UNIT-1

C++

→ C++ is an object-oriented programming language. It was developed by Bjarne stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA.

→ C++ is an extension of C with a major addition of the class construct feature of simula 67.

→ stroustrup initially called the new language C with classes. However later in 1983 the name was changed to C++.

→ The most important facilities that C++ adds on to C are 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.

Application of C++

→ Since C++ allows us to create hierarchy 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 close 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.

* The operator << is called the insertion or put to operator. In insertion the contents of the variable on its right to the object on its left.

Scanned by CamScanner

The iostream file

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier cout and the operator <<. Some old version of C++ use a header file called iostream.h. This is one of the changes introduced by ANSI C++.

```
# include <iostream>
```

Namespace

This defines a scope for the identifier 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;
```

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 keywords in C++.

* In C++ main() returns an integer type value to the operating system. Therefore every main() in C++ should end with a return(0) statement; otherwise a warning or an error might occur.

```
main()
{
  - - - - - -
  - - - - - -
}
```

* The multiple use of << in one statement is called cascading. When cascading an output operator we should ensure necessary blank spaces between different items.

```
cout << "sum = " << sum << "\n" << "Average = "
                << Average << endl;
```

Class

→ One of the major features of C++ is classes. They provide a method of binding together data and function which operate on them.

→ Classes are user defined data type.

A class is a way to bind the data and its associated functions together. It allows the data to be hidden if necessary from external use. When defining a class we are creating a new abstract data type that can be treated like any other built in data type.

Generally class specification has two parts

- class declaration
- class function definations

→ The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

Syntax

```
class class_name
{
  private:
      variable declarations;
      function declarations;
  public:
      variable declarations;
      function declarations;
};
```

Scanned by CamScanner

→ The class declaration is similar to a struct declaration. The keyword class specifies that what follows is an abstract data of type class_name. The body of a class is enclosed within braces and terminated by a semicolon.

→ class body contains the declaration of variables and functions. These functions and variables are collectively called class members.

→ They are usually grouped under two sections namely private and public to denote which is of the members are private and which of them are public.

→ The keywords private and public are known as visibility labels.

→ The class member that have been declared as private can be accessed only from within the class. on the other hand public members can be accessed from outside the class.

→ The use of the keyword private is optional.

→ The variable declared inside the class are known as data member and the function are known as member function.

→ Only member functions can have access to the private data member and private functions.

→ The binding of data and functions together into a single class type variable is referred to as encapsulation.

Ex

```
class item
{
  int number;   // variable declaration
  float cost;   // private by default

public:
    void getdata (int a, float b); // function declaration
    void putdata (void);

};
```

Accessing class members :-

The private data of a class can be accessed only through the member functions of that class. The main() cannot contain statement that access number and cost directly.

Ex syntax for calling member function.

```
object_name. function_name(actual_arguments);
```

Ex a.getdata (100, 75.5);

Defining Member functions

Member functions can be defined in two places.

- outside the defination
- inside the defination

Outside the defination

Member functions that are declared inside a class have to be defined separately outside the class. Their definations are very much like the normal functions. They should have a function header and a function body.

Syntax

```
return type class_name :: function_name
                    (argument declaration)
{
    function body
}
```

Scanned by CamScanner

Eg

```
Void item :: getdata (int a, float b)
    {
        number = a;
        cost = b;
    }
Void item :: putdata (void)
    {
        cout << "Number : " << number << "\n";
        cout << " cost : " << cost << "\n";
    }
```

The member function have some special characteristics that are often used in the program development. These characteristics are

→ Several different classes can use the same function name. The 'membership label' will resolve their scope.

→ Member functions can access the private data of the class. A non-member function cannot do so.

→ A member function can call another member function directly without using the dot operator.

Inside the class defination :-

Another method of defining a member function is to replace the function declaration by the actual function defination inside the class.

