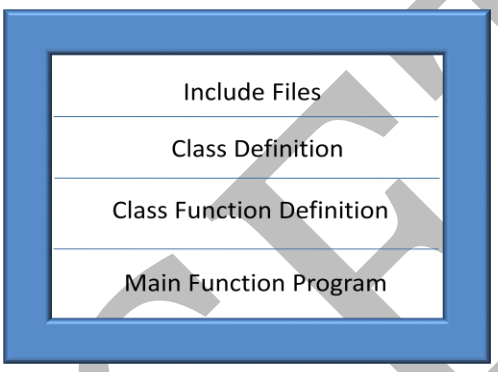
Overview of C++

Basic Introduction:

- C++ is derived from C Language. It is a Superset of C.
- Earlier C++ was known as C with classes.
- C++ was developed by **Bjarne Stroustrup** in the 1980 at Bell Labs.
- Most C Programs can be compiled in C++ compiler.
- C++ expressions are the same as C expressions.

Structure of C++ Program: Layout of C++ Program



Simple C++ Program

```
// Hello World program                    comment

#include <iostream.h>                     Allows access to an I/O library

int main() {                              Starts definition of special function main()

  cout << "Hello World\n";                output (print) a string

  return 0;

}                                         Program returns a status code (0 means OK)
```

Input Statement:

Cin : Extracting Input from User Using Keyboard

Syntax :

cin >> variable;

(The operator >> called as **extraction operator** or **get from operator**)

Example: 1

```
int number1;
int number2;

cout<<"Enter First Number: ";
cin>>number1;

cout<<"Enter Second Number: ";
cin>>number2;
```

Example: 2

```
int a,b;
cin >> a >> b;
```

Output Statement:

Cout: Display Output to User Using Screen(Monitor)

Syntax :

```
cout << variable;
```

Example: 1

```
int tempNumber=6;
cout << tempNumber;
```

Example: 2

```
int a=10;
int b=20;
cout << a << b;
```

Structures

Definition:

A Structure contains one or more data items of different type in which the individual elements can differ in type.

Declaring a structure:

The structure can be declared with the keyword struct following the name and opening brace with data elements of different type then closing brace with semicolon as below

Syntax:

```
struct structure_name
```

```
{
    structure_element 1;
    structure_element 2;
    ……
    structure_element n;
};
 struct structure_name v1,v2,……..vn;
```

Example:

```
 stuct book
       {
         char name[10];
         float price;
         int pages;
        };
        struct book b1,b2,b3;
```

Accessing structure elements

Members of structures are accessed using the member access operators – dot operator (.) and arrow operator(->)

Example:

```
stuct std
       {
         char name[10];
         int no;
         int marks;
        };
        struct std s;
```

For accessing the structure member from the above example

s.no; s.name; s.marks;

Where s is the structure variable

Example Program

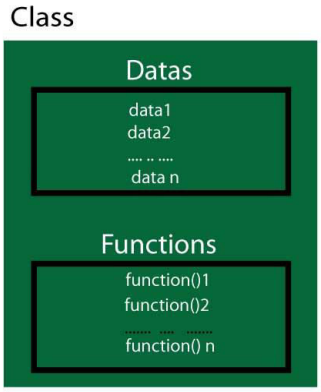
```
#include<iostream.h>
#include<conio.h>
   Struct School
   {
        Int roll;
        Char name [25];
        Char address [25];
        Float weight;
   } stu;
   Void main ()
```

```
{
Cout<<"\n enter the roll no";
Cin>>stu.roll;
Cout<<"\n enter name";
Cin>>stu.name;
Cout<<"\n enter address";
Cin>>stu.address;
Cout<<"\n ether weight";
Cin>>stu.weight;
Cout<<"\n the student details are:";
Cout<<"\n Name :"<< stu.name;
Cout<<"\n Roll no"<<stu.roll;
Cout<<"\n address"<<stu.address;
Cout<<"\n weight"<<stu.weight;
getch ();
}
```

Class

Definition

A **class** is a user **defined** type or data structure declared with keyword **class** that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a **class** is private).



Defining the Class

```
class class_name
  {
 // some data
 // some functions
```

```
};
```

Example of Class

```
class temp
    {
        private:
         int data1;
        float data2;
        public:
        void func1()
        {
        data1=2;
        }
        float func2()
        {
         data2=3.5;
         retrun data;
         }
    };
```

Accessing Class Members

Data members and member functions can be accessed in similar way the member of structure is accessed using member operator(.). For the class and object defined above, *func1()* for object *obj2*can be called using code:

```
obj2.func1();
```

Similary, the data member can be accessed as:

```
object_name.data_memeber;
```

Reference Variables (Object)

When class is defined, only specification for the object is defined. Object has same relationship to class as variable has with the data type. Objects can be defined in similar way as structure is defined.

