UNIT - 1

INTRODUCTION TO C++

INPUT and OUTPUT IN C++

Introduction to C++

Object-Oriented Technology

This world is made up off several objects and objects communicate with each other. Similar objects can be grouped together. For example, take a college. It can have two working sections like *teaching* and *non-teaching*. These two groups can be further sub divided.

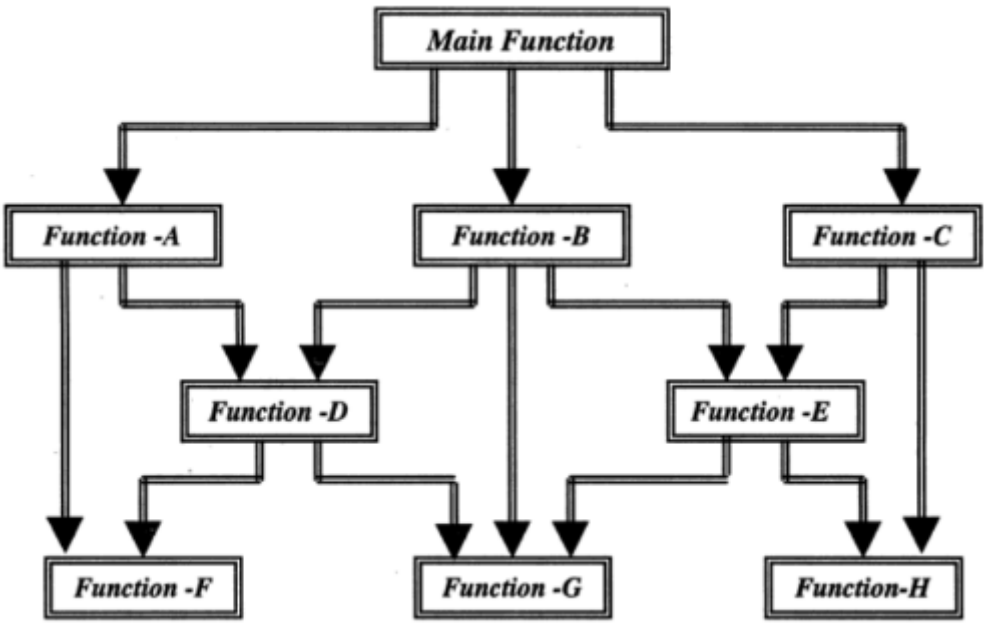
We can also group the people in a college into several departments. Generally a college will have several lecturers in each department. A college can have one or more laboratories and there might be separate lab technicians to assist in conducting practical. A college will have a Principal for managing the entire college. Each department might have its own head who is responsible for managing their own department. Like this the entire world can be viewed as a collection of objects that communicate with each other to perform activities.

This theory of objects can be applied to software to solve complex problems.

Disadvantages of Conventional Programming

Traditional programming languages like COBOL, FORTRAN etc are known as procedural languages. In these languages the program is written as a set of statements which contains the logic and control that instructs the compiler or interpreter how to perform a certain task. It becomes difficult to manage the code when the program is quite large.

To reduce complexity, functions or modules were introduced which divides a large program into a set of cohesive units. Each unit is called a function and each function can call other functions as shown below:



Even though functions reduces complexity, with a lot of functions sharing a common set of global data may lead to logical errors.

Following are the disadvantages of procedural or structured languages:

1. Programs are divided into a set of functions which can share data. Hence, no data security.
2. Focus is on the code to perform the task but not on the data needed.
3. Data is shared globally by several functions which might lead to logical errors.
4. No restrictions on which functions can share the global data.

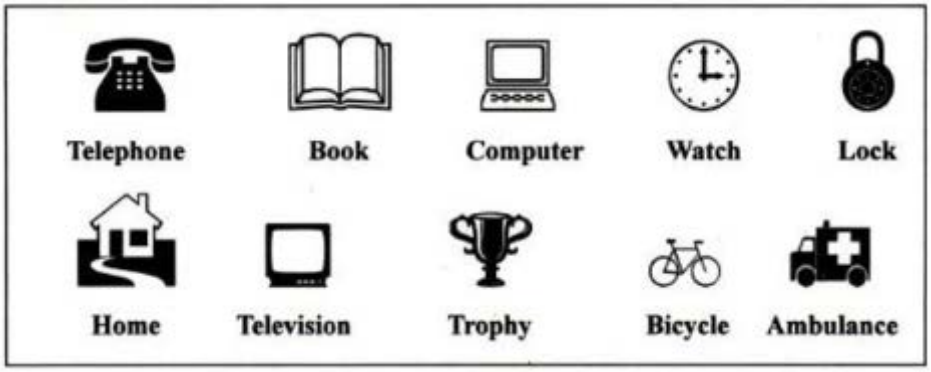
Concepts of Object-Oriented Programming

Following are the key concepts in object-oriented programming:

Objects

Object is a real world entity or a prime run-time entity in object-oriented programming. Object occupies space in memory. Every object has its own properties and behavior. An object is an instance or specimen of a class. Every object is unique. Each object contains state, which is the collection of data values associated with the properties of an object.

