NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF MECHATRONICS ENGINEERING

COURSE MATERIALS

MRT 363 OBJECT ORIENTED PROGRAMMING

VISION OF THE INSTITUTION
To mould true citizens who are millennium leaders and catalysts of change through excellence in education.
MISSION OF THE INSTITUTION
NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values. We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2013
- Course offered: B.Tech Mechatronics Engineering
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of Dr. A P J Abdul Kalam Technological University.

DEPARTMENT VISION

To develop professionally ethical and socially responsible Mechatronics engineers to serve the humanity through quality professional education.

DEPARTMENT MISSION

1) The department is committed to impart the right blend of knowledge and quality education to create professionally ethical and socially responsible graduates.

2) The department is committed to impart the awareness to meet the current challenges in technology.

3) Establish state-of-the-art laboratories to promote practical knowledge of mechatronics to meet the needs of the society

PROGRAMME EDUCATIONAL OBJECTIVES

I. Graduates shall have the ability to work in multidisciplinary environment with good professional and commitment.

II. Graduates shall have the ability to solve the complex engineering problems by applying electrical, mechanical, electronics and computer knowledge and engage in lifelong learning in their profession.

III. Graduates shall have the ability to lead and contribute in a team with entrepreneur skills, professional, social and ethical responsibilities.

IV. Graduates shall have ability to acquire scientific and engineering fundamentals necessary for higher studies and research.

PROGRAM OUTCOME (PO'S)

Engineering Graduates will be able to:

PO 1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO 2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO 3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO 11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOME(PSO'S)

PSO 1: Design and develop Mechatronics systems to solve the complex engineering problem by integrating electronics, mechanical and control systems.

PSO 2: Apply the engineering knowledge to conduct investigations of complex engineering problem related to instrumentation, control, automation, robotics and provide solutions.

COURSE OUTCOME

After the completion of the course the student will be able to

CO 1	Understand the special features of object oriented programming approach in connection with C++
CO 2	Apply the concept of constructors.
CO 3	Apply the concept of Operator Overloading.
CO 4	Evaluate the different exception handling mechanisms.
CO 5	Apply virtual and pure virtual functions and complex programming situations.
CO6	Illustrate the process of data file manipulations using C++.

CO VS PO'S AND PSO'S MAPPING

CO	PO1	PO 2	PO3	PO 4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PS0 1	PSO 2
CO 1	3	2	1	-	1	2	-	1	-	3	-	3	3	3
CO 2	3	3	1	2	2	-	-	2	-	3	1	3	2	3
CO 3	3	3	1	2	1	-	-	2	-	3	1	3	3	3
CO 4	3	3	2	3	-	-	-	-	-	-	-	3	2	2
CO 5	3	2	1	3	-	-	-	1	-	2	1	3	3	2
CO6	3	1	2	-	2	-	-	-	-	3	-	3	2	2

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

SYLLABUS

Module 1: Object oriented programming concepts - objects - classes - methods and messages - abstraction and encapsulation - inheritance - abstract classes - polymorphism. Introduction to C++ - classes - access specifiers - function and data members - default arguments - function overloading - friend functions - const and volatile functions - static members.

Module 2: Objects - pointers and objects - constant objects - nested classes - local classes-Constructors - default constructor - Parameterized constructors - Constructor with dynamic allocation - copy constructor - destructors.

Module 3: Operator overloading - overloading through friend functions - overloading the assignment operator - type conversion - explicit constructor.

Module 4: Function and class templates - Exception handling - try-catch- throw paradigm - exception specification - terminate and Unexpected functions - Uncaught exception.

Module 5: Inheritance - public, private, and protected derivations - multiple inheritance - virtual base class - abstract class - composite objects Runtime polymorphism - virtual functions - pure virtual functions - RTTI - typeid - dynamic casting - RTTI and templates - cross casting - down casting .

Module 6: Streams and formatted I/O - I/O manipulators - file handling - random access - object serialization - namespaces - std namespace - ANSI String Objects - standard template library.

QUESTION BANK

MODULE I				
Q:NO:	**QUESTIONS**	**CO**	**KL**	**PAGE NO:**
1	How are data and functions organized in an object oriented program?	CO1	K2	13
2	What is Object oriented programming? Write the features of Object Oriented Programming?	CO1	K1	14
3	Distinguish between the following terms a)Objects and classes b) Data abstraction and data encapsulation c) Inheritance and Polymorphism	CO1	K2	17
4	Describe Inheritance as applied to OOP?	CO1	K2	18
5	Write and explain the four features of object Oriented Programming?	CO1	K2	22
6	Explain friend function with example?	CO1	K5	52
7	Explain static function with example?	CO1	K5	53
8	Explain access modifier public with example?	CO1	K5	38
9	Explain two different methods for creating objects of the class?	CO1	K2	23
10	How to access class members ?	CO1	K1	24
11	How can you define a function? Write the syntax for function definition? With example	CO1	K2	32

12	How Cascading of I/O Operators can be used?	CO1	K2	19
13	Define member function of a class (1) Inside class (2) Outside class	CO1	K2	29
14	Discuss about access modifier private with example?	CO1	K5	26
MODULE II				
1	Define and write the syntax of pointer to object concept?	CO2	K2	68
2	What are concept objects?	CO2	K1	71
3	Explain constructor, Briefly describe types of constructor?	CO2	K3	74
4	Briefly describe default constructor with example?	CO2	K5	75
5	Explain copy constructor with example?	CO2	K5	77
6	Explain parameterized constructor with example?	CO2	K5	76
MODULE III				
1	Define operator overloading?	CO3	K2	86
2	Explain the rules for overloading the operators?	CO3	K1	87
3	Explain overloading unary operator with a program?	CO3	K5	88
4	Explain overloading binary operator overloaded using friend function?	CO3	K5	94

5	Explain overloading binary operator with example?	CO3	K5	91
6	Briefly describe assignment operator with example?	CO3	K5	95
7	With the help of an example describe implicit type conversion?	CO3	K4	99
8	With the help of an example describe explicit type conversion?	CO3	K4	100
9	Explain type conversion and its different type?	CO3	K2	99
MODULE IV				
1	Define function template with example? Write its syntax?	CO4	K2	102
2	Write the program to handle different data types without separate code for each of them?	CO4	K6	101
3	Explain exception handling ? write the syntax for exception handlers?	CO4	K2	104
4	Define 1. Try block 2. Catch block 3. Throw keyword	CO4	K1	106
5	Write a program to implement try , catch and throw keyword.	CO4	K5	108
6	Define exception specification? Write its syntax?	CO4	K2	115

7	Two special library functions are implemented in C++ to process exception not properly handled by catch blocks or exception thrown outside of a valid try block? Explain them briefly?	CO4	K4	115
8	Define uncaught exception? Explain the functions to handle uncaught exceptions with example?	CO4	K2	116
9	Briefly describe catch all handlers with an example?	CO4	K2	117
10	Differentiate Single try multiple catch and catch all paradigm?	CO4	K2	109
	MODULE V			
1	Explain inheritance? Why and when to use inheritance?	CO5	K4	120
2	Write the syntax for inheritance? Explain with an example?	CO5	K5	121
3	Illustrate with a real life example how inheritance is implemented in C++ program?	CO5	K6	122
4	Explain different modes of inheritance?	CO5	K2	122
5	Illustrate with a real life example how multiple inheritance is implemented in C++ program?	CO5	K6	125
6	Illustrate with a real life example how single level inheritance is implemented in C++ program?	CO5	K6	124

7	Illustrate with a real life example how hierarchical inheritance is implemented in C++ program?	CO5	K6	127
8	Illustrate with a real life example how multi level inheritance is implemented in C++ program?	CO5	K6	126
9	Illustrate with a real life example how hybrid inheritance is implemented in C++ program?	CO5	K6	128
10	Explain abstract class with the help of an example?	CO5	K2	139
11	Distinguish upcasting and down casting?	CO5	K5	149
12	Write the Pros and Cons of RTTI?	CO5	K2	170
13	Write the syntax of Pure virtual function?	CO5	K2	137
14	Explain dynamic cast operator with help of an example?	CO5	K1	147
15	Differentiate between static binding and late binding?	CO5	K2	141
16	Explain the use of typeid operator in C++?	CO5	K2	151

MODULE VI				
1	The basic data type for I/O in C++ is the stream. How?	CO6	K5	174
2	In C++ program, to get information into a file or a program, we need to explicitly instruct the computer to acquire the desired information. How?	CO6	K6	176

3	Write the header files used for formatted I/O in C++?	CO3	K3	177
4	Write a stream input program for the sum of 3 integers?	CO6	K2	179
5	Write a sample program to writing to a file?	CO6	K6	181
6	Write a sample program to reading from a file?	CO6	K6	182
7	Write a sample program to close a file?	CO6	K6	183
8	Write special operations in a file?	CO6	K2	184
9	C++ standard template library contain 3 well structured components? Explain	CO6	K2	204
10	Explain the use of namespace?	CO6	K1	197
11	How object serialization achieved in C++?	CO6	K2	192
12	What are the different file opening modes in C++?	CO6	K2	180
13	How the random access is achieved in file?	CO6	K2	186

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

S:NO	TOPIC	PAGE NO:
1	C++ signal handling	234
2	Multidimensional Array in C/ C++	235
3	Setting up C++ development environment	239

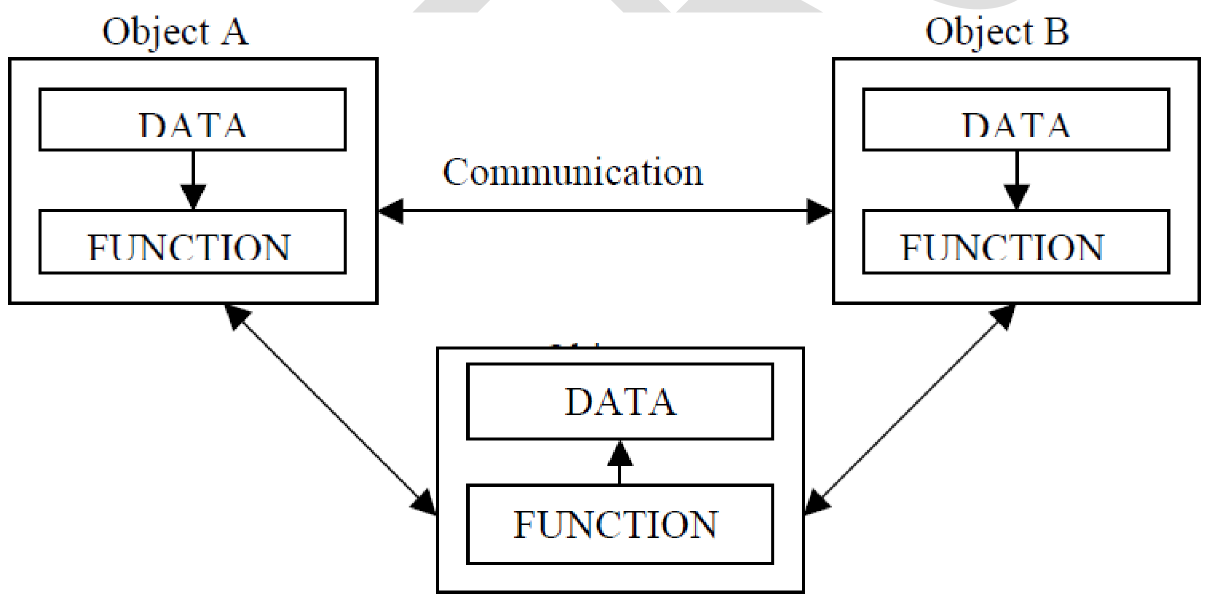
MODULE 1

Basic Concepts of Object Oriented Programming

Object Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

The organization of data and function in object-oriented programs is shown in fig.1.3. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.

Organization of data and function in OOP



Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.

• New data and functions can be easily added whenever necessary.
• Follows bottom up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

Basic Concepts of Object Oriented Programming
It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:
• Objects
• Classes
• Data abstraction and encapsulation
• Inheritance
• Polymorphism
• Dynamic binding
• Message passing

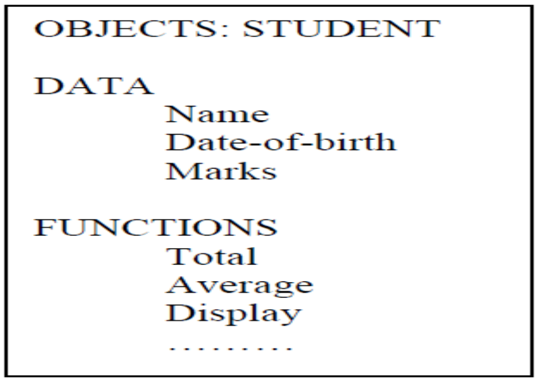
Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are to object in a program, then the customer object may send a message to the count object requesting for the bank balance. Each object contain data, and code to manipulate data. Objects can interact without having to know details of each other's data or code. It is a sufficient to know the type of message accepted, and the type of response returned by the objects.

fig 1.5 shows two notations that are popularly used in object-oriented analysis and design.



OBJECTS: STUDENT

DATA
- Name
- Date-of-birth
- Marks

FUNCTIONS
- Total
- Average
- Display
- ....

Fig. 1.5 representing an object

Classes

The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types. For examples, Mango, Apple and orange members of class fruit.

Classes are user-defined that types and behave like the built-in types of a programming language. The syntax used to create an object is not different then the syntax used to create an integer object in C. If fruit has been defines as a class, then the statement

Fruit Mango;

Will create an object **mango** belonging to the class **fruit.**

Data Abstraction and Encapsulation

The wrapping up of data and function into a single unit (called class) is known as *encapsulation*. Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

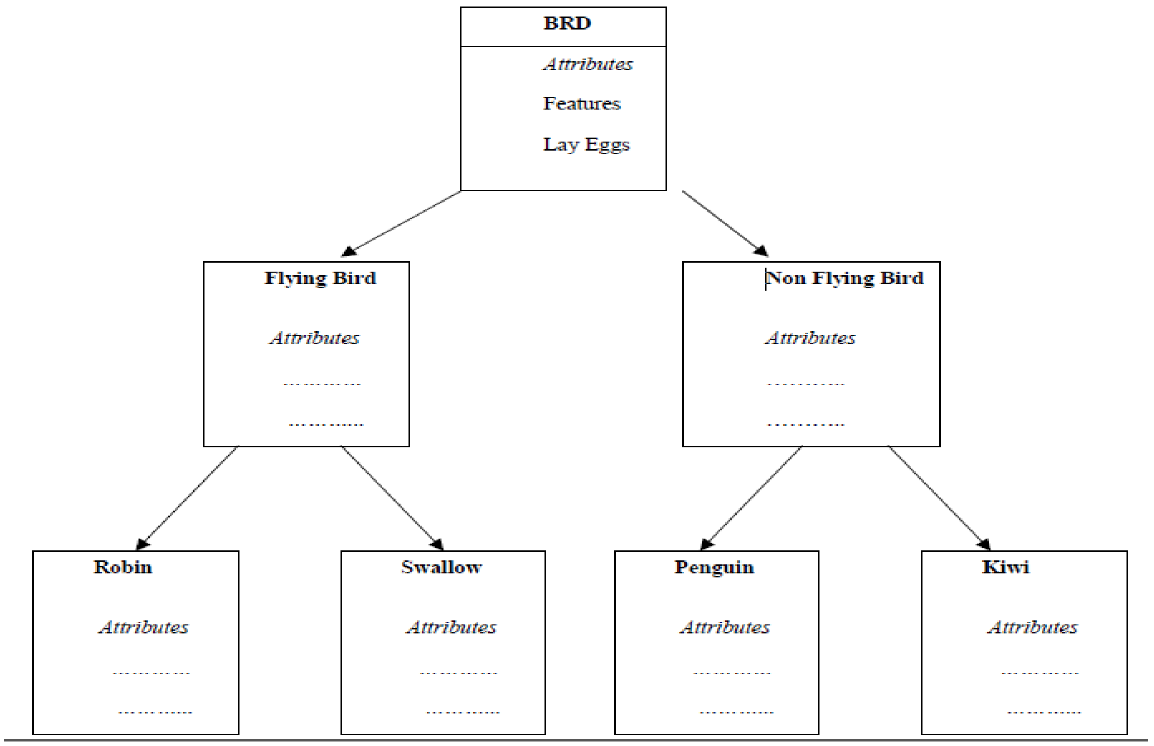
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.

The attributes are some time called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member function*.

Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of *hierarchical classification*. For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig 1.6.In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes.

Fig. 1.6 Property inheritances



Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to

exhibit different behaviors in different instances is known as *operator overloading*.
Fig. 1.7 illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as *function overloading.*

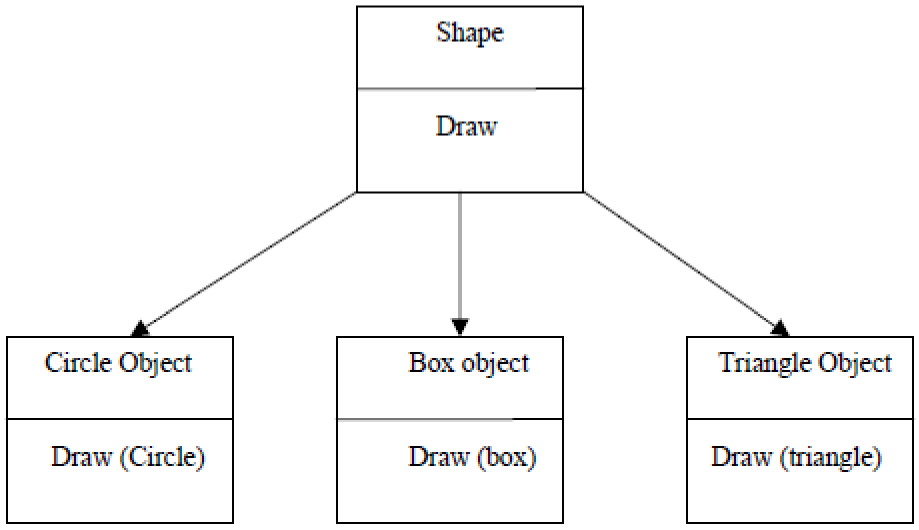


Fig. 1.7 Polymorphism

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

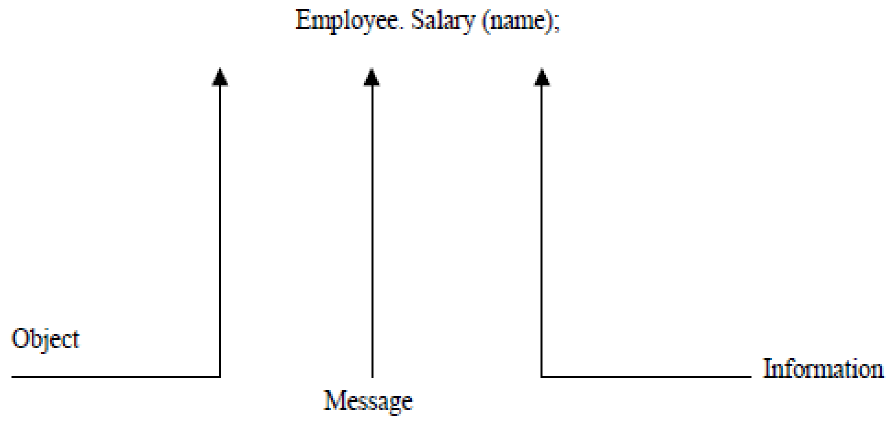
Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another.
Message passing involves specifying the name of object, the name of the function

(message) and the information to be sent. Example:



Application of OOP

Real-business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem. The areas of application of OOP include:

- Real-time system
- Simulation and modeling
- Object-oriented data bases
- Hypertext, Hypermedia, and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

Introduction of C++

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories USA, in the early 1980's.

Stroustrup, a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C.

The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature ZX. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented

version of C.
C+ + is a superset of C. Almost all c programs are also C++ programs. However, there are a few minor differences that will prevent a c program to run under C++ complier. The most important facilities that C++ adds on to C care classes, inheritance, function overloading and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

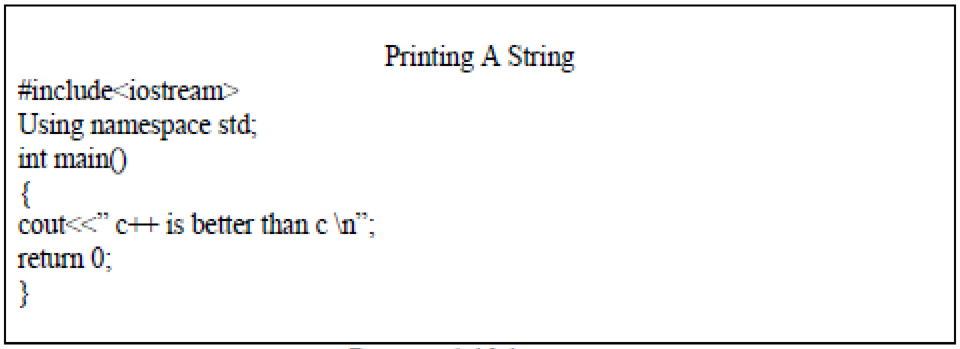
1.9.1 Application of C++

C++ is a versatile language for handling very large programs; it is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life applications systems.

- Since C++ allow us to create hierarchy related objects, we can build special object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get closed to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general-purpose language in the near future.

1.10 Simple C++ Program

Let us begin with a simple example of a C++ program that prints a string on the screen.



Printing A String

```
#include<iostream>
Using namespace std;
int main()
{
cout<<" c++ is better than c \n";
return 0;
}
```

Program 1.10.1
This simple program demonstrates several C++ features.

1.10.1 Program feature

Like C, the C++ program is a collection of function. The above example contain only one function **main().** As usual execution begins at main(). Every C++ program must have a **main()**. C++ is a free form language. With a few exception, the compiler ignore carriage return and white spaces. Like C, the C++ statements terminate with semicolons.

1.10.2 Comments

C++ introduces a new comment symbol // (double slash). Comment start with a double slash symbol and terminate at the end of the line.
A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.
The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
// This is an example of
// C++ program to illustrate
```

The C comment symbols /*,*/ are still valid and are more suitable for multiline comments. The following comment is allowed:

```
/* This is an example of C++ program to illustrate
some of its features */
```

1.10.3.Output Operator

Cout<<"C++ is better than C.";
Causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout(pronounced as C out) is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. The operator << is called the insertion or put to operator.

1.10.4 The iostream File

We have used the following #include directive in the program:
#include <iostream>
The #include directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file. The header file **iostream.h** should be included at the beginning of all programs that use input/output statements.

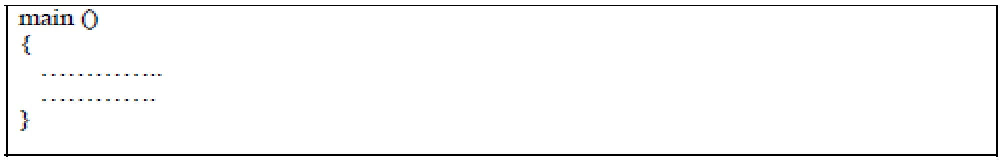
1.10.5 Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the

identifier defined in the **namespace** scope we must include the using directive, like Using namespace std;
Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. **Using** and **namespace** are the new keyword of C++.

1.10.6 Return Type of main()

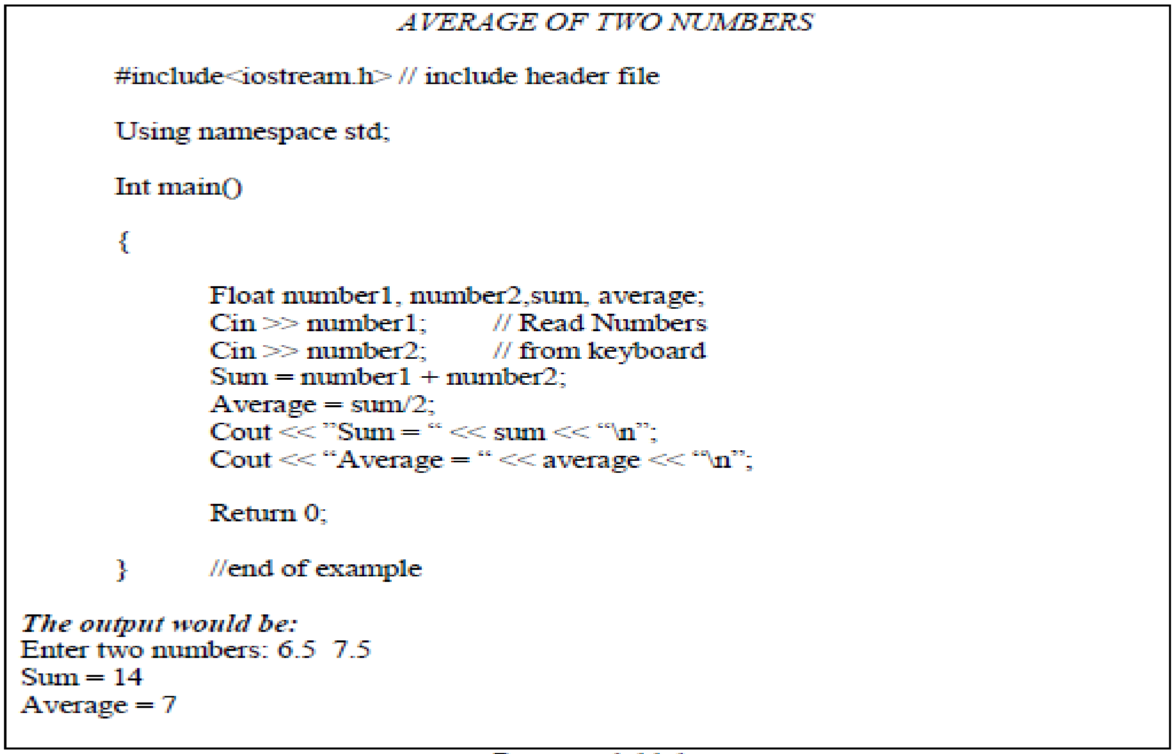
In C++, main () returns an integer value to the operating system. Therefore, every main () in C++ should end with a return (0) statement; otherwise a warning an error might occur. Since main () returns an integer type for main () is explicitly specified as **int.** Note that the default return type for all function in C++ is **int.** The following main without type and return will run with a warning:



```
main ()
{
   ..............
   ..............
}
```

1.11 More C++ Statements

Assume that we should like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in program 1.11.1



AVERAGE OF TWO NUMBERS

```
#include<iostream.h> // include header file

Using namespace std;

Int main()

{

        Float number1, number2,sum, average;
        Cin >> number1;        // Read Numbers
        Cin >> number2;        // from keyboard
        Sum = number1 + number2;
        Average = sum/2;
        Cout << "Sum = " << sum << "\n";
        Cout << "Average = " << average << "\n";

        Return 0;

}       //end of example
```

The output would be:

```
Enter two numbers: 6.5  7.5
Sum = 14
Average = 7
```

Program 1.11.1

1.11.1 Variables

The program uses four variables number1, number2, sum and average. They are declared as type float by the statement.

float number1, number2, sum, average;

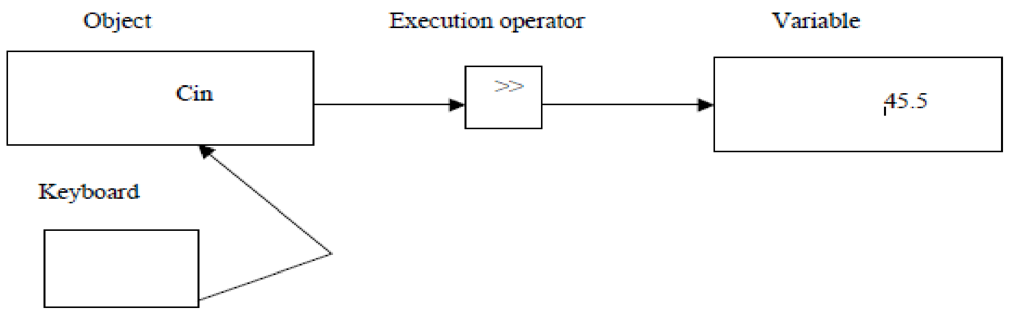
All variable must be declared before they are used in the program.

1.11.2 Input Operator The statement

cin >> number1;

Is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier cin (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as extraction or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right fig 1.8. This corresponds to a familiar scanf() operation. Like <<, the operator >> can also be overloaded.



1.8 Input using extraction operator

1.11.3 Cascading of I/O Operators

We have used the insertion operator << repeatedly in the last two statements for printing results.

The statement

Cout << "Sum = " << sum << "\n";

First sends the string "Sum = " to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

Cout << "Sum = " << sum << "\n"

<< "Average = " << average << "\n";

This is one statement but provides two line of output. If you want only one line of output, the statement will be:

Cout << "Sum = " << sum << ","

<< "Average = " << average << "\n";

The output will be:

Sum = 14, average = 7

We can also cascade input iperator >> as shown below:

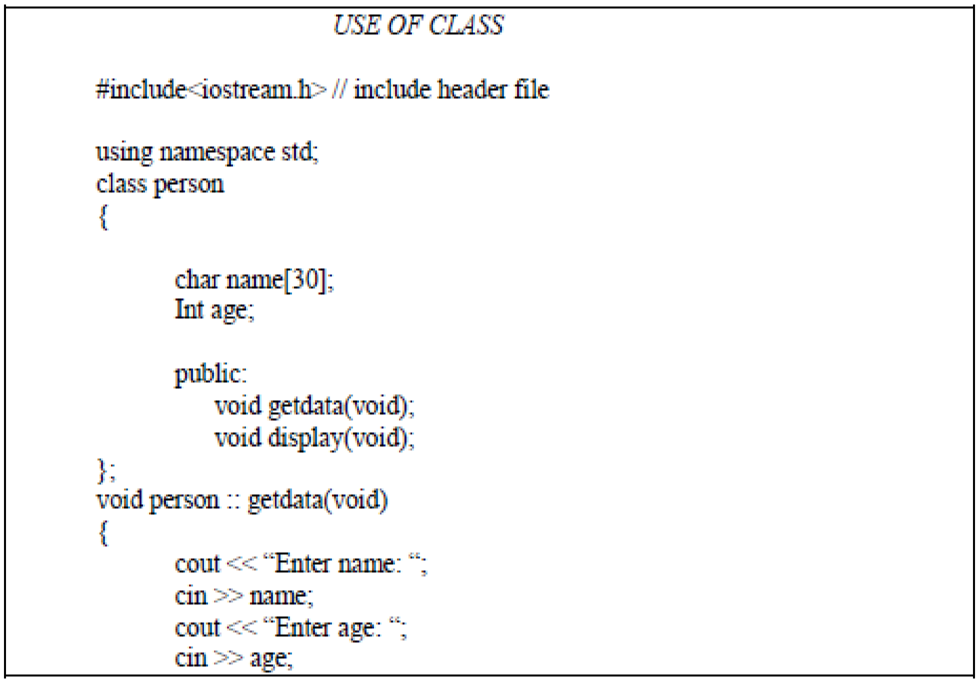
Cin >> number1 >> number2;

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to munber1 and 20 to number2.

1.12 An Example with Class

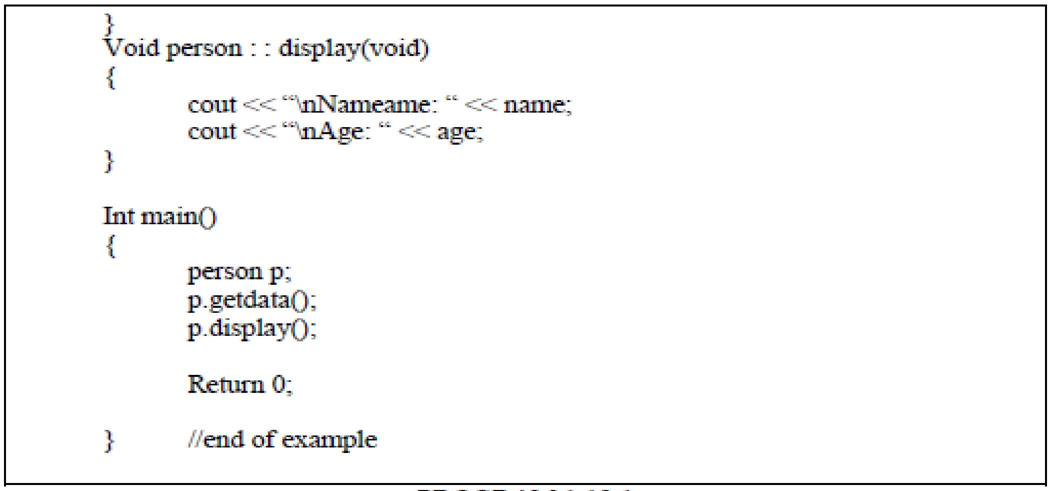
- One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

Program 1.12.1 shows the use of class in a C++ program.



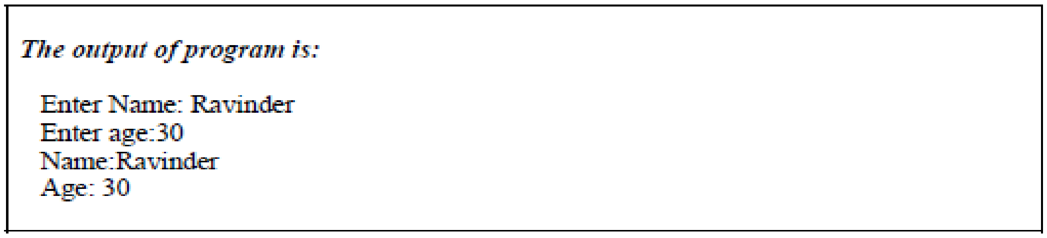
USE OF CLASS