```
class item
{
    int number;
    float cost;
public:
    Void getdata (int a, float b);
    Void putdata (void)
    {
        cout << number << "\n";
        cout << cost << "\n";
    }
};
```

When a function is defined inside a class it is treated as an inline function. Therefore the restrictions and limitations that apply to an inline function are also applicable here. Normally only small functions are defined inside the class definition.

Nesting of Member function :-

A member function of a class can be called only by an object of that class using a dot operator. However there is an exception to this. 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.

Private Member functions

It is normal practice to place all the data item in a private section and all the functions in public. Some situations may require certain functions to be hidden from the outside calls.

→ Task such as alerting an account in a customer file or providing increment to an employee are events of serious consequences and therefore the functions handling to an employee are events of serious such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class.

Scanned by CamScanner

```
class sample
{
  int m;
  void read (void);
public:
    void update (void);
    void write (void);
};
```

Object

→ Object 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. They may also represent user-defined data such as vectors time and lists.

→ Programming problem is analyzed in terms of objects and the nature of communication between them.

→ When a program is executed the object interact by sending messages to one another.

Object : STUDENT

DATA:
- Name
- Date of birth
- Marks

FUNCTIONS
- Total
- Average
- Display

Object as function Arguments

An object may be used as a function argument. This can be done in two ways.
- A copy of the entire object is passed to the function.
- Only the address of the object is transfered to the function.

→ The first method is called pass-by-value. since a copy of the object is passed to the function any changes made to the object inside the function do not affect the ~~ts~~ object used to call the function.

→ The second method is called pass-by-reference. When an address of the object is passed the called function works directly on the actual object used in the call.

→ If Any changes made to the object inside the function will reflect in the actual object.

→ The pass-by-reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class it is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called ~~called~~ constructor because it constructs the values of data members of the class.

Scanned by CamScanner

A constructor is declared and defined as follows

```
class Integer
{
 int m,n;
public:
    integer(void);   // constructor declared
    ------
    ------
    ------
};
integer :: integer(void)  // constructor defined
 {
  m=0; n=0;
 }
```

When a class contain a constructor like the one defined above it guaranteed that an object created by the class will be initialized automatically. For example the declaration

integer int 1; // object int 1 created.

→ A constructor that accepts no parameters is called the default constructor. The default constructor for class A is A::A(). If no such constructor is defined the the compiler supplies a default constructor. Therefore a statement such as

A a;

invokes the default constructor of the compiler to create the object a.

The constructor functions have some special characteristics. These are.

→ They should be declared in the public section.
→ They are invoked automatically when the objects are created.
→ They donot have returnes types. not even void and therefore and they cannot return values.
→ They cannot be inherited, though a derivod class can call the base constructor.
→ Like other C++ functions they can have default arguments.
→ constructors cannot be virtual.
→ We cannot refer to their address
→ An object with a constructor cannot be used as a member of a union.
→ They make 'implicit calls' to the operators new and delete when memory allocation is required.

Parameterized constructor

The construction that can take arguments are called parameterized constructors.

```
class integer
{
 int m,n;
public:
    integer (int x, int y); // parameterized constru.
    ------
    ------
};
integer :: integer (int x, int y)
 {
  m=x; n=y;
 }
```

Scanned by CamScanner

We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways.

- By calling the constructor expicity.
- By calling the constructor implicity.

integer int L = integer (0,100); // explict call
integer int L (0,100); // implicit call

Ex

```
# include <iostream>
using namespace std;
class cube
{
 public :
    int side ;
 cube (int a)
  {
   side = a
  }
};
 int main ()
 {
   cube c1 (10);
   cube c2 (20);
   cout << c1. side;
   cout << c2. side;
 }
```

Defaut construction :-

Defaut construction is the construction which doesn't take any argument. It has no parameter.

Ex

