f) User defined data

- Distance
- Currency
- Time
- Date
- Complex Number

In computer programming, all these objects of the real world can be modeled by combining data and function together to make object of the program which is not possible in procedure oriented programming.

2. Class

Object consists of data and function tied together in a single unit. Functions are used to manipulate on the data. The entire construct of objects can be represented by a user defined data type in programming. The class is the user defined data type used to declare the objects. Actually objects are the variable of the user defined data type implemented as class. Once a class is defined, we can create any number of objects of its type. Each object that is created from the user defined type implemented as class is associated with the data type of that class. For example, manager, peon, secretary clerk are the objects of the class employee. Similarly, car, bus, jeep, truck are the objects of the class vehicle. Classes are user defined data type (like a struct in C programming language) and behave much like built in data type (like int, char, float) of programming language. It specifies what data and functions will be included in objects of that class. Defining class doesn't create an object; however defining process specifies the data and function to be in the objects of its type.

One of the objects of student can have following values

Name = "Bishal"
Registration_number = 200876255
Marks = {66, 77, 51, 48, 82}

The function Sort_name () will sort and display list of students on the basis of name in alphabetical order. Similarly, function Tot_marks () will sum the marks obtained by the student. The function Percentage_marks() will calculate the percentage and the Decide_division () function will decide division based on percentage obtained by the student.

Each class describes a possibly infinite set of individual objects, each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operations with other instances of the class. The following points give the idea of class:
A class is a template that specifies data and their operations.
A class is an abstraction of the real world entities with similar properties.
Ideally, the class is an implementation of abstract data type.

3. Abstraction

Abstraction is representing essential features of an object without including the background details or explanation. It focuses the outside view of an object, separating its essential behavior from its implementation.

We can manage complexity through abstraction. Let's take an example of vehicle. It is constructed from thousands of parts. The abstraction allows the driver of the vehicle to drive without having detail knowledge of the complexity of the parts. The driver can drive the whole vehicle treating like a single object.

Similarly Operating System like Windows, UNIX provides abstraction to the user. The user can view his files and folders without knowing internal detail of Hard disk like the sector number, track number, cylinder number or head number. Operating System hides the truth about the disk hardware and presents a simple file-oriented interface.

The class is a construct in object oriented programming for creating user-defined data for abstraction. When data and it operation are presented together, the construct is call ADT (Abstract Data Type). In OOP classes are used in creating ADT. For example, a student class can be made and can be available to be used in programs. The programmer can implement the class in creating objects and its manipulation without knowing its implementation. The program can use the function Sort_name() to sort the names in alphabetical order without knowing whether the implementation uses bubble sort, merge sort, quick sort algorithms.

4. Encapsulation

The mechanism of wrapping up of data and function into a single unit is called encapsulation. Because of encapsulation data and its manipulating function can be kept together. We can assume encapsulation as a protective wrapper that prevents the data being accessed by other code defined outside the wrapper. By making use of encapsulation we can easily achieve abstraction.

The purpose of a class is to encapsulate complexity. Each data or function in a class can be marked as private or public. The public interface of a class represents everything that external users of the class may know about the data and function. The private function and data can only be accessed by code that is a member of a class. The code other than member of a class cannot access a private function or data. This insulation of data from direct access by the program is called data hiding. After hiding data by making them private, it then safe from accidental alteration.

The public interface should be carefully designed no to expose too much of the inner working of a class.

5. Inheritance

Inheritance is the process by which objects of one class acquire the characteristics of object of another class. We can use additional features to an existing class without modifying it. This is possible by deriving a new class (derived class) from the existing one (base class).This process of deriving a new class from the existing base class is called inheritance.

It provides the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the existing code. We naturally view the