```
#include<iostream.h> // include header file

using namespace std;
class person
{

        char name[30];
        Int age;

        public:
            void getdata(void);
            void display(void);
};
void person :: getdata(void)
{
        cout << "Enter name: ";
        cin >> name;
        cout << "Enter age: ";
        cin >> age;
```



```
}
Void person : : display(void)
{
        cout << "\nNameame: " << name;
        cout << "\nAge: " << age;
}

Int main()
{
        person p;
        p.getdata();
        p.display();

        Return 0;

}       //end of example
```

PROGRAM 1.12.1



The output of program is:

```
Enter Name: Ravinder
Enter age:30
Name:Ravinder
Age: 30
```

The program define **person** as a new data of type class. The class person includes two basic data type items and two function to operate on that data. These functions are called **member function**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as objects. Here, p is an object of type **person**. Class object are used to invoke the function defined in that class.

Structure of C++ Program

The structure of C++ program is divided into four different sections:

(1) Header File Section

(2) Class Declaration section

(3) Member Function definition section

(4) Main function section

(1) Header File Section:

o This section contains various header files.

o You can include various header files in to your program using this section.

For example:

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

o Header file contains declaration and definition of various built in functions as well as object. In order to use this built in functions or object we need to include particular header file in our program.

(2) Class Declaration Section:

o This section contains declaration of class.

o You can declare class and then declare data members and member functions inside that class.

For example:

```
class Demo
{
int a, b;
public:
```

```
void input(); void output();
}
```

o You can also inherit one class from another existing class in this section.

(3) Member Function Definition Section:

o This section is optional in the structure of C++ program.

o Because you can define member functions inside the class or outside the class. If all the member functions are defined inside the class then there is no need of this section.

o This section is used only when you want to define member function outside the class.

o This section contains definition of the member functions that are declared inside the class.

For example:

```
void Demo:: input ()
{
cout << “Enter Value of A:”;
cin >> a;
cout << “Enter Value of B:”;
cin >> b;
}
```

(4) Main Function Section:

o In this section you can create an object of the class and then using this object you can call various functions defined inside the class as per your requirement.

For example:

```
void main ()
{ Demo d1; d1.input (); d1.output (); }
```

We can also compare the structure of C++ program with client server application.

In client server application client send request to the server and server sends response to the client.
In above C++ structure the class declaration section and member function definition section both together works as a server and main () function section works as a client. Because in main () function section we create an object of the class and then using that object we make a call to the function declared in the class.

What is Class?

A class is a user defined data type that allows us to bind data and its associated functions together as a single unit. Thus class provides the facility of data encapsulation.
Class provides the facility of data hiding using the concept of visibility mode such as public, private and protected.
Once a class is defined we can create an object of the class to access variables and functions defined inside the class.

Syntax:

```
class Class_Name
{
Private:
Data-Type Variable_Name;
Function declaration or Function Definition;
Public:
Data-Type Variable_Name;
Function declaration or Function Definition;
};
```

Class can be created using the **class keyword**. The class definition starts with curly bracket and ends with curly bracket followed by semicolon. We can declare variables as well as functions inside the curly bracket as shown in the syntax. The variables defined inside class are known as **data member** and the function declared inside the class are known as **member function**.

In order to provide **data hiding** facility class provides the concept of **visibility mode** such as **private, public or protected**. If you don't specify any visibility mode for the member of the class then **by default all the members of the class are considered as private**.

The data member and member function declared as a public can be accessed directly using the object of the class. But the data member and member function declared as private cannot be accessed directly using the object of the class.

Example:

```
class test
{
int a, b;
public:
void input ();
{
cout<<"Enter Value of a and b";
cin>>a>>b;
}
void output ()
{
cout<<"A="<<a<<"B="<<b;
}
};
```

Create Object in C++: There are two different methods for creating objects of the class:

(1) We can create object at the time of specifying a class after the closing curly bracket.

Example:

```
Class test
{
int a,b;
public:
```

```
void input ();
void ouput ();
}t1,t2,t3;
```

Here, t1, t2 and t3 are the objects of class test.

(2) We can create object inside the main function using name of the class.

The general syntax for creating object inside main function is as below:

Class_Name Object_Name;

Example:

Test t1, t2, t3;

Here, t1, t2 and t3 are the objects of class test.

How to Access Class Members?

The data member and member function declared as a public can be accessed directly using the object of the class. But the data member and member function declared as private cannot be accessed directly using the object of the class.
We can access private member of the class using public member of the class.
The general syntax for accessing public member using object is given below:

Object_Name. Data_Member = value;

Object_Name. Member_Function (Argument_List);

Example:

T1. input ();

T1. output ();

Example:

```
class Test
{
int b;
Public:
int a;
void inputb()
{
b=20;
}
};
int main()
{
Test t1;
T1.a=10; //works because a is public
T1.b=20; //error because b is private
T1.inputb (); //works because inputb () is public
return 0;
}
```

Access Modifiers in C++

Access modifiers are used to implement important feature of Object Oriented Programming known as Data Hiding.

Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: **If we do not specify any access modifiers for the members inside the class then by default**

the access modifier for the members will be Private.

Let us now look at each one these access modifiers in details:

Public: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

// C++ program to demonstrate public access modifier

```
#include<iostream>

using namespace std;

 // class definition

class Circle

{

   public:

      double radius;


      double  compute_area()

      {

         return 3.14*radius*radius;

      }


};


// main function

int main()

{

   Circle obj;


   // accessing public datamember outside class

   obj.radius = 5.5;
```

```
    cout << "Radius is:" << obj.radius << "\n";

    cout << "Area is:" << obj.compute_area();

    return 0;

}
```

Output:

```
Radius is:5.5

Area is:94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

- **Private**: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class. Example:

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double  compute_area()
        {   // member function can access private
            // data member radius
            return 3.14*radius*radius;
        }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;
    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

- The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class. **Output**:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
     double radius;
            ^
31:9: error: within this context
   obj.radius = 1.5;
       ^
```

- However we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

```
// C++ program to demonstrate private

// access modifier


#include<iostream>

using namespace std;


class Circle

{

    // private data member

    private:

        double radius;


    // public member function

    public:

        double  compute_area(double r)

        {   // member function can access private

            // data member radius

            radius = r;


            double area = 3.14*radius*radius;


            cout << "Radius is:" << radius << endl;

            cout << "Area is: " << area;

        }


};
```

```
// main function

int main()

{

    // creating object of the class

    Circle obj;


    // trying to access private data member

    // directly outside the class

    obj.compute_area(1.5);



    return 0;

}
```

- **Output**:

```
Radius is:1.5
Area is: 7.065
```

Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
// C++ program to demonstrate

// protected access modifier

#include <bits/stdc++.h>

using namespace std;


// base class

class Parent

{

    // protected data members

    protected:

    int id_protected;


};


// sub class or derived class

class Child : public Parent

{



    public:

    void setId(int id)

    {


        // Child class is able to access the inherited

        // protected data members of base class


        id_protected = id;
    }

    void displayId()
    {
        cout << "id_protected is:" << id_protected << endl;
    }
};

// main function
int main() {

    Child obj1;

    // member function of derived class can
    // access the protected data members of base class

    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

- **Output**:

```
id_protected is:81
```

Define Member Function of Class

Member function can be defined in two different way:
(1) Inside class

(2) Outside class

(1) Inside Class:/Inline functions

When we declare the function in the class at the same time we can also give the definition of the function in the class as shown below:

```
class Test
{
int a,b;
public:
void input ()
{
cout<<"Enter Value of a";
cin>>a>>b;
}
};
```

The function defined inside class becomes inline by default.

(2) Outside Class:

We can also define the member function outside the class. But at that time we have to instruct compiler this function belongs to which class using scope resolution operator as follow:

Syntax:

```
Return-Type Class_Name :: Function_Name (parameter list)
{
Function definition
}
```

Example:

```
class Test
{
int a,b;
Public:
void input ();
};
void test :: input ()
```

```
{
cout<<"Enter Value of a";
cin>>a>>b;
}
```

Making outside function inline

The function defined inside the class becomes inline by default so all the restriction that applied to inline function is also applied to the member function defined inside the class.
However we can also make the function inline which is defined outside the class.
To make the outside function inline we have to just precede the definition with the keyword inline.

Example.

```
class Test
{
int a, b;
public:
void input ()
{
cout<<"Enter value of a and b";
cin>>a>>b;
}
void output ();
};
inline void Test :: output ()
{
cout<<"A="<<a<<endl<<"B="<<b;
}
```

Private Member Function in C++

Like data member of the class a member function can also be declared as private so that it cannot be accessed outside the class.
If we declare private member function then it cannot be accessed directly using the

object of the class so we have to access it from the public member function of the same class.

Example:

```
class Circle
{
int r;
float area ();
public:
void getRadius ();
void DisplayArea ();
};
int Circle ::area ()
{
return (3.14 * r * r); }
void Circle :: getRadius ()
{
cout<<"Enter Radius";
cin>>r;
}
void Circle ::DisplayArea ()
{
Cout<<"Area="<<area ();
}
int main()
{
Circle C1;
C1.getRadius();
C1.DisplayArea ();
return 0;
}
```

In the above example we have declared three member function getRadius (), DisplayArea () and area (). In which area () is defined as private. We have called only two member functions getRadius () and DisplayArea () using the name of the object inside main function. We have called the third member function area () from inside the DisplayArea () member function. This concept is known as private member function.

functons

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

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

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

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

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

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

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the

type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers

int max(int num1, int num2) {
   // local variable declaration
   int result;

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

   return result;
}
```

Function Declarations

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

A function declaration has the following parts −

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

For the above defined function max(), following is the function declaration −

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

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

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

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

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

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

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

```
#include <iostream>

using namespace std;


// function declaration

int max(int num1, int num2);


int main () {

  // local variable declaration:

  int a = 100;

  int b = 200;

  int ret;

   // calling a function to get max value.

  ret = max(a, b);

  cout << "Max value is : " << ret << endl;

   return 0;

}

 // function returning the max between two numbers

int max(int num1, int num2) {

  // local variable declaration
  int result;

   if (num1 > num2)

     result = num1;

  else

     result = num2;



  return result;

}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

```
Max value is : 200
```

Function Arguments

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

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

While calling a function, there are two ways that arguments can be passed to a function −

Sr.No	Call Type & Description
1	**Call by Value** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

2	**Call by Pointer** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	**Call by Reference** This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

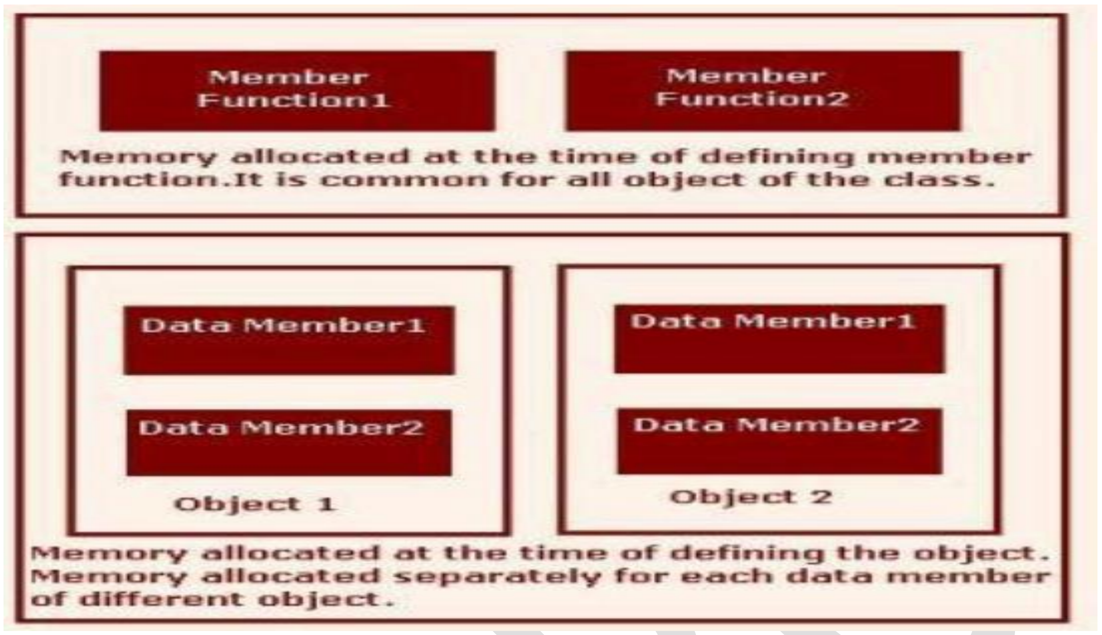
```
#include <iostream>
using namespace std;
 int sum(int a, int b = 20) {
  int result;
  result = a + b;
  return (result);}
int main () {
  // local variable declaration:
  int a = 100;
  int b = 200;

  int result;

   // calling a function to add the values.

  result = sum(a, b);

  cout << "Total value is :" << result << endl;

  // calling a function again as follows.

  result = sum(a);

  cout << "Total value is :" << result << endl;

   return 0; }
```

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

```
Total value is :300
Total value is :120
```

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.
The memory allocation of class members is shown below:


Member Function1
Member Function2
Memory allocated at the time of defining member function.It is common for all object of the class.
Data Member1
Data Member2
Object 1
Data Member1
Data Member2
Object 2
Memory allocated at the time of defining the object. Memory allocated separately for each data member of different object.

Hence data member of the class can contain different value for the different object, memory allocation is performed separately for each data member for different object at the time of creating an object.
Member function remains common for all object. Memory allocation is done only once for member function at the time of defining it.

Create an Array of Object

Like data member of the class we can also create an array of objects. Array of object is useful when we want to create large number of objects of the same class.

Example:

```
#include<iostream.h>
class Student
{
int rollno;  char name[20];
public:
```

```
void Input();  void Output();
};
void Student::Input()
{ cout<<"Enter Roll Number:"; cin>>rollno;  cout<<"Enter Name:";  cin>>name;  }
void Student::Output()
{ cout<<"Roll Number:"<<rollno<<endl;   cout<<"Name:"<<name<<endl;
}
int main()
{
Student S[3];  int i; for(i=0;i<3;i++)  S[i].Input();
for(i=0;i<3;i++)
S[i].Output();
return 0;
}
```

Output:

```
Enter Roll Number: 1
Enter Name: Mukesh
Enter Roll Number: 2
Enter Name: Ruchin
Enter Roll Number: 3
Enter Name: Mehul
```

Static Data Member of Class

In C++ memory is allocated separately for each data member for different objects. So if you change the value of data member of class using one object will not affect the value of the same data member for other object of the same class. However sometimes it is required that all the objects of the same class **share some common variables.** This can be accomplished using the concept of **static data member**.
We can declare static data members using **static keyword**.
Static data member having several characteristics which are given below:
(1) Hence all the object of the same class share static data member memory is allocated only once to the static data member. It remains common for all the objects of the same class.
(2) Static data member is initialized to 0 when first object of the class is created.

Static data member is declared in the class but it must be defined outside the class using class name and scope resolution operator (::) because memory allocation for static data member of the class is performed different then normal data member of the class and it is not the part of class object.

Example:

```
#include<iostream.h>
#include<conio.h>
class item
{
```

```
static int count;
public:
void DisplayCounter()
{
count++;
cout<<"count:"<<count;
} };
int item::count;
int main()
{
item a,b,c;
a.DisplayCounter();
b.DisplayCounter();
c.DisplayCounter();
return 0;
}
```

Output:

```
Count: 1
Count: 2
Count: 3
```

Static Member Function of Class

Like data member of the class, member function of the class can also be declared as a static.

Static member functions are designed to work with static data members.

In order to make a member function as static we need to precede the function declaration with **static keyword**.

Static member function having several characteristics which are given below:

(1) Static member function can access only static member of the class in which it is declared.

(2) Static member function is not a part of class object so it can not be called using object of the class. **Static member function can be called using class name and scope resolution operator as shown below:**

Class_Name :: Function_Name ();

Example:

```
#include <iostream.h>
#include <conio.h>
class test
{
```

```
private:
static int count;
public:
void setCount(void);
static void DisplayCounter(void);
};
void test :: setCount(void)
{
count++;
}
void test :: DisplayCounter(void)
{
cout<<"Count:"<< count << endl;
}
int test::count;
int main(void)
{
test t1, t2;
clrscr();
t1.setCount();
test :: DisplayCounter();
t2.setCount();
test :: DisplayCounter();
getch();
return(0);
}
```

Output:

```
Count: 1
Count: 2
```

Friend Function:A friend function is a function that is not a member of a class but it can access private and protected member of the class in which it is declared as friend.

Since friend function is not a member of class it can not be accessed using object of the class. It is called in the same way as normal external function is called.

It works same as your real life friend. Your friend is not a member of your family but still he knows about you and your family.

Sometimes it is required that private member of the class can be accessed outside the class at that time we have to use friend function.

A function can be declared as a friend by preceding function declaration with friend keyword as shown below:

friend Return_Type Function_Name (Argument List);

Example:

```
#include<iostream.h>
class Circle
{
int r;
public:
void input()
{
cout<<"Enter Radius:";
cin>>r;
}
friend float area(Circle C);
};
float area(Circle C)
{
return (3.14*C.r*C.r);
}
int main()
{
Circle C1;
C1.input();  cout<<"Area of Circle is:"<<area(C1);  return 0;
}
```

Friend function having following characteristics:

(1) A friend function can be declared inside class but it is not member of the class.
(2) It can be declared either public or private without affecting its meaning.
(3) A friend function is not a member of class so it is not called using object of the class. It is called like normal external function.
(4) A friend function accepts object as an argument to access private or public member of the class.
(5) A friend function can be declared as friend in any number of classes.

Const member functions in C++

A function becomes const when const keyword is used in function's declaration. The idea of const functions is not allow them to modify the object on which they are called. It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.
Following is a simple example of const function.

```
#include<iostream>

using namespace std;


class Test {

    int value;

public:

    Test(int v = 0) {value = v;}


    // We get compiler error if we add a line like "value = 100;"

    // in this function.

    int getValue() const {return value;}

};


int main() {

    Test t(20);

    cout<<t.getValue();

    return 0;

}
```

Run on IDE

Output:

```
20
```

When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects.

For example the following program has compiler errors.

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


class Test {
   int value;
public:
   Test(int v = 0) {value = v;}
   int getValue() {return value;}
};


int main() {
   const Test t;
   cout << t.getValue();
   return 0;
}
```

Run on IDE

Output:

```
 passing 'const Test' as 'this' argument of 'int
Test::getValue()' discards qualifiers
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

C++ Tokens

C++ Tokens are the smallest individual units of a program.

Following are the C++ tokens : (most of c++ tokens are basically similar to the C tokens)

- Keywords
- Identifiers
- Constants
- Variables
- Operators

Keywords

The reserved words of C++ may be conveniently placed into several groups. In the first group we put those that were also present in the C programming language and have been carried over into C++. There are 32 of these, and here they are:

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

Identifiers

Identifiers refers to the name of variables, functions, arrays, classes, etc. created by the user. Identifiers are the fundamental requirement of any language.

Identifier naming conventions

- Only alphabetic characters, digits and underscores are permitted.
- First letter must be an alphabet or underscore (_).
- Identifiers are case sensitive.
- Reserved keywords can not be used as an identifier's name.

Constants

Constants refers to fixed values that do not change during the execution of a program.

Declaration of a constant :

const [data_type] [constant_name]=[value];

Consider the example

```
#include <iostream.h>
int main()
{
        const int  max_length=100;            //          integer constant
        const char choice='Y';                //          character constant
        const char title[]="www.includehelp.com";   //          string constant
        const float temp=12.34;               // float constant

        cout<<"max_length :"<<max_length<<endl;
        cout<<"choice :"<<choice<<endl;
        cout<<"title :"<<title<<endl;
        cout<<"temp :"<<temp<<endl;
        return 0;
}
```

Output

```
max_length :100
choice :Y
title :www.includehelp.com
temp :12.34
```

Variable

A variable is a meaningful name of data storage location in computer memory. When using a variable you refer to memory address of computer.

We know that in C, all variables must be declared before they are used, this is true with C++.

The main difference in C and C++ with regards to the place of their declaration in the program...

C requires all the variables to be defined in the beginning of scope.

C++ allows the declaration of a variable anywhere in the scope, this means that a variable can be declared right at the place of its first use.

> **Syntax to declare a variable :**
>
> [data_type] [variable_name];

Consider the example

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

```
int main()
{
        int a,b;
        cout<<" Enter first number :";
        cin>>a;
        cout<<" Enter second number:";
        cin>>b;

        int sum;  // declaration
        /*this type of declaration will not allow in C*/
        sum=a+b;
        cout<<" Sum is : "<<sum <<"\n";
        return 0;
}
```

Output

```
Enter first number :55
Enter second number:15
Sum is : 70
```

Operators in C / C++

We can define operators as symbols that helps us to perform specific mathematical and logical computations on operands. In other words we can say that an operator operates the operands. For example, consider the below statement:

```
c = a + b;
```

Here, '+' is the operator known as *addition operator* and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'.

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language −

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200

/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	**Increment operator**, increases integer value by one	A++ will give 11
--	**Decrement operator**, decreases integer value by one	A-- will give 9

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.

<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then −

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows −

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

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

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

Show Examples

Operator	Description	Example

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

Assignment Operators

There are following assignment operators supported by C++ language −

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C

+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

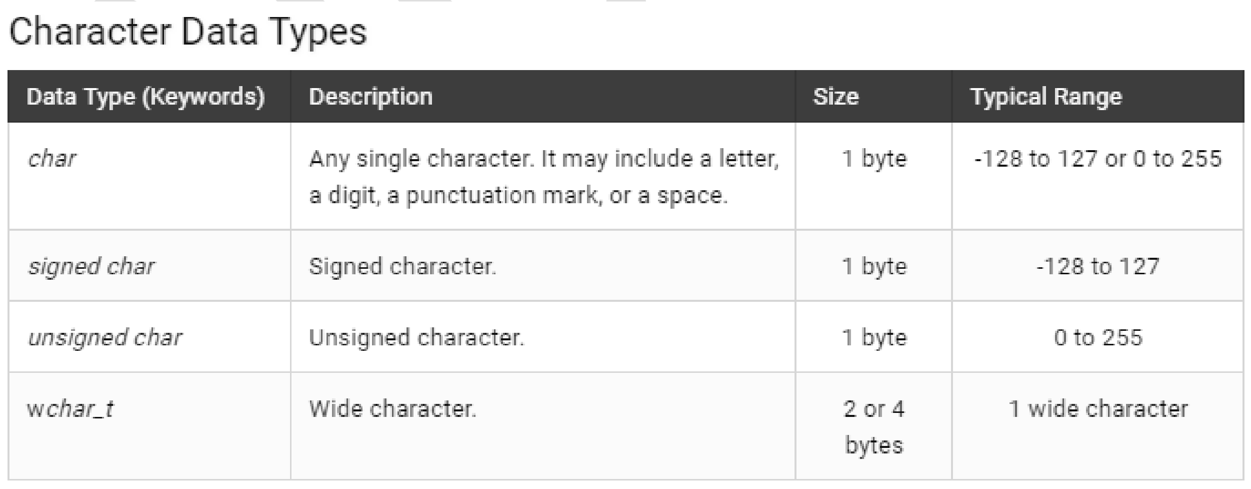
Misc Operators

The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	**sizeof** sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast** Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	***** Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

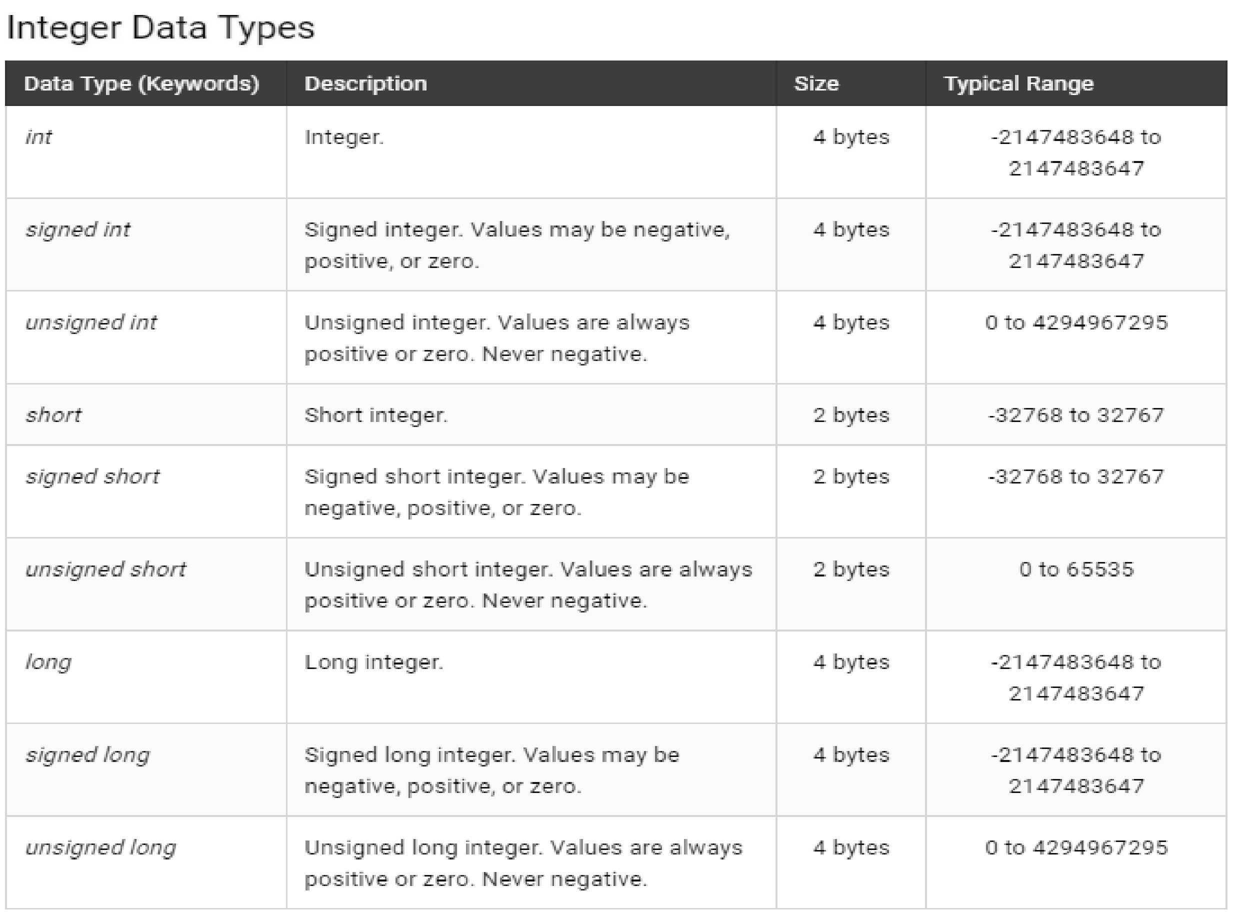
Data Types available in C++:

1. Primary(Built-in) Data Types:
   - character
   - integer
   - floating point
   - boolean
   - double floating point
   - void
   - wide character
2. User Defined Data Types:
   - Structure
   - Union
   - Class
   - Enumeration
3. Derived Data Types:
   - Array
   - Function
   - Pointer
   - Reference



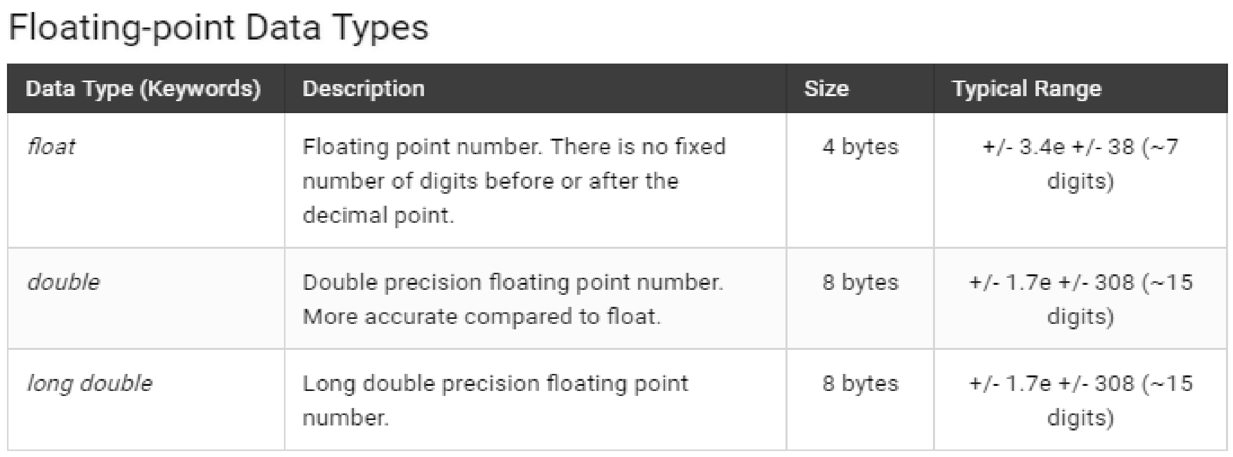
Character Data Types

Data Type (Keywords)	Description	Size	Typical Range
char	Any single character. It may include a letter, a digit, a punctuation mark, or a space.	1 byte	-128 to 127 or 0 to 255
signed char	Signed character.	1 byte	-128 to 127
unsigned char	Unsigned character.	1 byte	0 to 255
wchar_t	Wide character.	2 or 4 bytes	1 wide character



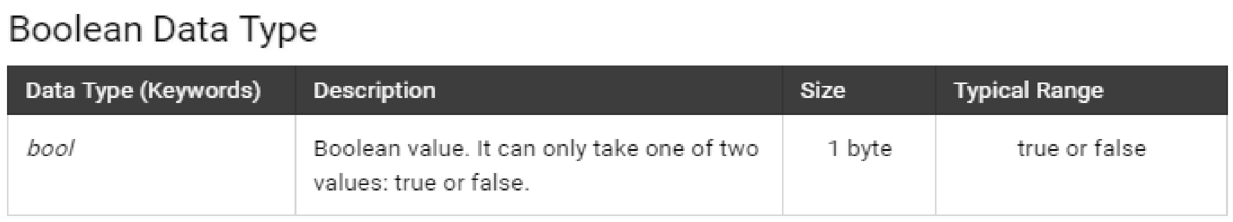
Integer Data Types

Data Type (Keywords)	Description	Size	Typical Range
int	Integer.	4 bytes	-2147483648 to 2147483647
signed int	Signed integer. Values may be negative, positive, or zero.	4 bytes	-2147483648 to 2147483647
unsigned int	Unsigned integer. Values are always positive or zero. Never negative.	4 bytes	0 to 4294967295
short	Short integer.	2 bytes	-32768 to 32767
signed short	Signed short integer. Values may be negative, positive, or zero.	2 bytes	-32768 to 32767
unsigned short	Unsigned short integer. Values are always positive or zero. Never negative.	2 bytes	0 to 65535
long	Long integer.	4 bytes	-2147483648 to 2147483647
signed long	Signed long integer. Values may be negative, positive, or zero.	4 bytes	-2147483648 to 2147483647
unsigned long	Unsigned long integer. Values are always positive or zero. Never negative.	4 bytes	0 to 4294967295



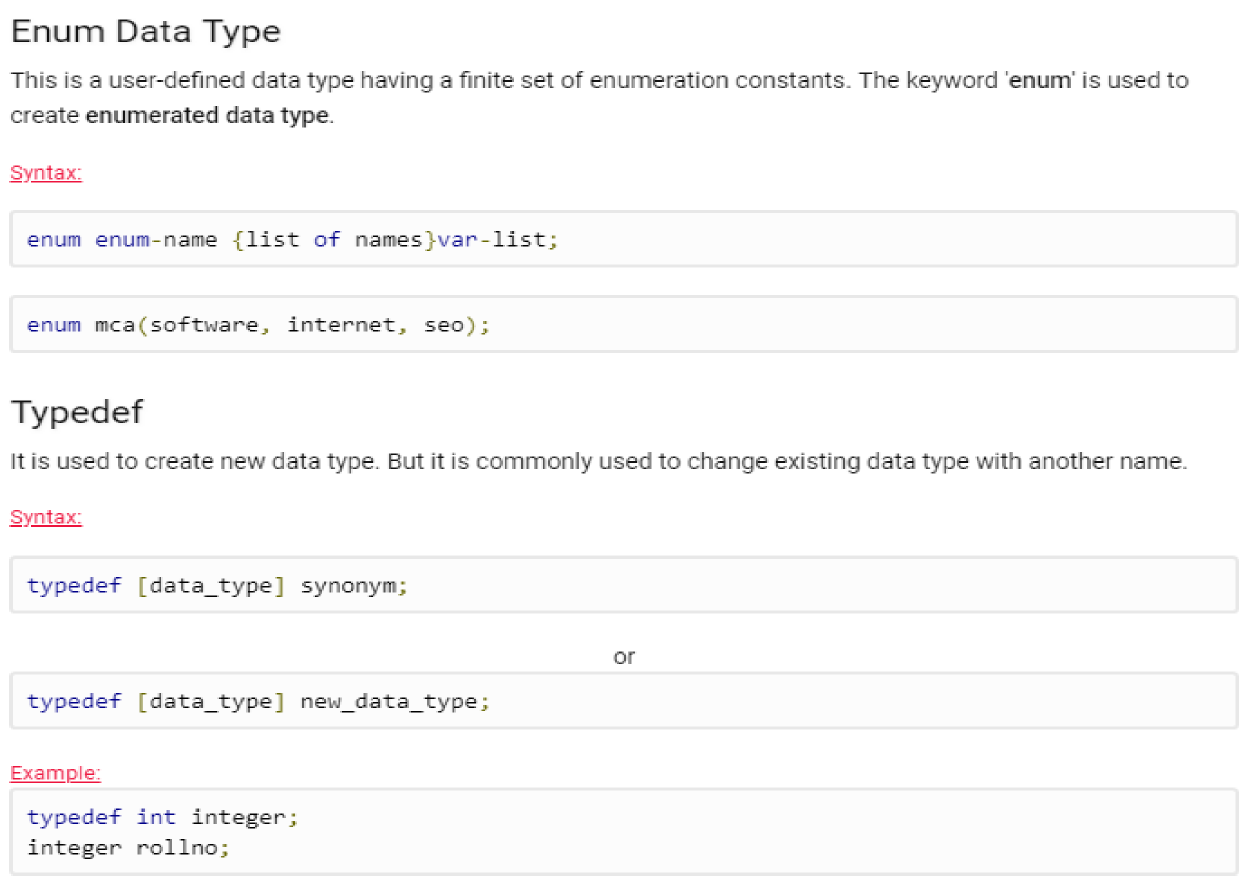
Floating-point Data Types

Data Type (Keywords)	Description	Size	Typical Range
float	Floating point number. There is no fixed number of digits before or after the decimal point.	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number. More accurate compared to float.	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8 bytes	+/- 1.7e +/- 308 (~15 digits)



Boolean Data Type

Data Type (Keywords)	Description	Size	Typical Range
bool	Boolean value. It can only take one of two values: true or false.	1 byte	true or false



Enum Data Type

This is a user-defined data type having a finite set of enumeration constants. The keyword '**enum**' is used to create **enumerated data type**.

Syntax:

