Inheritance is very popular Concept in OOP's This provides the Capability to a user to use the Predefined Code or the code that is not created by the user himself but if he may wants to use that code then he can use that code This is Called Inheritance but Always Remember in Inheritance a user only using the code but he will not be able to change the code that is previously created he can only use that code.

**3) Data Encapsulation :-** Data Encapsulation is also Known as Data Hiding as we know with the inheritance concept of OOP's a user can use any code that is previously created but if a user wants to use that code then it is must that previously code must be Public as the name suggests public means for other peoples but if a code is Private then it will be known as Encapsulate and user will not be able to use that code So With the help of OOP's we can alter or change the code means we can make the Code as Private or public This allows us to make our code either as public or private

**4) Polymorphism :-** Poly Means many and morphism means many function The Concepts Introduces in the form of Many behaviors of an object Like an Operator + is used for Addition of Two Numbers and + is also used for Joining two names The Polymorphism in C++ Introduces in the Form of Functions Overloading and in the Form of Constructor Overloading

**5) Dynamic Binding:-** Binding is used when we call the Code of the Procedure in Binding all the Code that is Linked with the single procedure is Called When a Call is Made to that Procedure Then the Compiler will found the Entire code of the Single Procedure if A Compiler will Fond all the Code of Single Procedure in Compile Time then it is Called as the Early Binding Because Compiler Knows about the code at the time of Compilation but in the Late Binding Compiler will understand all the Code at Run Time or at the Time of the Execution.

**6) Message Communication:** - Message Communication is occurred when an object passes the Call to Method of Class for Execution We Know for executing any method from the class First we have to create the object of class when an object passes References to function of class then In Message Communication First of all we have to Create the Object of the Class the we make Communication between the Object and the Methods of the Class.

### **Benefits of OOPs**

- OOP provides a clear modular structure for programs.
- It is good for defining abstract data types.
- Implementation details are hidden from other modules and other modules has a clearly defined interface.
- It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- objects, methods, instance, message passing, inheritance are some important properties provided by these particular languages
- encapsulation, polymorphism, abstraction are also counts in these fundamentals of programming language.
- It implements real life scenario.
- In OOP, programmer not only defines data types but also deals with operations applied for data structures.

### **Features Of Object Oriented Programming :**

---

- More reliable software development is possible.
- Enhanced form of c programming language.
- The most important Feature is that it's procedural and object oriented nature.
- Much suitable for large projects.
- Fairly efficient languages.
- It has the feature of memory management.

### **The characteristics of OOP are:**

1. Class definitions – Basic building blocks OOP and a single entity which has data and operations on data together
2. Objects – The instances of a class which are used in real functionality – its variables and
3. operations
4. Abstraction – Specifying what to do but not how to do ; a flexible feature for having a overall view of an object's functionality.
5. Encapsulation – Binding data and operations of data together in a single unit – A class adhere thi Feature
6. Inheritance and class hierarchy – Reusability and extension of existing classes

7. Polymorphism – Multiple definitions for a single name - functions with same name with different
8. functionality; saves time in investing many function names Operator and Function overloading
9. Generic classes – Class definitions for unspecified data. They are known as container classes. They are flexible and reusable.
10. Class libraries – Built-in language specific classes

## Applications of Object Oriented Programming

Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

**C++** is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. **C++** runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This **C++** tutorial adopts a simple and practical approach to describe the concepts of **C++** for beginners to advanced software engineers.

## Why to Learn C++

**C++** is a MUST for students and working professionals to become a great Software Engineer. I will list down some of the key advantages of learning C++:

- C++ is very close to hardware, so you get a chance to work at a low level which gives you a lot of control in terms of memory management, better performance and finally a robust software development.
- **C++ programming** gives you a clear understanding about Object Oriented Programming. You will understand low level implementation of polymorphism when you will implement virtual tables and virtual table pointers, or dynamic type identification.
- C++ is one of the every green programming languages and loved by millions of software developers. If you are a great C++ programmer then you will never sit without work and more importantly you will get highly paid for your work.
- C++ is the most widely used programming languages in application and system programming. So you can choose your area of interest of software development.
- C++ really teaches you the difference between compiler, linker and loader, different data types, storage classes, variable types their scopes etc.

There are 1000s of good reasons to learn C++ Programming. But one thing for sure, to learn any programming language, not only C++, you just need to code, and code and finally code until you become expert.

## Hello World using C++

Just to give you a little excitement about **C++ programming**, I'm going to give you a small conventional C++ Hello World program, You can try it using Demo link

C++ is a super set of C programming with additional implementation of object-oriented concepts.

```
#include <iostream>
using namespace std;

// main() is where program execution begins.
int main() {
    cout << "Hello World"; // prints Hello World
    return 0; }
```

There are many C++ compilers available which you can use to compile and run above mentioned program:

- Apple C++. Xcode
- Bloodshed Dev-C++
- Clang C++
- IBM C++
- Intel C++
- Microsoft Visual C++
- Oracle C++
- HP C++

It is really impossible to give a complete list of all the available compilers. The C++ world is just too large and too much new is happening.

## Applications of C++ Programming

As mentioned before, C++ is one of the most widely used programming languages. It has its presence in almost every area of software development. I'm going to list few of them here:

- **Application Software Development** - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.
- **Programming Languages Development** - C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX's C Shell, PHP and Python, and Verilog etc.
- **Computation Programming** - C++ is the best friends of scientists because of fast speed and computational efficiencies.
- **Games Development** - C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.
- **Embedded System** - C++ is being heavily used in developing Medical and Engineering Applications like softwares for MRI machines, high-end CAD/CAM systems etc.

This list goes on, there are various areas where software developers are happily using C++ to provide great softwares. I highly recommend you to learn C++ and contribute great softwares to the community.

## **Standard Libraries**

Standard C++ consists of three important parts –

- The core language giving all the building blocks including variables, data types and literals, etc.
- The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
- The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

## **The ANSI Standard**

The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.

The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

## **Learning C++**

The most important thing while learning C++ is to focus on concepts.

The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.

C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

## **Use of C++**

C++ is used by hundreds of thousands of programmers in essentially every application domain.

C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under realtime constraints.

C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.

Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.

## **Local Environment Setup**

If you are still willing to set up your environment for C++, you need to have the following two softwares on your computer.

### **Text Editor**

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or UNIX.

The files you create with your editor are called source files and for C++ they typically are named with the extension .cpp, .cp, or .c.

A text editor should be in place to start your C++ programming.

## C++ Compiler

This is an actual C++ compiler, which will be used to compile your source code into final executable program.