```
class cube
{
 public :
   int side;
  cube ()
    {
     side = 10;
    }
};
 int main()
  {
   cube c;
   cout << c.side;
  }
```

→ A defaut constructor is so important for initialization of subject members, that even if we do not define a constructor expicity.

Copy constructors :-

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.

Constructor overloading :-

Just like other member functions, construction can also be overloaded. Infact when you have both defaut and parameterized constructors defined in your class you are having overloaded constructors. one with parameter and other with no parameter. You can have any number of constructors in a class that differ in parameter list.

→ If you define any constructor expicity then the compiler will not provide defaut constructor and you will have to define it yourself.

Scanned by CamScanner

Ex

```
# include <iostream>
using name_space std;
class complex
 {
 private :
      int real, img;
 public :
      complex () { real=0 ; img=0;}
      complex (int a) { real=a ; img=a;}
      complex (int a, int b) { real=a ; img=b;}
Void add (complex , complex);
Void disc();
  };
complex :: complex (int a)
      {
      real = a
      img = a
      }
complex :: complex (int a , int b)
     {
     real = a;
     img = b;
     }
Void complex :: add (complex c1 , complex c2)
     {
     real = c1.real + c2.real;
     img = c1.img + c2 img;
     }
Void complex :: disc()
   {
   cout << " Addition is = " << real << " + "
          << img << " i " << endl;
   }
int main()
```

```
{
complex ca(2), cb(5,6), cc;
ca.disc();
cb.disc();
cc. add (ca,cb);
cc.disc();
```

copy constructor :-

→ A copy constructor takes a reference to an object of the same class as itself as an argument.

→ A copy constructor is used to declare and initialize an object from another object.

→ The process of initializing through a copy constructor is known as copy initialization.

→ Typical ~~constructor of class~~ declaration of a copy constructor.

→ Forcing a copy constructor to be generated by the compiler.

→ Avoiding implicit generation of the copy constructor.

Ex

```
# include <iostream>
Using namespace std;
class A
  {
   int n;
   A (int n=1) : n (n) { }
   A (const A & a) : n(a.n) { }
  };
Class B : A
  {
    B::B()
    B::B (const & b)
  };
class C:B
  {
   C() : B() { }
```

Scanned by CamScanner

```
private :
    A(int
    c(const c &)
};
int main()
{
    A a1(7);
    A a2(a1);
    B b;
    B b2 = b;
    A a3 = b;
    A va(10);
    C c;
```

Inline Function

→ one of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times.

→ To eliminate the cost of calls to small functions c++ proposes a new features called inline function.

→ It make program execute faster.

→ inline datatype function name (arguments)

→ Inline function is a function that is expanded in line when it is called.

→ Inline function are those functions whose definition is samall and can be submited at the place where its function call.

Criteria for Inline function :-

→ It contains a loop.

→ compiler may not perform inlining in such circumstances.

* If a function contains a loop
* If a function contain static variables
* If a function is recursive
* If a function contains switch or goto statement.
* If a function return type is other than void and the return statement doesn't exist in function body.

Advantages

→ Function call overhead doesn't occur.

→ It also saves the overhead of push/pop variables on the stack when function is called.

→ It also saves overhead of a return call from a function.

→ When you inline a function you may enable compiler to perform context specific optimization on the body of function. Such optimization are not possible for normal functions calls.

→ Inline function may be usefull for embedded systems because inline can yield less code than the function call preamble and return.

disadvantages

→ If you use too many inline functions then the size of the binary executable file will be large because of the duplication of same code.

→ Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

Scanned by CamScanner

→ Inline function may not be useful for many embedded systems. Because the embedded systems code size is more important than speed.

→ Inline function might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

Example

①

```
#include <iostream>
using namespace std;
inline int cube(int s)
  {
    return s*s*s;
  }
int main()
  {
  cout << "The cube of 3 is :"
  return 0;
  }
```

Output : The cube of 3 is : 27

②