```
enum enum-name {list of names}var-list;
```

```
enum mca(software, internet, seo);
```

Typedef

It is used to create new data type. But it is commonly used to change existing data type with another name.

Syntax:

```
typedef [data_type] synonym;
```

or

```
typedef [data_type] new_data_type;
```

Example:

```
typedef int integer;
integer rollno;
```

A loop statement allows us to execute a statement or group of statements multiple times

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

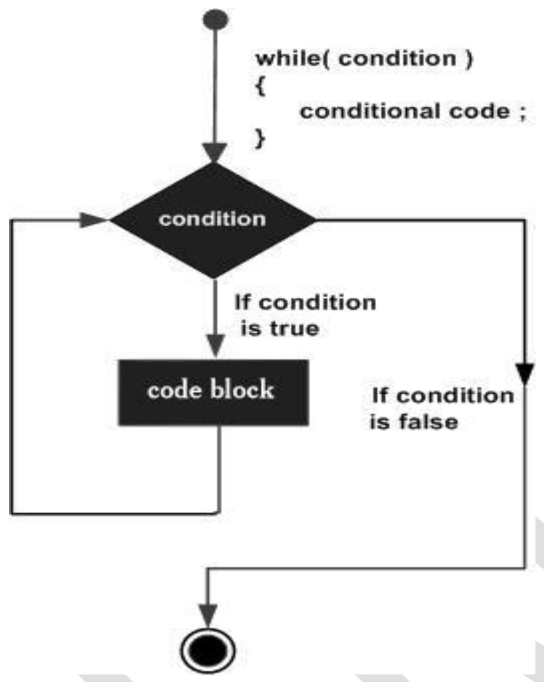
The syntax of a while loop in C++ is −

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

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

Live Demo

```
#include <iostream>

using namespace std;


int main () {

  // Local variable declaration:

  int a = 10;


  // while loop execution
  while( a < 20 ) {
     cout << "value of a: " << a << endl;
     a++;
  }

  return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a for loop in C++ is −

```
for ( init; condition; increment ) {
  statement(s);
}
```

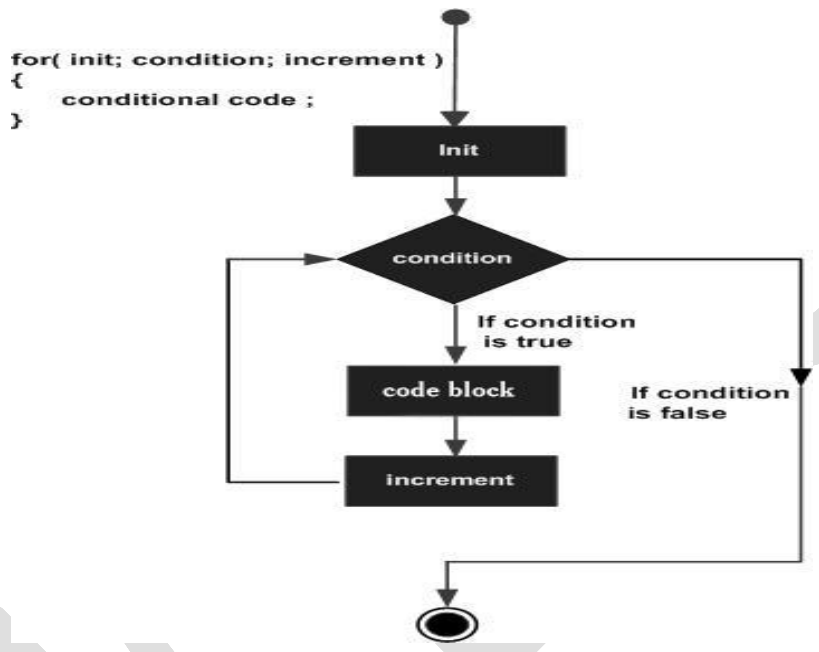
Here is the flow of control in a for loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement can be left blank, as long as a semicolon appears after the

condition.

- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram



Example

```
#include <iostream>

using namespace std;


int main () {
   // for loop execution
   for( int a = 10; a < 20; a = a + 1 ) {
      cout << "value of a: " << a << endl;
   }
```

```
  return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

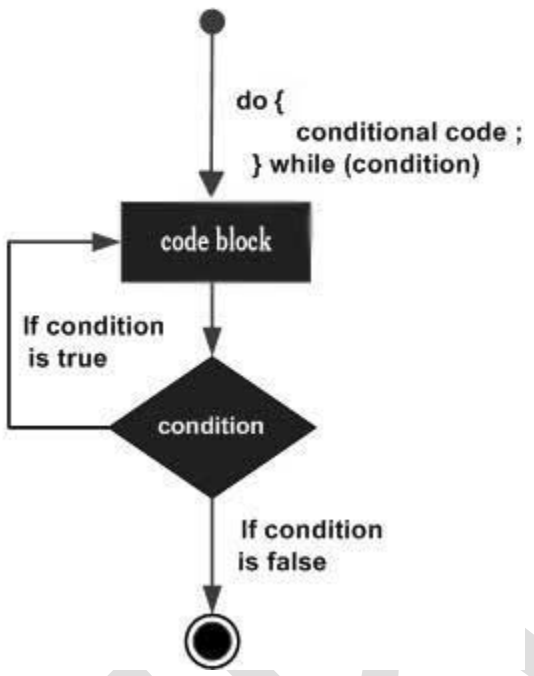
The syntax of a do...while loop in C++ is −

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram



Example

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

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

  // do loop execution
  do {
    cout << "value of a: " << a << endl;
    a = a + 1;
  } while( a < 20 );

  return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

MODULE II

Objects -pointers and objects -constant objects -nested classes-local classes-Constructors -default constructor -Parameterized constructors -Constructor with dynamic allocation -copy constructor -destructors.

What is Class?

A class is a user defined data type that allows us to bind data and its associated functions together as a single unit. Thus class provides the facility of data encapsulation.
Class provides the facility of data hiding using the concept of visibility mode such as public, private and protected.
Once a class is defined we can create an object of the class to access variables and functions defined inside the class.

Syntax:

```
class Class_Name
{
Private:
Data-Type Variable_Name;
Function declaration or Function Definition;
Public:
Data-Type Variable_Name;
Function declaration or Function Definition;
};
```

Class can be created using the **class keyword**. The class definition starts with curly bracket and ends with curly bracket followed by semicolon.
We can declare variables as well as functions inside the curly bracket as shown in the syntax. The variables defined inside class are known as **data member** and the function declared inside the class are known as **member function**.
In order to provide **data hiding** facility class provides the concept of **visibility mode** such as **private, public or protected**. If you don't specify any visibility mode for the member of the class then **by default all the members of the class**

are considered as private.
The data member and member function declared as a public can be accessed directly using the object of the class. But the data member and member function declared as private cannot be accessed directly using the object of the class.

Example:

```
class test
{
int a, b;
public:
void input ();
{
cout<<"Enter Value of a and b";
cin>>a>>b;
}
void output ()
{
cout<<"A="<<a<<"B="<<b;
}
};
```

Create Object in C++: There are two different methods for creating objects of the class:

(1) We can create object at the time of specifying a class after the closing curly bracket.

Example:

```
Class test
{
int a,b;
public:
void input ();
void ouput ();
}t1,t2,t3;
```

Here, t1, t2 and t3 are the objects of class test.

(2) We can create object inside the main function using name of the class.

The general syntax for creating object inside main function is as below:

Class_Name Object_Name;

Example:

Test t1, t2, t3;

Here, t1, t2 and t3 are the objects of class test.

How to Access Class Members?

The data member and member function declared as a public can be accessed directly using the object of the class. But the data member and member function declared as private cannot be accessed directly using the object of the class.

We can access private member of the class using public member of the class.

The general syntax for accessing public member using object is given below:

Object_Name. Data_Member = value;

Object_Name. Member_Function (Argument_List);

Example:

T1. input ();

T1. output ();

Example:

```
class Test
{
int b;
Public:
int a;
void inputb()
{
b=20;
}
};
int main()
```

```
{
Test t1;
t1.a=10; //works because a is public
t1.b=20; //error because b is private
t1.inputb (); //works because inputb () is public
return 0;
}
```

POINTERS AND OBJECTS:

pointers

Similar to C, in C++, variables are used to hold data values during program execution. When declared, every variable occupies certain memory locations. It is possible to access and display the address of the memory location of a variable using '&' operator. Memory is arranged in series of bytes. These series of bytes are numbered from zero onward. The number specified to a cell is known as memory address. A pointer variable stores the memory address of any type of variable. The pointer variable and normal variable should be of the same type. The pointer is denoted by

A byte is nothing but a combination of eight bits, as shown in Figure. The binary numbers 0 and 1 are known as bits. Each byte in the memory is specified with a unique (matchless) memory address. The memory address is an unsigned integer starting from zero to uppermost addressing capacity of the microprocessor. The number of memory locations pointed by a pointer depends on the type of the pointer. The programmer should not worry about the addressing procedure of the variables. The compiler takes care of this. The pointers are either 16 bits or 32 bits long.

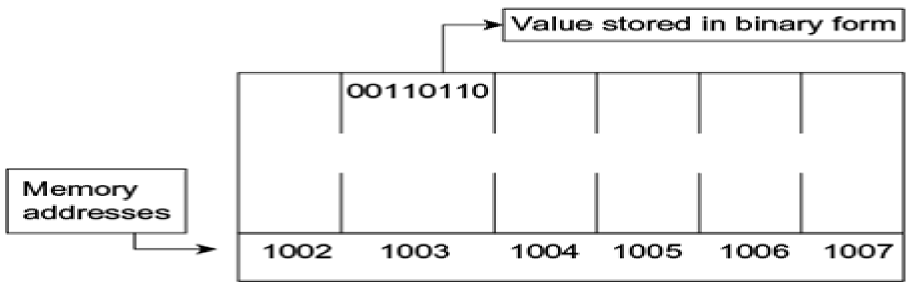


Fig: Memory representation

The allocation of memory during program run time is called dynamic memory allocation. Such a type of memory allocation is essential for data structure, and it is efficiently handled by pointers. Arrays are another reason for using pointers. Arrays are used to store more values. Actually, the name of the array is a pointer. Command-line arguments are one more reason for using pointers. These arguments are passed to programs and are stored in an array of pointers argv [].

POINTERS

A pointer is a memory variable that stores a memory address. Pointers can have any name that is legal for other variables, and it is declared in the same fashion as other variables, but it is always denoted by '*' operator.

Features Of Pointers

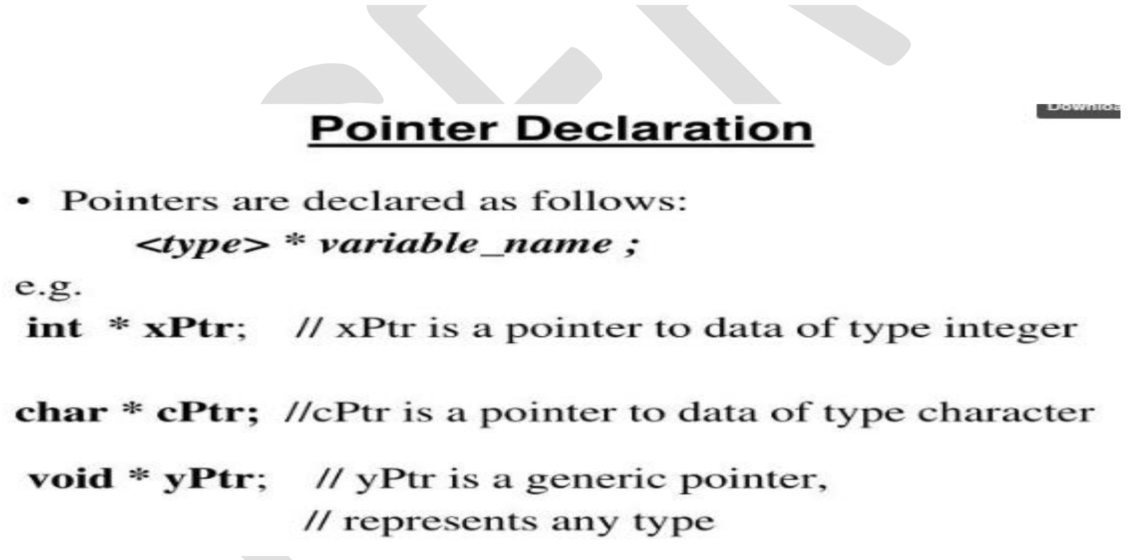
1. Pointers save memory space.
2. Execution time with pointers is faster, because data are manipulated with the address, that is, direct access to memory location.
3. Memory is accessed efficiently with the pointers. The pointer assigns as well as releases the memory space. Memory is dynamically allocated.
4. Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
5. We can access the elements of any type of array, irrespective of its subscript range.
6. Pointers are used for file handling.
7. Pointers are used to allocate memory in a dynamic manner.
8. In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class. The compiler will generate an error message "cannot convert 'A* to B*,'" where A is the base class and B is the derived class.

Pointer Declaration

Pointer variables can be declared as follows:

```
Example
int *x;
float *f;
char *y;
```

1. In the first statement, 'x' is an integer pointer, and it informs the compiler that it holds the address of any integer variable. In the same way, 'f' is a float pointer that stores the address of any float variable, and 'y' is a character pointer which stores the address of any character variable.
2. The indirection operator (*) is also called the dereference operator. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.



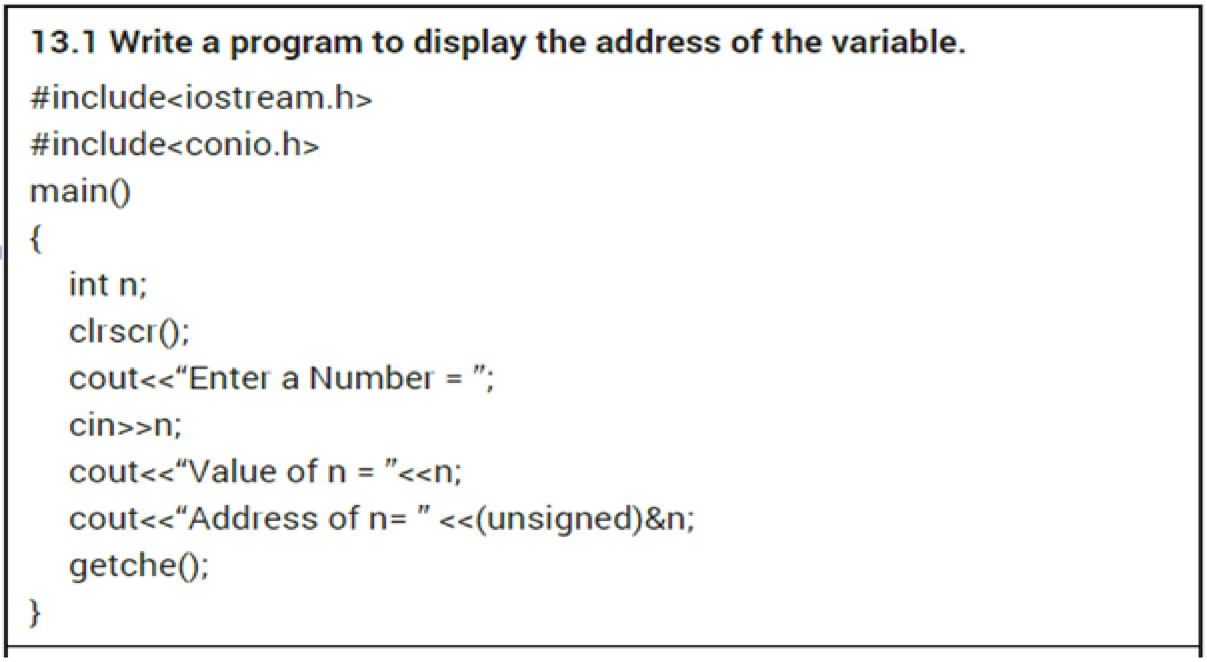
Pointer Declaration

- Pointers are declared as follows:

  <type> * variable_name ;

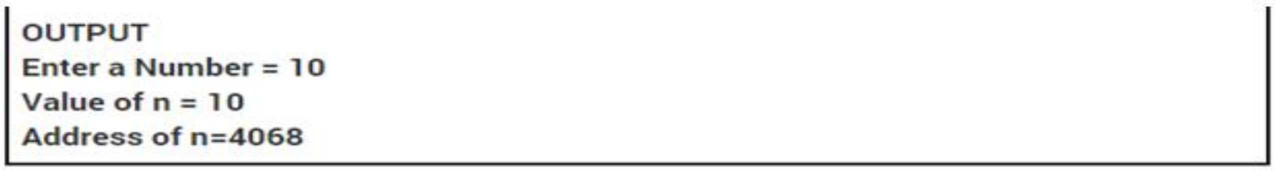
e.g.

```
int  * xPtr;    // xPtr is a pointer to data of type integer

char * cPtr;  //cPtr is a pointer to data of type character

void * yPtr;    // yPtr is a generic pointer,
                // represents any type
```

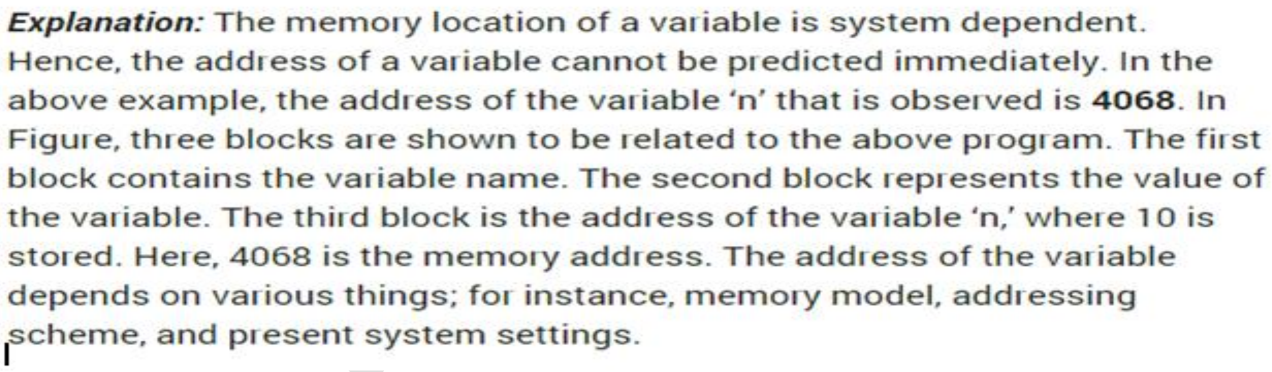


13.1 Write a program to display the address of the variable.

```
#include<iostream.h>
#include<conio.h>
main()
{
   int n;
   clrscr();
   cout<<"Enter a Number = ";
   cin>>n;
   cout<<"Value of n = "<<n;
   cout<<"Address of n= " <<(unsigned)&n;
   getche();
}
```



```
OUTPUT
Enter a Number = 10
Value of n = 10
Address of n=4068
```



Explanation: The memory location of a variable is system dependent. Hence, the address of a variable cannot be predicted immediately. In the above example, the address of the variable 'n' that is observed is **4068**. In Figure, three blocks are shown to be related to the above program. The first block contains the variable name. The second block represents the value of the variable. The third block is the address of the variable 'n,' where 10 is stored. Here, 4068 is the memory address. The address of the variable depends on various things; for instance, memory model, addressing scheme, and present system settings.

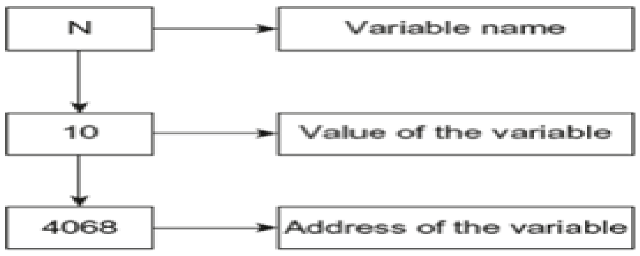


Fig: Variable and its memory address

C++ allows you to have pointers to objects. The pointers pointing to objects are referred to as Object Pointers.

C++ Declaration and Use of Object Pointers

Just like other pointers, the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :

class-name ∗ object-pointer ;

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type. For example, to declare ptr as an object pointer of Sample class type, we shall write

Sample ∗optr ;

where Sample is already defined class. When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

Pointer to Object

In C++ you can declare a pointer that contains the address of the object of type class.
Suppose we have created a class named base as shown below:

```
class Base
{
public:
int x;
void display ()
{
cout<<"X="<<x<<endl;
}
};
```

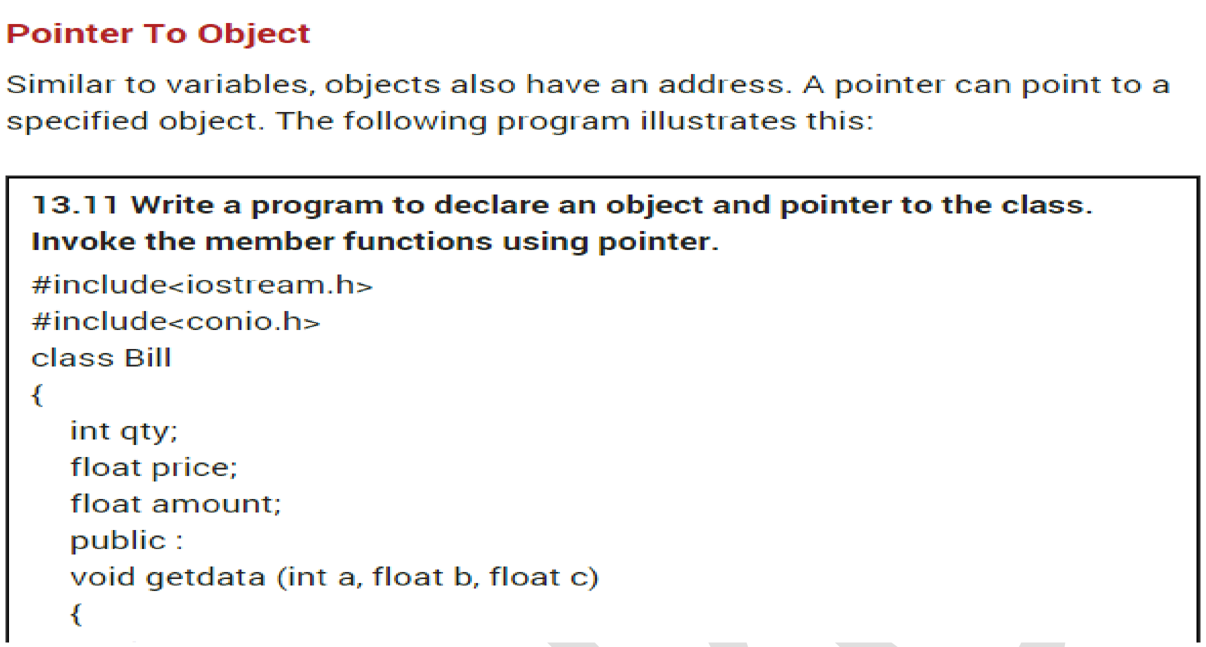
Now you can declare a pointer that contains the address of the object of class base as shown below:

Base *ptr; // declare a pointer of base class
Base B1; // declare an object of base class
Ptr = &B1; // assign address of object b1 to base class pointer
Using this pointer you can access the public member of the base class as shown below:
ptr->x = 10;
ptr->display ();

```
#include <iostream.h>
class Base
{
public:
int x;
void display ()
{
cout<<"X="<<x<<endl;
}
};
int main ()
{
Base B1;
Base *ptr;
ptr = &B1;
ptr->x = 10;
ptr->display();
}
```

Output:

```
X= 10
```

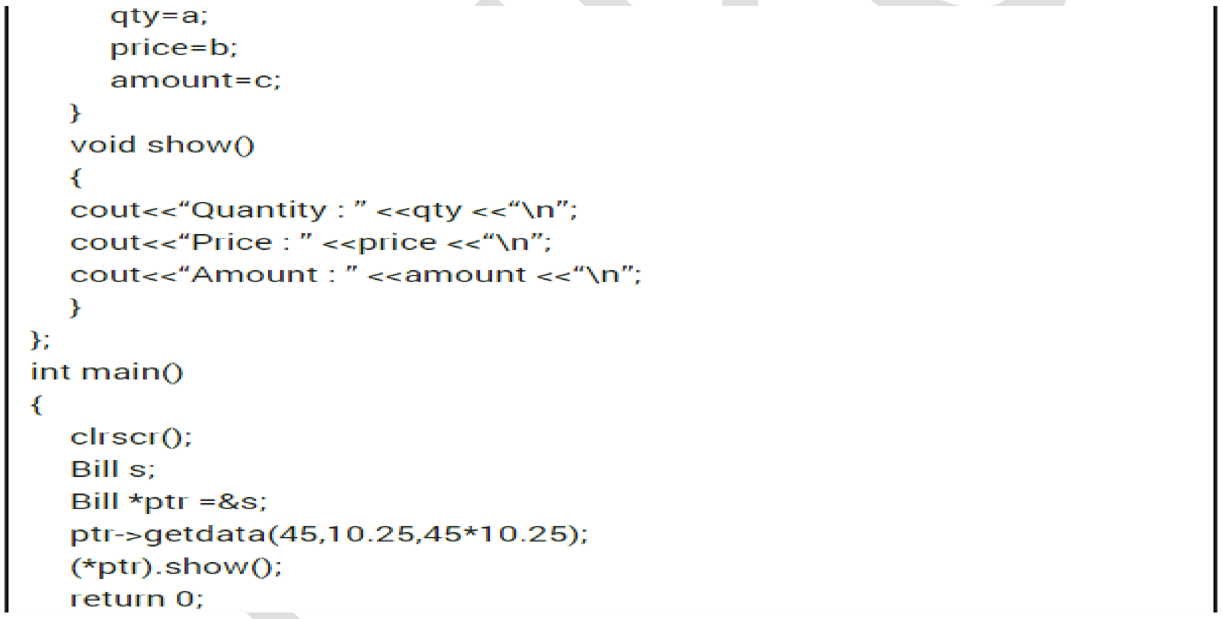


Pointer To Object

Similar to variables, objects also have an address. A pointer can point to a specified object. The following program illustrates this:

13.11 Write a program to declare an object and pointer to the class. Invoke the member functions using pointer.