Most C++ compilers don't care what extension you give to your source code, but if you don't specify otherwise, many will use .cpp by default.

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective Operating Systems.

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
- **Methods** – A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** – Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

## C++ Program Structure

Let us look at a simple code that would print the words *Hello World*.

```
#include <iostream>
using namespace std;

// main() is where program execution begins.
int main() {
    cout << "Hello World"; // prints Hello World
    return 0;
}
```

Let us look at the various parts of the above program –

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
- The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
- The next line **// main() is where program execution begins.** is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line **int main()** is the main function where program execution begins.



- The next line `cout << "Hello World";` causes the message "Hello World" to be displayed on the screen.
- The next line `return 0;` terminates `main()` function and causes it to return the value 0 to the calling process.

## Compile and Execute C++ Program

Let's look at how to save the file, compile and run the program. Please follow the steps given below

- Open a text editor and add the code as above.
- Save the file as: `hello.cpp`
- Open a command prompt and go to the directory where you saved the file.
- Type `'g++ hello.cpp'` and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line and would generate `a.out` executable file.
- Now, type `'a.out'` to run your program.
- You will be able to see `'Hello World'` printed on the window.
- Make sure that `g++` is in your path and that you are running it in the directory containing file `hello.cpp`.
- You can compile C/C++ programs using makefile. For more details, you can check our ['Makefile Tutorial'](#).

## Semicolons and Blocks in C++

- In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.
- For example, following are three different statements –
  - `x = y;`
  - `y = y + 1;`
  - `add(x, y);`
- A block is a set of logically connected statements that are surrounded by opening and closing braces. For example –
  - ```
{
  cout << "Hello World"; // prints Hello World
  return 0;
}
```
- C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example –
  - `x = y;`
  - `y = y + 1;`
  - `add(x, y);`
  - is the same as
  - `x = y; y = y + 1; add(x, y);`

## C++ Identifiers

- A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores, and digits (0 to 9).
- C++ does not allow punctuation characters such as @, \$, and % within identifiers. C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.
- Here are some examples of acceptable identifiers –
  - mohd            zara        abc        move\_name    a\_123
  - myname50    \_temp        j        a23b9        retVal

## C++ Keywords

- The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

|            |          |                  |          |
|------------|----------|------------------|----------|
| asm        | else     | new              | this     |
| auto       | enum     | operator         | throw    |
| bool       | explicit | private          | true     |
| break      | export   | protected        | try      |
| case       | extern   | public           | typedef  |
| catch      | false    | register         | typeid   |
| char       | float    | reinterpret_cast | typename |
| class      | for      | return           | union    |
| const      | friend   | short            | unsigned |
| const_cast | goto     | signed           | using    |

|              |           |             |          |
|--------------|-----------|-------------|----------|
| continue     | if        | sizeof      | virtual  |
| default      | inline    | static      | void     |
| delete       | int       | static_cast | volatile |
| do           | long      | struct      | wchar_t  |
| double       | mutable   | switch      | while    |
| dynamic_cast | namespace | template    |          |

## • Trigraphs

- A few characters have an alternative representation, called a trigraph sequence. A trigraph is a three-character sequence that represents a single character and the sequence always starts with two question marks.
- Trigraphs are expanded anywhere they appear, including within string literals and character literals, in comments, and in preprocessor directives.
- Following are most frequently used trigraph sequences –

| Trigraph | Replacement |
|----------|-------------|
| ??=      | #           |
| ??/      | \           |
| ??'      | ^           |
| ??(      | [           |
| ??)      | ]           |
| ??!      |             |

|     |   |
|-----|---|
| ??< | { |
| ??> | } |
| ??- | ~ |

- All the compilers do not support trigraphs and they are not advised to be used because of their confusing nature.

## comments Line :-

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

C++ comments start with `/*` and end with `*/`. For example –

```
/* This is a comment */
```

```
/* C++ comments can also
 * span multiple lines
 */
```

## Data Types in C++

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types –

| Type           | Keyword |
|----------------|---------|
| Boolean        | bool    |
| Character      | char    |
| Integer        | int     |
| Floating point | float   |

|                       |         |
|-----------------------|---------|
| Double floating point | double  |
| Valueless             | void    |
| Wide character        | wchar_t |

Several of the basic types can be modified using one or more of these type modifiers –

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

| Type               | Typical Bit Width | Typical Range             |
|--------------------|-------------------|---------------------------|
| Char               | 1byte             | -127 to 127 or 0 to 255   |
| unsigned char      | 1byte             | 0 to 255                  |
| signed char        | 1byte             | -127 to 127               |
| Int                | 4bytes            | -2147483648 to 2147483647 |
| unsigned int       | 4bytes            | 0 to 4294967295           |
| signed int         | 4bytes            | -2147483648 to 2147483647 |
| short int          | 2bytes            | -32768 to 32767           |
| unsigned short int | 2bytes            | 0 to 65,535               |
| signed short int   | 2bytes            | -32768 to 32767           |

|                        |              |                                 |
|------------------------|--------------|---------------------------------|
| long int               | 8bytes       | -2,147,483,648 to 2,147,483,647 |
| signed long int        | 8bytes       | same as long int                |
| unsigned long int      | 8bytes       | 0 to 4,294,967,295              |
| long long int          | 8bytes       | $-(2^{63})$ to $(2^{63})-1$     |
| unsigned long long int | 8bytes       | 0 to 18,446,744,073,709,551,615 |
| Float                  | 4bytes       |                                 |
| Double                 | 8bytes       |                                 |
| long double            | 12bytes      |                                 |
| wchar_t                | 2 or 4 bytes | 1 wide character                |

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

[Live Demo](#)

```
#include <iostream>
using namespace std;

int main() {
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int) << endl;
    cout << "Size of short int : " << sizeof(short int) << endl;
    cout << "Size of long int : " << sizeof(long int) << endl;
    cout << "Size of float : " << sizeof(float) << endl;
    cout << "Size of double : " << sizeof(double) << endl;
    cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

    return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine –

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

## Variable in C++

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive –

There are following basic types of variable in C++ as explained in last chapter –

| Sr.No | Type & Description                                                           |
|-------|------------------------------------------------------------------------------|
| 1     | <b>bool</b><br>Stores either value true or false.                            |
| 2     | <b>char</b><br>Typically a single octet (one byte). This is an integer type. |
| 3     | <b>int</b><br>The most natural size of integer for the machine.              |
| 4     | <b>float</b><br>A single-precision floating point value.                     |
| 5     | <b>double</b><br>A double-precision floating point value.                    |
| 6     | <b>void</b><br>Represents the absence of type.                               |

7

**wchar\_t**

A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like **Enumeration, Pointer, Array, Reference, Data structures, and Classes**.

Following section will cover how to define, declare and use various types of variables.

## Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows –

```
type variable_list;
```

Here, **type** must be a valid C++ data type including char, w\_char, int, float, double, bool or any user-defined object, etc., and **variable\_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here –

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows –

```
type variable_name = value;
```

Some examples are –

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;         // definition and initializing d and f.
byte z = 22;              // definition and initializes z.
char x = 'x';             // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

## Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.



## Example

Try the following example where a variable has been declared at the top, but it has been defined inside the main function –

```
#include <iostream>
using namespace std;

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {
    // Variable definition:
    int a, b;
    int c;
    float f;

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

    cout << c << endl ;

    f = 70.0/3.0;
    cout << f << endl ;

    return 0;
}
```

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

```
30
23.3333
```

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example –

```
// function declaration
int func();
int main() {
    // function call
    int i = func();
}

// function definition
int func() {
    return 0;
}
```

1. `#include <iostream>`

# Simple C++ Programs

C++ programs are frequently asked in the interview. These programs can be asked from basics, array, string, pointer, linked list, file handling etc. Let's see the list of top c++ programs.

---

## Fibonacci Series

```
using namespace std;
int main() {
int n1=0,n2=1,n3,i,number;
cout<<"Enter the number of elements: ";
cin>>number;
cout<<n1<<" "<<n2<<" "; //printing 0 and 1
for(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed
{
n3=n1+n2;
cout<<n3<<" ";
n1=n2;
n2=n3;
}
return 0;
}
```

**Input:** 10

**Output:** 0 1 1 2 3 5 8 13 21 34

## 2) Prime number

Write a c++ program to check prime number.

```
#include <iostream>
using namespace std;
int main()
{
int n, i, m=0, flag=0;
cout << "Enter the Number to check Prime: ";
cin >> n;
m=n/2;
for(i = 2; i <= m; i++)
{
```

```

if(n % i == 0)
{
cout<<"Number is not Prime."<<endl;
flag=1;
break;
}
}
if (flag==0)
cout << "Number is Prime."<<endl;
return 0;
}

```

**Output:-**

```

Enter the Number to check Prime: 17
Number is Prime.
Enter the Number to check Prime: 57
Number is not Prime.

```

### 3) Palindrome number

Write a c++ program to check palindrome number.

```

#include <iostream>
using namespace std;
int main()
{
int n,r,sum=0,temp;
cout<<"Enter the Number=";
cin>>n;
temp=n;
while(n>0)
{
r=n%10;
sum=(sum*10)+r;
n=n/10;
}
if(temp==sum)
cout<<"Number is Palindrome.";
else
cout<<"Number is not Palindrome.";
return 0;
}

```

Output:

```
Enter the Number=121
Number is Palindrome.
Enter the number=113
Number is not Palindrome.
```

#### 4) Factorial

Write a c++ program to print factorial of a number.

```
#include <iostream>
using namespace std;
int main()
{
    int i,fact=1,number;
    cout<<"Enter any Number: ";
    cin>>number;
    for(i=1;i<=number;i++){
        fact=fact*i;
    }
    cout<<"Factorial of " <<number<<" is: " <<fact<<endl;
    return 0;
}
```

Output:

```
Enter any Number: 5
Factorial of 5 is: 120
```

#### 5) Armstrong number

Write a c++ program to check armstrong number.

```
#include <iostream>
using namespace std;
int main()
{
    int n,r,sum=0,temp;
    cout<<"Enter the Number= ";
    cin>>n;
    temp=n;
    while(n>0)
    {
        r=n%10;
```

```

sum=sum+(r*r*r);
n=n/10;
}
if(temp==sum)
cout<<"Armstrong Number."<<endl;
else
cout<<"Not Armstrong Number."<<endl;
return 0;
}

```

Output:

```

Enter the Number= 371
Armstrong Number.
Enter the Number= 342
Not Armstrong Number.

```

## 6) Sum of Digits

Write a c++ program to print sum of digits.

```

#include <iostream>
using namespace std;
int main()
{
int n,sum=0,m;
cout<<"Enter a number: ";
cin>>n;
while(n>0)
{
m=n%10;
sum=sum+m;
n=n/10;
}
cout<<"Sum is= "<<sum<<endl;
return 0;
}

```

Output:

```

Enter a number: 23
Sum is= 5
Enter a number: 624
Sum is= 12

```

## 7) Reverse Number

Write a c++ program to reverse given number.

```
#include <iostream>
using namespace std;
int main()
{
int n, reverse=0, rem;
cout<<"Enter a number: ";
cin>>n;
while(n!=0)
{
rem=n%10;
reverse=reverse*10+rem;
n/=10;
}
cout<<"Reversed Number: "<<reverse<<endl;
return 0;
}
```

Output:

```
Enter a number: 234
Reversed Number: 432
```

## 8) Swap two numbers without using third variable

Write a c++ program to swap two numbers without using third variable.

```
#include <iostream>
using namespace std;
int main()
{
int a=5, b=10;
cout<<"Before swap a= "<<a<<" b= "<<b<<endl;
a=a*b; //a=50 (5*10)
b=a/b; //b=5 (50/10)
a=a/b; //a=10 (50/5)
cout<<"After swap a= "<<a<<" b= "<<b<<endl;
return 0;
}
```

Output:

```
Before swap a= 5 b= 10
After swap a= 10 b= 5
```

## 9) Number Triangle

Write a c++ program to print number triangle.

```
#include <iostream>
using namespace std;
int main()
{
int i,j,k,l,n;
cout<<"Enter the Range=";
cin>>n;
for(i=1;i<=n;i++)
{
for(j=1;j<=n-i;j++)
{
cout<<" ";
}
for(k=1;k<=i;k++)
{
cout<<k;
}
for(l=i-1;l>=1;l--)
{
cout<<l;
}
cout<<"\n";
}
return 0;
}
```

Output:

```
Enter the Range=5
1
121
12321
1234321
123454321
Enter the Range=6
1
```

```
121
2321
1234321
123454321
12345654321
```

## (10). Fibonacci Series in C++

```
#include<iostream>
using namespace std;
void printFibonacci(int n){
    static int n1=0, n2=1, n3;
    if(n>0){
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
        cout<<n3<<" ";
        printFibonacci(n-1);
    }
}
int main(){
    int n;
    cout<<"Enter the number of elements: ";
    cin>>n;
    cout<<"Fibonacci Series: ";
    cout<<"0 "<<"1 ";
    printFibonacci(n-2); //n-2 because 2 numbers are already printed
    return 0;
}
```

Output:

```
Enter the number of elements: 15
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```



# C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

---

## Advantage of functions in C

There are many advantages of functions.