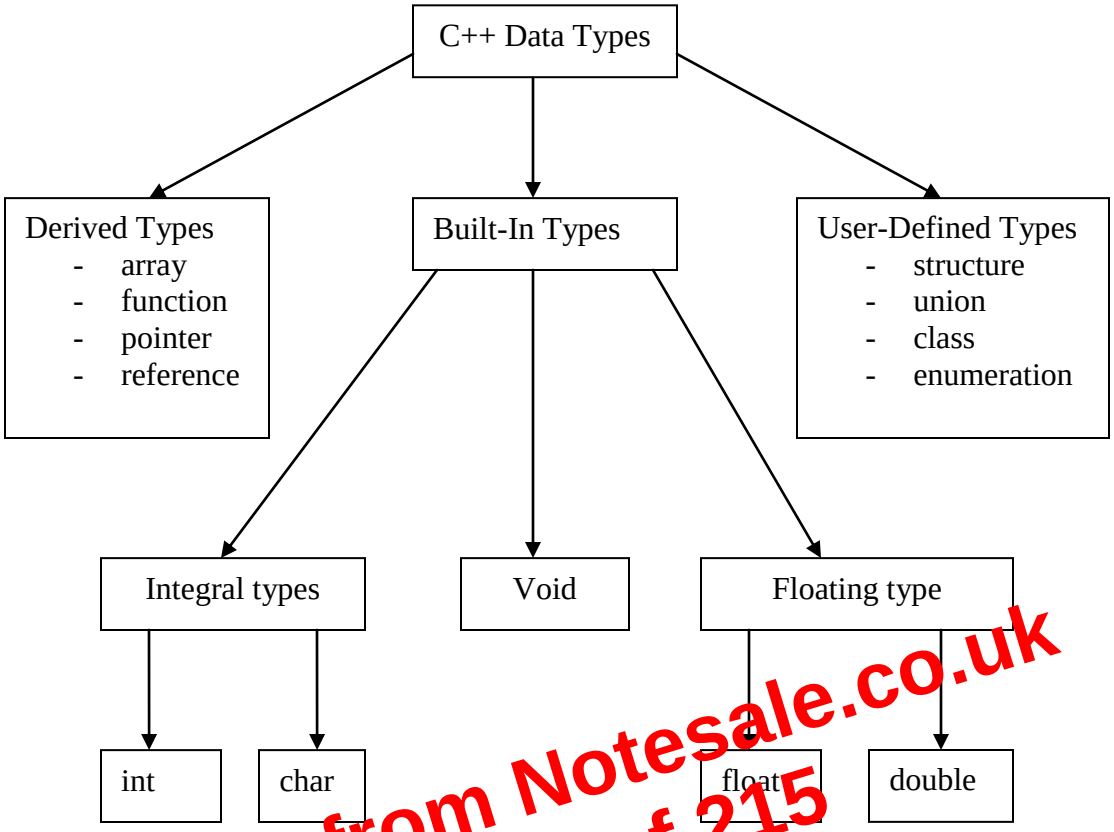


Fig. Hierarchy of C++ Data types

Enumerated Data Type:

Like structures, enumerated data type is another user defined data type. Enumerated means that all the values are listed. They are used when we know a finite list of values that a data type can take on or it is an alternative way for creating symbolic constants. The 'enum' keyword automatically lists a list of words and assign them values 0,1,2…

Eg. enum shape {circle, square, triangle};
enum Boolean {true, false};
enum switch {on, off};

The above example is equivalent to

```
const  circle = 0;
const square = 1;
const triangle = 2;
```

We can even use standard arithmetic operator on enum types. We can also use relational operators when suitable. This is because, the enum data types are internally treated as integers.

Once we specify the data type, we need to define variables of that type.

Eg. shape s1,s2;

It should be noted that, the inline keyword merely sends request, not a command, to a compiler. The compiler may not always accept this request. Some situations where inline expansion may not work are

- for functions having loop, switch or goto statements
- for recursive functions
- functions with static variables
- for functions not returning values, if a return statement exists

Inline functions must be defined before they are called.

Eg.

```
#include<iostream.h>

inline float lbtokg(float lbs)
{
        return (0.453 * lbs);
}


main()
{
        float lbs, kgs;
        cout<<"Enter weight in lbs:";
        cin>>lbs;
        kgs=lbtokg(lbs);
        cout<<"Weight in kg is "<<kgs;
        return (0);
}
```


Exercise:

When do we use inline function? Explain with example.
When do we use default argument? Explain with example.

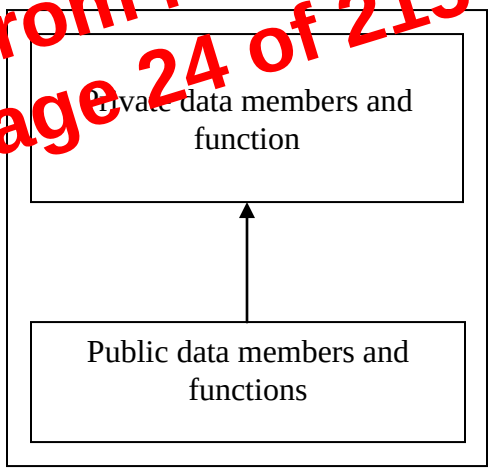
Function Overloading

Function that share the same name are said to be **overloaded functions** and the process is referred to as **function overloading**. i.e. function overloading is the process of using the same name for two or more functions. Each redefinition of a function must use different type of parameters, or different sequence of parameters or different number of parameters. The number, type or sequence of parameters for a function is called the

```
private:
        Variable declarations
        Function declarations
public:
        Variable declarations
        Function declarations

};
```

- the class specification starts with a keyword "class" (like "struct" for structures), followed by a class-name. The class-name is an identifier.
- The body of the class is enclosed within braces and terminated by a semicolon.
- The functions and variables are collectively called class-members. The variables are, specially, called data members while the functions are called member functions.
- The two new keywords inside the above specification are – private and public. Those keywords are termed as **access-specifiers**, they are also termed as **visibility labels**. These are followed by colons.
    - The class members that have been declared as private can be accessed only from within the class. i.e., only the member functions can have access to the private data members and private functions.
    - On the other hand, public members can be accessed from anywhere outside the class (using object and dot operator).



    - All the class members are private by default. So, the keyword "private" is optional.
    - If both the labels are missing, then, by default, all the members will be private and the class will be completely inaccessible by the outsiders (hence it won't be useful at all).

The key feature of OOP is data hiding. Generally, data within a class is made private and the functions are public. So, data will be safe from accidental manipulations, while the functions can be accessed from outside the class. However, it is not always necessary that

```
                                        area=l*b;
        }
        Area(int a)
        {
                                          l=a;
                                         b=0;
                                       area=l*l;
        }
};
```

Destructors

Destructors are the special function that destroys the object that has been created by a constructor. In other words, they are used to release dynamically allocated memory and to perform other "cleanup" activities. Destructors, too, have special name, a class name preceded by a tilde sign (~).