```
#include<iostream.h>
#include<conio.h>
class Bill
{
    int qty;
    float price;
    float amount;
    public :
    void getdata (int a, float b, float c)
    {
```



```
        qty=a;
        price=b;
        amount=c;
    }
    void show()
    {
    cout<<"Quantity : " <<qty <<"\n";
    cout<<"Price : " <<price <<"\n";
    cout<<"Amount : " <<amount <<"\n";
    }
};
int main()
{
    clrscr();
    Bill s;
    Bill *ptr =&s;
    ptr->getdata(45,10.25,45*10.25);
    (*ptr).show();
    return 0;
```

```
}
```

OUTPUT

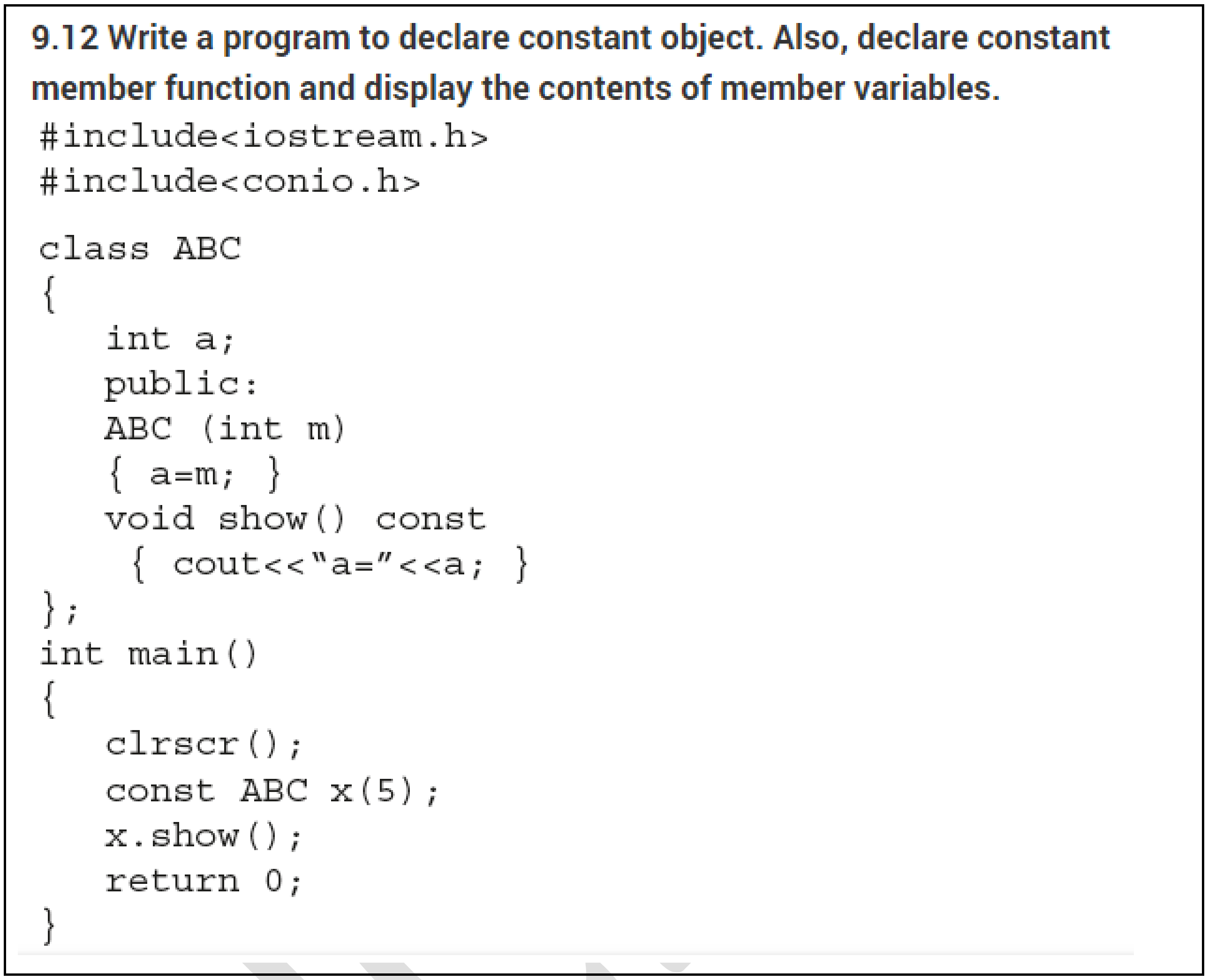
```
Quantity : 45
Price : 10.25
Amount : 461.25
```

Explanation: In the above program, the class Bill contains two float and one int members. The class Bill also contains the member function getdata() and show() to read and display the data. In function main(), s is an object of class Bill, and ptr is a pointer of the same class. The address of object s is assigned to pointer ptr. Using pointer ptr with arrow operator (->) and dot operator (.), members and functions are invoked. The statements used for invoking functions are as given below.

```
ptr->getdata (45,10.25,45*10.25);
(*ptr).show();
```

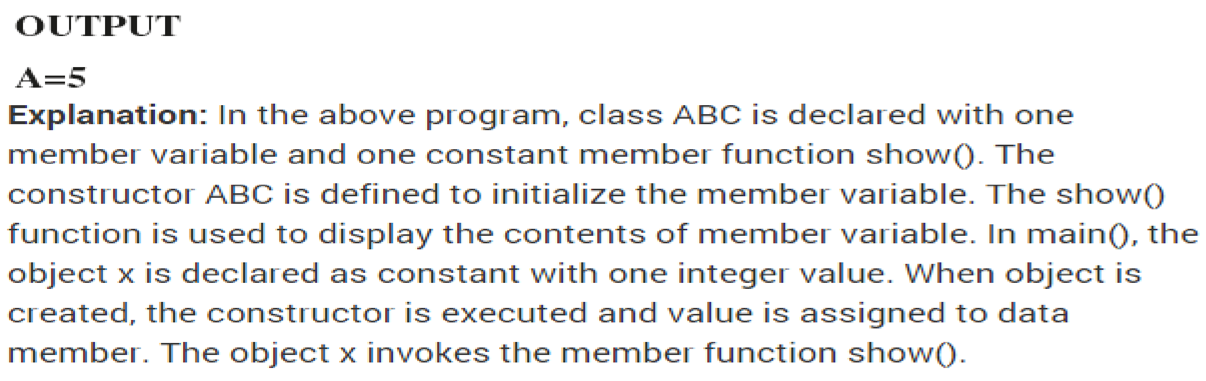
The const Objects

In the previous chapter, we have studied the constant functions. The const declared functions do not allow the operations that alter the values. In the same fashion, we can also make the object constant by the keyword const. Only constructor can initialize data member variables of constant object. The data member of constant objects can be read-only and any effort to alter values of variables will generate an error. The data members of constant object are also called read-only data members. The constant object can access only constant functions. If constant object tries to invoke a non-member function, an error message will be displayed.



9.12 Write a program to declare constant object. Also, declare constant member function and display the contents of member variables.

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

class ABC
{
   int a;
   public:
   ABC (int m)
   { a=m; }
   void show() const
    { cout<<"a="<<a; }
};
int main()
{
   clrscr();
   const ABC x(5);
   x.show();
   return 0;
}
```



OUTPUT

```
A=5
```

Explanation: In the above program, class ABC is declared with one member variable and one constant member function show(). The constructor ABC is defined to initialize the member variable. The show() function is used to display the contents of member variable. In main(), the object x is declared as constant with one integer value. When object is created, the constructor is executed and value is assigned to data member. The object x invokes the member function show().

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 {

      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()

{   }
```

LOCAL CLASS AND NESTED CLASS

A class which is declared inside a function is called a local class. A local class is accessible only within the function it is declared. Following guidelines should be followed while using local classes:

- Local classes can access global variables only along with scope resolution operator.
- Local classes can access static variables declared inside a function.
- Local classes cannot access auto variables declared inside a function.
- Local classes cannot have static variables.
- Member functions must be defined inside the class.
- Private members of the class cannot be accessed by the enclosing function if it is not declared as a friend function.

Below example demonstrates a local class:

```
#include<iostream>

using namespace std;

const float PI = 3.1415;

int main()

{

class Circle

{public:

int r;

void area()

{cout<<"Area of circle is: "<<(::PI*r*r);

}

void set_radius(int radius)

{

r = radius;

}};

Circle c;

c.set_radius(10);

c.area();

return 0;}
```

Output of the above program is as follows:

```
Area of circle is: 314.15
```

C++ allows programmers to declare one class inside another class. Such classes are called nested classes. When a class B is declared inside class A, class B cannot access the members of class A. But class A can access the members of class B through an object of class B. Following are the properties of a nested class:

- A nested class is declared inside another class.
- The scope of inner class is restricted by the outer class.
- While declaring an object of inner class, the name of the inner class must be preceded by the

outer class name and scope resolution operator.

Constructor in C++

Constructor is a member function of the class.
It is called special member function because the name of the constructor is same as name of the class.
Constructor is used to construct the values for the data member of the class automatically when the object of class is created.
Like other member function there is no need to call constructor explicitly. It is invoked automatically each time the object of its class is created.
Every class having at least one constructor defined in it. If you do not define any constructor in the class then compiler will automatically create a constructor inside the class and assigns default value (0) to the data member of the class.

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object.

```
class A
{
 int x;
 public:
 A();  //Constructor
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
 int i;
 public:
 A(); //Constructor declared
};

A::A()  // Constructor definition
{
 i=1;
}
```

Types of Constructors

Constructors are of three types :

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

```
class_name ()
{ Constructor Definition }
```

Example :

```
class Cube
{
int side;
public:
Cube()
 {
 side=10;
 }
};

int main()
{
Cube c;
cout << c.side;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 public:
 int side;
};

int main()
{
 Cube c;
 cout << c.side;
}
```

Output : 0

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

Example :

```
class Cube
{
 public:
 int side;
 Cube(int x)
  {
   side=x;
  }
};

int main()
{
 Cube c1(10);
 Cube c2(20);
 Cube c3(30);
 cout << c1.side;

 cout << c2.side;

 cout << c3.side;

}
```

OUTPUT : 10 20 30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

Copy Constructor

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.

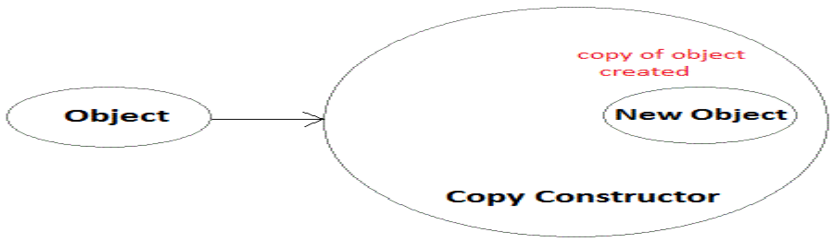
Copy Constructor in C++

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name.he compiler provides a default Copy Constructor to all the classes.

Syntax of Copy Constructor

```
Classname(const classname & objectname)
{
. . . .
}
```

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.



Below is a sample program on Copy Constructor:

```
#include<iostream>
using namespace std;
class Samplecopyconstructor
{
private:
int x, y; // data members
public:
Samplecopyconstructor(int x1, int y1)
{
x = x1;
y = y1;
}
// Copy constructor
Samplecopyconstructor (const Samplecopyconstructor &sam)
{
x = sam.x;
y = sam.y;
}
void display()
{
cout<<x<<" "<<y<<endl;
}
};
int main()
{
Samplecopyconstructor obj1(10, 15); // Normal constructor
Samplecopyconstructor obj2 = obj1; // Copy constructor
cout<<"Normal constructor : ";
obj1.display();
cout<<"Copy constructor : ";
obj2.display();
return 0;
}
```

Output:

```
Normal constructor : 10 15
```

```
Copy constructor : 10 15
```

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.

2. When an object of the class is passed (to a function) by value as an argument.

3. When an object is constructed based on another object of the same class.

4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the return value optimization (sometimes referred to as RVO).

When is user defined copy constructor needed?

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection..etc.

Constructor Overloading

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

```
class Student

{

 int rollno;
 string name;
 public:
 Student(int x)
 {
  rollno=x;
  name="None";
 }
 Student(int x, string str)
 {
  rollno=x ;
  name=str ;
 }
};

int main()
{
 Student A(10);
 Student B(11,"Ram");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write `Student S;` in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

Destructors

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** `~` sign as prefix to it.

```
class A
{
 public:
```

```
 ~A();
};
```

Destructors will never have any arguments.

Example to see how Constructor and Destructor is called

```
class A
{
A()
 {
  cout << "Constructor called";
 }

~A()
 {
  cout << "Destructor called";
 }
};

int main()
{
 A obj1;   // Constructor Called
 int x=1
 if(x)
  {
   A obj2;  // Constructor Called
  }   // Destructor Called for obj2
} //  Destructor called for obj1
```

Single Definition for both Default and Parameterized Constructor

In this example we will use **default argument** to have a single definition for both defualt and parameterized constructor.

```
class Dual
{
 int a;
 public:
 Dual(int x=0)
  {
   a=x;
  }
};

int main()
{
 Dual obj1;
 Dual obj2(10);
}
```

Here, in this program, a single Constructor definition will take care for both these object initializations. We don't need separate default and parameterized constructors.

The constructor that does not accept any argument is known as default constructor.
Following are important characteristics of Constructor:
(1) It is called automatically when object of its class is created.
(2) It does not return any value.
(3) It must be defined inside public section of the class.
(4) It can have default arguments.
(5) Inheritance of constructor is not possible.
(6) It can not virtual.
Disadvantage of default constructor is that each time an object is created it will assign same default values to the data members of the class. It is not possible to *assign different values to the data members for the different object of the class using default constructor.*

Parameterized Constructor

Disadvantage of default constructor is that each time an object is created it will assign same default values to the data members of the class.
However sometimes it is required to assign different values to the data members for the different object of the class. This can be accomplished using the concept of parameterized constructor.
The constructor that accepts parameters as an argument is called ***Parameterized constructor****.*

Parameterized Constructor can be defined inside class as shown below: Class Rectangle

```
{
int Height, Width;
public:
Rectangle (int h, int w)
{
Height = h;
Width = w;
}
}
```

Parameterized Constructor can be defined outside class as shown below:

```
Class Rectangle
{
int Height, Width;
public:
Rectangle (int h, int w);
}
Rectangle :: Rectangle (int h, int w)
{
Height = h;
Width = w;
}
```

Now when you create an object of the class Rectangle as shown below:
Rectangle R1;
It will not invoke parameterized constructor but it will invoke default constructor and assigns default value 0 to its data member Height and Width.
In order to invoke parameterized constructor we need to pass arguments while creating object. We can pass arguments using two different methods:
(1) Implicit:
Rectangle R1 (10, 20);
It will assign value of 10 to Height and value of 20 to Width.
(2) Explicit:
Rectangle R2 = Rectangle (30, 40);
It will assign value of 30 to Height and value of 40 to Width.

Copy Constructor

Sometimes it is required to copy the value of data member of one object into another object while creating it. This can be accomplished using the concept of copy constructor.
The constructor that accepts reference to the object as an argument is known as copy constructor.
Copy Constructor can be defined inside class as shown below:

```
Class Rectangle
{
int Height, Width;
public:
Rectangle (Rectangle &r)
{
Height = r.Height;
Width = r.Width;
}
}
```

Copy Constructor can be defined outside class as shown below:

```
Class Rectangle
{
int Height, Width;
public:
Rectangle (Rectangle &r);
}
Rectangle :: Rectangle (Rectangle &r)
{
Height = r.Height;
Width = r.Width;
}
```

In order to invoke copy constructor we need to pass object as an arguments while creating object. We can pass arguments using two different methods:
(1) Implicit:
Rectangle R2 (R1);
It will assign value of data member of object R1 into data member of object R2.
(2) Explicit:
Rectangle R2 = Rectangle (R1);
It will assign value of data member of object R1 into data member of object R2.

Constructor with dynamic allocation

Dynamically allocated array in the constructor.

Constructors can be used to initialize member objects as well as allocate memory. This can allow an object to use only that amount of memory that is required immediately. This memory allocation at run-time is also known as dynamic memory allocation. The *new* operator is used for this purpose.

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

Live Demo

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


class Box {
  public:
    Box() {
      cout << "Constructor called!" <<endl;
    }
    ~Box() {
      cout << "Destructor called!" <<endl;
    }
};
int main() {
  Box *myBoxArray = new Box[4];
  delete [] myBoxArray; // Delete array
  return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

MODULE III

Operator overloading - overloading through friend functions - overloading the assignment operator - type conversion - explicit constructor.

Operator Overloading in C++

Operator overloading is a type of polymorphism in which a single operator is overloaded to give user defined meaning to it. Operator overloading provides a flexibility option for creating new definitions of C++ operators. There are some C++ operators which we can't overload.

The lists of such operators are:

- Class member access operator (. (dot), .* (dot-asterisk))
- Scope resolution operator (::)
- Conditional Operator (?:)
- Size Operator (sizeof)

  - These are the lists of few excluded operators and are very few when compared to large sets of operators which can be used for the concept of operator overloading. An overloaded operator is used to perform an operation on the user-defined data type. Let us take an example of the addition operator (+) operator has been overloaded to perform addition on various variable types, like for integer, floating point, String (concatenation) etc.
  - syntax:

```
• return type className :: operator op (arg_list)
• {
•  //Function body;
• }
```

  - Here, the return type is the type of value returned by the specified operation and op is the operator being overloaded.

In C++, we can make operators to work for user defined classes. 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.

Write the rules for overloading the operators.

o Only existing operators can be overloaded. New operators cannot be created.

o The overloaded operator must have atleast one operand that is of user defined type.

o The basic meaning of an operator cannot be changed. That is the plus operator cannot be used to subtract one value from the other.

o Overloaded operator follow the syntax rules of the original operators. They cannot be overridden.

o There are some operators that cannot be overloaded. They are Size of, . ,: :,?:.

o Friend function cannot be used to overload certain operators (= ,( ) ,[ ] ,->).However member functions can be used to overload them.

o Unary operators, overload by means of a member function, take no explicit arguments and return no explicit values,but,those overloaded by means of a friend function, take one reference argument.

o Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

o When using binary operator overloaded through a member function, the left hand operand must be an object of the relevant class.

o Binary arithmetic operators such as +,-,*,and / must explicitly return a value. They must not attempt to change their own arguments.

A simple and complete example

```
#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 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
```

UNARY OPERATOR OVERLOADING

An Unary operator is an operator that operates on a single operand and returns a new value.

To write a program to find the complex numbers using unary operator overloading.

Unary operators:

- Increment (++) Unary operator.
- Decrement (--) Unary operator.
- The minus (-) unary.
- The logical not (!) operator.

Unary Operator Overloading Algorithm/Steps:

- Step 1: Start the program.
- Step 2: Declare the class.
- Step 3: Declare the variables and its member function.
- Step 4: Using the function getvalue() to get the two numbers.
- Step 5: Define the function operator ++ to increment the values

- Step 6: Define the function operator - -to decrement the values.
- Step 7: Define the display function.
- Step 8: Declare the class object.
- Step 9: Call the function getvalue()
- Step 10: Call the function operator ++() by incrementing the class object and call the function display.
- Step 11: Call the function operator - -() by decrementing the class object and call the function display.
- Step 12: Stop the program.

Simple Program for Unary Operator Overloading Program

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

class complex {
   int a, b, c;
public:

   complex() {
   }

   void getvalue() {
      cout << "Enter the Two Numbers:";
      cin >> a>>b;
   }

   void operator++() {
      a = ++a;
```

```
        b = ++b;
    }

    void operator--() {
        a = --a;
        b = --b;
    }

    void display() {
        cout << a << "+\t" << b << "i" << endl;
    }
};

void main() {
```

```
    clrscr();
    complex obj;
    obj.getvalue();
    obj++;
    cout << "Increment Complex Number\n";
    obj.display();
    obj--;
    cout << "Decrement Complex Number\n";
    obj.display();
    getch();
}
```

Sample Output

```
Enter the two numbers: 3 6
Increment Complex Number
4 +             7i
Decrement Complex Number
```

```
3 +           6i
```

BINARY OPERATOR OVERLOADING

To write a program to add two complex numbers using binary operator overloading.

Binary Operator Overloading Algorithm/Steps:

- Step 1: Start the program.
- Step 2: Declare the class.
- Step 3: Declare the variables and its member function.
- Step 4: Using the function getvalue() to get the two numbers.
- Step 5: Define the function operator +() to add two complex numbers.
- Step 6: Define the function operator –()to subtract two complex numbers.
- Step 7: Define the display function.
- Step 8: Declare the class objects obj1,obj2 and result.
- Step 9: Call the function getvalue using obj1 and obj2
- Step 10: Calculate the value for the object result by calling the function operator + and operator -.
- Step 11: Call the display function using obj1 and obj2 and result.
- Step 12: Return the values.
- Step 13: Stop the program.

Binary Operator Overloading Example Program

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

class complex{
    int a, b;
public:

    void getvalue() {
        cout << "Enter the value of Complex Numbers a,b:";
        cin >> a>>b;
    }

    complex operator+(complex ob) {
        complex t;
        t.a = a + ob.a;
        t.b = b + ob.b;
        return (t);
    }

    complex operator-(complex ob) {
        complex t;
        t.a = a - ob.a;
        t.b = b - ob.b;
        return (t);
    }

    void display() {
        cout << a << "+" << b << "i" << "\n";
    }
};

void main() {
    clrscr();
    complex obj1, obj2, result, result1;

    obj1.getvalue();
    obj2.getvalue();
    result = obj1 + obj2;
    result1 = obj1 - obj2;

    cout << "Input Values:\n";
    obj1.display();
    obj2.display();

    cout << "Result:";
    result.display();
    result1.display();getch();
}
```

Sample Output:

```
Enter the value of Complex Numbers a, b
4           5
Enter the value of Complex Numbers a, b
2           2
Input Values
4 + 5i
2 + 2i
Result
6 +   7i
2 +   3i
```

Overloading with friend Function

The friend functions are more useful in operator overloading. They offer better flexibility, which is not provided by the member function of the class. The difference between member function and friend function is that the member function takes argument explicitly. On the contrary, the friend function needs the parameters to be explicitly passed. The syntax of operator overloading with friend function is as follows:

```
friend return-type operator operator-symbol (variable1, variable2)
{
   statement1;
   statement2;
}
```

The keyword friend precedes function prototype declaration. It must be written inside the class. The function can be defined inside or outside the class. The arguments used in friend function are generally objects of the friend classes. A friend function is similar to normal function; the only difference being that friend function can access private member of the class through the objects. friend function has no permission to access private members of a class directly. However, it can access the private members through objects of the same class.

10.9 Write a program to overload unary operator using friend function.

```
#include<iostream.h>
#include<constream.h>
class complex
{
   float real,imag;
   public:
   complex() // zero argument constructor
   {
   real=imag=0;
```

```
    }
    complex (float r, float i) //
    two argument constructor
    {
    real=r;
    imag=i;
    }
    friend complex operator - ( complex c)
    {
    c.real=-c.real;
    c.imag=-c.imag;
    return c;
    }
    void display()
    {
    cout<<"\n Real:"<<real;
    cout<<"\n Imag:"<<imag;
    }
};
void main()
```

```
 {
     clrscr();
     complex c1(1.5,2.5),c2;
     c1.display();
     c2=-c1;
     cout<<"\n\n After Negation \n";
     c2.display();
 }
```

OUTPUT

```
Real : 1.5
Imag : 2.5
After Negation
Real : -1.5
Imag : -2.5
```

Explanation: In the above program, operator – is overloaded using friend function. The operator function is defined as friend. The statement c2=–c1 invokes the operator function. This statement also returns the negated values of c1 without affecting actual value of c1 and assigns it to object c2.

The negation operation can also be used with an object to alter its own data member variables. In such a case, the object itself acts as a source and destination object. This can be accomplished by sending reference of object. The following program illustrates this.

Assignment operator in C++

1. Assignment Operator is Used to assign value to an variable.
2. Assignment operator is denoted by equal to sign.

3. Assignment operator have Two values L-Value and R-value. Operator copies R-Value into L-Value.
4. It is a binary operator.

C++ Overloading Assignment Operator

1. C++ Overloading assignment operator can be done in object oriented programming.
2. By overloading assignment operator, all values of one object (i.e instance variables) can be copied to another object.
3. Assignment operator must be overloaded by a non-static member function only.
4. If the overloading function for the assignment operator is not written in the class, the compiler generates the function to overload the assignment operator.

Syntax

```
Return_Type operator = (const Class_Name &)
```

Way of overloading Assignment Operator

```
#include<iostream>
using namespace std;
 class Marks
{
  private:
     int m1;
     int m2;
    public:

  //Default constructor
  Marks() {
      m1 = 0;
```

```
        m2 = 0;
    }
     // Parametrised constructor
    Marks(int i, int j) {
        m1 = i;
        m2 = j;
    }
   // Overloading of Assignment Operator
    void operator=(Marks M ) {
        m1 = M.m1;
        m2 = M.m2;
    }
    void Display() {
      cout << "Marks in 1st Subject:" << m1;
      cout << "Marks in 2nd Subject:" << m2;
    }
};
int main()
{
  // Make two objects of class Marks
   Marks Mark1(45, 89);
   Marks Mark2(36, 59);

   cout << " Marks of first student : ";
   Mark1.Display();
   cout << " Marks of Second student :";
   Mark2.Display();

   // use assignment operator
   Mark1 = Mark2;

   cout << " Mark in 1st Subject :";
   Mark1.Display();
  return 0;
}
```

Explanation

```
private:
    int m1;
    int m2;
```

Here, in Class Marks contains private Data Members m1 and m2.

```
Marks Mark1(45, 89);
Marks Mark2(36, 59);
```

In the main function, we have made two objects ‘Mark1’ and ‘Mark2’ of class ‘Marks’. We have initialized values of two objects using parametrised constructor.

```
void operator=(const Marks &M ) {
            m1 = M.m1;
            m2 = M.m2;
}
```

As shown in above code, we overload the assignment operator, Therefore, ‘Mark1=Mark2’ from main function will copy content of object ‘Mark2’ into ‘Mark1’.

Output

```
Marks of first student :
Mark in 1st Subject : 45
Marks in 2nd Subject : 89

Marks of Second student :
Mark in 1st Subject : 36
Marks in 2nd Subject : 59
Marks of First student :
Mark in 1st Subject : 36
Marks in 2nd Subject : 59
```

Type Conversion in C

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as 'automatic type conversion'.
   - Done by the compiler on its own, without any external trigger from the user.
   - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
   - All the data types of the variables are upgraded to the data type of the variable with largest data type.
   -
   - **bool -> char -> short int -> int ->**
   - **unsigned int -> long -> unsigned ->**
   - **long long -> float -> double -> long double**
   - It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

   Example of Type Implicit Conversion:

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
    int x = 10;    // integer x
    char y = 'a';  // character c
     // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;
    // x is implicitly converted to float
    float z = x + 1.0;
     printf("x = %d, z = %f", x, z);
    return 0;
}
```

1. Output:

```
x = 107, z = 108.000000
```

2. **Explicit Type Conversion**– This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type.
   The syntax in C:

```
(type) expression
```

Type indicated the data type to which the final result is converted.

```
// C program to demonstrate explicit type casting

#include<stdio.h>


int main()

{

    double x = 1.2;


    // Explicit conversion from double to int

    int sum = (int)x + 1;


    printf("sum = %d", sum);


    return 0;

}
```

Output:

```
sum = 2
```

Advantages of Type Conversion

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps us to compute expressions containing variables of different data types.

MODULE IV

Function and class templates -Exception handling -try-catch-throw paradigm -exception specification -terminate and Unexpected functions -Uncaught exception.

TEMPLATES

C++ template is used in situation where we need to write the same function for different data types. For example, if we need a function to add two variables. The variable can be integer, float or double. For this purpose we have to write one function for each data type. To avoid writing the same function for different data types we use **template**.

There are two types of templates in C++ :

- Function template
- Class template

C++ Function templates are those functions which can handle different data types without separate code for each of them.

```
#include<iostream.h>
#include<conio.h>
template <class T>
T Sum(T n1, T n2)                    // Template function
{
        T rs;

        rs = n1 + n2;

        return rs;

}
```

```
    void main()
    {
        int A=10,B=20,C;
        long I=11,J=22,K;

        C = Sum(A,B);               // Calling template function
        cout<<"\nThe sum of integer values : "<<C;

        K = Sum(I,J);               // Calling template function
        cout<<"\nThe sum of long values : "<<K;

    }

Output :

        The sum of integer values : 30
        The sum of long values : 33
```

To make a function templates, we must write the following statement before function definition.