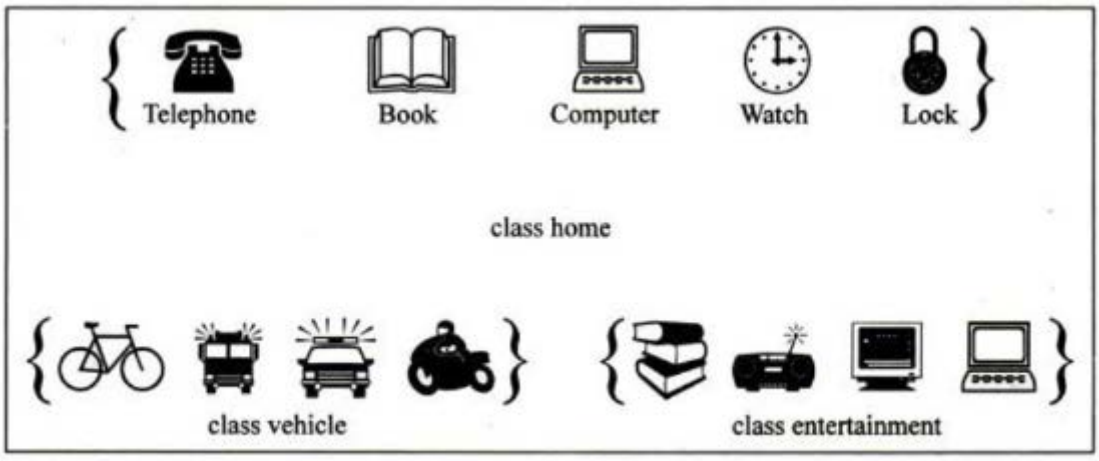
Some of the real world examples for objects are shown below:



Classes

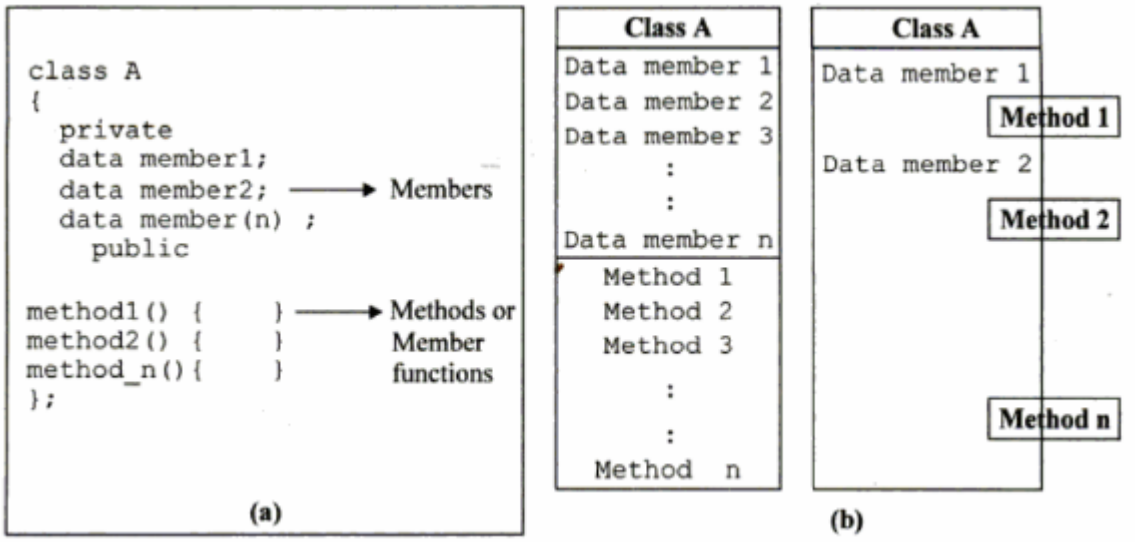
A class is a collection of similar objects that have similar properties and exhibit similar behavior. A class is a template or blueprint for creating objects. After a class is created, we can create any number of objects. A class does not occupy any memory like an object.