Eg.

A destructor for the class Area will look like

```
                                                    ~Area()
                                                  { - - - - - - -
    - - - - - -
        }
```

Destructor gets invoked, automatically, when an object goes out of scope (i.e. exit from the program, or block or function). They are also declared in the public section. Destructor never takes any argument, nor does it return any value. So, they cannot be overloaded.


```
#include<iostream.h>
#include<conio.h>
class A
{
        static int count;
        public:
        A()
        {
            count++;
            cout<<count<<endl;
        }
        ~A()
        {
            cout<<count<<endl;
            count--;
        }
};

int A::count;

main()
{
```

```
        y = -y;
        z = -z;
}

main()
{
        abc a;
        a.getdata(4,-5,6);
        a.display();
        -a;
        a.display();
        getch();
        return 0;
}
```

Operator Return Values

The operator++() function can return a value. If we use a statement like this

```
c1 = ++c2;
```

For this we have to define the ++ operator to have a return type object of a class in the operator++ function. That is the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator and assign this value to c1. the example given below illustrates this.


```
class Counter
{
        int count;
        public:
        Counter()
        {
                count = 0;
        }
        Counter(int c)
        {
                count = c;
        }

        Counter operator++()
        {
                return Counter(++count);
        }
        void put_count()
        {
                cout<<count<<endl;
```

- One can not alter the precedence of operators
- One can not change the number of operands that an operator takes.

Type Conversion (Data Conversion)

We use the assignment operator (=) to assign value of one variable to another. For example

x = y;

Where x and y are integer variables. We have also noticed that = assigns the value of one user defined object to another, provided that they are of the same type. For example

d2 = d1;

Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler does not need any special instructions to deal with the assignment of user-defined objects such as distance objects.

The assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign.

But if the variables on different sides of the = are of different types, then the type of variable on the right side of = needs to be converted to the type of left side variable before the assignment takes place.

Type conversion is the conversion of one data type to another data type.

Conversion Between Basic Types

Consider the statement,

intvar = floatvar;

where intvar is of type int and floatvar is of type float. Here the compiler will call a special routine to convert the value of floatvar, which is expressed in floating point format, to an integer format so that it can be assigned to intvar. There are many such

```
        float total_value;
        total_value = s;
        cout<<"\nData of s ";
        s.putdata();
        cout<<"Total float value = "<<total_value;
        getch();
        return 0;
}
```

Another example

```
class DistConv
{
        private:
        int kilometers;
        double meters;
        static double kilometersPerMile;
        public:
   // This function converts a built-in type (i.e. miles) to the
   // user-defined type (i.e. DistConv)
        DistConv(double mile)  // Constructor with one argument
        {
                double km = kilometersPerMile * mile ; // converts miles to
                                //kilometers
                kilometers = int(km);          // converts float km to
                                //int and assigns to kilometer

                meters = (km - kilometers) * 1000 ;    // converts to meters
        }
        DistConv(int k, float m)   // constructor with two arguments
        {
                kilometers = k ;
                meters = m ;
        }
   // ********Conversion Function************
   operator double()              // converts user-defined type i.e.
                           // DistConv to a basic-type
   {                         // (double) i.e. meters
      double K = meters/1000 ;       // Converts the meters to
                              // kilometers
      K += double(kilometers) ;       // Adds the kilometers
      return K / kilometersPerMile ;    // Converts to miles
   }
```

Lab Sheet-2

1. Write a program to overload += operator. (You can overload this operator using Distance class as d1 += d2)

2. Write a program to overload = operator (It is already overloaded in C++). Use Distance class to test the program.

3. Write a program to overload ++ operator using friend function.

4. Create a class called **Length** that has data members meter and centimeter. Overload + operator to add two objects of class Length. (For example L3 = L1 + L2). Also facilitate the operations like L4 = L1 + 5 and L5 = 5 + L4 where L1, L2, L3, L4 and L5 are objects of class Length. Use constructors and member functions to initialize and display values.

5. Write a conversion routine in c++ that can convert user-defined data distance to basic data float. Assume that the class distance contains two data members (feet (integer type) and inch (floating point type). NOTE 1-meter = 3.33 feet and 1 feet = 12 inches)

6. Define a class to hold rectangular co-ordinates, i.e. x and y co-ordinates. Let P1 and P2 be the objects of this class where P1 is initialized to (20, 30). Facilitate the operation P2 = P1++ in such a way that the value in P2 is (21, 31) afterward.

7. Write a program to overload + operator to concatenate two strings.

Inheritance

Inheritance is the most powerful feature of object-oriented programming. Inheritance is the process of creating new classes, called derived classes, from existing classes or base classes. The class inherits all the capabilities of the base class but can add refinements of its own.

When new class is created based on some other class using inheritance, the newly created class is called the derived class or sub-class, while the class on which it is based is called base class or superclass.

Inheritance is also called a 'kind of relationship'. Inheritance supports the concept of '**reusability**'. Once a class has been written and tested, its features can be adapted by other programmers whenever required.