```
    template <class T>
```

Here T is the type name, which is dynamically determined by the compiler according to the parameter passed to fu nction definition.

Like function templates, we can also use templates with class to make member function common for different data types.

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

```
#include<conio.h>

template <class T>
class Addition                              // Template class
{
            public:

            T Add(T, T);
};

template <class T>
T Addition<T>::Add(T  n1, T n2)        // Template member function
{
            T rs;

            rs = n1 + n2;

            return rs;
}

void main()
{
            Addition  <int>obj1;
            Addition  <long>obj2;

            int A=10,B=20,C;
            long I=11,J=22,K;

            C = obj1.Add(A,B);             // Calling template member function
```

```
                cout<<"\nThe sum of integer values : "<<C;

                K = obj2.Add(I,J);              // Calling template member function
                cout<<"\nThe sum of long values : "<<K;

        }

        Output :

                        The sum of integer values : 30
                        The sum of long values : 33
```

EXCEPTION HANDLING IN C++

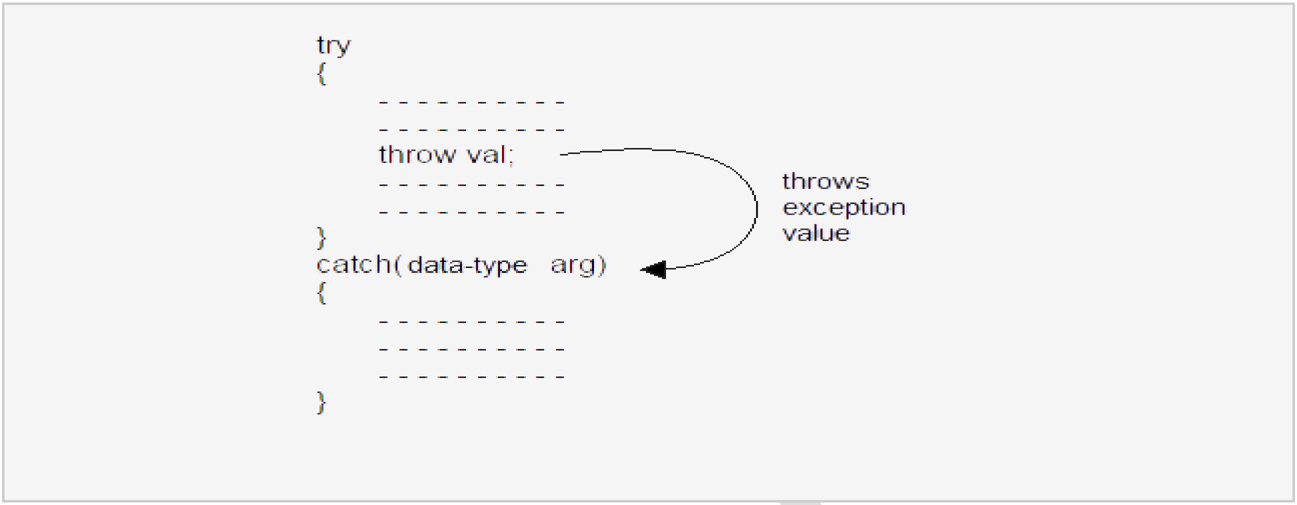
An exception is a situation, which occured by the runtime error. In other words, an exception is a runtime error. An exception may result in loss of data or an abnormal execution of program.

Exception handling is a mechanism that allows you to take appropriate action to avoid runtime errors.

C++ provides three keywords to support exception handling.

- **Try :** The try block contain statements which may generate exceptions.
- **Throw :** When an exception occur in try block, it is thrown to the catch block using throw keyword.
- **Catch :**The catch block defines the action to be taken, when an exception occur.

The general form of try-catch block in c++.



```
try
{
    - - - - - - - - - -
    - - - - - - - - - -
    throw val;
    - - - - - - - - - -
    - - - - - - - - - -
}
catch( data-type  arg)
{
    - - - - - - - - - -
    - - - - - - - - - -
    - - - - - - - - - -
}
```

Explain with an example. How exception handling is carried out in C++.

Exceptions are run time anomalies. They include conditions like division by zero or access to an array outside to its bound etc.

Types: Synchronous exception , Asynchronous exception.

Errors such as "out of range index" and "overflow belongs to the **synchronous type of exceptions.**

The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called **asynchronous exceptions.**

The proposed exception handling mechanism in C++ is designated to handle only synchronous exceptions.

The purpose of exception handling mechanism to provide means to detect and report an exceptional circumstances so that appropriate action can be taken. The mechanism suggest a separate machine handling code that perform the following tasks.

Find the problem (Hit the exception)

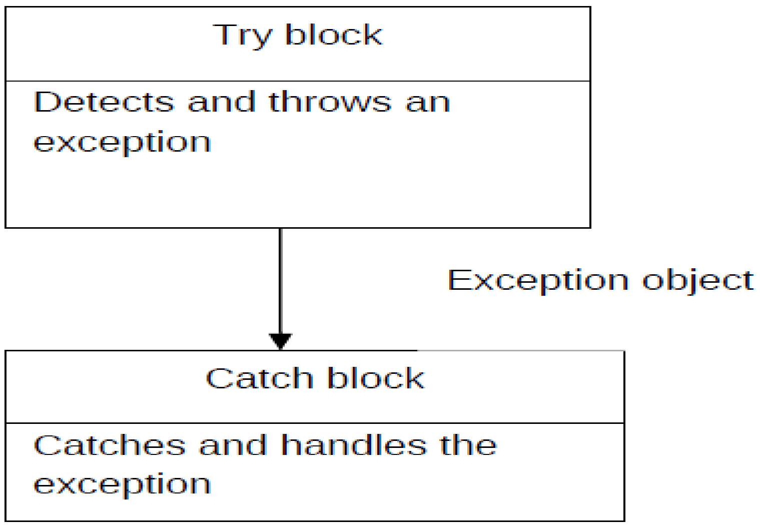
Inform that an error has occurred. (Throw exception)

Receive error information (Catch exception)

Take corrective action (Handle exception)

C++ exception handling mechanism is basically built upon three keywords, namely, **try , throw**

and **catch.** The keyword try is used to preface a block of statements which may generate exceptions. This block of statements is known as try block. When an exception is detected it is thrown using a throw statement in the try block. A catch block defined by the keyword catch catches the exception thrown by the throw statement in the try block and handles it appropriately.


Try block
Detects and throws an exception
Exception object
Catch block
Catches and handles the exception

- try block throwing an exception
- invoking function that generates exception
- throwing mechanism
- catching mechanism
- multiple catch statements
- catch all exceptions
- Rethrowing an exception

General form

```
try
{
…..
throw exception;
…….
}
Catch ( type arg)
{
……
}
```

Exceptions that has to be caught when functions are used- The form is as follows:

```
Type function (arg list)
{
……
Throw (object)
…..
}
try
{
…..
Invoke function here;

…….
}
Catch ( type arg)
{
Handles exception here
}
```

Multiple catch statements:

```
try
{
…..
throw exception;
…….
}
Catch ( type arg)
{
……// catch block1
}
Catch ( type arg)
{
……//catch block2
}
Catch ( type arg)
{
……//catch block n
}
```

Generic exception handling is done using ellipse as follows:

```
Catch ( . . .)
{
……
}
```

Define a DivideBy Zero definition and use it to throw exceptions on attempts to divide by zero.

```
#include<iostream.h>

Void mian()

{

int a,b;

Cout<<"\n enter values of a and b ";

Cin>>a;

Cin>>b;

try{

if(b!= 0)

{

Cout<<"\n Result = "<<a/b;

}

else

{

throw(b);

}}

Catch(int i)

{

Cout<<"caught divide by zero exception ";

}}
```

Example of simple try-throw-catch

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

```
    void main()
    {
        int n1,n2,result;
        cout<<"\nEnter 1st number : ";
        cin>>n1;
        cout<<"\nEnter 2nd number : ";
        cin>>n2;
        try
        {
          if(n2==0)
            throw n2;         //Statement 1
          else
          {
            result = n1 / n2;
            cout<<"\nThe result is : "<<result;
          }
        }
        catch(int x)
        {
          cout<<"\nCan't divide by : "<<x;
        }

        cout<<"\nEnd of program.";
    }
```

Output :

```
        Enter 1st number : 45
        Enter 2nd number : 0
        Can't divide by : 0
```

```
End of program
```

The catch block contain the code to handle exception. The catch block is similar to function definition.

```
catch(data-type arg)
{  - - - - - - - - -
 - - - - - - - - - -};
```

Data-type specifies the type of exception that catch block will handle, Catch block will receive value, send by throw keyword in try block.

A **single try statement** can have multiple catch statements. Execution of particular catch block depends on the type of exception thrown by the throw keyword. If throw keyword send exception of integer type, catch block with integer parameter will get execute.

Example of multiple catch blocks

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a=2;

      try
      {

         if(a==1)
            throw a;                 //throwing integer exception

         else if(a==2)
```

```
              throw 'A';          //throwing character exception

          else if(a==3)
              throw 4.5;          //throwing float exception

        }
        catch(int a)
        {
          cout<<"\nInteger exception caught.";
        }
        catch(char ch)
        {
          cout<<"\nCharacter exception caught.";
        }
        catch(double d)
        {
          cout<<"\nDouble exception caught.";
        }

        cout<<"\nEnd of program.";

    }
```

Output :

```
        Character exception caught.
        End of program.
```

The above example will caught only three types of exceptions that are integer, character and double. If an exception occur of long type, no catch block will get execute and abnormal program termination will occur. To avoid this, We can use the catch statement with three dots as parameter (...) so that it can handle all types of exceptions.

Example to catch all exceptions

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a=1;

      try
      {

         if(a==1)
            throw a;              //throwing integer exception

         else if(a==2)
            throw 'A';            //throwing character exception

         else if(a==3)
            throw 4.5;            //throwing float exception

      }
      catch(...)
      {
         cout<<"\nException occur.";
```

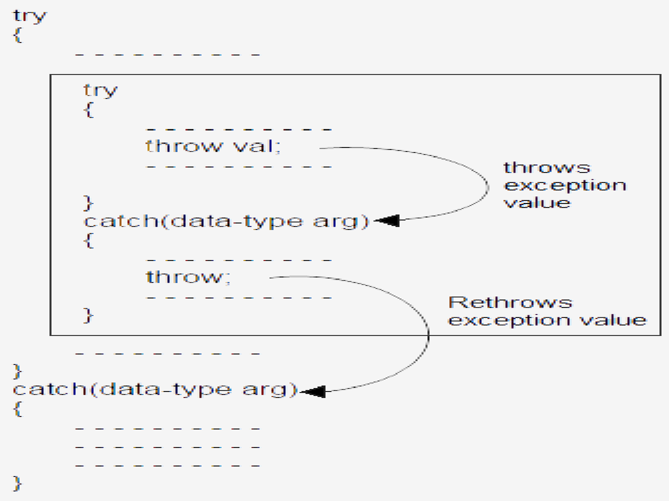
```
        }


        cout<<"\nEnd of program.";

    }
```

Output :

```
        Exception occur.

        End of program.
```

Rethrowing exception is possible, where we have an inner and outer try-catch statements (Nested try-catch). An exception to be thrown from inner catch block to outer catch block is called rethrowing exception.

Syntax of rethrowing exceptions



```
try
{
     - - - - - - - - - -
     try
     {
          - - - - - - - - - -
          throw val;
          - - - - - - - - - -
     }
     catch(data-type arg)
     {
          - - - - - - - - - -
          throw;
          - - - - - - - - - -
     }
     - - - - - - - - - -
}
catch(data-type arg)
{
     - - - - - - - - - -
     - - - - - - - - - -
     - - - - - - - - - -
}
```

Example of rethrowing exceptions

```
    #include<iostream.h>

    #include<conio.h>

    void main()
```

```
{
    int a=1;

     try
     {
          try
          {
               throw a;
          }
          catch(int x)
          {
             cout<<"\nException in inner try-catch block.";

             throw x;
          }

     }
     catch(int n)
     {
        cout<<"\nException in outer try-catch block.";
     }

     cout<<"\nEnd of program.";

}
```

Output :

```
     Exception in inner try-catch block.
     Exception in outer try-catch block.
     End of program.
```

We can restrict the type of exception to be thrown, from a function to its calling statement, by adding throw keyword to a function definition.

Example of restricting exceptions

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

void Demo() throw(int ,double)
{
    int a=2;

        if(a==1)
            throw a;              //throwing integer exception

        else if(a==2)
            throw 'A';            //throwing character exception

        else if(a==3)
throw 4.5;              //throwing float exception
}
void main()
{
      try
      {
         Demo();
      }
      catch(int n)
```

```
        {
          cout<<"\nException  caught.";
        }
        cout<<"\nEnd  of program.";
    }
```

The above program will abort because we have restricted the Demo() function to throw only integer and double type exceptions and Demo() is throwing character type exception.

Exception Specifications

C++ provides a mechanism to ensure that a given function is limited to throwing only a specified list of exceptions. An exception specification at the beginning of any function acts as a guarantee to the function's caller that the function will not directly or indirectly throw any exception not contained in the exception specification.

It is possible to restrict a function to throw only specified exceptions.This is achieved by adding a throw list clause to the function definition.The general form of using an exception specification is:

Type function(arg-list) throw (type list)

{

……..

Function body

………..

}

The type-list specifies the type of exceptions that may be thrown.Throwing any other type of exception will cause abnormal program termination.If we wish to prevent a function from throwing any exception,we may do so by making the type-list empty.that is we must use,

Throw(); //empty list

In the function header line.

unexpected() and terminate() Functions

Not all thrown errors can be caught and successfully dealt with by a catch block. In some situations, the best way to handle an exception is to terminate the program. Two special library functions are implemented in C++ to process exceptions not properly handled by catch blocks or exceptions thrown outside of a valid try block. These functions are unexpected() and terminate().

When a function with an exception specification throws an exception that is not listed in its exception specification, the function void unexpected() is called. Next, unexpected() calls a function specified by the set_ unexpected() function. By default, unexpected() calls the function terminate().

In some cases, the exception handling mechanism fails and a call to void terminate() is made. This terminate() call occurs in any of the following situations:

- When terminate() is explicitly called
- When no catch can be matched to a thrown object
- When the stack becomes corrupted during the exception-handling process
- When a system defined unexpected() is called

The terminate() function calls a function specified by the set_terminate() function. By default, terminate calls abort() , which exits from the program.

A terminate function cannot return to its caller, either by using return or by throwing an exception.

Uncaught exceptions

In the past few examples, there are quite a few cases where a function assumes its caller (or another function somewhere up the call stack) will handle the exception. In the following example, mySqrt() assumes someone will handle the exception that it throws -- but what happens if nobody actually does?

Here's our square root program again, minus the try block in main():

```
#include <iostream>
#include <cmath> // for sqrt() function

// A modular square root function
double mySqrt(double x)
{
  // If the user entered a negative number, this is an error condition
  if (x < 0.0)
      throw "Can not take sqrt of negative number"; // throw exception of type const char*

  return sqrt(x);
}

int main()
{
  std::cout << "Enter a number: ";
  double x;
  std::cin >> x;

  // Look ma, no exception handler!
  std::cout << "The sqrt of " << x << " is " << mySqrt(x) << '\n';

  return 0;
}
```

Now, let's say the user enters -4, and mySqrt(-4) raises an exception. Function mySqrt() doesn't handle the exception, so the program stack unwinds and control returns to main(). But there's no exception handler here either, so main() terminates. At this point, we just terminated our application!

When main() terminates with an unhandled exception, the operating system will generally notify you that an unhandled exception error has occurred. How it does this depends on the operating system, but possibilities include printing an error message, popping up an error dialog, or simply crashing. Some OSes are less graceful than others.

Generally this is something you want to avoid altogether!

Catch-all handlers

And now we find ourselves in a conundrum: functions can potentially throw exceptions of any data type, and if an exception is not caught, it will propagate to the top of your program and cause it to terminate. Since it's possible to call functions without knowing how they are even implemented (and thus, what type of exceptions they may throw), how can we possibly prevent this from happening?

Fortunately, C++ provides us with a mechanism to catch all types of exceptions. This is known as a **catch-all handler**. A catch-all handler works just like a normal catch block, except that instead of using a specific type to catch, it uses the ellipses operator (…) as the type to catch.

Here's an simple example:

```
#include <iostream>

int main()
{
        try
        {
                throw 5; // throw an int exception
        }
        catch (double x)
        {
                std::cout << "We caught an exception of type double: " << x << '\n';
        }
        catch (...) // catch-all handler
        {
                std::cout << "We caught an exception of an undetermined type\n";
        }
}
```

Because there is no specific exception handler for type int, the catch-all handler catches this exception. This example produces the following result:

We caught an exception of an undetermined type

The catch-all handler should be placed last in the catch block chain. This is to ensure that exceptions can be caught by exception handlers tailored to specific data types if those handlers exist. Often, the catch-all handler block is left empty:

```
catch(...) {} // ignore any unanticipated exceptions
```

This will catch any unanticipated exceptions and prevent them from stack unwinding to the top of your program, but does no specific error handling.

MODULE V

Inheritance - public, private, and protected derivations - multiple inheritance - virtual base class - abstract class - composite objects Runtime polymorphism - virtual functions - pure virtual functions - RTTI - typeid - dynamic casting - RTTI and templates - cross casting - down casting .

Inheritance in C++

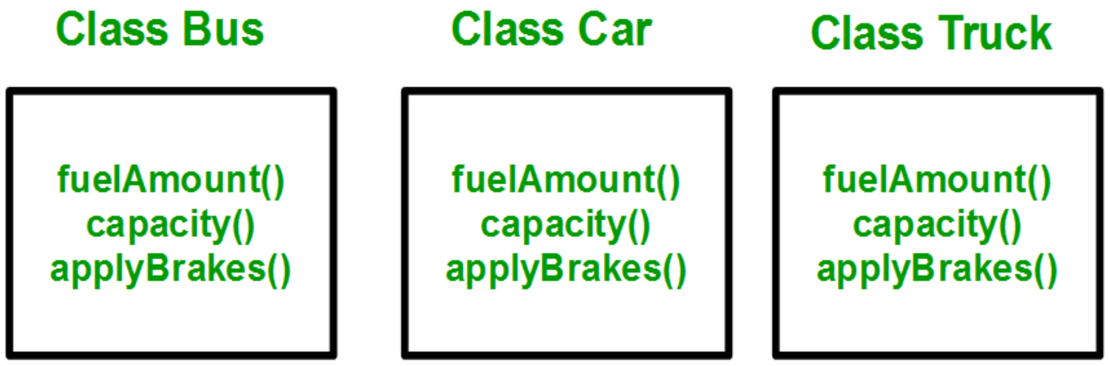
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

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

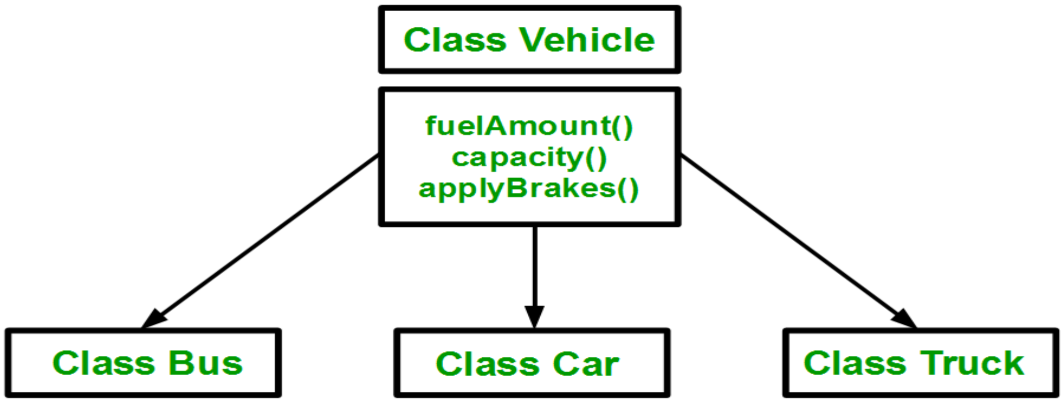
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

Implementing inheritance in C++:

For creating a sub-class which is inherited from the base class we have to follow the belowsyntax.

Syntax:

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.
Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

```
// C++ program to demonstrate implementation

// of Inheritance


#include <bits/stdc++.h>

using namespace std;


//Base class

class Parent

{

    public:

      int id_p;

};


// Sub class inheriting from Base Class(Parent)

class Child : public Parent

{

    public:

      int id_c;

};


//main function

int main()

   {


        Child obj1;


        // An object of class child has all data members
        // and member functions of class parent

        obj1.id_c = 7;

        obj1.id_p = 91;

        cout << "Child id is " << obj1.id_c << endl;

        cout << "Parent id is " << obj1.id_p << endl;


        return 0;

   }
```

Run on IDE

Output:

```
Child id is 7

Parent id is 91
```

In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

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.

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};
```

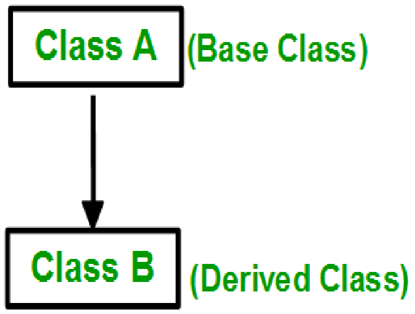
```
class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

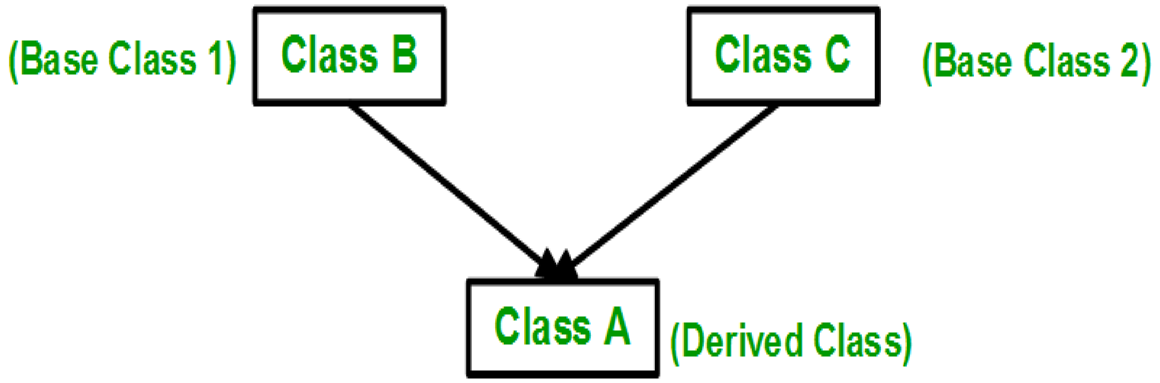
```
class subclass_name : access_mode base_class
{
  //body of subclass
};
```

```
// 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:

```
6.    This is a vehicle
```

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....

{

  //body of subclass

};
```

Here, the number of base classes will be separated by a comma (', ') and access mode for every base class must be specified.

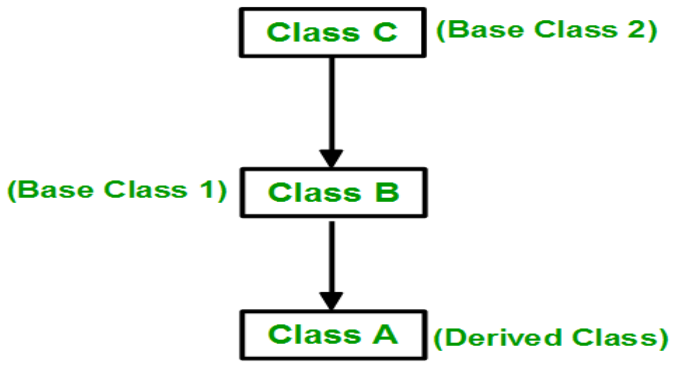
```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

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

// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

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

};
```

```
// 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
This is a 4 wheeler Vehicle
```

Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

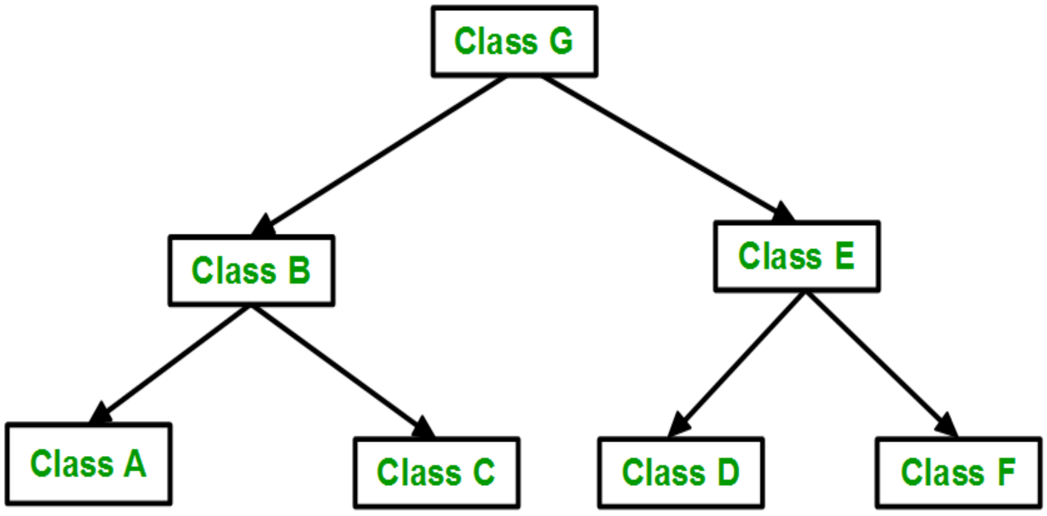


```
// C++ program to implement

// 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;
}
```

7. output:

```
8.  This is a Vehicle
9.  Objects with 4 wheels are vehicles
10. Car has 4 Wheels
```

Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



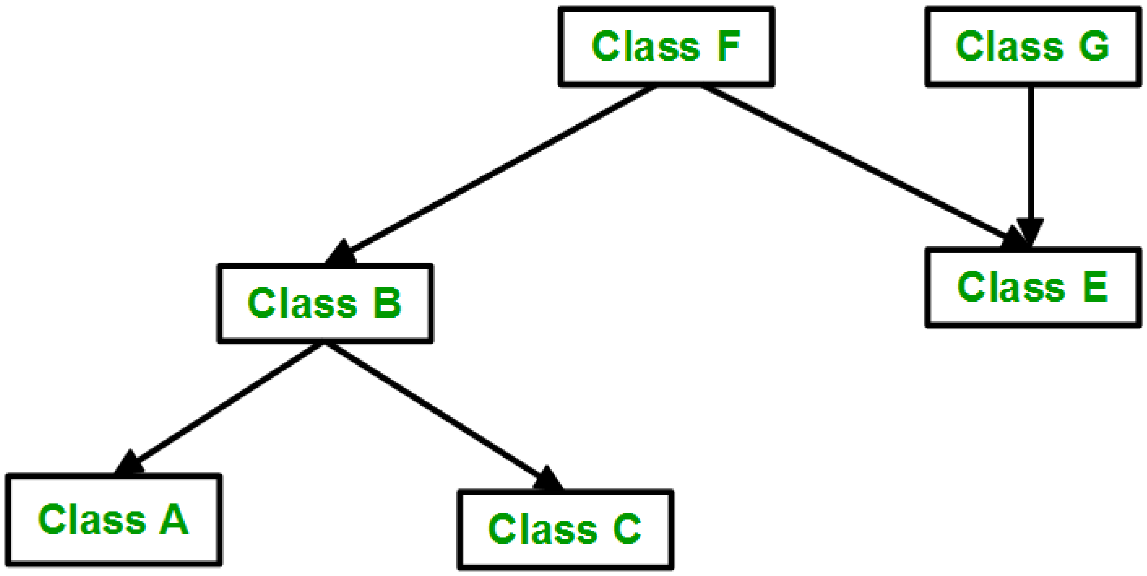
```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

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


// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};
```

```
// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:



```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

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

//base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};

// first sub class
class Car: public Vehicle
```

```
{


};


// second sub class
class Bus: public Vehicle, public Fare
{


};


// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle
```

Virtual base class is used in situation where a derived have multiple copies of base class.

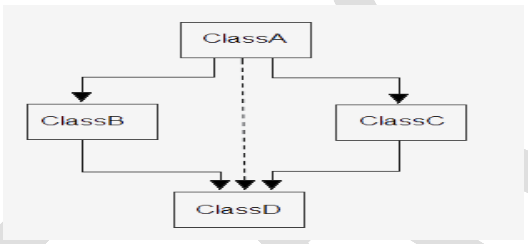
What is a virtual base class?

- 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 figure:



Example without using virtual base class

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

class ClassA
{
      public:
      int a;
};

class ClassB : public  ClassA
{
      public:
      int b;
```

```
};
class ClassC : public  ClassA
{
        public:
        int c;
};

class ClassD : public  ClassB, public  ClassC
{
        public:
        int d;
};

void  main()
{

                        ClassD obj;

                        obj.a  = 10;        //Statement 1, Error occur
                        obj.a  = 100;       //Statement 2, Error occur

                        obj.b  = 20;
                        obj.c  = 30;
                        obj.d  = 40;

                        cout<< "\n  A : "<< obj.a;
                        cout<< "\n  B : "<< obj.b;
                        cout<< "\n  C : "<< obj.c;
                        cout<< "\n  D : "<< obj.d;
```

```
            }


 Output :
          A from ClassB  : 10
          A from ClassC  : 100
          B : 20
          C : 30
          D : 40
```

In the above example, both **ClassB** & **ClassC** inherit **ClassA**, they both have single copy of **ClassA**. However **ClassD** inherit both **ClassB** & **ClassC**, therefore **ClassD** have two copies of **ClassA**, one from **ClassB** and another from **ClassC**.

Statement 1 and 2 in above example will generate error, bco'z compiler can't differentiate between two copies of **ClassA** in **ClassD**.

To remove multiple copies of **ClassA** from **ClassD**, we must inherit **ClassA** in **ClassB**and **ClassC** as **virtual** class.

Example using virtual base class

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

    class ClassA
    {
          public:
          int a;
    };

    class ClassB : virtual public  ClassA
    {
        public:
        int b;
};
class ClassC : virtual public ClassA
{
        public:
        int c;
};

class ClassD : public ClassB, public ClassC
{
        public:
        int d;
};
void main()
{
                ClassD obj;

                obj.a = 10;        //Statement 1
                obj.a = 100;       //Statement 2

                obj.b = 20;
                obj.c = 30;
                obj.d = 40;

                cout<< "\n A : "<< obj.a;
                cout<< "\n B : "<< obj.b;
                cout<< "\n C : "<< obj.c;
                cout<< "\n D : "<< obj.d;
```

```
            }
 Output  :
         A : 100
         B : 20
         C : 30
         D : 40
```

According to the above example, **Class D** have only one copy of **Class A** and statement 4 will overwrite the value of **a**, given in statement 3.

C++ VIRTUAL FUNCTION: Giving new implementation of base class method into derived class and the calling of this new implemented function with derived class's object is called function overriding.

Giving new implementation of derived class method into base class and the calling of this new implemented function with base class's object is done by making base class function as virtual function.

Virtual function is used in situation, when we need to invoke derived class function using base class pointer. We must declare base class function as virtual using **virtual** keyword preceding its normal declaration. The base class object must be of pointer type so that we can dynamically replace the address of base class function with derived class function. This is how we can achieve "Runtime Polymorphism".

If we doesn't use virtual keyword in base class, base class pointer will always execute function defined in base class.

Example of virtual function

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

class BaseClass
{
        public:
        virtual void Display()
        {
                cout<<"\n\tThis is Display() method of Base Class";
        }

        void Show()
        {
                cout<<"\n\tThis is Show() method of Base Class";
        }

};

class DerivedClass : public BaseClass
{

        public:
        void Display()
        {
                cout<<"\n\tThis is Display() method of Derived Class";
        }

        void Show()
        {
                cout<<"\n\tThis is Show() method of Derived Class";
        }

};
void main()
```

```
 {

        DerivedClass D;

        BaseClass *B;          //Creating Base Class Pointer
        B = new BaseClass;

        B->Display();          //This will invoke Display() method of Base Class
        B->Show();             //This will invoke Show() method of Base Class

        B=&D;

        B->Display();          //This will invoke Display() method of Derived Class
                              //bcoz Display() method is virtual in Base Class

        B->Show();             //This will invoke Show() method of Base Class
                              //bcoz Show() method is not virtual in Base Class

 }

Output :

        This is Display() method of Base Class
        This is Show() method of Base Class
        This is Display() method of Derived Class
        This is Show() method of Base Class
```

PURE VIRTUAL FUNCTION IN C++

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. Pure virtual function doesn't have body or implementation. We must implement all pure virtual functions in derived class.