### 1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

### 2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

---

## Types of Functions

There are two types of functions in C programming:

**1. Library Functions:** are the functions which are declared in the C++ header files such as `ceil(x)`, `cos(x)`, `exp(x)`, etc.

**2. User-defined functions:** are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.

## Declaration of a function

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...)
```

```
{  
//code to be executed  
}
```

---

## C++ Function Example

Let's see the simple example of C++ function.

```
#include <iostream>  
using namespace std;  
void func() {  
    static int i=0; //static variable  
    int j=0; //local variable  
    i++;  
    j++;  
    cout<<"i=" << i<<" and j=" <<j<<endl;  
}  
int main()  
{  
    func();  
    func();  
    func();  
}
```

Output:

```
i= 1 and j= 1  
i= 2 and j= 1  
i= 3 and j= 1
```

# Passing Arguments to Function

In programming, argument (parameter) refers to the data which is passed to a function (function definition) while calling it.

In the above example, two variables, `num1` and `num2` are passed to function during function call. These arguments are known as actual arguments.

The value of `num1` and `num2` are initialized to variables `a` and `b` respectively. These arguments `a` and `b` are called formal arguments.

This is demonstrated in figure below:

```
# include <iostream>
using namespace std;

int add(int, int);

int main() {
    ... ..
    sum = add(num1, num2); // Actual parameters: num1 and num2
    ... ..
}

int add(int a, int b) { // Formal parameters: a and b
    ... ..
    add = a+b;
    ... ..
}
```

## Notes on passing arguments

- The numbers of actual arguments and formal argument should be the same. (Exception: [Function Overloading](#))
- The type of first actual argument should match the type of first formal argument. Similarly, type of second actual argument should match the type of second formal argument and so on.
- You may call function a without passing any argument. The number(s) of argument passed to a function depends on how programmer want to solve the problem.
- You may assign default values to the argument. These arguments are known as [default arguments](#).
- In the above program, both arguments are of `int` type. But it's not necessary to have both arguments of same type.

## Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.

Let's understand call by value and call by reference in C++ language one by one.

---

### Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
    int data = 3;
    change(data);
    cout << "Value of the data is: " << data<< endl;
    return 0;
}
void change(int data)
{
    data = 5;
}
```

Output:

```
Value of the data is: 3
```

## Call by reference in C++

In call by reference, original value is modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

**Note:** To understand the call by reference, you must have the basic knowledge of pointers. Let's try to understand the concept of call by reference in C++ language by the example given below:

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
    int swap;
    swap=*x;
    *x=*y;
    *y=swap;
}
int main()
{
    int x=500, y=100;
    swap(&x, &y); // passing value to function
    cout<<"Value of x is: "<<x<<endl;
    cout<<"Value of y is: "<<y<<endl;
    return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

## Difference between call by value and call by reference in C++

| Call by value                                                        | Call by reference                                                       |
|----------------------------------------------------------------------|-------------------------------------------------------------------------|
| A copy of value is passed to the function                            | An address of value is passed to the function                           |
| Changes made inside the function is not reflected on other functions | Changes made inside the function is reflected outside the function also |

|                                                                          |                                                                     |
|--------------------------------------------------------------------------|---------------------------------------------------------------------|
| Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in same memory location |
|--------------------------------------------------------------------------|---------------------------------------------------------------------|

## **What is an Inline function in C++?**

One of the major objectives of using functions in a program is to save memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing tasks such as jumping to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads and sometimes maybe the time taken for jumping to the calling function will be greater than the time taken to execute that function.

C++ has a different solution to this problem. To eliminate the time of calls to small functions, C++ proposes a new function called inline function. An inline function is a function that is expanded in line when it is invoked thus saving time. The compiler replaces the function call with the corresponding function code that reduces the overhead of function calls.

We should note that inlining is only a request to the compiler, not a command. The compiler can ignore and skip the request for inlining. The compiler may not perform inlining in the following circumstances:

- If a function contains a loop. (for, while, do-while)
- If a function is recursive.
- If a function contains static variables.
- If a function contains a switch command or goto statement.
- For a function not returning values, if a return statement exists.

### **Syntax:**

For an inline function, declaration and definition must be done together.

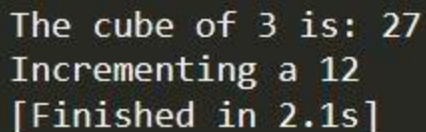
```
inline function-header  
  
{  
function-body  
}
```

### **Example:**

```
#include <iostream>  
using namespace std;  
inline int cube(int s)  
{  
return s*s*s;  
}
```

```
}
inline int inc(int a)
{
return ++a;
}
int main()
{
int a = 11;
cout << "The cube of 3 is: " << cube(3) << "\n";
cout << "Incrementing a " << inc(a) << "\n";
return 0;
}
```

## Output:



```
The cube of 3 is: 27
Incrementing a 12
[Finished in 2.1s]
```

## Explanation:

It is a simple example which shows two inline functions declared using the inline keyword as a prefix.

Moving on with this article on Inline function in C++

## When to use Inline function?

We can use Inline function as per our needs. Some useful recommendation are mentioned below-

- We can use the inline function when performance is needed.
- We can use the inline function over macros.
- We prefer to use the inline keyword outside the class with the function definition to hide implementation details of the function.

Moving on with this article on Inline function in C++

## Points to be remembered while using Inline functions

- We must keep inline functions small, small inline functions have better efficiency and good results.
- Inline functions do increase efficiency, but we should not turn all the functions inline. Because if we make large functions inline, it may lead to code bloat and might end up decreasing the efficiency.

- It is recommended to define large functions outside the definition of a class using scope resolution:: operator, because if we define such functions inside a class definition, then they might become inline automatically and again affecting the efficiency of our code.

Moving on with this article on Inline function in C++

## Advantages of Inline function

- Function call overhead doesn't occur.
- It saves the overhead of a return call from a function.
- It saves the overhead of push/pop variables on the stack when the function is called.
- When we use the inline function it may enable the compiler to perform context-specific optimization on the function body, such optimizations are not possible for normal function calls.
- It increases the locality of reference by utilizing the instruction cache.
- An inline function may be useful for embedded systems because inline can yield less code than the function call preamble and return.

## Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.