Real world examples of classes are shown below:



Methods

An operation represents the behavior of an object. When an operation is coded in a class, it is called a method. All objects in a class perform certain common actions or operations. Following figure shows a class, its data members, and methods written in different styles:

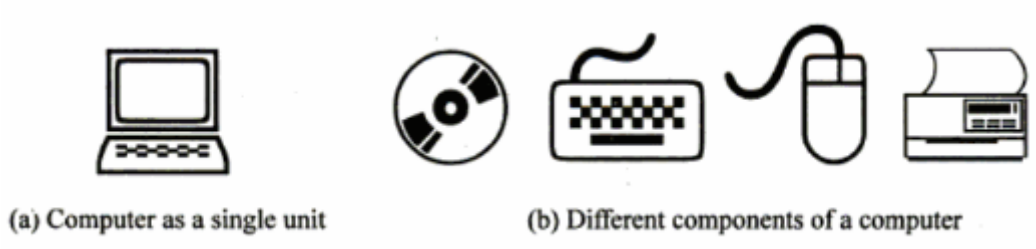


Data Abstraction

Abstraction refers to the process of presenting essential features without including any complex background details. A class uses this abstraction to represent only essential properties and behavior.

A powerful way to achieve abstraction is by manipulating hierarchical classifications. This allows us to divide the system into several logical layers which can be maintained separately.

Real world example for abstraction is a computer which is a single unit made up of several units (parts) as shown below:

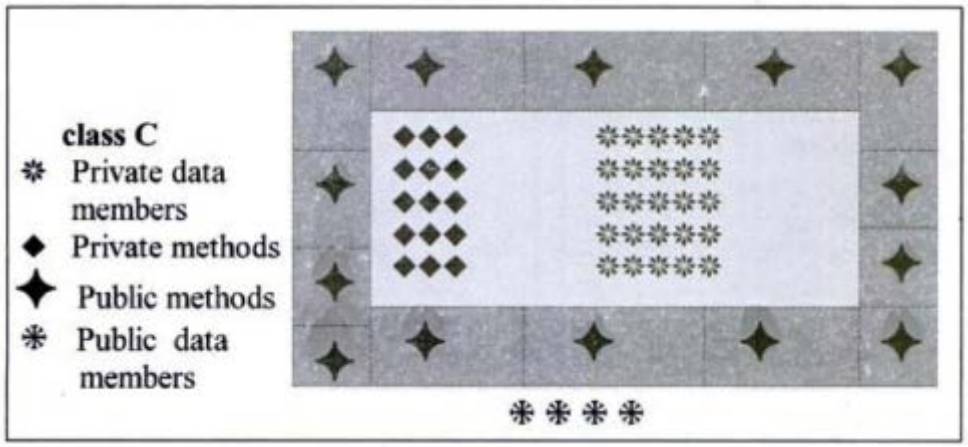


(a) Computer as a single unit (b) Different components of a computer

Encapsulation

Wrapping up of code and data together into a single unit is known as encapsulation. Encapsulation is implemented by classes. A class contains data which is generally private and is accessible to the outside world only through public methods.

A side effect of encapsulation is data hiding which allows the users to use an object without knowing its internal details. Following figure represents different sections of encapsulation:



Inheritance

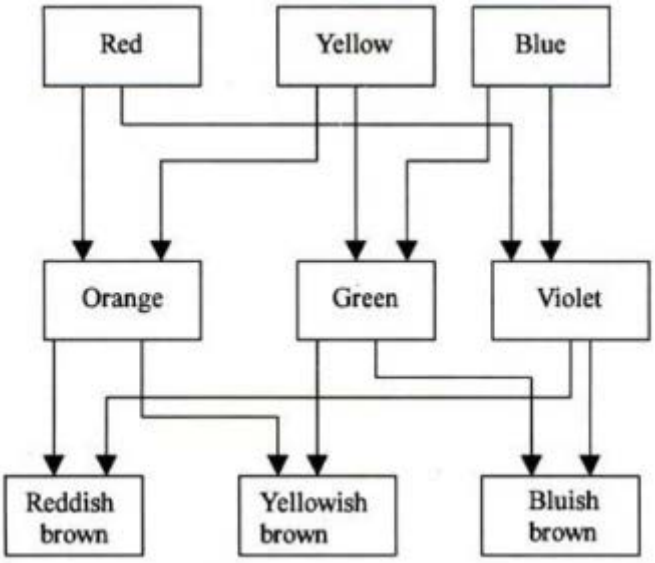
Deriving properties of an object from one class to object of an another class is known as inheritance. Inheritance promotes the feature of object-oriented programming, reusability.

Using inheritance a new class can be created without effecting the existing class. We can use already available functionality of the existing class and add new functionality in the new class based on the requirements.