Pure virtual function is also known as abstract function.

A class with at least one pure virtual function or abstract function is called abstract class. We can't create an object of abstract class. Member functions of abstract class will be invoked by derived class object.

Example of pure virtual function

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

class BaseClass       //Abstract class
{

        public:
         virtual void Display1()=0;     //Pure virtual function or abstract function
         virtual void Display2()=0;     //Pure virtual function or abstract function

         void Display3()
         {
                cout<<"\n\tThis is Display3() method of Base Class";
         }

};

class DerivedClass : public BaseClass
{

        public:
          void Display1()
          {
                cout<<"\n\tThis is Display1() method of Derived Class";
          }

          void Display2()
          {
                cout<<"\n\tThis is Display2() method of Derived Class";
          }

    };

    void main()
    {

          DerivedClass D;

          D.Display1();         // This will invoke Display1() method of Derived Class
          D.Display2();         // This will invoke Display2() method of Derived Class
          D.Display3();         // This will invoke Display3() method of Base Class

    }

 Output :

          This is Display1() method of Derived Class
          This is Display2() method of Derived Class
          This is Display3() method of Base Class
```

Abstract class is used in situation, when we have partial set of implementation of methods in a class. For example, consider a class have four methods. Out of four methods, we have an implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use abstract class.

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.

A class with at least one **pure virtual function** or **abstract function** is called abstract class.

Pure virtual function is also known as abstract function.

- We can't create an object of abstract class b'coz it has partial implementation of methods.
- Abstract function doesn't have body
- We must implement all abstract functions in derived class.

Example of C++ Abstract class

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

class BaseClass      //Abstract class
{

    public:
     virtual void Display1()=0;     //Pure virtual function or abstract function
     virtual void Display2()=0;     //Pure virtual function or abstract function

     void Display3()
     {
          cout<<"\n\tThis is Display3() method of Base Class";
     }

};
```

```
class DerivedClass : public BaseClass
{

    public:
     void Display1()
     {
           cout<<"\n\tThis is Display1() method of Derived Class";
     }

     void Display2()
     {
           cout<<"\n\tThis is Display2() method of Derived Class";
     }

};

void main()
{

    DerivedClass D;

    D.Display1();       // This will invoke Display1() method of Derived Class
    D.Display2();       // This will invoke Display2() method of Derived Class
    D.Display3();       // This will invoke Display3() method of Base Class

}
```

Output :

```
This is Display1() method of Derived Class
This is Display2() method of Derived Class
This is Display3() method of Base Class
```

composite objects Runtime polymorphism

Virtual Functions

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

Problem without Virtual Keyword

```
class Base
{
 public:
 void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}

int main()
{
 Base* b;      //Base class pointer
 Derived d;    //Derived class object
 b = &d;
 b->show();    //Early Binding Ocuurs
}
```

Output : Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Using Virtual Keyword

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

```
class Base
{
 public:
 virtual void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}
```

```
int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Late Binding Ocuurs
}
```

```
Output : Derived class
```

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

Using Virtual Keyword and Accessing Private Method of Derived class

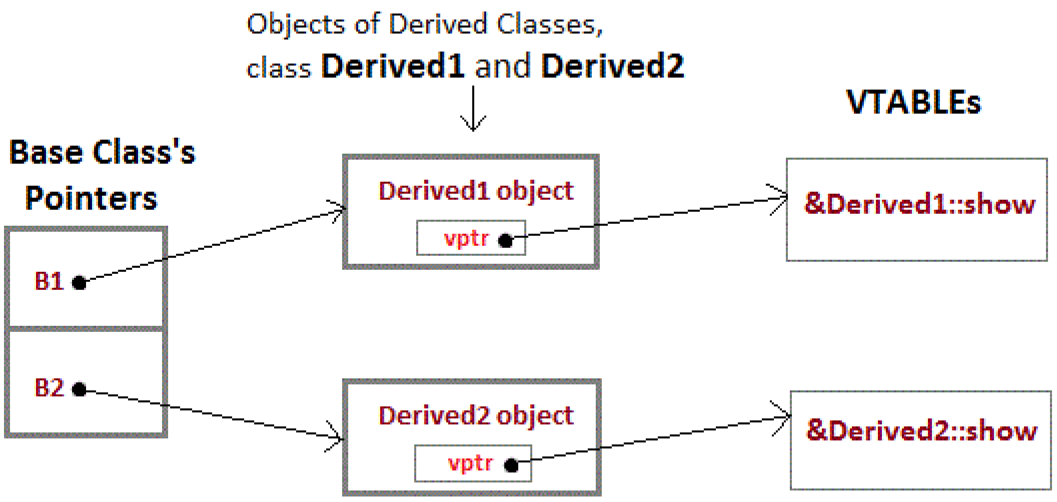
We can call **private** function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

```
#include
using namespace std;

class A
{
   public:
   virtual void show()
   {
      cout << "Base class\n";
   }
};

class B: public A
{
private:
   virtual void show()
   {
      cout << "Derived class\n";
   }
};
```

```
int main()
{
    A *a;
    B b;
    a = &b;
    a -> show();
}
```

Output : Derived class

Mechanism of Late Binding



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

To accomplich late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resovle the call by binding the correct function using the vpointer.

Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the copiler uses **VPTR**(vpointer) to point to the Virtual Function.

RTTI

RTTI is short for **Run-time Type Identification**. RTTI is to provide a standard way for a program to determine the type of object during runtime.

In other words, RTTI allows programs that use pointers or references to base classes to retrieve the actual derived types of the objects to which these pointers or references refer.

RTTI is provided through two operators:

1. The **typeid** operator, which returns the actual type of the object referred to by a pointer (or a reference).
2. The **dynamic_cast** operator, which safely converts from a pointer (or reference) to a base type to a pointer (or reference) to a derived type.

The dynamic_cast Operator

An attempt to **convert** an object **into** a **more specific** object.

Let's look at the code. If you do not understand what's going on, please do not worry, we'll get to it later.

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

class A
{
public:
        virtual void f(){cout << "A::f()"  << endl;}
};

class B : public A
{
public:
        void f(){cout << "B::f()"  << endl;}
};

int main()
{
        A a;
        B b;
        a.f();      // A::f()
        b.f();      // B::f()

        A *pA = &a;
        B *pB = &b;
        pA->f();     // A::f()
        pB->f();     // B::f()

        pA = &b;
```

```
            // pB = &a;     // not allowed
            pB = dynamic_cast<B*>(&a;); // allowed but it returns NULL

            return 0;
}
```

The **dynamic_cast** operator is intended to be the most heavily used RTTI component. It doesn't give us what type of object a pointer points to. Instead, it answers the question of whether we can **safely assign** the address of an object to a pointer of a particular type.

Unlike other casts, a **dynamic_cast** involves a run-time type check. If the object bound to the **pointer** is not an object of the target type, it fails and the value is **0**. If it's a **reference** type when it fails, then an exception of type **bad_cast** is thrown. So, if we want **dynamic_cast** to throw an exception (**bad_cast**) instead of returning **0**, cast to a reference instead of to a pointer. Note also that the dynamic_cast is the only cast that relies on run-time checking.

"The need for **dynamic_cast** generally arises because you want to perform derived class operation on a derived class object, but you have only a pointer or reference-to-base"

Let's look at the example code:

```
class Base { };

class Derived : public Base { };

int main()
{
            Base b;
            Derived d;

            Base  *pb = dynamic_cast<Base*>(&d;);            // #1
            Derived  *pd = dynamic_cast<Derived*>(&b;);      // #2

            return 0;
}
```

The #1 is ok because dynamic_cast is always successful when we cast a class to one of its base classes

The #2 conversion has a compilation error:

```
error C2683: 'dynamic_cast' : 'Base' is not a polymorphic type.
```

It's because base-to-derived conversions are not allowed with dynamic_cast unless the base class is **polymorphic**.

So, if we make the Base class polymorphic by adding virtual function.

Upcasting and Downcasting

Converting a derived-class reference or pointer to a base-class reference or pointer is called **upcasting**. It is always allowed for public inheritance without the need for an explicit type cast.

Actually this rule is part of expressing the **is-a** relationship. A **Derived** object is a **Base** object in that it inherits all the data members and member functions of a **Base** object. Thus, anything that we can do to a **Base** object, we can do to a **Derived** class object.

The **downcasting**, the opposite of upcasting, is a process converting a base-class pointer or reference to a derived-class pointer or reference.

It is not allowed without an explicit type cast. That's because a derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

Here is a self explanatory example

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

class Employee {
private:
        int id;
public:
        void show_id(){}
};

class Programmer : public Employee {
public:
        void coding(){}
};

int main()
{
        Employee employee;
        Programmer programmer;

        // upcast - implicit upcast allowed
        Employee *pEmp = &programmer;
        // downcast - explicit type cast required
        Programmer *pProg = (Programmer *)&employee;



        // Upcasting: safe - progrommer is an Employee
        // and has his id to do show_id().
        pEmp->show_id();
        pProg->show_id();

        // Downcasting: unsafe - Employee does not have
        // the method, coding().
    // compile error: 'coding' : is not a member of 'Employee'
        // pEmp->coding();
        pProg->coding();

        return 0;
}
```

The typeid

typeid operator allows us to determine whether two objects are the same type.

In the previous example for Upcasting and Downcasting, **employee** gets the method **coding()** which is not desirable. So, we need to check if a pointer is pointing to the Programmer object before we use the method, coding().

Here is a new code showing how to use **typeid**:

```
class Employee {
private:
        int id;
public:
        void show_id(){}
};

class Programmer : public Employee {
public:
        void coding(){}
};

#include <typeinfo>

int main()
{
        Employee lee;
        Programmer park;

        Employee *pEmpA = &lee;
        Employee *pEmpB = &park;

        // check if two object is the same
        if(typeid(Programmer) == typeid(lee)) {
                Programmer *pProg = (Programmer *)&lee;
                pProg->coding();
        }
        if(typeid(Programmer) == typeid(park)) {
                Programmer *pProg = (Programmer *)&park;
                pProg->coding();
        }

        pEmpA->show_id();
        pEmpB->show_id();

        return 0;
}
```

So, only a programmer uses the **coding()** method.

Note that we included <typeinfo> in the example. The **typeid** operator returns a reference to a **type_info** object, where **type_info** is a class defined in the **typeinfo** header file.

Runtime Type Information (RTTI)

Runtime Type Information (RTTI) is the concept of determining the type of any variable during

Execution (runtime.) The RTTI mechanism contains:

- The operator dynamic_cast
- The operator typeid
- The struct type_info

RTTI can only be used with polymorphic types. This means that with each class you make, you must have at least one virtual function (either directly or through inheritance.)

Compatibility note: On some compilers you have to enable support of RTTI to keep track of dynamic types. So to make use of dynamic_cast (see next section) you have to enable this feature. See you compiler documentation for more detail.

Dynamic_cast

The dynamic_cast can only be used with pointers and references to objects. It makes sure that the result of the type conversion is valid and complete object of the requested class. This is way a dynamic_cast will always be successful if we use it to cast a class to one of its base classes. Take a look at the example:

```
        class Base_Class { };
```

```
          class Derived_Class: public Base_Class { };

          Base_Class a; Base_Class * ptr_a;
          Derived_Class b; Derived_Class * ptr_b;

          ptr_a = dynamic_cast<Base_Class *>(&b);
          ptr_b = dynamic_cast<Derived_Class *>(&a);
```

The first dynamic_cast statement will work because we cast from derived to base. The second dynamic_cast statement will produce a compilation error because base to derived conversion is not allowed with dynamic_cast unless the base class is polymorphic.

If a class is polymorphic then dynamic_cast will perform a special check during execution. This check ensures that the expression is a valid and complete object of the requested class. Take a look at the example:

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class Base_Class { virtual void dummy() {} };
class Derived_Class: public Base_Class { int a; };

int main () {
  try {
          Base_Class * ptr_a = new Derived_Class;
          Base_Class * ptr_b = new Base_Class;
          Derived_Class * ptr_c;

          ptr_c = dynamic_cast< Derived_Class *>(ptr_a);
          if (ptr_c ==0) cout << "Null pointer on first type-cast" << endl;

          ptr_c = dynamic_cast< Derived_Class *>(ptr_b);
          if (ptr_c ==0) cout << "Null pointer on second type-cast" << endl;

          } catch (exception& my_ex) {cout << "Exception: " << my_ex.what();}
  return 0;
}
```

In the example we perform two dynamic_casts from pointer objects of type Base_Class* (namely ptr_a and ptr_b) to a pointer object of type Derived_Class*.

If everything goes well then the first one should be successful and the second one will fail. The pointers ptr_a and ptr_b are both of the type Base_Class. The pointer ptr_a points to an object of the type Derived_Class. The pointer ptr_b points to an object of the type Base_Class. So when the dynamic type cast is performed then ptr_a is pointing to a full object of class Derived_Class, but the pointer ptr_b points to an object of class Base_Class. This object is an incomplete object of class

Derived_Class; thus this cast will fail!

Because this dynamic_cast fails a null pointer is returned to indicate a failure. When a reference type is converted with dynamic_cast and the conversion fails then there will be an exception thrown out instead of the null pointer. The exception will be of the type bad_cast.

With dynamic_cast it is also possible to cast null pointers even between the pointers of unrelated classes. Dynamic_cast can cast pointers of any type to void pointer(void*).

Typeid and typ_info

If a class hierarchy is used then the programmer doesn't have to worry (in most cases) about the data-type of a pointer or reference, because the polymorphic mechanism takes care of it. In some cases the programmer wants to know if an object of a derived class is used. Then the programmer can make use of dynamic_cast. (If the dynamic cast is successful, then the pointer will point to an object of a derived class or to a class that is derived from that derived class.) But there are circumstances that the programmer (not often) wants to know the prizes data-type. Then the programmer can use the typeid operator.

The typeid operator can be used with:

- Variables
- Expressions
- Data-types

Take a look at the typeid example:

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

int main ()
{
        int * a;
        int b;

        a=0; b=0;
        if (typeid(a) != typeid(b))
        {
                cout << "a and b are of different types:\n";
                cout << "a is: " << typeid(a).name() << '\n';
                cout << "b is: " << typeid(b).name() << '\n';
        }
        return 0;
}
```

Note: the extra header file typeinfo.

The result of a typeid is a const type_info&. The class type_info is part of the standard C++ library and contains information about data-types. (This information can be different. It all depends on how it is implemented.)

A bad_typeid exception is thrown by typeid, if the type that is evaluated by typeid is a pointer that is preceded by a dereference operator and that pointer has a null value.

RTTI

RTTI is short for **Run-time Type Identification**. RTTI is to provide a standard way for a program to determine the type of object during runtime.

In other words, RTTI allows programs that use pointers or references to base classes to retrieve the actual derived types of the objects to which these pointers or references refer.

RTTI is provided through two operators:

3. The **typeid** operator, which returns the actual type of the object referred to by a pointer (or a reference).
4. The **dynamic_cast** operator, which safely converts from a pointer (or reference) to a base type to a pointer (or reference) to a derived type.

The dynamic_cast Operator

An attempt to **convert** an object **into** a **more specific** object.

Let's look at the code. If you do not understand what's going on, please do not worry, we'll get to it later.

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

class A
{
public:
        virtual void f(){cout << "A::f()" << endl;}
};

class B : public A
{
public:
        void f(){cout << "B::f()" << endl;}
};

int main()
{
        A a;
        B b;
        a.f();      // A::f()
        b.f();      // B::f()

        A *pA = &a;
        B *pB = &b;
        pA->f();    // A::f()
        pB->f();    // B::f()
        pA = &b;
        // pB = &a;     // not allowed
        pB = dynamic_cast<B*>(&a;);  // allowed but it returns NULL

        return 0;
}
```

The **dynamic_cast** operator is intended to be the most heavily used RTTI component. It doesn't give us what type of object a pointer points to. Instead, it answers the question of whether we can **safely assign** the address of an object to a pointer of a particular type.

Unlike other casts, a **dynamic_cast** involves a run-time type check. If the object bound to the **pointer** is not an object of the target type, it fails and the value is **0**. If it's a **reference** type when it fails, then an exception of type **bad_cast** is thrown. So, if we want **dynamic_cast** to throw an exception (**bad_cast**) instead of returning **0**, cast to a reference instead of to a pointer. Note also that the dynamic_cast is the only cast that relies on run-time checking.

"The need for **dynamic_cast** generally arises because you want to perform derived class operation on a derived class object, but you have only a pointer or reference-to-base" said Scott Meyers in his book "Effective C++".

Let's look at the example code:

```
class Base { };

class Derived : public Base { };

int main()
```

```
{
        Base b;
        Derived d;

        Base *pb = dynamic_cast<Base*>(&d;);          // #1
        Derived *pd = dynamic_cast<Derived*>(&b;);    // #2

        return 0;
}
```

The #1 is ok because dynamic_cast is always successful when we cast a class to one of its base classes

The #2 conversion has a compilation error:

```
error C2683: 'dynamic_cast' : 'Base' is not a polymorphic type.
```

It's because base-to-derived conversions are not allowed with dynamic_cast unless the base class is **polymorphic**.

So, if we make the Base class polymorphic by adding virtual function.

as in the code sample below, it will be compiled successfully.

```
class Base {virtual void vf(){}};
class Derived : public Base { };

int main()
{
        Base b;
        Derived d;

        Base *pb = dynamic_cast<Base*>(&d;);                    // #1
        Derived *pd = dynamic_cast<Derived*>(&b;);     // #2

        return 0;
}
```

But at runtime, the #2 cast fails and produces **null pointer**.

Let's look at another example.

```
class Base { virtual void vf(){} };

class Derived : public Base { };

int main()
{
        Base *pBDerived = new Derived;
        Base *pBBase = new Base;
        Derived *pd;

        pd = dynamic_cast<Derived*>(pBDerived);        #1
```

```
            pd = dynamic_cast<Derived*>(pBBase);            #2

            return 0;
}
```

The example has two dynamic casts from pointers of type **Base** to a point of type **Derived**. But only the #1 is successful.

Even though **pBDerived** and **pBBase** are pointers of type **Base***, **pBDerived** points to an object of type **Derived**, while **pBBase** points to an object of type **Base**. Thus, when their respective type-castings are performed using **dynamic_cast**, **pBDerived** is pointing to a full object of class **Derived**, whereas **pBBase** is pointing to an object of class **Base**, which is an incomplete object of class **Derived**.

In general, the expression

```
dynamic_cast<Type *>(ptr)
```

converts the pointer **ptr** to a pointer of type **Type*** if the pointer-to object (*ptr) is of type **Type** or else derived directly or indirectly from type **Type**. Otherwise, the expression evaluates to **0**, the null pointer.

Dynamic_cast - example

In the code below, there is one function call in main() that's not working. Which one?

```
#include <iostream>
```

```
using namespace std;

class A
{
public:
        virtual void g(){}
};
class B : public A
{
public:
        virtual void g(){}
};
class C : public B
{
public:
        virtual void g(){}
};
class D : public C
{
public:
        virtual void g(){}
};

A* f1()
{
        A *pa = new C;
        B *pb = dynamic_cast<B*>(pa);
        return pb;
}
```

```
A* f2()
{
        A *pb = new B;
        C *pc = dynamic_cast<C*>(pb);
        return pc;
}

A* f3()
{
        A *pa = new D;
        B *pb = dynamic_cast<B*>(pa);
        return pb;
}

int main()
{
   f1()->g();   // (1)
   f2()->g();   // (2)
   f3()->g();   // (3)

   return 0;
}
```

Answer (2). It's a downcasting.

Dynamic_cast - another example

In this example, the **DoSomething(Window* w)** is passed down **Window** pointer. It calls **scroll()** method which is only available from **Scroll** object. So, in this case, we need to check if the object is the **Scroll** type or not before the call to the **scroll()** method.

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

class Window
{
public:
        Window(){}
        Window(const string s):name(s) {};
        virtual ~Window() {};
        void getName() { cout << name << endl;};
private:
        string name;
};

class ScrollWindow : public Window
{
public:
        ScrollWindow(string s) : Window(s) {};
        ~ScrollWindow() {};
        void scroll() { cout << "scroll()" << endl;};
};

void DoSomething(Window *w)
{
        w->getName();
        // w->scroll();  // class "Window"  has no member scroll

        // check if the pointer is pointing to a scroll window
        ScrollWindow *sw = dynamic_cast<ScrollWindow*>(w);

        // if not null, it's a scroll window object
        if(sw) sw->scroll();
}

int main()
{
        Window *w = new Window("plain  window");
        ScrollWindow *sw = new ScrollWindow("scroll  window");

        DoSomething(w);
        DoSomething(sw);

        return 0;
}
```

Upcasting and Downcasting

Converting a derived-class reference or pointer to a base-class reference or pointer is called **upcasting**. It is always allowed for public inheritance without the need for an explicit type cast.

Actually this rule is part of expressing the **is-a** relationship. A **Derived** object is a **Base** object in that it inherits all the data members and member functions of a **Base** object. Thus, anything that we can do to a **Base** object, we can do to a **Derived** class object.

The **downcasting**, the opposite of upcasting, is a process converting a base-class pointer or reference to a derived-class pointer or reference.

It is not allowed without an explicit type cast. That's because a derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

Here is a self explanatory example

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

class Employee {
private:
        int id;
public:
        void show_id(){}
};

class Programmer : public Employee {
public:
        void coding(){}
};

int main()
{
        Employee employee;
```

```
            Programmer programmer;

            // upcast - implicit  upcast allowed
            Employee  *pEmp = &programmer;

            // downcast - explicit  type cast required
            Programmer *pProg = (Programmer *)&employee;



            // Upcasting: safe - progrommer is an Employee
            // and has his id to do show_id().
            pEmp->show_id();
            pProg->show_id();

            // Downcasting: unsafe - Employee does not have
            // the method, coding().
       // compile error: 'coding' : is not a member of 'Employee'
            // pEmp->coding();
            pProg->coding();

            return 0;
}
```

More on Upcasting and Downcasting

The typeid

typeid operator allows us to determine whether two objects are the same type.

In the previous example for Upcasting and Downcasting, **employee** gets the method **coding()** which is not desirable. So, we need to check if a pointer is pointing to the Programmer object before we use the method, coding().

Here is a new code showing how to use **typeid**:

```
class Employee {
private:
        int id;
public:
        void show_id(){}
};

class Programmer : public Employee {
public:
        void coding(){}
};

#include <typeinfo>

int main()
{
        Employee lee;
        Programmer park;

        Employee *pEmpA = &lee;
        Employee *pEmpB = &park;
```

```
            // check if two object is the same
            if(typeid(Programmer) == typeid(lee)) {
                        Programmer *pProg = (Programmer *)&lee;
                        pProg->coding();
            }
            if(typeid(Programmer) == typeid(park)) {
                        Programmer *pProg = (Programmer *)&park;
                        pProg->coding();
            }

            pEmpA->show_id();
            pEmpB->show_id();

            return 0;
}
```

So, only a programmer uses the **coding()** method.

Note that we included <typeinfo> in the example. The **typeid** operator returns a reference to a **type_info** object, where **type_info** is a class defined in the **typeinfo** header file.

RTTI - pros and cons

This is from Google C++ Style Guide.

RTTI allows a programmer to query the C++ class of an object at run time. This is done by use of **typeid** or **dynamic_cast**. Avoid using Run Time Type Information (RTTI).

1. **Pros**
   The standard alternatives to RTTI (described below) require modification or redesign of the class hierarchy in question. Sometimes such modifications are infeasible or undesirable, particularly in widely-used or mature code.
   RTTI can be useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type. It is also useful in managing the relationship between objects and their mocks.
   RTTI is useful when considering multiple abstract objects. Consider

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast&LT;Derived*&GT;(other);
  if (that == NULL)
    return false;
  ...
}
```

9. **Cons**
   Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed.
   Undisciplined use of RTTI makes code hard to maintain. It can lead to type-based decision trees or switch statements scattered throughout the code, all of which must be examined when making further changes.
10. **Decision**
   RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unittests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code. If you find yourself needing to

write code that behaves differently based on the class of an object, consider one of the following alternatives to querying the type:

1. Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
2. If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

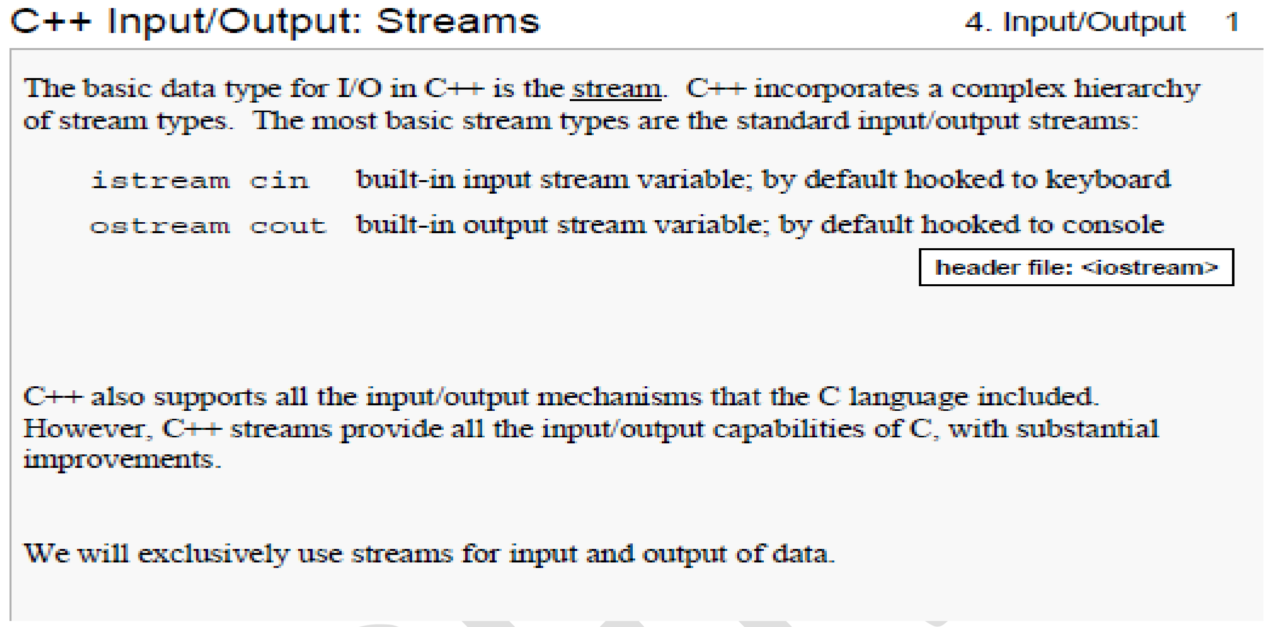
When the logic of a program guarantees that a given instance of a base class is in fact an instance of a particular derived class, then a **dynamic_cast** may be used freely on the object. Usually one can use a **static_cast** as an alternative in such situations. Decision trees based on type are a strong indication that your code is on the wrong track.

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

Code such as this usually breaks when additional subclasses are added to the class hierarchy. Moreover, when properties of a subclass change, it is difficult to find and modify all the affected code segments.
Do not hand-implement an RTTI-like workaround. The arguments against RTTI apply just as much to workarounds like class hierarchies with type tags. Moreover, workarounds disguise your true intent.

MODULE VI

Streams and formatted I/O - I/O manipulators - file handling - random access - object serialization - namespaces - std namespace - ANSI String Objects - standard template library.



C++ Input/Output: Streams

4. Input/Output 1

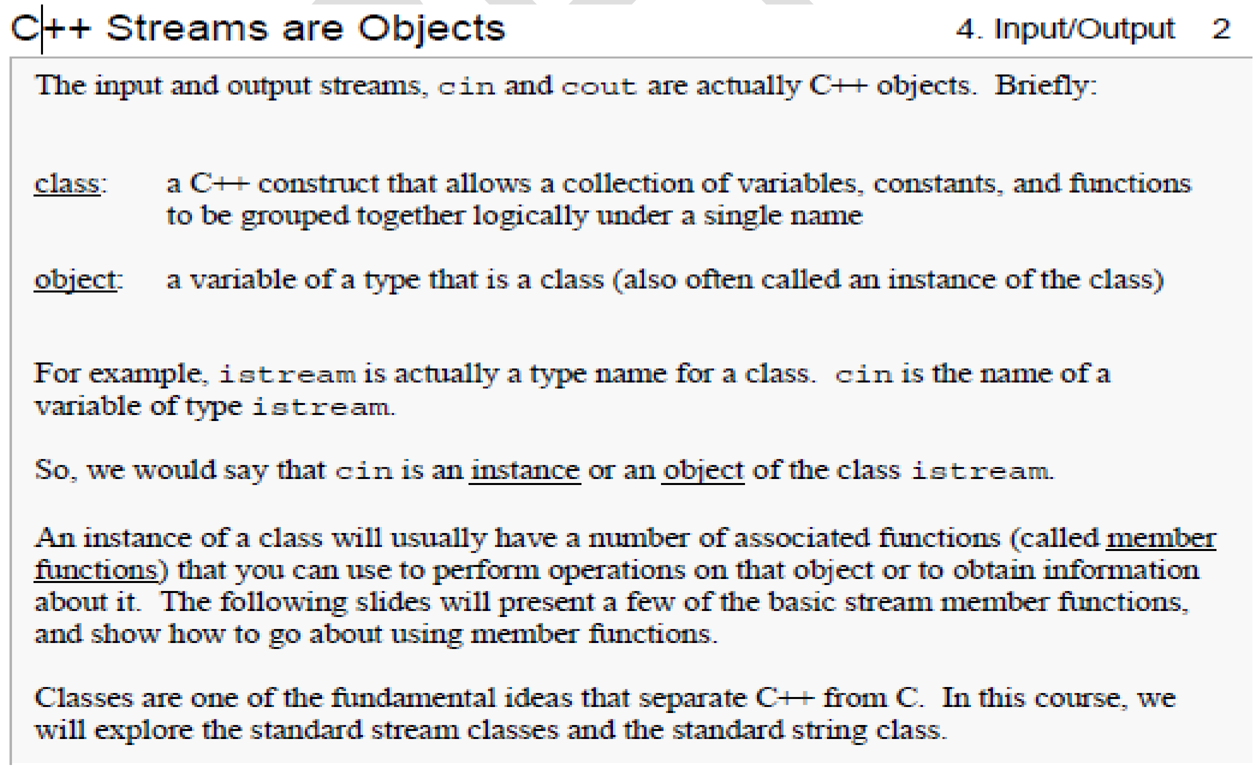
The basic data type for I/O in C++ is the <u>stream</u>. C++ incorporates a complex hierarchy of stream types. The most basic stream types are the standard input/output streams:

`istream cin`	built-in input stream variable; by default hooked to keyboard
`ostream cout`	built-in output stream variable; by default hooked to console

header file: <iostream>

C++ also supports all the input/output mechanisms that the C language included. However, C++ streams provide all the input/output capabilities of C, with substantial improvements.