Base class

```
class aclass
{
      public:
         void disp(void)
          {
              cout<<"Base"<<endl;
          }
};
class bclass : public aclass
{
      public:
         void disp(void)
          {
               cout<<"Derived"<<endl;
           }
};
void main()
{
      bclass Bvar;
      Bvar.disp();
}
```

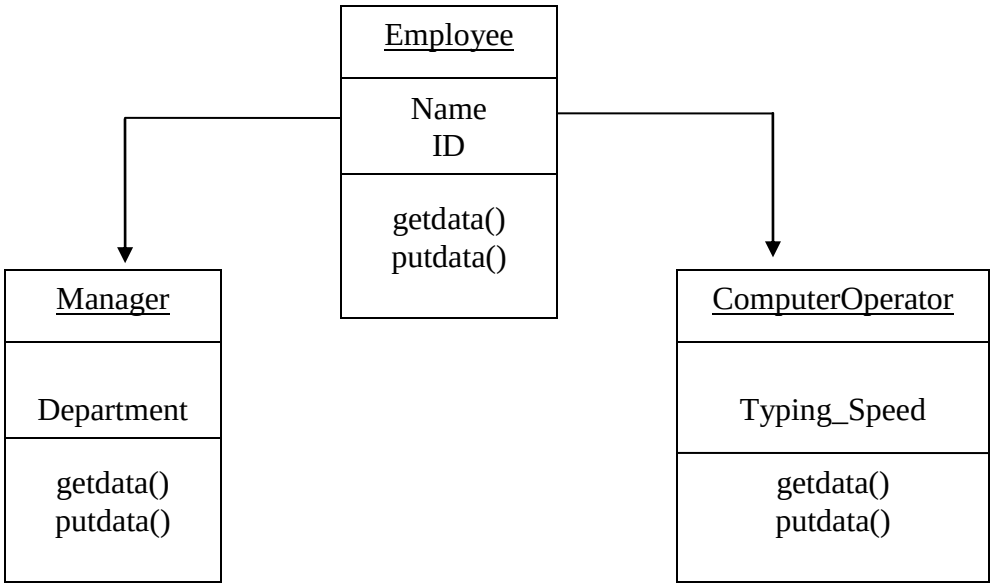
Here, the function disp() is overridden.

- If the function is invoked from an object of the derived class, then the function in the derived is executed.
- If the function is invoked from an object of the base class, then the base class member function is invoked.

Ambiguities in Multiple Inheritance

When a class inherits from multiple base classes, a whole part of ambiguities creep in. for eg- what happens when two base classes contain a function of the same name? Consider the following program,

```
#include<iostream.h>
class base1
{
     public:
         void disp(void)
          {
              cout<<"Base1"<<endl;
           }
};
class base2
{
     public:
```



1. Write a C++ program to represent the above inheritance scheme. Also write a main() function to test the classes, Manager and ComputerOperator, by creating their objects, taking input and displaying the corresponding values.

2. Imagine a college hires some lecturers. Some lecturers are paid in period basis, while others are paid in month basis. Create a class called **lecturer** that stores the **ID**, and the **name of lecturers**. From this class derive two classes: **PartTime**, which adds payperhr (type float), and **FullTime**, which adds paypermonth (type float). Each of these three classes should have a **readdata()** function to get its data from the user, and a **printdata()** function to display its data.

   Write a **main()** program to test the **FullTime** and **PartTime** classes by creating instances of them, asking the user to fill in their data with **readdata()**, and then displaying the data with **printdata()**.

3. An industry seals lorry and taxi. Create a class **Automobile** that stores production date and price. From this class derive another two classes: **Lorry**, which adds weight capacity in kilogram and **Taxi**, which adds seat-capacity in number. Each of these classes should have member functions to get data and set data. Use user-defined constructors to initialize these objects.

4. Create a class called **cricketer** with member variables to represent name, age and no of matches played. From this class derive two classes: **Bowler** and **Batsman**. Bowler class has no_of_wickets as member variable and Batsman class has no_of_runs and centuries as member variables. Use appropriate member functions in all classes to read and display respective data.

5. Define a base class **Shape** having data member radius (int). Derive new classes called **Circle** and **Sphere** from this class. Write methods to compute the area of circle and sphere.

6. Create classes **Book** having data members name of author (string), price (float) and class **Stock** having data members number of books (int) and category (string). Create another class **Library** which derives from both the classes Book and Stock. All the classes should have functions having same name. Write program to test these classes.

Virtual Function and Run Time Polymorphism

Pointers

Pointers have a reputation for being hard to understand. One important use for pointers is in the dynamic allocation of memory, carried out in C++ with the keyword new and its partner delete

Addresses (Pointer Constants)

Every byte in the computer's memory has an address. Addresses are numbers, just as they are for houses on a street.

The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575; for 16 MB of memory, it is 16,777,215.

The Address of Operator &

You can find out the address occupied by a variable by using the address of operator &.

New and Delete Operator

Pointer provides the necessary support for C++ powerful dynamic memory allocation system. Dynamic allocation is the means by which a program can obtain memory while it is running.

For eg- int arr[100];

```
main()
{
        MyClass m1,m2;
        m1.TestThisPointer();
        m2.TestThisPointer();
        getch();
        return 0;
}
```

Accessing Member Data with this

When we call a member function, it comes into existence with the value of **this** set to the address of the object for which it was called. The **this** pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to.

```
class MyClass
{
        int x;
        public:
        void test()
        {
                this->x = 10;
                cout<<this->x;
        }
};
main()
{
        MyClass m;
        m.test();
        getch();
        return 0;
}
```


This program simply prints the value 10. The test() member function accesses the variable **x** as

```
this->x
```

This is exactly the same as referring to **x** directly.