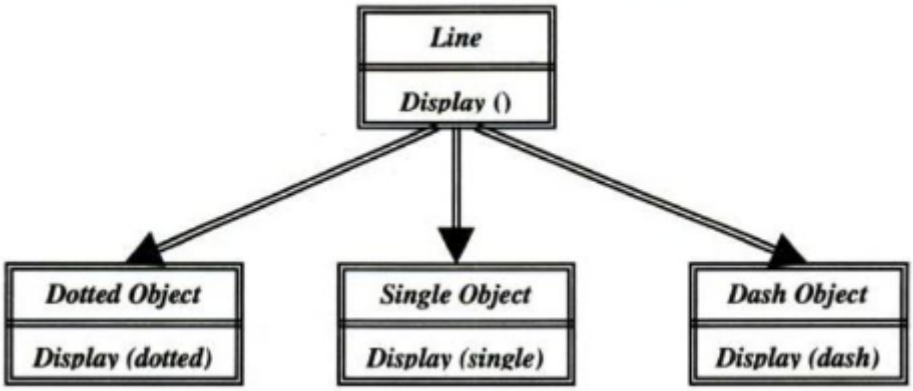
As a real world example, consider the following colour hierarchy, where Red, Yellow and Blue are existing classes and all other classes are derived classes:

Polymorphism

Single method exhibiting different behaviours in the same class or different classes is known as polymorphism. We can code a generic interface (set of methods) to a set of associated actions.



As a real world example for polymorphism consider a generic **Line** class which contains a method **Display()**. Now consider three specific classes: **DottedObject, SingleObject,** and **DashObject** which in turn also contains the same **Display()** method as shown below:



Dynamic Binding

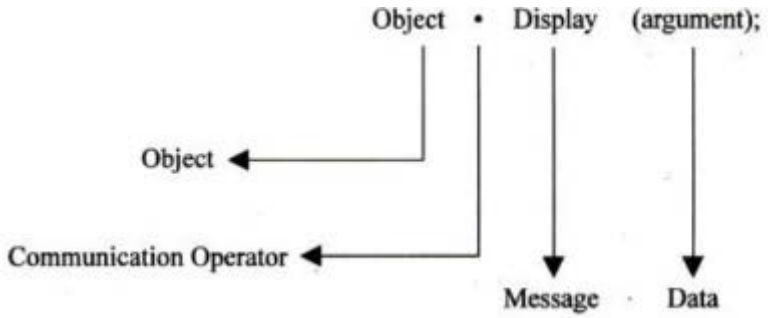
The process of associating an action to a message at run-time is known as dynamic binding which is also known as late binding. The action to be associated is recognized when the object is created which is done at run-time.

As an example, consider the above hierarchy in which a **Line** object can call the **Display()** method in any of the three child classes: **DottedObject, SingleObject,** or **DashObject**.

Message Passing

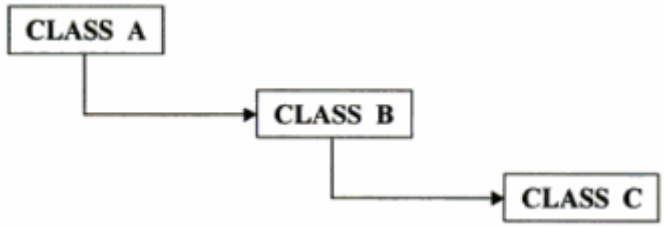
Objects communicate through messages. A message is a request for an action. Object sending the message can also pass data (parameters). Object which receives the message invokes a corresponding member function (action).

Consider the following example which represents message passing:



Reusability

Using the code available in an existing class to create new classes is known as reusability which is an important feature of object-orientation. In the below example, class **A** is an already existing class from which class **B** is created and from class **B**, class **C** is created:

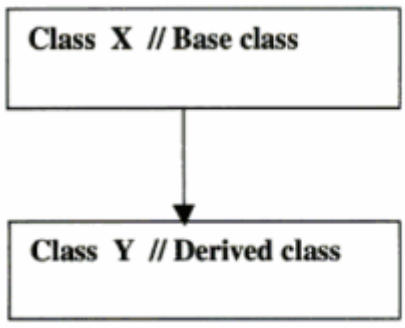


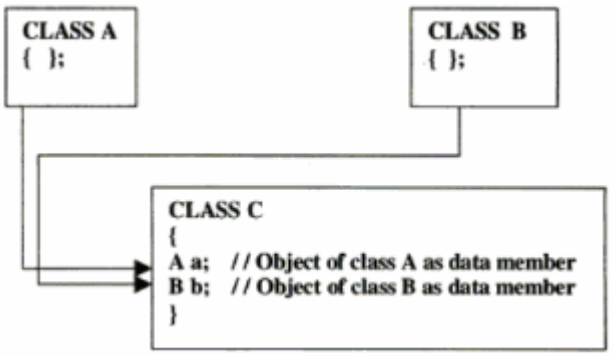
Delegation

In object-oriented programming, two classes can be joined either by inheritance or delegation. Both of them promotes reusability of code.

As shown in the below figure (a), class **Y** is derived from class **X**. Here class Y is a **kind of** X. This kind of relationship between Y and X classes is known as inheritance. Inheritance is also known as **is a** relationship.

Also as shown in figure (b) below, objects of classes **A** and **B** are members of class **C**. This kind of relationship is known as **has a** relationship and is also called as delegation or composition.