```
# include <iostream>
using namespace std;
class Operation
  {
    int a,b, add,sub,mul;
    float div;
public :
    void get();
    void sum();
    void difference();
    void product();
    void division();
  };
inline void operation :: get()
  {
    cout << " Enter first value : ";
    cin >> a;
    cout << " Enter second value :";
    cin >> b;
  }
inline void operation :: sum()
  {
    add = a+b;
    cout << " Addition of two numbers :";
  }
inline void operation :: ~~sub()~~ difference()
  {
  sub = a-b;
  cout << " Difference between two numbers is :";
  }
inline void operation :: product()
  {
    mul = a*b;
    cout << " product of two numbers is :";
  }
inline void operation :: division()
  {
    div = a/b;
    cout << " Division of two numbers is :";
  }

int main()
  {
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    return 0;
```

Scanned by CamScanner

Static class Members :-

→ Static member of a c++ class We can define class members static using static Keywords. 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.

Static Data Members :-

A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a static variable has certain special characteristics are

→ Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

→ It is visible only within the class, but its lifetime is the entire program.

→ It is initialized to zero when the first object of its class is created. No other initialization is permitted.

Static variables are normally used to maintain values common to the entire class. For example a static data member can be used as a counter that records the occurences of all the objects.

ex

```
# include <iostrem>
using namespace std;
class item
{
  static int count;
  int number;
Public:
  void getdata (int a)
  {
    number = a;
    ~~count++;~~
    count++;
  }
  void getcount (void)
  {
    cout << " count : ";
    cout << count << "\n"
  }
};
int item :: count;
int main ()
{
item a, b, c;
a.getcount ();
b.getcount ();
c.getcount ();
a.getdata (100);
b.getdata (200);
c.getdata (300);
cout << " After reading data" << endl;
```

Scanned by CamScanner

```
a.getcount();
b.getcount();
c.get count();
return 0;
}
```

Static Member Function :-

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 function have a class scope and they do not have access to the this pointer of the class.

properties :-

* A static function can have access to only other static members (function or variables) declared in the same class.

* A static member function can be called using class name (instead of its objects)

class name :: function name.

The static function showcount() display the number of objects created till that moment. A count of numbers of objects created is maintained by the static variable count.

→ The function showcount() displays the code number of each object.

Ex

```
# include <iostream>
using namespace std;
class test
{
    int code;
    static int count;
public:
    void setcode()
    {
        code = ++count;
    }
    void showcode()
    {
        cout << "object number :" << code << "\n";
    }
    static void showcount()
    {
        cout << " count :" << count << "\n";
    }
};
int test :: count;
int main()
{
    test t1, t2;
    t1.setcode();
    t2.setcode();
    test :: showcount();
    test t3;
    t3.setcode
    test :: showcount();
    t1.showcode();
    t2.showcode();
    t3.showcode();
    return 0;
}
```

Scanned by CamScanner

This pointer :-

→ Every object in c++ has access to its own address through an important pointer called this pointer.

→ The this pointer is an implicit parameter to all member functions. Therefore inside a member function this may be used to refer to the invoking object.

→ Friend functions do not have a this this pointer because friends are not members of a class. only member functions have a this pointer.

→ compiler supplies an implicit pointer along with function names as 'this'.

→ The this pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all non static function.

→ 'This' pointer is a constant pointer that holds the memory address of the current object.

→ 'this' is unavailable for static member function

```
static void fun2() { cout<<"inside fun2()";
        this→fun1();}
```

Ex

```
# include <iostream>
using namespace std;
class Test
{
private:
    int x;
public:
    void setx(int x)
    {
        this→x=x;
    }
    void display()
    {
        cout<<" x="<<x<<endl;
    }
};
int main()
{
    Test t1;
    int x=20;
    t1.setx(x);
    t1.display();
    return 0;
}
```

Friend Function :-

→ Friend function has privileges to access all private and protected members of the class

→ Friends are not member function.

→ A friend can be a function, function template, or member function, or a class or class template in which case the entire class and all of its members are friends.

→ If the friend function of class X is a member function of class Y. then it must be invoked through an object of class Y.

→ Friend function can access the non-public members of the class only through an object.

→ A friend function is a non-member function of the class that has been granted access to all private members of the class.

→ We simply declare the function within the class by a prefixing its declaration with keyword friend.

Scanned by CamScanner

→ Function defination must not use keyword friend.

→ Defination of friend function is specified outside the class body and is not treated as a part of the class.

→ The major difference between member function and friend function is that the member function is accessed through the object while friend function requires object to be passed as parameter.

→ friend function need not have a 'this' pointer as it does not belong to any object.

→ The friend declaration is unaffected by its location in the class.

→ On one hand, friend functions increase flexibility in programming on the other they are against the principles of OOP.

Advantages of Friend Function :-

→ When two or more classes contain members that are interrelated with the other parts of your program then friend functions prove to be useful.

→ Friend function may be use for in operator overloading.

→ Friend function may facilitate the creation of some type of I/O functions.

Example ①

```
#include <iostream>
using namespace std;
class complex
{
private :
    int real, img;
public :
  complex () { real=0; img=0;}
  complex (int, int)
  getvalue();
  display complex();
friend complex add (complex, complex)
};
complex :: complex (int, int)
  {
    real = x;
    img = y;
  }
complex :: getvalue()
  {
    cout << " Enter real value : ";
    cin >> real;
    cout << " Enter img value : ";
    cin >> img;
  }
complex add( complex c1, complex c2)
  {
    complex c;
    c.img = c1.img + c2.img;
    c.real = c1.real + c2.real;
    return c;
  }
complex :: display complex()
  {
```

Scanned by CamScanner

```
cout << " real << "+" << img << "i" << endl;
 }
int main()
 {
  complex ca, cb, cc;
  ca.getvalue();
  ca.display complex();
  cb.getvalue();
  cb.display complex();
  cc = add (ca, cb);
  cc.display complex();
 }
```

②

```
# include <iostream>
using namespace std;
class Distance
 {
  private:
     int inch, feet;
  public:
     Distance () { inch=0; feet=0;}
     Distance (int x, int y) { inch=x; feet=y;}
  void getdistance();
  friend distance add (Distance, Distance)
  void displaysum();
 };
void Distance :: getdistance();
 {
  cout << " Enter inch :";
  cin >> inch;
  cout << " Enter feet :";
  cin >> feet;
 }
Distance add (Distance d1, Distance d2)
 {
  Distance d;
  d.feet = d1.feet + d2.feet;
  d.inch = d1.inch + d2.inch;
  if (d.inch > 12)
     d.feet = d.feet + (d.inch/12);
     d.inch = d.inch % 12;
     return d;
 }
void Distance :: displaysum()
 {
  cout << " sum is :" << feet << "\t" << inch <<
                                  "\t" << endl;
 }
int main()
 {
  Distance d1, d2, d3;
  d1.getdistance();
  d2.getdistance();
  d3 = add (d1, d2);
  d3.displaysum();
 }
```

Dynamic Memory allocation and deallocation :-

→ C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in better and easier way.

Scanned by CamScanner

→ Dynamic variables are not declared with ordinary variable declarations, they are explicitly allocated and deallocated at run time.

→ Memory allocation is accomplished using the new operator and deallocation is accomplished using the delete operator.

→ When a program requires a variables, it uses new to allocate the variable. When the program no longer needs the variables, it uses delete to deallocate it.

→ The lifetime of a dynamically allocated variable therefore, is the time between the execution of the new and delete statements.

→ Normally, the new operator creates an unintialized variable of the specified type and returns a pointer to it.

→ Dymic variables can be destroyed at any time during program execution. The delete operation used for this purpose

→ When insufficient memory available at the program then the new operator returns NULL pointer.

```
int *p = NULL;   // pointer initialized with NULL
p = new int;
```

Combine declaration of pointer

```
int *p = new int;
```

Ex

```
int *p = new int(25)
```

```
delete [] pointer-variable;
delete p;
```

Scanned by CamScanner