Using this for Returning Values

A more practical use for **this** is in returning values from member functions and overloaded operator.

```
class alpha
{
        int data;
```

```
        public:
        alpha()
        {}
        alpha(int x)
        {
                data = x;
        }
        void display()
        {
                cout<<data<<endl;
        }
        alpha& operator = (alpha &a)
        {
                data = a.data;
                return *this;
        }
};
main()
{
        alpha a1(50);
        alpha a2;
        a2 = a1; // calls overloaded operator =
        a1.display();
        a2.display();
        getch();
        return 0;
}
```



```
//using this pointer for returning value from member function
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
```

```
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        Distance compare(Distance);
};
Distance Distance :: compare(Distance d2)
{
        float dist1 = meter + (float)centimeter/100;
        float dist2 = d2.meter + (float)d2.centimeter/100;
        if(dist1 < dist2)
                return *this;
        else
                return d2;
}
main()
{
        Distance d1(4,50);
        Distance d2,d3,d4;
        d2.getDist();
        d4 = d1.compare(d2);
        cout<<"Small length is ";
        d4.show();
        getch();
        return 0;
}
```

The function put() writes a single character to the associated stream. Similarly, the function get() reads a single character from the associated stream.

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        char ch;
        fstream file;
        file.open("myfile.txt",ios::out); // write only
        cin.get(ch);
        while(ch != '\n')
        {
                file.put(ch);
                cin.get(ch);
        }
        file.close();
        file.open("myfile.txt",ios::in);  // read only
        while(file)
        {
                file.get(ch);
                cout<<ch;
        }
        file.close();
        getch();
}
```


write() and read() Functions

The functions write() and read(), unlike the functions put() and get(), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. For example- an **int** takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit **int** will take four bytes to store it in the character form. **write()** and **read()** functions take two arguments. The first is the address of the variable, and the second is the length of that variable in bytes. The address of the variable must be cast to type char* (pointer to character type).

```
//writing array of integers using write()
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        int x[] = {100,200,300,400};
        int i;
        fstream file;
```

Templates

(Meaning to word: A document or file or entity having a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used)

A template is one of the recently added feature in c++. It supports the generic data types and generic programming. Generic programming is an approach where generic data types are used as parameters in algorithms so that they can work for a variety of suitable data types.

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

A template is a way to specify generic code, with a placeholder for the type. Note that the type is the only "parameter" of a template, but a very powerful one, since anything from a function to a class (or a routine) can be specified in "general" terms without concerning yourself about the specific type.

Templates offer several advantages:

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.

Specifying a class

Specification of a class consists of two parts.

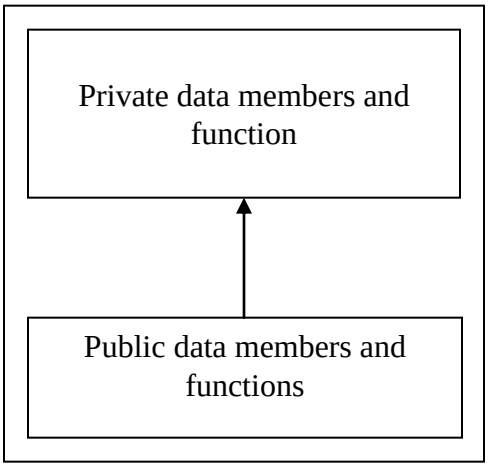
- class declaration
- function definition

Syntax:

```
class class-name
{
    private:
        Variable declarations
        Function declarations
    public:
        Variable declarations
        Function declarations

};
```

- the class specification starts with a keyword "class" (like "struct" for structures), followed by a class-name. The class-name is an identifier.
- The body of the class is enclosed within braces and terminated by a semicolon.
- The functions and variables are collectively called class-members. The variables are, specially, called data members while the functions are called member functions.
- The two new keywords inside the above specification are – private and public. Those keywords are termed as **access specifiers**, they are also termed as **visibility labels**. These are followed by colons.
    - The class members that have been declared as private can be accessed only from within the class, i.e., only the member functions can have access to the private data members and private functions.
    - On the other hand, public members can be accessed from anywhere outside the class (using object and dot operator).



    - All the class members are private by default. So, the keyword "private" is optional.

Outside the class

In this approach, the member functions are **only declared** inside the class, whereas its definition is written outside the class. (As has been done in the previous example of class Employee).

General form:

```
return-type class-name::function-name(argument-list)
{
        - - - - - - - - -
        - - - - - - - -  --  - //function body
         - - - -- - -- --
}
```

The function is, generally, defined immediately after the class-specifier. The function-name is proceeded by

- return-type of the function
- class-name to which the function belongs
- symbol with double colons(::). This symbol is called the scope resolution operator. This operator tells the compiler that the function belongs to the class class-name.

Eg.