We will exclusively use streams for input and output of data.



C++ Streams are Objects

4. Input/Output 2

The input and output streams, `cin` and `cout` are actually C++ objects. Briefly:

<u>class</u>:	a C++ construct that allows a collection of variables, constants, and functions to be grouped together logically under a single name
<u>object</u>:	a variable of a type that is a class (also often called an instance of the class)

For example, `istream` is actually a type name for a class. `cin` is the name of a variable of type `istream`.

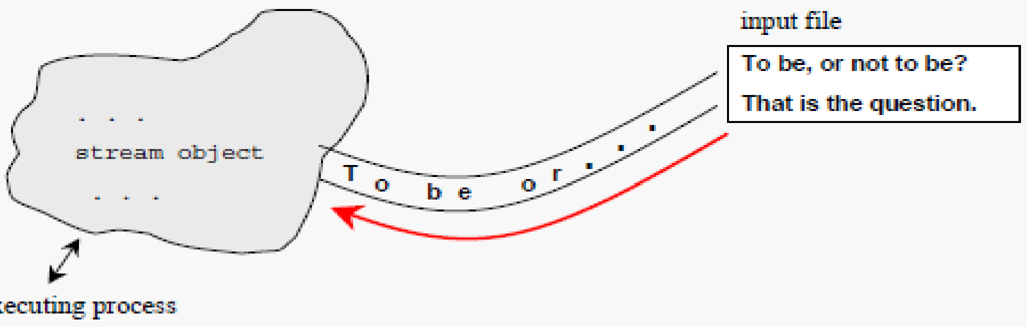
So, we would say that `cin` is an <u>instance</u> or an <u>object</u> of the class `istream`.

An instance of a class will usually have a number of associated functions (called <u>member functions</u>) that you can use to perform operations on that object or to obtain information about it. The following slides will present a few of the basic stream member functions, and show how to go about using member functions.

Classes are one of the fundamental ideas that separate C++ from C. In this course, we will explore the standard stream classes and the standard string class.

Conceptual Model of a Stream

A stream provides a connection between the process that initializes it and an object, such as a file, which may be viewed as a sequence of data. In the simplest view, a stream object is simply a serialized view of that other object. For example, for an input stream:



We think of data as flowing in the stream to the process, which can remove data from the stream as desired. The data in the stream cannot be lost by "flowing past" before the program has a chance to remove it.

The stream object provides the process with an "interface" to the data.

Output: the Insertion Operator

To get information out of a file or a program, we need to explicitly instruct the computer to output the desired information.

One way of accomplishing this in C++ is with the use of an output stream.

In order to use the standard I/O streams, we must have in our program the pre-compiler directive:

```
#include <iostream>
```

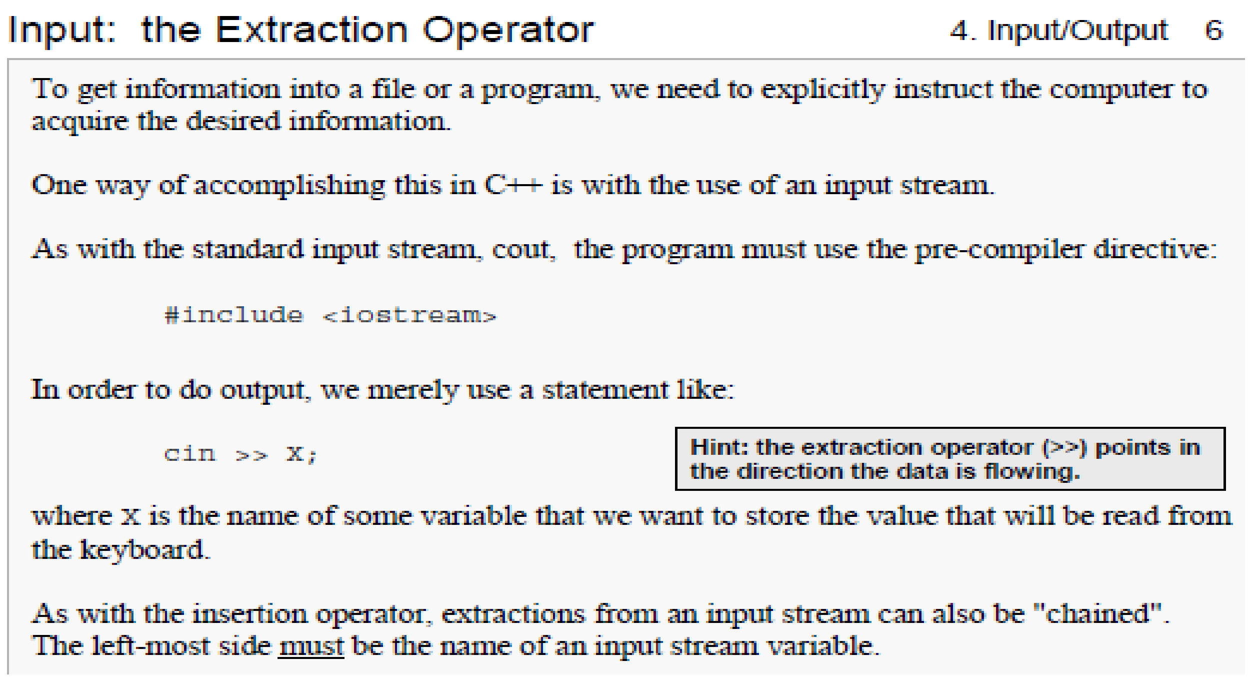
In order to do output to the screen, we merely use a statement like:

```
cout << " X = " << X;
```

Hint: the insertion operator (<<) points in the direction the data is flowing.

where X is the name of some variable or constant that we want to write to the screen.

Insertions to an output stream can be "chained" together as shown here. The left-most side must be the name of an output stream variable, such as `cout`.



Input: the Extraction Operator

4. Input/Output 6

To get information into a file or a program, we need to explicitly instruct the computer to acquire the desired information.

One way of accomplishing this in C++ is with the use of an input stream.

As with the standard input stream, cout, the program must use the pre-compiler directive:

```
#include <iostream>
```

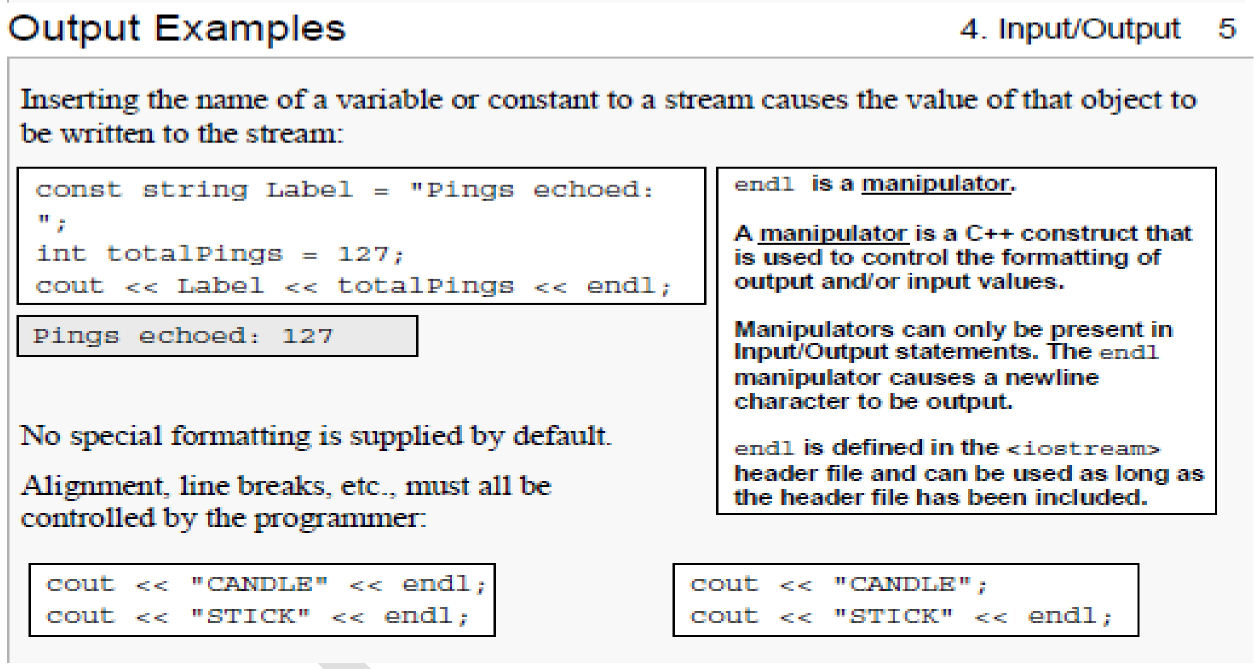
In order to do output, we merely use a statement like:

```
cin >> X;
```

Hint: the extraction operator (>>) points in the direction the data is flowing.

where X is the name of some variable that we want to store the value that will be read from the keyboard.

As with the insertion operator, extractions from an input stream can also be "chained". The left-most side must be the name of an input stream variable.



Output Examples

4. Input/Output 5

Inserting the name of a variable or constant to a stream causes the value of that object to be written to the stream:

```
const string Label = "Pings echoed:
";
int totalPings = 127;
cout << Label << totalPings << endl;
```

```
Pings echoed: 127
```

`endl` is a manipulator.

A manipulator is a C++ construct that is used to control the formatting of output and/or input values.

Manipulators can only be present in Input/Output statements. The `endl` manipulator causes a newline character to be output.

`endl` is defined in the `<iostream>` header file and can be used as long as the header file has been included.

No special formatting is supplied by default.

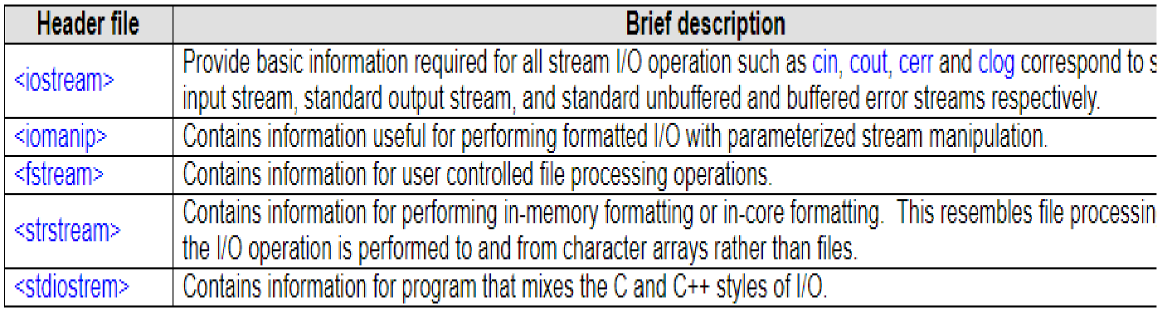
Alignment, line breaks, etc., must all be controlled by the programmer:

```
cout << "CANDLE" << endl;
cout << "STICK" << endl;
```

```
cout << "CANDLE";
cout << "STICK" << endl;
```

18.1 iostream Library

- In C Formatted I/O you have learned the formatted I/O in C by calling various standard functions. In this Module we will discuss how this formatted I/O implemented in C++ by using member functions and stream manipulators.
- If you have completed this C++ Data Encapsulation until C++ Polymorphism, you should be familiar with class object. In C++ we will deal a lot with classes. It is readily available for us to use.
- We will only discuss the formatted I/O here, for file I/O and some of the member functions mentioned in this Module, will be presented in another Module. The discussion here will be straight to the point because some of the terms used in this Module have been discussed extensively in C Formatted I/O.
- The header files used for formatted I/O in C++ are:



Header file	Brief description
<iostream>	Provide basic information required for all stream I/O operation such as cin, cout, cerr and clog correspond to s input stream, standard output stream, and standard unbuffered and buffered error streams respectively.
<iomanip>	Contains information useful for performing formatted I/O with parameterized stream manipulation.
<fstream>	Contains information for user controlled file processing operations.
<strstream>	Contains information for performing in-memory formatting or in-core formatting. This resembles file processin the I/O operation is performed to and from character arrays rather than files.
<stdiostrem>	Contains information for program that mixes the C and C++ styles of I/O.

Table 18.1: iostream library

- The compilers that fully comply with the C++ standard that use the template based header files won't need the **.h** extension. Please refer to Namespaces for more information.
- The iostream class hierarchy is shown below. From the base class ios, we have a derived class:

Class	Brief description
Istream	Class for stream input operation.
Ostream	Class for stream output operation.

Table 18.2: ios derived classes

- So, iostream support both stream input and output. The class hierarchy is shown below.

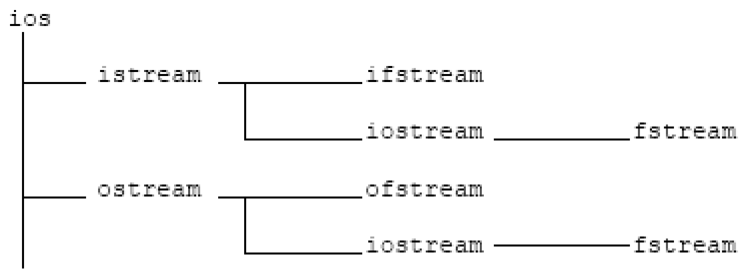


Figure 18.1: ios class hierarchy portion.

18.2 Left and Right Shift Operators

- We have used these operators in most of the previous tutorials for C++ codes.
- The left shift operator (<<) is overloaded to designate stream output and is called **stream insertion operator**.
- The right shift operator (>>) is overloaded to designate stream input and is called **stream extraction operator**.
- These operators used with the standard stream object (and with other user defined stream objects) is listed below:

Operators	Brief description
Cin	Object of istream class, connected to the **standard input device**, normally the keyboard.
Cout	Object of ostream class, connected to **standard output device**, normally the display/screen.
Cerr	Object of the ostream class connected to **standard error device**. This is unbuffered output, so each insertion to cerr causes its output to appear immediately.
Clog	Same as cerr but outputs to clog are buffered.

Table 18.3: iostream operators

- For file processing C++ uses (will be discussed in another Module) the following classes:

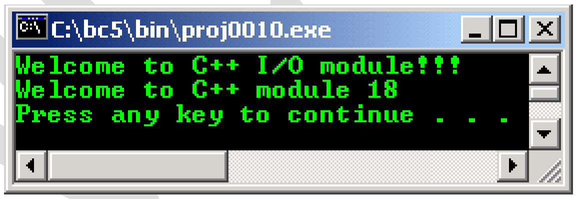
Class	Brief description
ifstream	To perform file input operations.
ofstream	For file output operation.
fstream	For file input/output operations.

Table 18.4: File input/output classes

- Stream output program example:

```
// string output using <<
#include <iostream>
using namespace std;

void main(void)
{
    cout<<"Welcome to C++ I/O module!!!"<<endl;
    cout<<"Welcome to ";
    cout<<"C++ module 18"<<endl;
    // endl is end line stream manipulator
    // issue a new line character and flushes the output buffer
    // output buffer may be flushed by cout<<flush; command
}
```

Output:

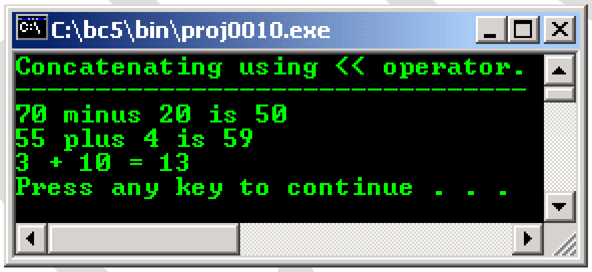


```
Welcome to C++ I/O module!!!
Welcome to C++ module 18
Press any key to continue . . .
```

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

    void main(void)
```

```
            {
                    int p = 3, q = 10;

                    cout << "Concatenating using << operator.\n"
                        <<"--------------------------------"<<endl;
                    cout << "70 minus 20 is "<<(70 - 20)<<endl;
                    cout << "55 plus 4 is "<<(55 + 4)<<endl;
                    cout <<p<<" + "<<q<<" = "<<(p+q)<<endl;
            }
```

Output:



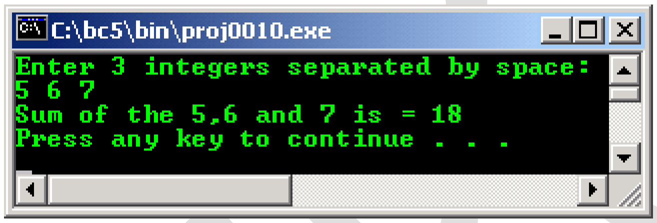
```
C:\bc5\bin\proj0010.exe
Concatenating using << operator.
--------------------------------
70 minus 20 is 50
55 plus 4 is 59
3 + 10 = 13
Press any key to continue . . .
```

- Stream input program example:

```
#include <iostream.h>

using namespace std;


void main(void)

{

        int p, q, r;


        cout << "Enter 3 integers separated by space: \n";

        cin>>p>>q>>r;
```

```
            // the >> operator skips whitespace characters such as tabs,
            // blank space and newline.  When eof is encountered, zero (false) is returned.
            cout<<"Sum of the "<<p<<","<<q<<" and "<<r<<" is = "<<(p+q+r)<<endl;
      }
```

Output:



```
Enter 3 integers separated by space:
5 6 7
Sum of the 5,6 and 7 is = 18
Press any key to continue . . .
```

std::**manipulators**

Stream manipulators

Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects, for example:

```
cout << boolalpha;
```

They are still regular functions and can also be called as any other function using a stream object as argument, for example:

```
boolalpha(cout);
```

Manipulators are used to change formatting parameters on streams and to insert or extract certain special characters.

File Handling using File Streams

File represents storage medium for storing data or information. Streams refer to sequence of bytes. In Files we store data i.e. text or binary data permanently and use these data to read or write in the form of input output operations by transferring bytes of data. So we use the term File Streams/File handling. We use the header file <fstream>

- **ofstream:** It represents output Stream and this is used for writing in files.
- **ifstream:** It represents input Stream and this is used for reading from files.

- **fstream:** It represents both output Stream and input Stream. So it can read from files and write to files.

Operations in File Handling:

- Creating a file: `open()`
- Reading data: `read()`
- Writing new data: `write()`
- Closing a file: `close()`

Creating/Opening a File

We create/open a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating. Syntax for file creation: `FilePointer.open("Path",ios::mode);`

- Example of file opened for writing: `st.open("E:\studytonight.txt",ios::out);`
- Example of file opened for reading: `st.open("E:\studytonight.txt",ios::in);`
- Example of file opened for appending: `st.open("E:\studytonight.txt",ios::app);`
- Example of file opened for truncating: `st.open("E:\studytonight.txt",ios::trunc);`

```
#include<iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
   fstream st; // Step 1: Creating object of fstream class
   st.open("E:\studytonight.txt",ios::out); // Step 2: Creating new file
   if(!st) // Step 3: Checking whether file exist
   {
      cout<<"File creation failed";
```

```
    }
    else
    {
        cout<<"New file created";
        st.close(); // Step 4: Closing file
    }
    getch();
    return 0;
}
```

Writing to a File

```
#include <iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
    fstream st; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
    {
        cout<<"New file created";
        st<<"Hello";    // Step 4: Writing to file
        st.close(); // Step 5: Closing file
    }
    getch();
    return 0;
}
```

Here we are sending output to a file. So, we use ios::out. As given in the program, information typed inside the quotes after **"FilePointer <<"** will be passed to output file.

Reading from a File

```
#include <iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
    fstream st; // step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::in);   // Step 2: Creating new file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"No such file";
    }
    else
    {
        char ch;
        while (!st.eof())
        {
            st >>ch;  // Step 4: Reading from file
            cout << ch;   // Message Read from file
        }
        st.close(); // Step 5: Closing file
    }
    getch();
    return 0;
}
```

Here we are reading input from a file. So, we use ios::in. As given in the program, information from the output file is obtained with the help of following syntax **"FilePointer >>variable"**.

Close a File

It is done by FilePointer.close().

```
#include <iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
    fstreamst; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out); // Step 2: Creating new file
    st.close(); // Step 4: Closing file
    getch();
    return 0;
}
```

Special operations in a File

There are few important functions to be used with file streams like:

- tellp() - It tells the current position of the put pointer.

  Syntax: filepointer.tellp()

- tellg() - It tells the current position of the get pointer.

  Syntax: filepointer.tellg()

- seekp() - It moves the put pointer to mentioned location.

Syntax: filepointer.seekp(no of bytes,reference mode)

- seekg() - It moves get pointer(input) to a specified location.

  Syntax: filepointer.seekg((no of bytes,reference point)

- put() - It writes a single character to file.
- get() - It reads a single character from file.

Note: *For seekp and seekg three reference points are passed:*
ios::beg *- beginning of the file*
ios::cur *- current position in the file*
ios::end *- end of the file*

Below is a program to show importance of tellp, tellg, seekp and seekg:

```
#include <iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
   fstream st; // Creating object of fstream class
   st.open("E:\studytonight.txt",ios::out);  // Creating new file
   if(!st) // Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created"<<endl;
      st<<"Hello Friends"; //Writing to file

      // Checking the file pointer position
      cout<<"File Pointer Position is "<<st.tellp()<<endl;
```

```
        st.seekp(-1, ios::cur); // Go one position back from current position

        //Checking the file pointer position
        cout<<"As per tellp File Pointer Position is "<<st.tellp()<<endl;

        st.close(); // closing file
    }
    st.open("E:\studytonight.txt",ios::in);   // Opening file in read mode
    if(!st) //Checking whether file exist
    {
        cout<<"No such file";
    }
    else
    {
        char ch;
        st.seekg(5, ios::beg);  // Go to position 5 from begning.
        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer position
        cout<<endl;
        st.seekg(1, ios::cur); //Go to position 1 from beginning.
        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer position
        st.close(); //Closing file
    }
    getch();
    return 0;
}
```

```
New file created
File Pointer Position is 13
As per tellp File Pointer Position is 12
As per tellg File Pointer Position is 5
As per tellg File Pointer Position is 6
```

C++ File Pointers and Random Access

Every file maintains two pointers called get_pointer (in input mode file) and put_pointer (in output mode file) which tells the current position in the file where reading or writing will takes place. (A file pointer in this context is not like a C++ pointer but it works like a book-mark in a book.). These pointers help attain random access in file. That means moving directly to any location in the file instead of moving through it sequentially.

There may be situations where random access in the best choice. For example, if you have to modify a value in record no 21, then using random access techniques, you can place the file pointer at the beginning of record 21 and then straight-way process the record. If sequential access is used, then you'll have to unnecessarily go through first twenty records in order to reach at record 21.

The seekg(), seekp(), tellg() and tellp() Functions

In C++, random access is achieved by manipulating seekg(), seekp(), tellg() and tellp() functions. The seekg() and tellg() functions allow you to set and examine the get_pointer, and the seekp() and tellp() functions perform these operations on the put_pointer.

The seekg() and tellg() functions are for input streams (ifstream) and seekp() and tellp() functions are for output streams (ofstream). However, if you use them with an fstream object then tellg() and tellp() return the same value. Also seekg() and seekp() work the same way in an fstream object. The most common forms of these functions are :

seekg()	istream & seekg(long);
	istream & seekg(long, seek_dir);
		Form 1
		Form 2

seekp()	ofstream & seekp(long);
	ofstream & seekp(long, seek_dir);
		Form 1
		Form 2

tellg()	long tellg()	
tellp()	long tellp()	

The working of seekg() & seekp() and tellg() & tellp() is just the same except that seekg() and tellg() work for ifstream objects and seekp() and tellp() work for ofstream objects. In the above table, seek_dir takes the definition enum seek_dir { beg, cur, end};.

The seekg() or seekp(), when used according to Form 1, then it moves the get_pointer or put_pointer to an absolute position. Here is an example:

```
ifstream fin;
ofstream fout;
:                           // file opening routine
fin.seekg(30);              // will move the get_pointer (in ifstream) to byte number 30 in the file
fout.seekp(30);             // will move the put_pointer (in ofstream) to byte number 30 in the file
```

When seekg() or seekp() function is used according to Form 2, then it moves the get_pointer or put_pointer to a position relative to the current position, following the definition of seek_dir. Since, seek_dir is an enumeration defined in the header file iostream.h, that has the following values:

```
ios::beg,          // refers to the beginning of the file
ios::cur,          // refers to the current position in the file
ios::end}          // refers to the end of the file
```

Here is an example.

```
fin.seekg(30, ios::beg);         // go to byte no. 30 from beginning of file linked with fin
fin.seekg(-2, ios::cur); // back up 2 bytes from the current position of get pointer
fin.seekg(0, ios::end);          // go to the end of the file
fin.seekg(-4, ios::end);// backup 4 bytes from the end of the file
```

The functions tellg() and tellp() return the position, in terms of byte number, of put_pointer and get_pointer respectively, in an output file and input file.

C++ File Pointers and Random Access Example

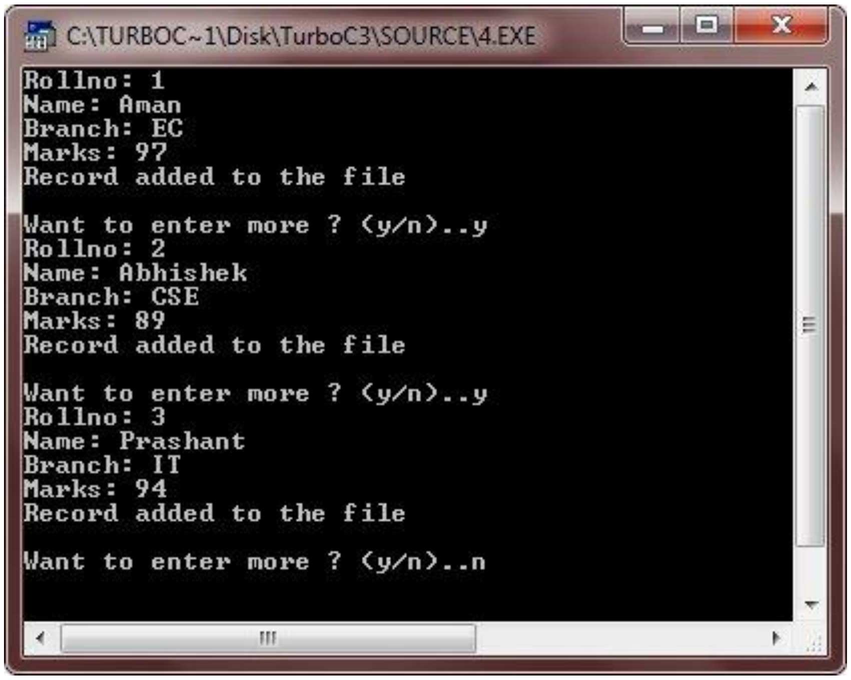
Here is an example program demonstrating the concept of file pointers and random access in a C++ program:

```
/* C++ File Pointers and Random Access
 * This program demonstrates the concept
 * of file pointers and random access in
 * C++ */

#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
```

```
class student
{
        int rollno;
        char name[20];
        char branch[3];
        float marks;
        char grade;

        public:
                void getdata()
                {
                        cout<<"Rollno: ";
                        cin>>rollno;
                        cout<<"Name: ";
                        cin>>name;
                        cout<<"Branch: ";
                        cin>>branch;
                        cout<<"Marks: ";
                        cin>>marks;

                        if(marks>=75)
                        {
                                grade = 'A';
                        }
                        else if(marks>=60)
                        {
                                grade = 'B';
                        }
                        else if(marks>=50)
                        {
                                grade = 'C';
                        }
                        else if(marks>=40)
                        {
                                grade = 'D';
                        }
                        else
                        {
                                grade = 'F';
                        }
                }

                void putdata()
                {
                        cout<<"Rollno: "<<rollno<<"\tName: "<<name<<"\n";
                        cout<<"Marks: "<<marks<<"\tGrade: "<<grade<<"\n";
                }

                int getrno()
                {
                        return rollno;
                }

                void modify();
```

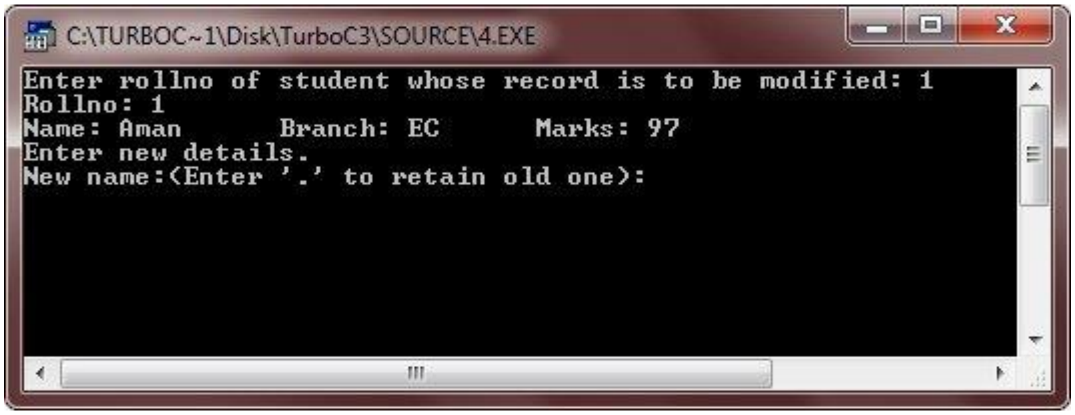
```
}stud1, stud;

void student::modify()
{
        cout<<"Rollno: "<<rollno<<"\n";
        cout<<"Name: "<<name<<"\tBranch: "<<branch<<"\tMarks: "<<marks<<"\n";

        cout<<"Enter new details.\n";
        char nam[20]=" ", br[3]=" ";
        float mks;
        cout<<"New name:(Enter '.' to retain old one): ";
        cin>>nam;
        cout<<"New branch:(Press '.' to retain old one): ";
        cin>>br;
        cout<<"New marks:(Press -1 to retain old one): ";
        cin>>mks;

        if(strcmp(nam, ".")!=0)
        {
                strcpy(name, nam);
        }
        if(strcmp(br, ".")!=0)
        {
                strcpy(branch, br);
        }
        if(mks  != -1)
        {
                marks = mks;
                if(marks>=75)
                {
                        grade = 'A';
                }
                else if(marks>=60)
                {
                        grade = 'B';
                }
                else if(marks>=50)
                {
                        grade = 'C';
                }
                else if(marks>=40)
                {
                        grade = 'D';
                }
                else
                {
                        grade = 'F';
                }
        }
}

void main()
{
        clrscr();
        fstream fio("marks.dat", ios::in | ios::out);
        char ans='y';
        while(ans=='y' || ans=='Y')
        {
                stud1.getdata();
                fio.write((char *)&stud1, sizeof(stud1));
                cout<<"Record added to the file\n";
                cout<<"\nWant to enter more ? (y/n)..";
                cin>>ans;
        }

        clrscr();
        int rno;
        long pos;
        char found='f';

        cout<<"Enter rollno of student whose record is to be modified: ";
        cin>>rno;

        fio.seekg(0);
        while(!fio.eof())
        {
                pos = fio.tellg();
                fio.read((char *)&stud1, sizeof(stud1));
                if(stud1.getrno() == rno)
                {
                        stud1.modify();
                        fio.seekg(pos);
                        fio.write((char *)&stud1, sizeof(stud1));
                        found = 't';
                        break;
                }
        }
        if(found=='f')
        {
                cout<<"\nRecord not found in the file..!!\n";
                cout<<"Press any key to exit...\n";
                getch();
                exit(2);
        }

    fio.seekg(0);
    cout<<"Now the file contains:\n";
    while(!fio.eof())
    {
                fio.read((char *)&stud, sizeof(stud));
                stud.putdata();
    }
    fio.close();
    getch();
}
```

Here are the sample runs of the above C++ program:



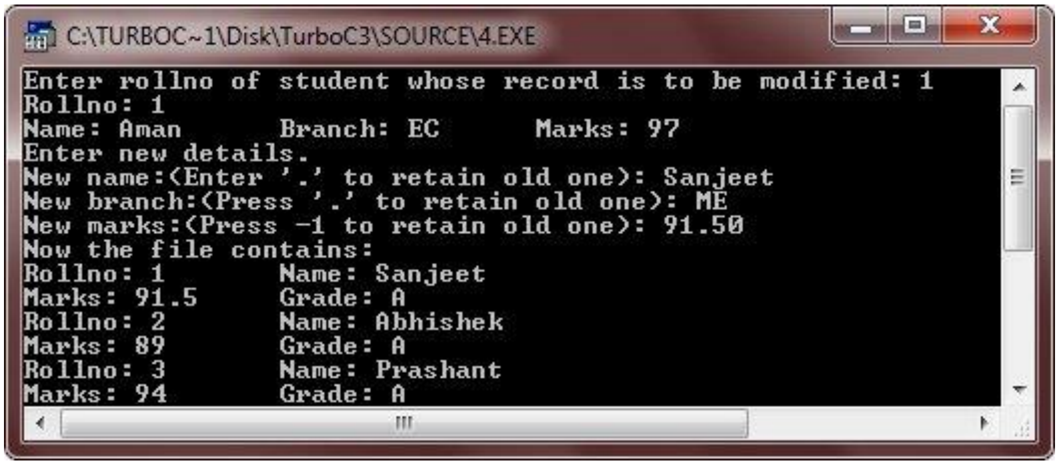
```
Rollno: 1
Name: Aman
Branch: EC
Marks: 97
Record added to the file