Genericity

The software components (like classes) can have more than one implementation based on the data type of arguments. Programmer can create a generic class or function which can work with different types of arguments. The **template** feature in C++ allows generic programming.

For example, a list data structure can operate on integers, characters, strings, floats or any other types of objects. So, a programmer can create a generic **List** class which can accept any kind of data.

Advantages of OOP

Object-oriented programming offers many advantages to the programmers and users. This technology allows programmers to create high quality software at an affordable price. Following are the advantages of OOP:

1. Object-oriented programs can be upgraded easily.
2. Using inheritance, code redundancy can be eliminated.
3. Data hiding allows the programmers to develop safe programs that do not disturb code in other parts of the program.
4. Encapsulation allows programmers to create classes with several features and only few public methods to be exposed.
5. All object-oriented languages can create, extend and reuse existing programs.
6. Object-oriented programming enhances the thought process of a programmer leading to rapid development of software.

History of C++

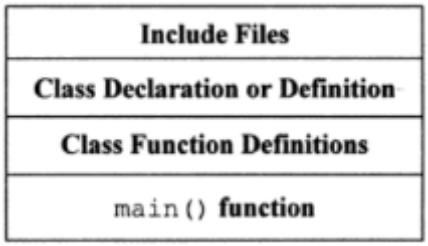
- C++ was developed by Bjarne Stroustrup in 1979 at AT&T Bell Labs.
- C++ is an object-oriented language which includes imperative as well as generic programming features.
- C++ was developed with features inherited from SIMULA 67 and ALGOL 68.
- C++ was initially called "C with classes".

Following are various applications of C++ language:

- Systems programming
- Embedded systems programming
- Weather control systems
- Defence systems
- Web applications
- Server programming
- Databases
- Web search
- e-commerce

Structure of a C++ Program

A C++ program can contain several parts or sections as shown below:



All sections except the **main()** function section are optional. Every C++ program must contain a main() function section. Classes in C++ are similar to structures in C. Classes in C++ can contain both variables and functions, where as it is not possible with structures in C. Functions can be defined inside and outside classes in C++.

Include Files Section

A C++ program might include one or more header files using the **#include** directive. The syntax for including header files is given below:

#include<stdio.h> [or] #include "stdio.h"

The extension of header files in C++ is ".h". In the above example, the header files **stdio.h**. is included. All the function declarations in that header file are available in our program. Header files can be included anywhere in the program. Good programming practice is to include them at the beginning of the program.

Class Declaration or Definition Section

One or more classes can be defined in this section. A class can be defined after the main() section. Good programming practice is to define classes before main() section.

A class definition must terminate with a semi-colon (;). A class definition contains variables, function declarations (prototypes), or function definitions.

Class Function Definitions Section

This contains the definitions of functions that were declared in the above section (class declaration section). Functions with less code are defined within the class. Such functions are treated as inline functions. Functions with large amount of code are defined outside the class.

main() Function Section

Every C++ program must contain a **main()** function as the execution starts from main() function itself. A main() function might contain the code to create and use the objects of classes created in above sections.

Header Files and Libraries in C++

A library provided by a programming language consists of pre-defined functionality arranged in several files. These libraries can be used by programmers to write programs with their own logic without focusing on housekeeping functionality.

C++ library provides several header files (.h files) which includes a lot of pre-defined functionality like reading and writing to files, memory management routines, console I/O routines etc.

Below are some of the header files and the library functions available in those header files in C++:

A Sample C++ Program

Let's consider the following C++ program which prints "Hello World" on to the console or terminal window. We will look at various elements used in the program.

Name of the header file	Description of the header file	Functions available in the header file
alloc.h	Memory management functions	calloc(), malloc(), free(), realloc() etc..
complex.h	Complex math functions	asin(), atan(), arg(), tanh() etc..
ctype.h	Character oriented functions	toupper(), tolower(), islower(), isupper() etc..
dos.h	DOS functions	getdate(), gettime(), sleep() etc..
graphics.h	Graphics related functions	arc(), bar(), circle(), getx() etc..
process.h	Process related functions	exit(), abort(), system() etc..
stdio.h	Standard I/O functions	puts(), gets(), fopen(), fclose(), printf(), scanf(), getchar(), putchar() etc..
stdlib.h	Standard library functions	atoi(), time(), abort() etc..
string.h	String manipulation functions	strcpy(), strcat(), strlen(), etc..
time.h	Declare functions that manipulate time and date	time(), stime(), difftime(), localtime() etc..

```
//C++ program to print "Hello World"
#include <iostream>
using namespace std;
int main()
{
        cout<<"Hello World";
        return 0;
```

```
}
```

Output of the above code is:
Hello World

In the above program, **iostream** is the name of a header file which contains I/O functionality. In order to take input and print some output using our C++ program, we have to include that header file using **#include** directive. Note that writing ".h" at the end of the header file name results in an error. Such style is deprecated in newer versions of C++.