```
void Employee::getdata()
{
        - - - - - - - - -
        - - - - - - - -  --  - //function body
         - - - -- - -- --
}
```


In this example, the return-type is void which means the function "getdata()" doesn't return any value. The scope-resolution operator tells that the function "getdata()" is a member of the class "Employee". The argument list is also empty.

The member functions have some special characteristics:

- A program may have several different classes. These classes can use same function name. The scope-resolution operator will resolve which belongs to whom.
- Member functions can directly access private data of that class. A non-member function cannot do so. (Exception is friend function)
- A member function can call another member function directly, without using the dot operator.

Inside the class

Function body can be included in the class itself by replacing function declaration by function definition. If it is done, the function is treated as an inline function. Hence, all the restrictions that apply to inline function, will also apply here.

Eg.

```
class A
{
        int a,b;
        public:
        void getdata()
        {
                cin>>a>>b;
        }
};
```

Here, getdata() is defined inside the class. So, it will act like an inline function.

A function defined outside the class can also be made 'inline' simply by using the qualifier 'inline' in the header line of a function definition.

Eg.

```
class A
{
        - - - - - - - - -
        - - - - - - - - - -
        public:
        void getdata(); // function declaration inside the class
};

inline void A::getdata()
{
        //function body
}
```

Function definition with the 'inline' qualifier/keyword. This qualifier will make the function 'getdata()' an inline function

Nested Member Functions

An object of the class using dot operator generally, calls a member function of a class. However, a member function can be called by using its name inside another member function of the same class. This is known as "Nesting of Member Functions".

Eg.

```
#include<iostream.h>

class Addition
{
        int a,b,sum;
        public:
        void read();
        void show();
        int add();
};
void Addition::read()
```

```
            cin>>n1>>n2;
        }
        friend int average(Avg a);
};
        //here "friend" is a keyword to specify that the function declared is a friend function
        //int is a return type of the friend function
        //average is the name of the friend function
        //Avg is the type of the argument type
        //a is an argument

int average(Avg a) //friend function definition
{
        return((a.n1 + a.n2)/2);
}

int main()
{
        Avg obj;
        obj.getn();
        cout<<"Mean:"<<average(obj);        //friend function called
        return(0);
}
```

Suppose we want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation there is nothing like a friend function. Here is an example that shows how friend functions can act as a bridge between two classes. Read program carefully!!!!!!!!


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

class beta;   // needed for frenfunction declaration

class alpha
{
        private:
        int data;
        public:
        void get_data()
        {
                cin>>data;
        }
        friend int frenfunction(alpha, beta);  // friend function
};
```

```
d4.display();
return(0);
}
```

Constructors are called thrice in this program. The first one d1 calls the parameterized constructor, while the other two calls the copy constructor

The output will be

```
An info stored in d1 5
An info stored in d2 5
An info stored in d3 5
An info stored in d4 5
```

Constructor Overloading

The process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor is defined in a class, it is called constructor overloading.

In the above example of class Data, we have defined three constructors. The first one is invoked when we don't pass any arguments. The second gets invoked when we supply one argument, while the third one gets invoked when an object is passed as an argument.

Eg.

```
Data obj1;
```

This statement would automatically invoke the first one.

```
Data obj2(5); invokes 2nd constructor
Data obj3(obj2); invokes 3rd constructor
```


Constructor with default argument

Like functions, constructors can also have default arguments.

Eg.

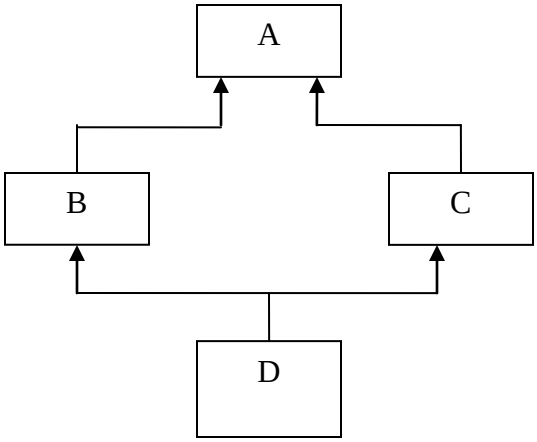
```
class A
{
        int a,b,c;
        pulic:
                A()
                {
                        a=0
                }
                A(int x,int y=10,int z=20)
                {
                        a=x;b=y;c=z;
                }
                void display()
                {
                        cout<<a<<b<<c;
                }
};
```

```
        fstream file;
        file.open("myfile.txt",ios::out); // write only
        cin.get(ch);
        while(ch != '\n')
        {
                file.put(ch);
                cin.get(ch);
        }
        file.close();
        file.open("myfile.txt",ios::in);  // read only
        while(file)
        {
                file.get(ch);
                cout<<ch;
        }
        file.close();
        getch();
}
```

write() and read() Functions

The functions write() and read(), unlike the functions put() and get(), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. For example, an **int** takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit **int** will take four bytes to store it in the character form. **write()** and **read()** functions take two arguments. The first is the address of the variable, and the second is the length of that variable in bytes. The address of the variable must be cast to type char* (pointer to character type).


```
//writing array of integers using write()
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        int x[] = {100,200,300,400};
        int i;
        fstream file;
        file.open("myfile.dat",ios::out|ios::binary);
        file.write((char*)&x,sizeof(x));
        file.close();
        for(i = 0;i < 4;i++)
        x[i] = 0;
        file.open("myfile.dat",ios::in|ios::binary);
        file.read((char*)&x,sizeof(x));
        for(i = 0;i < 4;i++)
        cout<<x[i];
```