Want to enter more ? (y/n)..y
Rollno: 2
Name: Abhishek
Branch: CSE
Marks: 89
Record added to the file

Want to enter more ? (y/n)..y
Rollno: 3
Name: Prashant
Branch: IT
Marks: 94
Record added to the file

Want to enter more ? (y/n)..n
```

After entering the 3 records, just press n, then press enter. After performing this as shown above, press ENTER and here are the sample runs after pressing the ENTER button:



```
Enter rollno of student whose record is to be modified: 1
Rollno: 1
Name: Aman      Branch: EC      Marks: 97
Enter new details.
New name:(Enter '.' to retain old one):
```

Now enter the roll number of that student whose record is to be modified. Here we enter 1, and then enter the new information for that roll number as shown here in the above and below outputs:



```
Enter rollno of student whose record is to be modified: 1
Rollno: 1
Name: Aman        Branch: EC      Marks: 97
Enter new details.
New name:(Enter '.' to retain old one): Sanjeet
New branch:(Press '.' to retain old one): ME
New marks:(Press -1 to retain old one): 91.50
Now the file contains:
Rollno: 1       Name: Sanjeet
Marks: 91.5     Grade: A
Rollno: 2       Name: Abhishek
Marks: 89       Grade: A
Rollno: 3       Name: Prashant
Marks: 94       Grade: A
```

Introduction to OBJECT SERIALIZATION

The C++ language provides a somewhat limited support for file processing. This is probably based on the time it was conceived and put to use. Many languages that were developed after C++, such as (Object) Pascal and Java provide a better support, probably because their libraries were implemented as the demand was made obvious. Based on this, C++ supports saving only values of primitive types such as short, int, char double. This can be done by using either the C FILE structure or C++' own fstream class.

Binary Serialization

Object serialization consists of saving the values that are part of an object, mostly the value gotten from declaring a variable of a class. AT the current standard, C++ doesn't inherently support object serialization. To perform this type of operation, you can use a technique known as binary serialization.

When you decide to save a value to a medium, the **fstream** class provides the option to save the value in binary format. This consists of saving each byte to the medium by aligning bytes in a contiguous manner, the same way the variables are stored in binary numbers.

To indicate that you want to save a value as binary, when declaring the **ofstream** variable, specify the **ios** option as **binary**. Here is an example:

```
#include <fstream>

#include <iostream>

using namespace std;



class Student

{

public:

        char  FullName[40];

        char  CompleteAddress[120];

        char  Gender;

        double Age;

        bool  LivesInASingleParentHome;

};



int main()

{

        Student one;



        strcpy(one.FullName, "Ernestine Waller");

        strcpy(one.CompleteAddress, "824 Larson Drv, Silver Spring, MD 20910");

        one.Gender = 'F';

        one.Age = 16.50;
        one.LivesInASingleParentHome = true;


        ofstream ofs("fifthgrade.ros", ios::binary);


        return 0;
}
```

Writing to the Stream

The **ios::binary** option lets the compiler know how the value will be stored. This declaration also initiates the file. To write the values to a stream, you can call the **fstream::write()**method.

After calling the write() method, you can write the value of the variable to the medium. Here is an example:

```
#include <fstream>

#include <iostream>

using namespace std;


class Student

{

public:

        char  FullName[40];

        char  CompleteAddress[120];

        char  Gender;

        double Age;

        bool  LivesInASingleParentHome;

};


int main()

{

        Student one;


        strcpy(one.FullName, "Ernestine Waller");
        strcpy(one.CompleteAddress, "824 Larson Drv, Silver Spring, MD 20910");

        one.Gender = 'F';

        one.Age = 16.50;

        one.LivesInASingleParentHome = true;


        ofstream ofs("fifthgrade.ros", ios::binary);


        ofs.write((char *)&one, sizeof(one));


        return 0;

}
```

Reading From the Stream

Reading an object saved in binary format is as easy as writing it. To read the value, call the ifstream::read() method. Here is an example:

```
#include <fstream>

#include <iostream>

using namespace std;


class Student

{

public:

        char  FullName[40];

        char  CompleteAddress[120];

        char  Gender;

        double Age;

        bool  LivesInASingleParentHome;

};
```

```
int main()
{
/*      Student one;

        strcpy(one.FullName, "Ernestine Waller");
        strcpy(one.CompleteAddress, "824 Larson Drv, Silver Spring, MD 20910");
        one.Gender = 'F';
        one.Age = 16.50;
        one.LivesInASingleParentHome = true;

        ofstream ofs("fifthgrade.ros", ios::binary);

        ofs.write((char *)&one, sizeof(one));
*/
        Student two;

        ifstream ifs("fifthgrade.ros", ios::binary);
        ifs.read((char *)&two, sizeof(two));

        cout << "Student Information\n";
        cout << "Student Name: " << two.FullName << endl;
        cout << "Address:    " << two.CompleteAddress << endl;
        if( two.Gender == 'f' || two.Gender == 'F' )
                cout << "Gender:     Female" << endl;
        else if( two.Gender == 'm' || two.Gender == 'M' )
                cout << "Gender:     Male" << endl;
        else
```

```
                cout << "Gender:      Unknown" << endl;
        cout << "Age:        " << two.Age << endl;
        if( two.LivesInASingleParentHome == true )
                cout << "Lives in a single parent home" << endl;
        else
                cout << "Doesn't live in a single parent home" << endl;

        cout << "\n";

        return 0;
}
```

Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is:

```
namespace identifier
{
entities
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

```
5
3.1416
```

Edi
&
Rur

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.

using

The keyword using is used to introduce a name from a namespace into the current declarative region. For example:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

```
5
2.7183
10
3.1416
```

Edi
&
Rur

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

```
5
10
3.1416
2.7183
```

Edi
&
Rur

In this case, since we have declared that we were using namespace first, all direct uses of x and y without name qualifiers were referring to their declarations in namespace first.

using and using namespace have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

```
5
3.1416
```

Edi
&
Rur

Namespace alias

We can declare alternate names for existing namespaces according to the following format:

namespace new_name = current_name;

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std; statement in all programs that used any entity defined in iostream.

C++ MANIPULATING STRINGS

A string is a sequence of character. As you know that C++ does not support built-in string type, you have used earlier those null character based terminated array of characters to store and manipulate strings. These strings are termed as *C Strings*. It often becomes inefficient performing operations on C strings. Programmers can also define their own string classes with appropriate member functions to manipulate strings. ANSI standard C++ introduces a new class called **string** which is an improvised version of C strings in several ways. In many cases, the strings object may be treated like any other built-in data type. The string is treated as another container class for C++.

C STYLE STRING

The C style string belongs to C language and continues to support in C++ also strings in C are the one-dimensional array of characters which gets terminated by \0 (null character).

This is how the strings in C are declared:

```
char ch[6] = {'H', 'e', 'l', 'l', 'o', '

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

'};
```

Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the \0 at the end of the string when it initializes the array.

String Class in C++

The string class is huge and includes many constructors, member functions, and operators.

Programmers may use the constructors, operators and member functions to achieve the following:

- Creating string objects
- Reading string objects from keyboard
- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string
- Adding objects of string
- Comparing strings
- Accessing characters of a string
- Obtaining the size or length of a string, etc...

Manipulate Null-terminated strings

C++ supports a wide range of functions that manipulate null-terminated strings. These are:

- **strcpy(str1, str2)**: Copies string str2 into string str1.
- **strcat(str1, str2)**: Concatenates string str2 onto the end of string str1.
- **strlen(str1)**: Returns the length of string str1.
- **strcmp(str1, str2)**: Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2.
- **strchr(str1, ch)**: Returns a pointer to the first occurrence of character ch in string str1.
- **strstr(str1, str2)**: Returns a pointer to the first occurrence of string str2 in string str1.

Important functions supported by String Class

- **append()**: This function appends a part of a string to another string
- **assign()**:This function assigns a partial string
- **at()**: This function obtains the character stored at a specified location
- **begin()**: This function returns a reference to the start of the string
- **capacity()**: This function gives the total element that can be stored
- **compare()**: This function compares a string against the invoking string
- **empty()**: This function returns true if the string is empty
- **end()**: This function returns a reference to the end of the string
- **erase()**: This function removes character as specified
- **find()**: This function searches for the occurrence of a specified substring
- **length()**: It gives the size of a string or the number of elements of a string
- **swap()**: This function swaps the given string with the invoking one

Important Constructors obtained by String Class

- **String()**: This constructor is used for creating an empty string
- **String(const char *str)**: This constructor is used for creating string objects from a null-terminated string
- **String(const string *str)**: This constructor is used for creating a string object from another string object

Operators used for String Objects

1. =: assignment
2. +: concatenation
3. ==: Equality
4. !=: Inequality
5. <: Less than
6. <=: Less than or equal
7. >: Greater than
8. >=: Greater than or equal
9. []: Subscription
10. <<: Output
11. >>: Input

The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of template classes is a prerequisite for working with STL.

STL has four components

- Algorithms
- Containers
- Functions
- Iterators

Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations
- Numeric
  - valarray class

Containers

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
  - vector
  - list
  - deque
  - arrays
  - forward_list(Introduced in C++11)
- Container Adaptors : provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity).
  - set
  - multiset
  - map
  - multimap

Functions

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

- Functors

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

- Iterators

Utility Library

Defined under <utility header>

- pair

Or

Hope you have already understood the concept of C++ Template which we have discussed earlier. The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components −

Sr.No	Component & Description
1	**Containers** Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
2	**Algorithms** Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
3	**Iterators** Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

We will discuss about all the three C++ STL components in next chapter while discussing C++ Standard Library. For now, keep in mind that all the three components have a rich set of pre-defined functions which help us in doing complicated tasks in very easy fashion.

Let us take the following program that demonstrates the vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows −

Live Demo

```
#include <iostream>

#include <vector>

using namespace std;


int main() {


   // create a vector to store int
```

```
vector<int> vec;

int i;


// display the original size of vec

cout << "vector size = " << vec.size() << endl;


// push 5 values into the vector

for(i = 0; i < 5; i++) {

   vec.push_back(i);

}


// display extended size of vec

cout << "extended vector size = " << vec.size() << endl;


// access 5 values from the vector

for(i = 0; i < 5; i++) {

   cout << "value of vec [" << i << "] = " << vec[i] << endl;

}


// use iterator to access the values

vector<int>::iterator v = vec.begin();

while( v != vec.end()) {

   cout << "value of v = " << *v << endl;

   v++;

}


return 0;
```

```
}
```

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

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
```

Here are following points to be noted related to various functions we used in the above example −

- The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.
- The size( ) function displays the size of the vector.
- The function begin( ) returns an iterator to the start of the vector.
- The function end( ) returns an iterator to the end of the vector.

Completed………

APPENDIX

CONTENT BEYOND THE SYLLABUS

C++ Signal Handling

- Signals are the interrupts which are delivered to a process by the operating system to stop its ongoing task and attend the task for which the interrupt has been generated.
- Signals can also be generated by the operating system on the basis of system or error condition.
- You can generate interrupts by pressing Ctrl+ C on Linux, UNIX, Mac OS X, or Windows system.

There are signals which cannot be caught by the program but there is a following list of signals which you can catch in your program and can take appropriate actions based on the signal.

These signals are defined in <csingnal> header file.

Here are the list of signals along with their description and working capability:

Signals	Description
SIGABRT	(Signal Abort) Abnormal termination of the program, such as a call to abort.
SIGFPE	(Signal floating- point exception) An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
SIGILL	(Signal Illegal Instruction) It is used for detecting an illegal instruction.
SIGINT	(Signal Interrupt) It is used to receipt an interactive program interrupt signal.
SIGSEGV	(Signal segmentation Violation) An invalid access to storage.

SIGTERM	(Signal Termination) A termination request sent to the program.
SIGHUP	(Signal Hang up) Hang Up (POSIX), its report that user's terminal is disconnected. It is used to report the termination of the controlling process.
SIGQUIT	Used to terminate a process and generate a core dump.
SIGTRAP	Trace trap.
SIGBUS	This is a BUS error which indicates an access to an invalid address.
SIGUSR1	User defined signal 1.
SIGUSR2	User defined signal 2.
SIGALRM	Alarm clock, which indicates an access to an invalid address.
SIGTERM	Used for termination. This signal can be blocked, handled, and ignored. Generated by kill command.
SIGCOUNT	This signal sent to process to make it continue.
SIGSTOP	Stop, unblockable. This signal is used to stop a process. This signal cannot be handled, ignored or blocked.

The signal() Function

C++ signal-handling library provides function signal to trap unexpected interrupts or events.

Syntax

```
1.  void (*signal (int sig, void (*func)(int)))(int);
```

Parameters

This function is set to handle the signal.

It specifies a way to handle the signals number specified by ***sig***.

Parameter ***func*** specifies one of the three ways in which a signal can be handled by a program.

- **Default handling (SIG_DFL):** The signal handled by the default action for that particular signal.
- **Ignore Signal (SIG_IGN):** The signal is ignored and the code execution will continue even if not purposeful.
- **Function handler:** A particular function is defined to handle the signal.

We must keep in mind that the signal that we would like to catch must be registered using a signal function and it must be associated with a signal handling function.

Note: The signal handling function should be of the void type.

Return value

The return type of this function is the same as the type of parameter func.

If the request of this function is successful, the function returns a pointer to the particular handler function which was in charge of handling this signal before the call, if any.

Data Races

Data race is undefined. If you call this function in a multi- threaded program then it will cause undefined behavior.

Exceptions

This function never throws exception.

Example 1

Let's see a simple example to demonstrate the use of signal() function:

```
#include <iostream>
#include <csignal>

using namespace std;

sig_atomic_t signalled = 0;

void handler(int sig)
{
  signalled = 1;
}

int main()
{
  signal(SIGINT, handler);

  raise(SIGINT);
  if (signalled)
    cout << "Signal is handled";
  else
    cout << "Signal is not handled";

  return 0;
}
```

Output:

```
Signal is handled
```

Example 2

Let's see another simple example:

```
#include <csignal>
#include <iostream>
```

```

namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
}

int main()
{
  // Install a signal handler
  std::signal(SIGINT, signal_handler);

  std::cout << "SignalValue: " << gSignalStatus << '\n';
  std::cout << "Sending signal" << SIGINT  << '\n';
  std::raise(SIGINT);
  std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

Output:

```
SignalValue: 0
Sending signal 2
SignalValue: 2
```

The raise() Function

The C++ signal raise() function is used to send signals to the current executing program.

<csignal> header file declared the function raise() to handle a particular signal.

Syntax

```
int raise (int sig);
```

Parameters

sig: The signal number to be sent for handling. It can take one of the following values:

- SIGINT
- SIGABRT
- SIGFPE
- SIGILL
- SIGSEGV
- SIGTERM
- SIGHUP

Return value

On success, it returns 0 and on failure, a non-zero is returned.

Data Races

Concurrently calling this function is safe, causing no data races.

Exceptions

This function never throws exceptions, if no function handlers have been defined with signal to handle the raised signal.

Example 1

Let's see a simple example to illustrate the use of raise() function when SIGABRT is passed:

```
#include <iostream>
#include <csignal>

using namespace std;

sig_atomic_t sig_value = 0;

void handler(int sig)
{
    sig_value = sig;
}

int main()
{
    signal(SIGABRT, handler);
    cout << "Before signal handler is called" << endl;
    cout << "Signal = " << sig_value << endl;
    raise(SIGABRT);
    cout << "After signal handler is called" << endl;
    cout << "Signal = " << sig_value << endl;

    return 0;
}
```

Output:

```
Before signal handler is called
Signal = 0
After signal handler is called
Signal = 6
```

Example 2

Let's see a simple example to illustrate the use of raise() function when SIGINT is passed:

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

sig_atomic_t s_value = 0;
void handle(int signal_)
{
    s_value = signal_;
}

int main()
```

```
{
    signal(SIGINT, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGINT);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 2
```

Example 3

Let's see a simple example to illustrate the use of raise() function when SIGTERM is passed:

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

sig_atomic_t s_value = 0;
void handle(int signal_)
{
    s_value = signal_;
}

int main()
{
    signal(SIGTERM, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGTERM);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 15
```

Example 4

Let's see a simple example to illustrate the use of raise() function when SIGSEGV is passed:

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

sig_atomic_t s_value = 0;
void handle(int signal_)
{
   s_value = signal_;
}

int main()
{
   signal(SIGSEGV, handle);
   cout << "Before called Signal = " << s_value << endl;
   raise(SIGSEGV);
   cout << "After called Signal = " << s_value << endl;
   return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 11
```

Example 5

Let's see a simple example to illustrate the use of raise() function when SIGFPE is passed:

```
#include <csignal>
#include <iostream>
```

```
using namespace std;

sig_atomic_t s_value = 0;
void handle(int signal_)
{
    s_value = signal_;
}

int main()
{
    signal(SIGFPE, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGFPE);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 8
```

2. Multidimensional Arrays in C / C++

Array- Basics

In C/C++, we can define multidimensional arrays in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).
General form of declaring N-dimensional arrays:

data_type array_name[size1][size2]....[sizeN];

data_type: Type of data to be stored in the array.
Here data_type is valid C/C++ data type
array_name: Name of the array
size1, size2,... ,sizeN: Sizes of the dimensions
Examples:
Two dimensional array:

int two_d[10][20];

Three dimensional array:

int three_d[10][20][30];

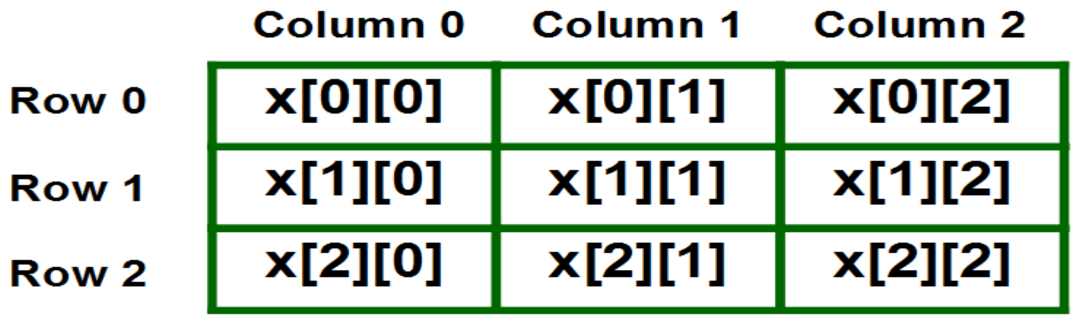
Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
For example:
The array **int x[10][20]** can store total (10*20) = 200 elements.
Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

Two-Dimensional Array

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one – dimensional array for easier understanding.

- The basic form of declaring a two-dimensional array of size x, y:
  Syntax:
- **data_type array_name[x][y];**
- **data_type**: Type of data to be stored. Valid C/C++ data type.
- We can declare a two dimensional integer array say 'x' of size 10,20 as:
- int x[10][20];
- Elements in two-dimensional arrays are commonly referred by x[i][j] where i is the row number and 'j' is the column number.
- A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array 'x' with 3 rows and 3 columns is shown below:



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initializing Two – Dimensional Arrays: There are two ways in which a Two-Dimensional array can be initialized.

First Method:
int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}

The above array have 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in the order, first 4 elements from the left in first row, next 4 elements in second row and so on.

Better Method:
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};

This type of initialization make use of nested braces. Each set of inner braces represents one row. In the above example there are total three rows so there are three sets of inner braces.

Accessing Elements of Two-Dimensional Arrays: Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.
Example:

int x[2][1];

The above example represents the element present in third row and second column.

Note: In arrays if size of array is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is 2+1 = 3.
To output all the elements of a Two-Dimensional array we can use nested for loops. We will require two for loops. One to traverse the rows and another to traverse columns.

filter_none

edit
play_arrow
brightness_4

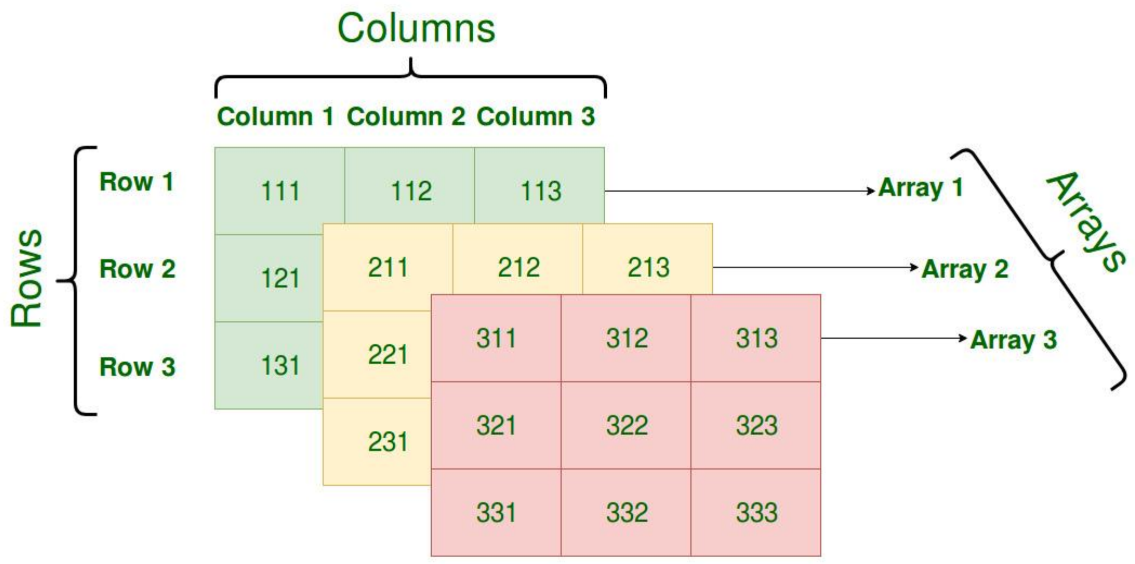
```
// C++ Program to print the elements of a

// Two-Dimensional array

#include<iostream>

using namespace std;



int main()

{

    // an array with 3 rows and 2 columns.

    int x[3][2] = {{0,1}, {2,3}, {4,5}};



    // output each array element's value

    for (int i = 0; i < 3; i++)

    {

        for (int j = 0; j < 2; j++)

        {

            cout << "Element at x[" << i

                 << "][" << j << "]: ";

            cout << x[i][j]<<endl;
```

```
      }

    }



    return 0;

}
```

Output:

```
Element at x[0][0]: 0
Element at x[0][1]: 1
Element at x[1][0]: 2
Element at x[1][1]: 3
Element at x[2][0]: 4
Element at x[2][1]: 5
```

Three-Dimensional Array



Initializing Three-Dimensional Array: Initialization in Three-Dimensional array is same as that of Two-dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15, 16, 17, 18, 19,
                 20, 21, 22, 23};
```

Better Method:

```
int x[2][3][4] =
 {
   { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
   { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
 };
```

Accessing elements in Three-Dimensional Arrays: Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

filter_none

edit
play_arrow
brightness_4

```
// C++ program to print elements of Three-Dimensional

// Array

#include<iostream>

using namespace std;



int main()

{

    // initializing the 3-dimensional array

    int x[2][3][2] =

    {

        { {0,1}, {2,3}, {4,5} },

        { {6,7}, {8,9}, {10,11} }

    };



    // output each element's value

    for (int i = 0; i < 2; ++i)

    {

        for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        cout << "Element at x[" << i << "][" << j
             << "][" << k << "] = " << x[i][j][k]
             << endl;
      }
    }
  }
  return 0;
}
```

Output:

```
Element at x[0][0][0] = 0
Element at x[0][0][1] = 1
Element at x[0][1][0] = 2
Element at x[0][1][1] = 3
Element at x[0][2][0] = 4
Element at x[0][2][1] = 5
Element at x[1][0][0] = 6
Element at x[1][0][1] = 7
Element at x[1][1][0] = 8
Element at x[1][1][1] = 9
Element at x[1][2][0] = 10
```

Element at x[1][2][1] = 11

In similar ways, we can create arrays with any number of dimension. However the complexity also increases as the number of dimension increases.
The most used multidimensional array is the Two-Dimensional Array.

3 Setting up C++ Development Environment

C++ is a general-purpose programming language and widely used nowadays for competitive programming. It has imperative, object-oriented and generic programming features.
C++ runs on lots of platform like Windows, Linux, Unix, Mac, etc. Before we start programming with C++. We will need an environment to be set-up on our local computer to compile and run our C++ programs successfully. If you do not want to set up a local environment you can also use online IDEs for compiling your program.

Using online IDE: IDE stands for integrated development environment. IDE is a software application that provides facilities to a computer programmer for developing software. There are many online IDEs available which you can use to compile and run your programs easily without setting up a local development environment.

```
#include<iostream>

using namespace std;

main()

{

    cout << "Learning C++ at GeekforGeeks";

}
```

Setting up local environment

For setting up your own personal development environment on your local machine you need to install two important softwares:

1. **Text Editor**: Text Editors are type of programs used to edit or write texts. We will use text-editors to type our C++ programs. The normal extension of a text file is (.txt) but a text file containing C++ program should be saved with '.CPP' or '.C' extension. Files ending with the extension '.CPP' and '.C' are called source code files and they are supposed to contain source code written in C++ programming language. These extension helps the compiler to identify that the file contains a C++ program. Before beginning programming with C++, one must have a text-editor installed to write programs.
2. **C++ Compiler**: Once you have installed text-editor and typed and save your program in a file with '.CPP' extension, you will need a C++ compiler to compile this file. A compiler is a computer program which converts high-level language into machine understandable low-level language. In other words, we can say that it converts the source code written in a programming language into another computer language which the computer understands. For compiling a C++ program we will need a C++ compiler which will convert the source code written in C++ into machine codes. Below are the details about setting up compiler on different platforms.
   - **Linux Installation**: We will install the GNU GCC compiler on Linux. To install and work with the GCC compiler on your Linux machine, proceed according to below steps:
     - You have to first run the below two commands from your Linux terminal window:
     - sudo apt-get update
     - sudo apt-get install GCC

       This command will install the GCC compiler on your system. You may also run the below command:

       sudo apt-get install build-essential

       This command will install all the libraries which are required to compile and run a C++ program.
     - After completing the above step, you should check whether the GCC compiler is installed in your system correctly or not. To do this you have to run the below-given command from Linux terminal:
     - g++ --version
     - If you have completed the above two steps without any errors, then your Linux environment is set up and ready to be used to compile C++ programs. In further steps, we will learn how to compile and run a C++

program on Linux using GCC compiler.

- Write your program in a text file and save it with any file name and.CPP extension. We have written a program to display "Hello World" and saved it in a file with the filename "helloworld.cpp" on desktop.
- Now you have to open the Linux terminal and move to the directory where you have saved your file. Then you have to run the below command to compile your file:
- g++ *filename.cpp* -o *any-name*
  filename.cpp is the name of your source code file. In our case, the name is "helloworld.cpp" and *any-name* can be any name of your choice. This name will be assigned to the executable file which is created by the compiler after compilation. In our case, we choose *any-name* to be "hello".
  We will run the above command as:
  g++ helloworld.cpp -o hello
- After executing the above command, you will see a new file is created automatically in the same directory where you have saved the source file and the name of this file is the name you chose as *any-name*.
  Now to run your program you have to run the below command:
- ./hello

  This command will run your program in the terminal window.

- **Windows Installation**: There are lots of IDE available for windows operating system which you can use to work easily with C++ programming language. One of the popular IDE is **Code::Blocks**. To download Code::Blocks you may visit this link. Once you have downloaded the setup file of Code::Blocks from the given link open it and follow the instruction to install.
  - After successfully installing Code::Blocks, go to *File* menu -> Select *New* and *create an Empty* file.
  - Now write your C++ program in this empty file and save the file with a '.cpp' extension.
  - After saving the file with '.cpp' extension, go to *Build* menu and choose the *Build and Run* option.
- **Mac OS X Installation**: If you are a Mac user,you have to download Xcode. To download Xcode you have to visit the apple website or you can search it on apple app store. You may follow the link developer.apple.com/technologies/tools/ to download Xcode. You will find all the necessary install instructions there.
  - After successfully installing Xcode, open the Xcode application.
  - To create a new project. Go to File menu -> select New -> select Project. This will create a new project for you.
  - Now in the next window you have to choose a template for your project. To choose a C++ template choose *Application* option which is under

the *OS X* section on the left side bar. Now choose *command-line tools* from available options and hit *Next* button.

- On the next window provide all the necessary details like ‘name of organisation’, ‘Product Name’ etc. But make sure to choose the language as C++ . After filling the details hit the next button to proceed to further steps.
- Choose the location where you want to save your project. After this choose the *main.cpp* file from the directory list on the left side-bar.
- Now after opening the main.cpp file, you will see a pre written c++ program or template is provided. You may change this program as per your requirement. To run your C++ program you have to go to *Product* menu and choose the *Run* option from the dropdown.

--