The **cin** and **cout** elements are declared in **std** namespace inside **iostream** header file. So, to use either **cin** or **cout**, we have to write **using namespace <name>**. If you don't want to write this line, you have to write **std::cin** and **std::cout** in your program instead of **cin** and **cout**.

Every C++ program must contain a **main()** function which specifies the starting point of execution. Since return type of main is **int**, you should return an integer value back in your program.

cout is the standard output object. It is used in conjunction with **insertion operator (<<)** to print output on the screen.

The **return** statement returned a value 0 which tells the operating system that the program has completed successfully. A non-zero value tells that the program execution was unsuccessful.

Finally, the first line is a comment. We can write single line comments in C++ using // and multi-line comments using /* and */.

Differences between C and C++

Following are various difference between C and C++:

C	C++
C is a structural programming language.	C++ is both structural and object oriented programming language.
C follows top-down approach.	C++ follows bottom-up approach.
C doesn't support virtual functions.	C++ supports virtual functions.
C doesn't supports object orientation features.	C++ supports object orientation features.
Operator overloading is not possible in C.	C++ supports operator overloading.
Data security is very less in C.	Data security is more in C++.
C is a middle level language.	C++ is a high level language.
C programs are divided into modules.	C++ programs are divided into classes and functions.
In C, main can be called from other functions.	In C++, main cannot be called from other functions.
Namespaces are not available in C.	Namespaces are available in C++.
Exception handling is not supported.	Exception handling is supported.
Function overloading is not possible.	Function overloading is possible.
scanf() and printf() are used for I/O.	cin and cout are used for I/O.
File extension in .c.	File extension is .cpp.

Input and Output in C++

Any program in a given programming language needs significant amount of input from the user and also needs to display to the output several times. To support such I/O operations, C++ provides a library which contains several classes and functions.

Streams in C++

In C++, standard library maintains input and output as streams. A stream is a flow of data, measured in bytes, in sequence. If data is received from input devices, it is called a source stream and if the data is to be sent to output devices, it is called a destination stream.

The data in the source stream can be used as input to the program. Then it is called an input stream. The destination stream can be used by the program to send data to the output devices. Then it is called an output stream.

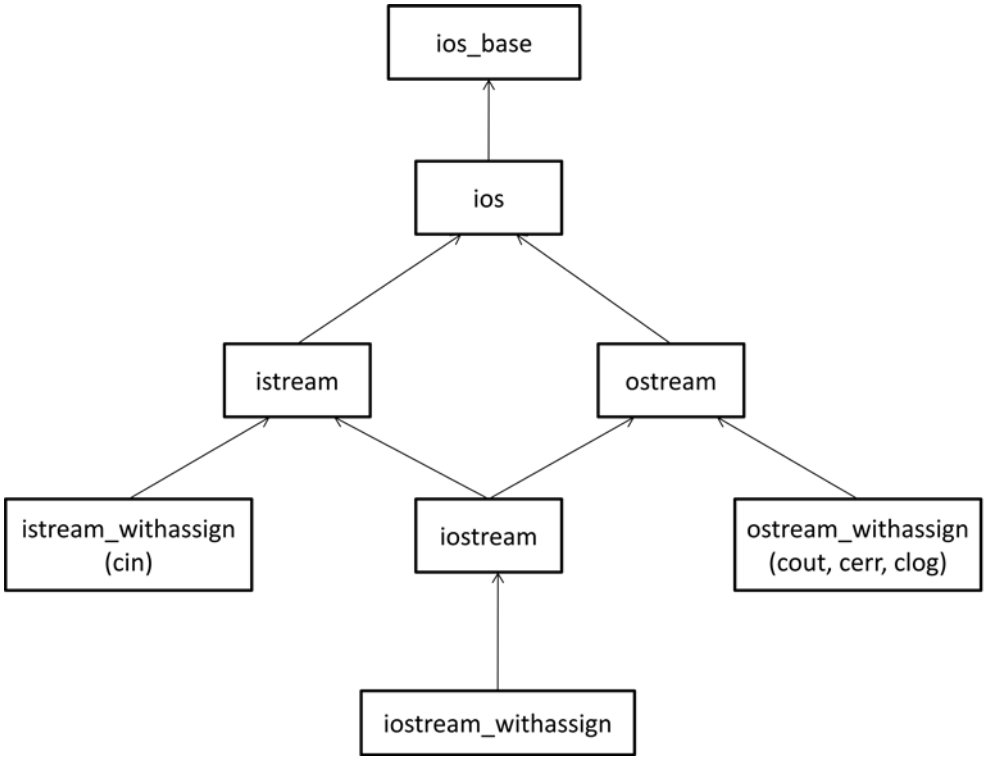
Pre-Defined Streams

C++ contains number of pre-defined streams. They are also called as standard I/O objects. These streams are available automatically when the execution of a program starts. The pre-defined streams available in C++ are:

cin: Standard input, usually keyboard. It handles input devices. It is analogous to **stdin** in C.
cout: Standard output, usually monitor. It handles output devices. It is analogous to **stdout** in C.
cerr: Standard error output, usually screen. It handles the unbuffered output data. It is analogous to **stderr** in C.
clog: A fully buffered version of **cerr**. It handles error messages that are sent from buffer to the standard error device. There is no equivalent of **clog** in C.