Following is a simple C++ example to demonstrate the use of default arguments. We don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

```
#include<iostream>
using namespace std;

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
}
```



```
    return 0;
}
```

Output:

```
25
50
80
```

## Friend class and function in C++

**Friend Class** A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

```
class Node {
private:
    int key;
    Node* next;
    /* Other members of Node Class */

    // Now class LinkedList can
    // access private members of Node
    friend class LinkedList;
};
```

**Friend Function** Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

- a) A method of another class
- b) A global function

```
class Node {
private:
    int key;
    Node* next;
```

```

    /* Other members of Node Class */
    friend int LinkedList::search();
    // Only search() of linkedList
    // can access internal members
};

```

Following are some important points about friend functions and classes:

- 1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
- 2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- 3) Friendship is not inherited (See [this](#) for more details)
- 4) The concept of friends is not there in Java.

## 1. A simple and complete C++ program to demonstrate friend Class

```

#include <iostream>
class A {
private:
    int a;

public:
    A() { a = 0; }
    friend class B; // Friend Class
};

class B {
private:
    int b;

public:
    void showA(A& x)
    {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};

int main()
{
    A a;
    B b;
    b.showA(a);
    return 0;
}

```

Output:

A::a=0

## 2. A simple and complete C++ program to demonstrate friend function of another class

```
#include <iostream>

class B;

class A {
public:
    void showB(B&);
};

class B {
private:
    int b;

public:
    B() { b = 0; }
    friend void A::showB(B& x); // Friend function
};

void A::showB(B& x)
{
    // Since showB() is friend of B, it can
    // access private members of B
    std::cout << "B::b = " << x.b;
}

int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

Output:

B::b = 0

## Simple Example Of Function based Program

### 1.Add two Number Using Function.

```
#include <iostream>
using namespace std;

//function declaration
int addition(int a,int b);

int main()
{
    int num1; //to store first number
    int num2; //to store second number
    int add;  //to store addition

    //read numbers
    cout<<"Enter first number: ";
    cin>>num1;
    cout<<"Enter second number: ";
    cin>>num2;

    //call function
    add=addition(num1,num2);
```

```
//print addition
cout<<"Addition is: "<<add<<endl;

return 0;
}

//function definition
int addition(int a,int b)
{
    return (a+b);
}
```

## Output

```
Enter first number: 100
Enter second number: 200
Addition is: 300
```

## 2. gretest among two number

```
#include <stdio.h>

// An example function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}

// main function that doesn't receive any parameter and
// returns integer.
int main(void)
{
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    printf("m is %d", m);
    return 0;
}
```

## 3. Check even or odd using function.

```
#include <iostream>
using namespace std;
void odd(int);
void even(int);
int main ()
{
    int num;
    cout<<"Enter any number";
    cin>>num;
    even(num);
    return 0;
}
void even(int a )
{
    if ( a%2==0)
        cout<<"\nEven number";
    else odd(a);
}
void odd(int)
{
    cout<<"\nOdd number"; }
}
```

## UNIT-2

### Classes And Objects:

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

### C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows –

```
class Box {
    public:
        double length;    // Length of a box
        double breadth;  // Breadth of a box
        double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

### Member Function :-

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them –

```
class Box {
    public:
        double length;           // Length of a box
        double breadth;         // Breadth of a box
        double height;          // Height of a box
        double getVolume(void); // Returns box volume
};
```

Member functions can be defined within the class definition or separately using **scope resolution**

## Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box –

```
Box Box1;           // Declare Box1 of type Box
Box Box2;           // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

## Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear –

```
#include <iostream>

using namespace std;

class Box {
public:
    double length;    // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};

int main() {
    Box Box1;        // Declare Box1 of type Box
    Box Box2;        // Declare Box2 of type Box
    double volume = 0.0;    // Store the volume of a box here

    // box 1 specification
    Box1.height = 5.0;
    Box1.length = 6.0;
    Box1.breadth = 7.0;

    // box 2 specification
    Box2.height = 10.0;
    Box2.length = 12.0;
    Box2.breadth = 13.0;

    // volume of box 1
    volume = Box1.height * Box1.length * Box1.breadth;
    cout << "Volume of Box1 : " << volume <<endl;

    // volume of box 2
    volume = Box2.height * Box2.length * Box2.breadth;
    cout << "Volume of Box2 : " << volume <<endl;
}
```



```
return 0;  
}
```

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

```
Volume of Box1 : 210  
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

## Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below –

| Sr.No | Concept & Description                                                                                                                                                                                                                             |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1     | <p>Class Member Functions</p> <p>A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.</p>                                                                     |
| 2     | <p>Class Access Modifiers</p> <p>A class member can be defined as public, private or protected. By default members would be assumed as private.</p>                                                                                               |
| 3     | <p>Constructor &amp; Destructor</p> <p>A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.</p> |
| 4     | <p>Copy Constructor</p> <p>The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.</p>                                                              |
| 5     | <p>Friend Functions</p> <p>A <b>friend</b> function is permitted full access to private and protected members of a class.</p>                                                                                                                     |
| 6     | <p>Inline Functions</p> <p>With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.</p>                                                                                     |
| 7     | <p>this Pointer</p> <p>Every object has a special pointer <b>this</b> which points to the object itself.</p>                                                                                                                                      |

A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.

## Member Function

A member function is defined outside the class using the ::(double colon symbol) scope resolution operator. This is useful when we did not want to define the function within the main program, which makes the program more understandable and easy to maintain.

*The general syntax of the member function of a class outside its scope :*

< return\_type > < class\_name > :: < member\_function > (arg1, arg2... argN)

The type of member arguments in the member function must exactly match with the types declared in the class definition of the < class\_name >. The scope resolution operator (::) is used along with the class name in the header of the function definition. It identifies the function as a member of a particular class. Without the scope resolution operator, the function definition would create an ordinary function, subject to the usual function rules of access and scope.

*The following program segment shows how a member function is declared outside the class declaration and how the object of a class is created to use the class's methods and variables :-*

```
#include <iostream>
using namespace std;
class TestMemberFunction
{
    public :
        int x, y;
        int sum(); // Memembr Function Declaration
};
int TestMemberFunction :: sum()
{
    return (x+y);
}
int main ()
{
    TestMemberFunction MemberFunction1;
    MemberFunction1.x = 1;
    MemberFunction1.y = 2;
    cout << "Sum is : " << MemberFunction1.sum();
    return 0;
}
```

## Nesting of Member Functions :-

A member function may call another member function within itself. This is called nesting of member functions. A member function can access not only the public functions but also the private functions of the class it belongs to.