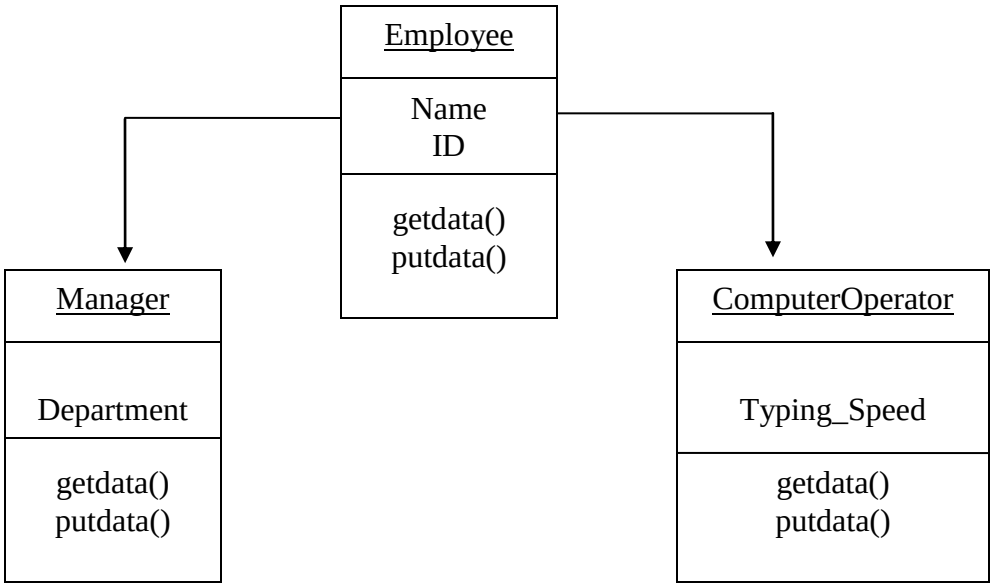
E.g-

```
class student
{
      ……
      …..
};
class test : public student
{
    …….
    …….
};
class sport : public student
{
    ……
    …...
};
class result : public test, public sport
{
   ……
   ….
};
```


In the example, there is one base class student. The class test and sport are derived from student and class result is derived from both test and sport classes. This shows the combination of multiple and hierarchical inheritance.

Sub-class Definition

A subclass can be defined by specifying its relationship with base class along with its own details. General form of defining sub class is



7. Write a C++ program to represent the above inheritance scheme. Also write a main() function to test the classes, Manager and ComputerOperator, by creating their objects, taking input and displaying the corresponding values.

8. Imagine a college hires some lecturers. Some lecturers are paid in period basis, while others are paid in month basis. Create a class called **lecturer** that stores the **ID**, and the **name of lecturers**. From this class derive two classes: **PartTime**, which adds payperhr (type float), and **FullTime**, which adds paypermonth (type float). Each of these three classes should have a **readdata()** function to get its data from the user, and a **printdata()** function to display its data.

   Write a **main()** program to test the **FullTime** and **PartTime** classes by creating instances of them, asking the user to fill in their data with **readdata()**, and then displaying the data with **printdata()**.

9. An industry seals lorry and taxi. Create a class **Automobile** that stores production date and price. From this class derive another two classes: **Lorry**, which adds weight capacity in kilogram and **Taxi**, which adds seat-capacity in number. Each of these classes should have member functions to get data and set data. Use user-defined constructors to initialize these objects.

10. Create a class called **cricketer** with member variables to represent name, age and no of matches played. From this class derive two classes: **Bowler** and **Batsman**. Bowler class has no_of_wickets as member variable and Batsman class has no_of_runs and centuries as member variables. Use appropriate member functions in all classes to read and display respective data.

Namespace

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;
}
```

Using scope resolution operator:
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
```


Example program1

```
#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;
}
```

Operator Overloading

The concept of overloading can be applied to operators as well. Operator overloading is the mechanism of giving special meanings to an operator. It provides a flexible option for the operations of new definitions for most of the C++ operators. In other words, operator overloading refers to giving the normal C++ operators (such as +, *, <=, += etc) additional meanings when they are applied to user-defined data types. In general,

a = b + c; works only with basic types like 'int' and 'float', and attempting to apply it when a, b and c are objects of a user defined class will cause complaints from the compiler. But, using overloading, we can make this statement legal even when a, b and c are user defined types (objects).

There are two types of operator overloading

- Unary operator overloading and
- Binary operator overloading

Unary operator overloading

Unary operators are those operators that act on a single operand. ++, -- are unary operators. The following program overloads the ++ unary operator for the distance class to increment the data number by one.


```
class Distance
{
        int feet;
        float inch;
        public:
        Distance (int f, float i);
        void operator++(void);
        void display();
};
Distance :: Distance (int f, float i)
{
        feet = f ; inch = i;
}
void Distance :: display()
{
        cout<<"Distance in feet"<<feet<<endl;
        cout<<"Distance in inch"<<inch<<endl;
}
void Distance :: operator ++(void)
{
        feet++;
        inch++;
}
void main()
{
```

```
        Distance dist(10,10);
        ++dist;
        dist.display();
}
```

In the above example, the ++ unary operator has been overloaded in the function void Distance :: operator ++(void). In this overloaded function, data members feet and inch are increased by one. This function is called at the second line '++dist' in the main.