Stream Classes

C++ provides several classes to work with stream based I/O. Such classes are known as stream classes and they are arranged in a hierarchy shown below:



All these classes are declared in **iostream.h** header file. So, we have to include this header file in order to use the stream classes. As shown in the above figure, **ios** is the base class for all the stream classes from which **istream** and **ostream** classes are derived. The **iostream** class is derived from both **istream** and **ostream** classes.

Following table lists out some of the stream classes and their contents:

Class	Content / Functions
ios	• It is an input and output stream class. • It is used to implement a buffer. It is a pointer to the buffer, streambuf. • ios matains information about the state of the buffer.
istream	• istream provides formatted input. • It is used to handle formatted as well as unformatted conversion of character from a streambuf. • istream declares functions such as peek(), tellg(), seekg(), getline(), read() etc. • istream class overloads the ">>" operator.
ostream	• It is used for general purpose output. • ostream declares functions such as tellp(), put(), write(), seekp(), etc. • ostream overloads the "<<" operator.
iostream	• It is used to handle both input and output streams.
istream_withassign	• It is derived from istream. • It is used for cin input.
iostream_withassign	• It is a bidirectional stream.

Formatted and Unformatted Data

Data which is received by the program without any modifications and sent to the output device without any modifications is known as unformatted data. On the other hand, sometimes we may want to apply some modifications to the actual data that is being received or sent. For example, we might want to display an integer in hexadecimal format in the output, leave some whitespace when printing a number and adjustments in the decimal point. Such modified data in known as formatted data.

As an example for formatted data, if we want to display a decimal number in hexadecimal format, we can use the **hex manipulator** as shown below:

```
cout<<hex<<15
```

Above line displays 15 in hexadecimal format as F.

Unformatted Console I/O

The input stream uses **cin** object of **istream** class to read data and the output stream uses **cout** object of **ostream** class to display data on output devices.

The cin statement uses >> (extraction operator) to read data from keyboard. Syntax of cin statement is shown below:

cin>>variable-name;

Example:

```
int a;
float f;
char ch;
cin>>a>>f>>ch;
```

In the above example, cin is used in cascading fashion. This is helpful for reading values into multiple variables in a single statement. No need of any format specifiers like %d, %f etc. The >> (extraction) operator will take care of that automatically.

If the data provided is greater than the number of variables in the cin statement, extra data will remain in the input stream.

The cout statement uses << (insertion operator) to display data to the output device. Syntax of cout statement is shown below:

cout<<variable-name;

Example:

```
cout<<a<<b<<c;
```

In the above example, a, b, and c are variables. No need to specify format specifiers like %d, %f, etc. The << (insertion) operator will take care of that automatically. All escape sequences like "\n", "\t", etc, can be used in cout statement.

```
//C++ program to read two numbers and print their sum
#include <iostream>
using namespace std;
int main()
{
        int a, b;
        cin>>a>>b;  //Reading values into variables a and b
        cout<<"Sum is: "<<(a+b);  //Printing the sum
        return 0;
}
```

Output of the above program is:
Sum is: 30
To read a single character, we can use **get()** function of **istream** class and to print a single character, we can use **put()** function of **ostream** class. Syntax of these methods is as follows:

get(char)
get(void)
put(char)

```
//C++ program to demonstrate get() and put() functions
#include <iostream>
using namespace std;
int main()
{
	char ch1,ch2,ch3;
	cin.get(ch1).get(ch2).get(ch3);
	cout.put(ch1).put(ch2).put(ch3);
	return 0;
}
```

Input: xyz

Output of the above program is: xyz

To read a string or a line of text, we can use **getline()** function and to print a string we can use **write()** function. Syntax of both of these methods is as follows:

getline(string, size)
write(string, size)

The *size* in the above syntax specifies the number of characters to read and print in the given string.

```
//C++ program to demonstrate getline() and write() functions
#include <iostream>
using namespace std;
int main()
{
	char str[30];
	cin.getline(str, 10);
	cout.write(str, 10);
	return 0;
}
```

Input: C++ rocks

Output of the above program is:

C++ rocks

istream Class Functions

The **istream** class is derived from **ios** class. The istream class contains the following functions:

Function	Purpose
get(ch)	Extract one character into ch.
get(str)	Extract characters into array *str*, until '\n'.
putback(ch)	Insert last character read back in to input stream.
ignore(MAX, DELIM)	Extract and discard MAX characters until the specified delimiter DELIM.
peek(ch)	Read one character without extracting it from the stream.
gcount()	Return number of characters read by immediate *get(), getline(), or read()* function.
read(str, MAX)	Extract upto MAX characters, until EOF from a file.
seekg()	Set distance of file pointer from start of file.
seekg(pos, seek_dir)	Set distance of file prointer from specified place in a file. *seek_dir* can be *ios::beg, ios::cur,* or *ios::end*.
tellg(pos)	Return position of file pointer from start of file.