*Let us see an simple example of showing nesting of the member functions :-*

```
#include <iostream>
using namespace std;
class NestingMemberFunction
{
    private :
        int myint;
    public :
        void set();
        int get();
};
void NestingMemberFunction :: set()
{
    myint = -1;
}
int NestingMemberFunction :: get()
{
    set();
    return myint;
}
int main ()
{
    int printvariable;
    NestingMemberFunction NestingMemberFunction1;
    printvariable = NestingMemberFunction1.get();
    cout << "The variable is : " << printvariable;
    return 0;
}
```

## Private Member Function

A function declared inside the class's private section is known as "**private member function**". A **private member function** is accessible through the only public member function. (Read more: data members and member functions in C++).

### Example:

In this example, there is a class named "**Student**", which has following data members and member functions:

- **Private**
  - **Data members**
    - rNo - to store roll number
    - perc - to store percentage

- 
- **Member functions**
  - inputOn() - to print a message "**Input start...**" before reading the roll number and percentage using public member function.
  - inputOff() - to print a message "**Input end...**" after reading the roll number and percentage using public member function.
- **Public**
  - **Member functions**
    - read() - to read roll number and percentage of the student
    - print() - to print roll number and percentage of the student

Here, inputOn() and inputOff() are the private member functions which are calling inside public member function read().

### Program:

```
#include <iostream>
using namespace std;

class Student
{
    private:
        int rNo;
        float perc;
        //private member functions
        void inputOn(void)
        {
            cout<<"Input start..."<<endl;
        }
        void inputOff(void)
        {
            cout<<"Input end..."<<endl;
        }

    public:
        //public member functions
        void read(void)
        {
            //calling first member function
            inputOn();
            //read rNo and perc
            cout<<"Enter roll number: ";
            cin>>rNo;
            cout<<"Enter percentage: ";
            cin>>perc;
        }
};
```

```

        //calling second member function

        inputOff();
    }
    void print(void)
    {
        cout<<endl;
        cout<<"Roll Number: "<<rNo<<endl;
        cout<<"Percentage: "<<perc<<"%"<<endl;
    }
};

//Main code
int main()
{
    //declaring object of class student
    Student std;

    //reading and printing details of a student
    std.read();
    std.print();

    return 0;
}

```

## Arrays within a Class

- Arrays can be declared as the members of a class.
- The arrays can be declared as private, public or protected members of the class.
- To understand the concept of arrays as members of a class, consider this example.

A program to demonstrate the concept of arrays as class member

### **Example**

```

#include<iostream>
const int size=5;
class student
{
int roll_no;
int marks[size];

```

```

public:
void getdata ();
void tot_marks ();
} ;

void student :: getdata ()
{
cout<<"\nEnter roll no: ";
Cin>>roll_no;
for(int i=0; i<size; i++)

{

cout<<"Enter marks in subject"<<(i+1)<<": ";
cin>>marks[i] ;
}

void student :: tot_marks() //calculating total marks
{
int total=0;
for(int i=0; i<size; i++)
total+ = marks[i];
cout<<"\n\nTotal marks "<<total;

}

void main() student stu;

stu.getdata() ;

stu.tot_marks() ;

getch();

}

```

## Output:

**Enter roll no: 101**

**Enter marks in subject 1: 67**

**Enter marks in subject 2 : 54**

**Enter marks in subject 3 : 68**

**Enter marks in subject 4 : 72**

**Enter marks in subject 5 : 82**

**Total marks = 343**

## Memory Allocation for Object of Class

Once you define class it will not allocate memory space for the data member of the class. The memory allocation for the data member of the class is performed separately each time when an object of the class is created. Since member functions defined inside class remains same for all objects, only memory allocation of member function is performed at the time of defining the class. Thus memory allocation is performed separately for different object of the same class. All the data members of each object will have separate memory space.

--→We can also dynamically allocate objects.

As we know that **Constructor** is a member function of a class which is called whenever a new object is created of that class. It is used to initialize that object. **Destructor** is also a class member function which is called whenever the object goes out of scope.

Destructor is used to release the memory assigned to the object. It is called in these conditions.

- When a local object goes out of scope
- For a global object, operator is applied to a pointer to the object of the class

We again use pointers while dynamically allocating memory to objects.

Let's see an example of array of objects.

```

#include <iostream>
using namespace std;

class A
{
    public:
    A() {
        cout << "Constructor" << endl;
    }
    ~A() {
        cout << "Destructor" << endl;
    }
};

int main()
{
    A* a = new A[4];
    delete [] a; // Delete array
    return 0;
}

```

## **Static data members**

Static data members are class members that are declared using the static keyword. There is only one copy of the static data member in the class, even if there are many class objects. This is because all the objects share the static data member. The static data member is always initialized to zero when the first class object is created.

The syntax of the static data members is given as follows –

**static data\_type data\_member\_name;**

In the above syntax, static keyword is used. The data\_type is the C++ data type such as int, float etc. The data\_member\_name is the name provided to the data member.

→A program that demonstrates the static data members in C++ is given as follows –

Example

```

#include <iostream>
#include<string.h>
using namespace std;

```



```

class Student
{
    private:
    int rollNo;
    char name[10];
    int marks;
    public:
    static int objectCount;
    Student() {
        objectCount++;
    }

    void getdata() {
        cout << "Enter roll number: "<<endl;
        cin >> rollNo;
        cout << "Enter name: "<<endl;
        cin >> name;
        cout << "Enter marks: "<<endl;
        cin >> marks;
    }

    void putdata() {
        cout<<"Roll Number = "<< rollNo <<endl;
        cout<<"Name = "<< name <<endl;

        cout<<"Marks = "<< marks <<endl;
        cout<<endl;
    }
};

int Student::objectCount = 0;

int main(void) {
    Student s1;
    s1.getdata();
    s1.putdata();
    Student s2;

    s2.getdata();
    s2.putdata();
    Student s3;
}

```

```

s3.getdata();
s3.putdata();
cout << "Total objects created = " << Student::objectCount << endl;
return 0;
}

```

The output of the above program is as follows –

```

Enter roll number: 1
Enter name: Mark
Enter marks: 78
Roll Number = 1
Name = Mark
Marks = 78

```

```

Enter roll number: 2
Enter name: Nancy
Enter marks: 55
Roll Number = 2
Name = Nancy
Marks = 55

```

```

Enter roll number: 3
Enter name: Susan
Enter marks: 90
Roll Number = 3
Name = Susan
Marks = 90
Total objects created = 3

```

In the above program, the class student has three data members denoting the student roll number, name and marks. The objectCount data member is a static data member that contains the number of objects created of class Student. Student() is a constructor that increments objectCount each time a new class object is created.