General syntax for defining operator overloading

return-type classname :: **operator** operator-to-overload (arg. list)

```
{
        //func body
}
```

The keyword '**operator**' is used to overload an operator. This declaration tells the compiler to call this member function whenever the ++ operator is encountered, provided the operands are of user-defined type.

//Another Example - overloading unary minus operator

```
class abc
{
        int x,y,z;
        public:
        void getdata(int a,int b,int c)
        {
                x = a;
                y = b;
                z = c;
        }
        void display()
        {
                cout<<x<<y<<z<<endl;
        }
        void operator -();
};
void abc::operator-()
{
        x = -x;
        y = -y;
        z = -z;
}

main()
{
        abc a;
        a.getdata(4,-5,6);
```

```
//Overloading == operator to compare two strings
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 20
class string
{
        private:
        char str[SZ];
        public:
        string()
        {
                strcpy(str," ");
        }
        string(char s[])
        {
                strcpy(str,s);
        }
        void getstring()
        {
                cout<<"\nEnter a string ";
                cin>>str;
        }
        int operator ==(string ss)
        {
                return(strcmp(str,ss.str) == 0)? 1:0;
        }
        void display()
        {
                cout<<str<<endl;
        }
};

main()
{
        clrscr();
        string s1 = "Nepal";
        string s2 = "Kathmandu";
        string s3;
        s3.getstring();
        if(s3 == s1)
        cout<<"\nYou typed Nepal";
        else if(s3 == s2)
        cout<<"\nYou typed Kathmandu";
        else
        cout<<"\nNot of both";
```

```
        else
                l2.display();
        getch();
        return 0;
}
```

Multiple Overloading

We have seen several different uses of + operator: - to add distance and to concatenate strings. We can put both these classes together in the same program, and C++ still knows how to interpret the + operator. It selects the correct function to carry out the addition based on the type of operand. Such an example is given below.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 40
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        Distance operator + (Distance);
};
Distance Distance :: operator + (Distance d2)
{
```

Another example

```
class DistConv
{
        private:
        int kilometers;
        double meters;
        static double kilometersPerMile;
        public:
   // This function converts a built-in type (i.e. miles) to the
   // user-defined type (i.e. DistConv)
        DistConv(double mile)  // Constructor with one argument
        {
                double km = kilometersPerMile * mile ;    // converts miles to
                                //kilometers

                kilometers = int(km);           // converts float km to
                                //int and assigns to kilometer

                meters = (km - kilometers) * 1000 ;    // converts to meters
        }
        DistConv(int k, float m)   // constructor with two arguments
        {
                kilometers = k;
                meters = m ;
        }
   // ********Conversion Function**********
   operator double()            // converts user-defined type i.e.
                        // DistConv to a basic-type
   {                     // (double) i.e. meters
      double K = meters/1000 ;       // Converts the meters to
                        // kilometers
      K += double(kilometers) ;       // Adds the kilometers
      return K / kilometersPerMile ;    // Converts to miles
   }

   void display(void)
   {
      cout << kilometers << " kilometers and " << meters << " meters" ;
   }
}; // End of the Class Definition

double DistConv::kilometersPerMile = 1.609344;

int main(void)
{
```

```
        }
};
class Derv2 : public Base
{
        public:
        void show()
        {
                cout<<"Derv2\n";
        }
};
int main()
{
        //Base b; // can't make object of abstract class
        Base *arr[2];
        Derv1 d1;
        Derv2 d2;
        arr[0] = &d1;
        arr[1] = &d2;
        arr[0]->show();
        arr[1]->show();
        getch();
        return 0;
}
```

Output of the program

Derv1

Derv2


```
//Example program: pure virtual function
#include<iostream.h>
#include<conio.h>
class person
{
        protected:
        char name[20];
        public:
        void getName()
        {
                cin>>name;
        }
        void putName()
        {
                cout<<endl<<name;
        }
        virtual void getData() = 0;
        virtual int isOutstanding() = 0;
};
```

Templates

(Meaning to word: A document or file or entity having a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used)

A template is one of the recently added feature in c++. It supports the generic data types and generic programming. Generic programming is an approach where generic data types are used as parameters in algorithms so that they can work for a variety of suitable data types.

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

A template is a way to specify generic code, with a placeholder for the type. Note that the type is the only "parameter" of a template, but a very powerful one, since anything from a function to a class (or a routine) can be specified in "general" terms without concerning yourself about the specific type.

Templates offer several advantages:

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

Templates and Macros

In many ways, templates work like preprocessor macros, replacing the templated variable with the given type. However, there are many differences between a macro like this:

```
#define min(i, j) (((i) < (j)) ? (i) : (j))
```

and a template:

```
template<class T> T min (T i, T j) { return ((i < j) ? i : j) }
```

Here are some problems with the macro:

- There is no way for the compiler to verify that the macro parameters are of compatible types. The macro is expanded without any special type checking.
- The i and j parameters are evaluated twice. For example, if either parameter has a post incremented variable, the increment is performed two times.
- Because macros are expanded by the preprocessor, compiler error messages will refer to the expanded macro, rather than the macro definition itself. Also, the macro will show up in expanded form during debugging.