Syntax to Define Object in C++

```
class_name variable name;
```

For the above defined class *temp*, objects for that class can be defined as:

```
temp obj1,obj2;
```

Constructor

- It is a special member function of a class , which is used to construct the memory of object & provides initialization.
- Its name is same as class name.
- It must be declared in public part otherwise result will be error.
- It does not return any value not even void otherwise result will be error.
- It can be defined by inline /offline method.
- Does not need to call because it get call automatically whenever object is created.
- It can be called explicitly also.
- It can take parameters.
- Constructer can be overloaded.
- It does not inherit.

Types of constructor

i. default constructor
ii. parameterized constructor
iii. default parameterized constructor
iv. copy constructor

There is no type of destructor.

Default constructor

A Default constructor is that will either have no parameters, or all the parameters have default values.

Parameterized constructor

These are the constructors with parameter

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. We will study copy constructors in detail later.

Syntax

```
class_Name (const class_Name &obj)
{
  // body of constructor
}
```

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

class marks
{
public:
   int maths;
   int science;

   //Default Constructor
```

```
    marks(){
      maths=0;
      science=0;
    }

    //Copy Constructor
    marks(const marks &obj){
      maths=obj.maths;
      science=obj.science;
    }

    display(){
      cout<<"Maths :   " << maths
      cout<<"Science : " << science;
    }
};
```

Destructor

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();
 };
```

Example to see how Constructor and Destructor are 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
```

Member Functions and Classes

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.

Example :

```
class Cube

{

 public:

 int side;

 int getVolume();     // Declaring function getVolume with no argument and return type int.

};
```

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube

{

 public:

 int side;

 int getVolume()

 {

  return side*side*side;       //returns volume of cube

 }
```

```
};
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

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

int Cube :: getVolume()     // defined outside class definition
{
 return side*side*side;
}
```

The maine function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

```
int main()
{
 Cube C1;
 C1.side=4;   // setting side value
 cout<< "Volume of cube C1 ="<< C1.getVolume();
}
```

Similarly we can define the getter and setter functions to access private data members, inside or outside the class definition

Friend Function

If a function is defined as a friend function then, the private and protected data of class can be accessed from that function. The complier knows a given function is a friend function by its keyword **friend**. The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend.

```
class class_name
    {
        ..... .... ........
        friend return_type function_name(argument/s);
        ..... .... ........

    }
```

Example to Demonstrate working of friend Function

```
/* C++ program to demonstrate the working of friend function.*/
#include <iostream>
using namespace std;
class Distance
{
   private:
      int meter;
   public:
      Distance(): meter(0){ }
      friend int func(Distance);  //friend function
};
int func(Distance d)          //function definition
{
   d.meter=5;        //accessing private data from non-member function
   return d.meter;
```

```
}
int main()
{
   Distance D;
   cout<<"Distance: "<<func(D);
   return 0;
}
```

Output

```
Distance: 5
```

Dynamic memory

In the programs all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

Operators new and new[]

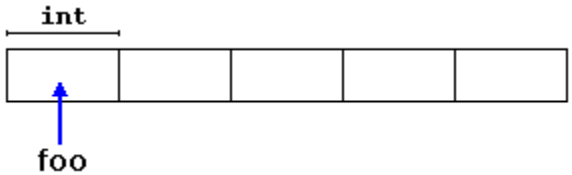
Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

pointer = new type
pointer = new type [number_of_elements]

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

```
1 int * foo;
2 foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.



Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0] or the expression *foo (both are equivalent). The second element can be accessed either with foo[1] or *(foo+1), and so on...

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a *constant expression*, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

C++ provides two standard mechanisms to check if the allocation was successful:

One is by handling exceptions. Using this method, an exception of type bad alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now, you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the method used by default by new, and is the one used in a declaration like:

```
foo = new int [5];  // if allocation fails, an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a *null pointer*, and the program continues its execution normally.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
foo = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

```
1 int * foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4   // error assigning memory. Take measures.
5 }
```

This nothrow method is likely to produce less efficient code than exceptions, since it

implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations. Still, most of the coming examples will use the nothrow mechanism due to its simplicity.

Operators delete and delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

```
1 delete pointer;
2 delete[] pointer;
```

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).

The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a*null pointer* (in the case of a *null pointer*, delete produces no effect).