Formatted Console I/O

C++ provides various console I/O functions for formatted input and output. The three ways to display formatted input and output are:

1. **ios** class functions and flags
2. Standard manipulators
3. User-defined manipulators

ios Class

The **ios** class is the base class for all input and output classes in C++. Following are some of the functions available in **ios** class:

Function	Working
width()	To set the width of the field. Output will be displayed in the given width.
precision()	To set the number of decimal places to display in a *float* value.
fill()	To set the character to be displayed in the blank space of a field.
setf()	To set various flags for formatting output.
unsetf()	To remove the flag settings.

The **width()** function can be used in two ways as shown below:

int width() : Returns the current width setting.

int width(int) : Sets the specified width and returns the previous width setting.

The **precision()** function can be used in two ways as shown below:

int precision() : Returns the current precision setting.

int precision(int) : Sets the specified precision and returns the previous precision setting.

The **fill()** function can be used in two ways as shown below:

char fill() : Returns the current fill character.

char fill(char) : Sets the fill character and returns the previous fill character.

The **setf()** function can be used in two ways as shown below:

long setf(long sb, long f) : The bits specified by the variable *f* are removed from data member *x_flags* in **ios** and set according the value specified by *sb*.

long setf(long) : Sets the flags in *x_flags* based on the given value.

Bit Fields

The **ios** class contains several flags and bit fields to adjust or format the output displayed on the screen. Available bit fields and flags in **ios** class are as follows:

Format	Flag	Bit Field
Left justification	ios::left	ios::adjustfield
Right justification	ios::right	ios::adjustfield
Padding after sign and base	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

ios::adjustfield is used with **setf()** function to set the alignment of padding to left, right, or internal.

ios::floatfield is used with **setf()** function to set the floating point notation to scientific or fixed.

ios::basefield is used with **setf()** function to set the display notation to decimal, octal or hexadecimal.

Flags without Bit Fields

Some flags does not contain any bit fields. Such flags are listed below:

Flag	Purpose
Ios::showbase	Uses base indicator on output.
Ios::showpos	Displays + preceding positive number.
Ios::showpoint	Shows trailing decimal point and zeros.
Ios::uppercase	Uses capital case for output.
Ios::skipws	Skips white space on input.
Ios::unitbuf	Flushes all streams after insertion.
Ios::stdio	Adjusts the stream with C standard input and output.
ios::boolapha	Converts boolean values to text.

Note: In the above table replace *Ios* with *ios (sorry for the typo).*

Manipulators

Manipulators are helper functions which can be used for formatting input and output data. The **ios** class and **iomanip.h** header file has several pre-defined manipulators.

Below table lists out several pre-defined manipulators. The manipulators **hex, oct, dec, ws, endl, and flush** are defined in **iostream.h**. The manipulators **setbase(), width(), fill(),** etc., that require an argument are defined in **iomanip.h**.

User-Defined Manipulators

Sometimes, to satisfy custom requirements, we have to create our own manipulator. Such manipulators which are created by the programmer or user are known as user-defined manipulators.

Syntax for creating a user-defined manipulator is as follows:

Manipulator	Function
setw(int n)	To set the field width to *n*
setbase	To set the base of the number system
setprecision(int p)	The precision is fixed to *p*
setfill(char f)	To set the character to be filled
setiosflags(long l)	Format flag is set to *l*
resetiosflags(long l)	Removes the flags indicated by *l*
endl	Gives a new line
skipws	Omits white space in input
noskipws	Does not omit white space in the input
ends	Adds null character to close an output string
flush	Flushes the buffer stream
lock	Locks the file associated with the file handle
ws	Omits the leading white spaces present before the first field
hex, oct, dec	Displays the number in hexadecimal or octal or in decimal format

```
ostream & m_name(ostream &os)
{
        Statement(s);
        return os;
}
```

In the above syntax, *m_name* is our user-defined manipulator name. As an example for user-defined manipulator let's create a manipulator which prints a new line.

```
ostream & newl(ostream & os)
{
        os<<"\n";
        return os;
}
```

After creating the manipulator named *newl*, we can use it our program as shown below:

```
int main()
{
        cout<<"Hello"<<newl<<"World"<<newl;
        return 0;
}
```