There are 2 member functions in class. The function getdata() obtains the data from the user and putdata() displays the data. The code snippet for this is as follows –

```

class Student {
private:
int rollNo;
char name[10];
int marks;

```

```

public:
static int objectCount;
Student() {
    objectCount++;
}

void getdata() {
    cout << "Enter roll number: "<<endl;
    cin >> rollNo;
    cout << "Enter name: "<<endl;
    cin >> name;
    cout << "Enter marks: "<<endl;
    cin >> marks;
}

void putdata() {
    cout<<"Roll Number = "<< rollNo <<endl;
    cout<<"Name = "<< name <<endl;
    cout<<"Marks = "<< marks <<endl;
    cout<<endl;
}
};

```

In the function main(), there are three objects of class Student i.e. s1, s2 and s3. For each of these objects getdata() and putdata() are called. At the end, the value of objectCount is displayed. This is given below –

```

int main(void) {
    Student s1;

    s1.getdata();
    s1.putdata();
}

```

```

Student s2;
s2.getdata();
s2.putdata();

Student s3;
s3.getdata();
s3.putdata();

cout << "Total objects created = " << Student::objectCount << endl;

return 0;
}

```

## Constructors in C++

### What is constructor?

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.

#### 1. program to illustrate the

#### // concept of Constructors

```

#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}

```

#### OUTPUT:

A: 10

B: 20

**Parameterized Constructors:**

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

## 2. // CPP program to illustrate // parameterized constructors

```
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

## Destructors in C++

What is destructor?

**Destructor is a member function which destructs or deletes an object.**When is destructor called?

**A destructor function is called automatically when the object goes out of scope:**

- (1) the function ends**
- (2) the program ends**
- (3) a block containing local variables ends**
- (4) a delete operator is called**

How destructors are different from a normal member function?

**Destructors have same name as the class preceded by a tilde (~)**  
**Destructors don't take any argument and don't return anything**

```
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};

String::String(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}

String::~~String()
{
    delete []s;
}
```

## UNIT-3

### Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

**Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.

**Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.

#### **Modes of Inheritance**

1. **Public mode:** If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode:** If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode:** If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

### Different Types of Inheritance

OOPs support the six different types of inheritance as given below :

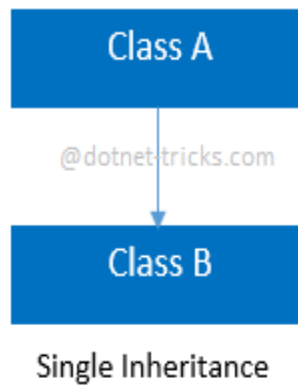
1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance

4. Multipath inheritance
5. Hierarchical Inheritance
6. Hybrid Inheritance

## Single inheritance

In this inheritance, a derived class is created from a single base class.

In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behavior of the parent class A.



```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle{

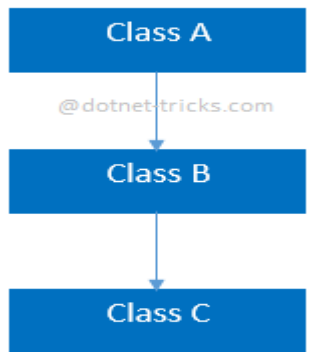
};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

### **OUTPUT:**

THIS IS A VEHICLE

## Multi-level inheritance



Multi-Level Inheritance

In this inheritance, a derived class is created from another derived class. In the given example, class C inherits the properties and behavior of class B and class B inherits the properties and behavior of class A. So, here A is the parent class of B and class B is the parent class of C. So, here class C implicitly inherits the properties and behavior of class A along with Class B i.e there is a multilevel of inheritance.

### **// C++ program to implement**

#### **// Multilevel Inheritance**

```
#include <iostream>
using namespace std;

// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class fourWheeler: public Vehicle
{ public:
    fourWheeler()
    {
        cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};

// sub class derived from two base classes
class Car: public fourWheeler{
public:
    car()
    {
        cout<<"Car has 4 Wheels"<<endl;
    }
}
```



```
};

// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes

    Car obj;

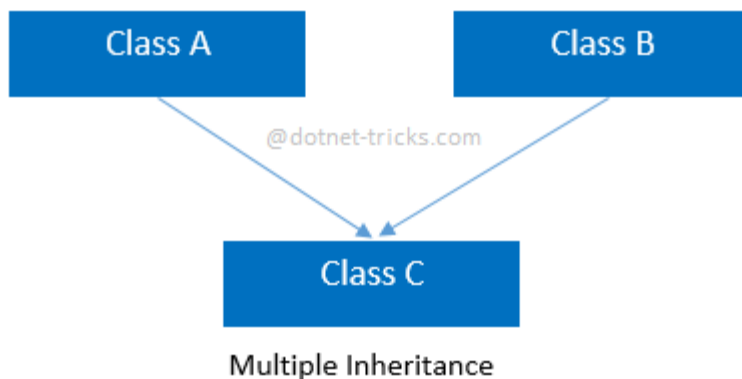
    return 0;
}
```

### **Output:**

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

## **Multipath inheritance**

In this inheritance, a derived class is created from another derived classes and the same base class of another derived classes. This inheritance is not supported by .NET Languages like C#, F# etc. In the given example, class D inherits the properties and behavior of class C and class B as well as Class A. Both class C and class B inherits the Class A. So, Class A is the parent for Class B and Class C as well as Class D. So it's making it Multipath inheritance.



```
#include <iostream>

using namespace std;

class Area
{
```

```

    public:
        int getArea(int l, int b)
        {
            return l * b;
        }
};

class Perimeter
{
    public:
        int getPerimeter(int l, int b)
        {
            return 2*(l + b);
        }
};

class Rectangle : public Area, public Perimeter
{
    int length;
    int breadth;
    public:
        Rectangle()
        {
            length = 7;
            breadth = 4;
        }
        int area()
        {
            return Area::getArea(length, breadth);
        }
        int perimeter()
        {
            return Perimeter::getPerimeter(length, breadth);
        }
};

int main()
{
    Rectangle rt;
    cout << "Area : " << rt.area() << endl;
    cout << "Perimeter : " << rt.perimeter() << endl;
    return 0;
}

```

### Output

```

Area : 28
Perimeter : 22

```

## What is a virtual base class?

- An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.
- C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

What is Virtual base class? Explain its uses.

- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

- Consider the following example :

```
class A
{
    public:
        int i;
};

class B : virtual public A
{
    public:
        int j;
};

class C: virtual public A
{
    public:
        int k;
};

class D: public B, public C
{
    public:
        int sum;
};

int main()
{
    D ob;
    ob.i = 10; //unambiguous since only one copy of i is inherited.
    ob.j = 20;
    ob.k = 30;
    ob.sum = ob.i + ob.j + ob.k;
    cout << "Value of i is : "<< ob.i<<"\n";
    cout << "Value of j is : "<< ob.j<<"\n"; cout << "Value of k is : "<< ob.k<<"\n";
    cout << "Sum is : "<< ob.sum <<"\n";

    return 0;
}
```

# Abstract Classes

---

An abstract class is, conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions.

A pure virtual function is one which **must be overridden** by any concrete (i.e., non-abstract) derived class. This is indicated in the declaration with the syntax "**= 0**" in the member function's declaration.

## Example

```
class AbstractClass {
public:
    virtual void AbstractMemberFunction() = 0; // Pure virtual function makes
                                              // this class Abstract class.
    virtual void NonAbstractMemberFunction1(); // Virtual function.

    void NonAbstractMemberFunction2();
};
```

# Nested Classes in C++

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

For example, program 1 compiles without any error and program 2 fails in compilation.

## Program 1

```
#include<iostream>

using namespace std;

/* start of Enclosing class declaration */
class Enclosing {
private:
    int x;

/* start of Nested class declaration */
class Nested {
    int y;
    void NestedFun(Enclosing *e) {
        cout<<e->x; // works fine: nested class can access
                   // private members of Enclosing class
    }
}
```

```

}; // declaration Nested class ends here
}; // declaration Enclosing class ends here

int main()
{

}

```

## **Program .2**

```

#include<iostream>

using namespace std;

/* start of Enclosing class declaration */
class Enclosing {

    int x;

    /* start of Nested class declaration */
    class Nested {
        int y;
    }; // declaration Nested class ends here

    void EnclosingFun(Nested *n) {
        cout<<n->y; // Compiler Error: y is private in Nested
    }
}; // declaration Enclosing class ends here

int main()
{

}

```

# Polymorphism in C++

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

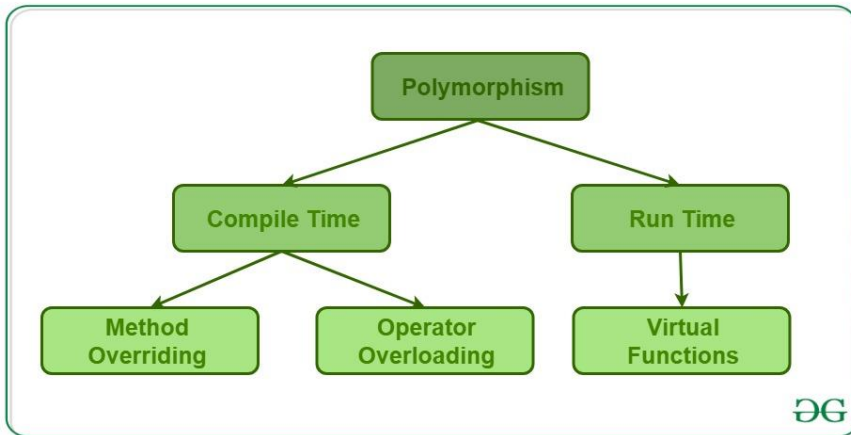
Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

Polymorphism is considered as one of the important features of Object Oriented Programming.

**In C++ polymorphism is mainly divided into two types:**

- Compile time Polymorphism

- Runtime Polymorphism



**Compile time polymorphism:** This type of polymorphism is achieved by function overloading or operator overloading.

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined –

```
#include <iostream>

using namespace std;
int main () {
    int var1;
    char var2[10];

    cout << "Address of var1 variable: ";
    cout << &var1 << endl;

    cout << "Address of var2 variable: ";
    cout << &var2 << endl;

    return 0;
}
```

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

```
Address of var1 variable: 0xbfefd5c0
Address of var2 variable: 0xbfefd5b6
```

# What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

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

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

## Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations –

```
#include <iostream>

using namespace std;

int main () {
    int var = 20;    // actual variable declaration.
    int *ip;        // pointer variable

    ip = &var;      // store address of var in pointer variable

    cout << "Value of var variable: ";
    cout << var << endl;

    // print the address stored in ip pointer variable
    cout << "Address stored in ip variable: ";
    cout << ip << endl;

    // access the value at the address available in pointer
    cout << "Value of *ip variable: ";
    cout << *ip << endl;

    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows –

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

## Pointers to Objects

A variable that holds an address value is called a pointer variable or simply pointer.

Pointer can point to objects as well as to simple data types and arrays.

sometimes we dont know, at the time that we write the program , how many objects we want to creat. when this is the case we can use **new** to creat objects while the program is running. new returns a pointer to an unnamed objects. lets see the example of student that will clear your idea about this topic

```
#include <iostream>
#include <string>
using namespace std;
class student
{
private:
    int rollno;
    string name;
public:
    student():rollno(0),name("")
    {}
    student(int r, string n): rollno(r),name (n)
    {}
    void get()
    {
        cout<<"enter roll no";
        cin>>rollno;
        cout<<"enter name";
        cin>>name;
    }
    void print()
    {
        cout<<"roll no is "<<rollno;
        cout<<"name is "<<name;
    }
};
void main ()
{
    student *ps=new student;
    (*ps).get();
    (*ps).print();
    delete ps;
}
```

## Virtual Function in C++

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a



reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve **Runtime polymorphism**
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

### Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have **virtual destructor** but it cannot have a virtual constructor.

### Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

Consider the following simple program showing run-time behavior of virtual functions.

```
// CPP program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }

    void show()
    {
        cout << "show base class" << endl;
    }
};

class derived : public base {
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};

int main()
{
    base* bptr;
    derived d;
    bptr = &d;
```

```
// virtual function, binded at runtime
bptr->print();

// Non-virtual function, binded at compile time
bptr->show();
}
```

**Output:**

```
print derived class
show base class
```

## Function Overloading in C++

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

```
#include <iostream>
using namespace std;

void print(int i) {
    cout << " Here is int " << i << endl;
}

void print(double f) {
    cout << " Here is float " << f << endl;
}

void print(char const *c) {
    cout << " Here is char* " << c << endl;
}

int main() {
    print(10);
    print(10.10);
    print("ten");
    return 0;
}
```

```
Here is int 10
Here is float 10.1
Here is char* ten
```

## Operator Overloading in C++

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

```
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0) {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

**Output:**

```
12 + i9
```