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type?
";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Edit & Run

```
 }
 return 0;
}
```

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:

```
p= new (nothrow) int[i];
```

There always exists the possibility that the user introduces a value for i so big that the system cannot allocate enough memory for it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program, and I got the text message we prepared for this case (Error: memory could not be allocated).

It is considered good practice for programs to always be able to handle failures to allocate memory, either by checking the pointer value (if nothrow) or by catching the proper exception.

Static class members

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Let us try the following example to understand the concept of static data members:

```
#include <iostream>

using namespace std;

class Box
{
```

```
   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;
```

```
   return 0;
}
```

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

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
```

```
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{

   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;
```

```
   return 0;
}
```

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

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

1.6 *Container Classes and integrators, Proxy Classes*

Container Class

A **container class** is a class designed to hold and organize multiple instances of another class. There are many different kinds of container classes. Commonly used container in programming is the **array**. An array container class instead because of the additional benefits it provides. Unlike built-in arrays, array container classes generally provide dynamically resizing (when elements are added or removed) and do bounds-checking. This not only makes array container classes more convenient than normal arrays, but safer too.

Container classes generally come in two different varieties. **Value containers** are **compositions** that store copies of the objects that they are holding (and thus are responsible for creating and destroying those copies). **Reference containers** are **aggregations** that store pointers or references to other objects (and thus are not responsible for creation or destruction of those objects).

An array container class

In this example, we are going to write an integer array class that implements most of the common functionality that containers should have. This array class is going to be a value container, which will hold copies of the elements its organizing.

First, let's create the IntArray.h file:

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
};

#endif
```

PROXY CLASSES:

It is desirable to hide the implementation details of a class to prevent access to proprietary information and proprietary program logic in a class. Providing clients of the class with a proxy class that k\nows only the public interface to the class enables the clients to use the class services without giving the client access to the class implementation details.

Steps :

Create the class definition and implementation files for the class whose private data need to be hidden.

Overloading: Function overloading

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

Ways to overload a function

1. By changing number of Arguments.
2. By having different types of argument.

Following is the example where same function **print()** is being used to print different data types:

```
#include <iostream>

using namespace std;


class printData
```

```
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }

      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void)
{
   printData pd;
   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");
   return 0;
}
```

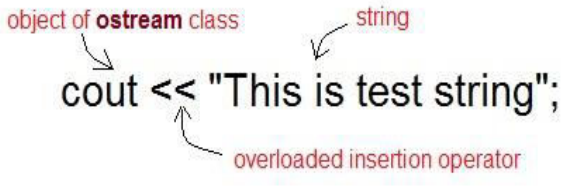
When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Overloading: Operator overloading

Operator Overloading

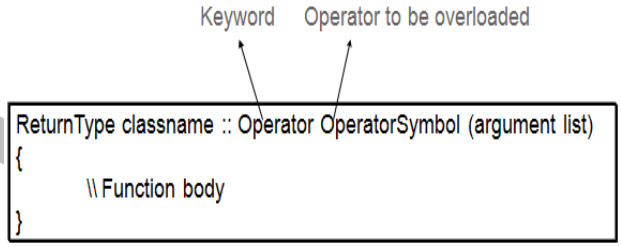
Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.



Almost any operator can be overloaded in C++. However there are few operators which cannot be overloaded. **Operators that are not overloaded** are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Example: overloading '+' Operator to add two time object

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

```
class time
{
 int h,m,s;
 public:
  time()
  {
   h=0, m=0; s=0;
  }
  void getTime();
  void show()
  {
   cout<< h<< ":"<< m<< ":"<< s;
  }
 time operator+(time);   //overloading '+' operator
};
time time::operator+(time t1)     //operator function
{
 time t;
 int a,b;
 a=s+t1.s;
 t.s=a%60;
 b=(a/60)+m+t1.m;
 t.m=b%60;
 t.h=(b/60)+h+t1.h;
 t.h=t.h%12;
 return t;
}
void time::getTime()
{
  cout<<"\n Enter the hour(0-11) ";
  cin>>h;
  cout<<"\n Enter the minute(0-59) ";
  cin>>m;
  cout<<"\n Enter the second(0-59) ";
  cin>>s;
}
void main()
{
 clrscr();
 time t1,t2,t3;
 cout<<"\n Enter the first time ";
 t1.getTime();
 cout<<"\n Enter the second time ";
 t2.getTime();
 t3=t1+t2;    //adding of two time object using '+' operator
 cout<<"\n First time ";
```

```
t1.show();
cout<<"\n Second time ";
t2.show();
cout<<"\n Sum of times ";
t3.show();
getch();
}
```

Overloadable/Non-overloadableOperators:

Following is the list of operators which can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which cannot be overloaded:

::	.*	.	?: