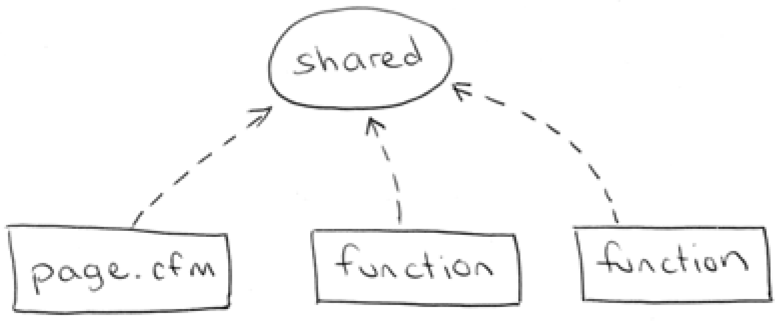
Q.2 a. What is object oriented programming? How does it differ from procedure oriented programming?

Answer:
In procedural programing our code is organised into small "procedures" that use and change our data. In ColdFusion, we write our procedures as either custom tags or functions. These functions typically take some input, do something, then produce some output. Ideally your functions would behave as "black boxes" where input data goes in and output data comes out.

The key idea here is that our functions have no intrinsic relationship with the data they operate on. As long as you provide the correct number and type of arguments, the function will do its work and faithfully return its output.

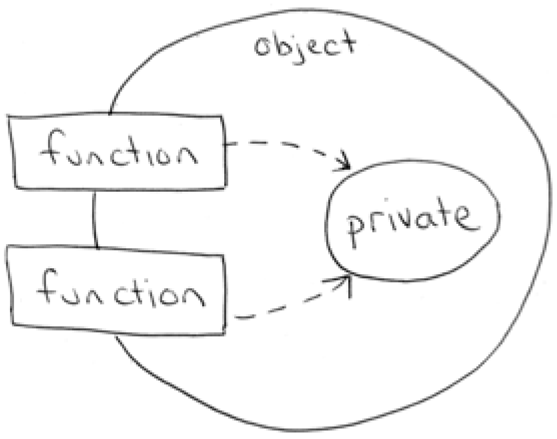
Sometimes our functions need to access data that is not provided as a parameter, i.e., we need access data that is outside the function. Data accessed in this way is considered "global" or "shared" data.



So in a procedural system our functions use data they are "given" (as parameters) but also directly access any shared data they need.

Object oriented programming

In object oriented programming, the data and related functions are bundled together into an "object". Ideally, the data inside an object can only be manipulated by calling the object's functions. This means that your data is locked away inside your objects and your functions provide the only means of doing something with that data. In a well designed object oriented system objects never access shared or global data, they are only permitted to use the data they have, or data they are given.



b. Explain the concept of polymorphism with the help of an example.

Answer: Page Number 16 of Text Book

c. Explain the difference between abstraction and encapsulation.

Answer:
Encapsulation protects abstractions. Encapsulation is the bodyguard; abstraction is the VIP.
Encapsulation provides the explicit boundary between an object's abstract interface (its abstraction) and its internal implementation details. Encapsulation puts the implementation details "in a capsule." Encapsulation tells users which features are stable, permanent services of the object and which features are implementation details that are subject to change without notice.
Encapsulation helps the developers of an abstraction: it provides the freedom to implement the abstraction in any way consistent with the interface. (Encapsulation tells developers exactly what users can and cannot access.) Encapsulation also helps the users of an abstraction: it provides protection by preventing dependence on volatile implementation details.
Abstraction provides business value; encapsulation "protects" these abstractions. If a developer provides a good abstraction, users won't be tempted to peek at the object's internal mechanisms. Encapsulation is simply a safety feature.
Abstraction is achieved by making class abstract having one or more methods abstract. Which is nothing but essential characteristic which should be implemented by the class extending it?
e.g. when you inventing/designing a car you define a characteristics like car should have 4 doors, break, steering wheel etc… so anyone uses this design should include this characteristics. Implementation is not the head each of abstraction. It will just define characteristics which should be included.

Q.3 a. what is the difference between class and structure?

Answer:

Diff b/w Class and Structure is:
Class is defined as-
>Class is a successor of Structure. By default all the members inside the class are private.
>Class is the advanced and the secured form of structure.
>Classes are reference typed.
>Class can be inherited.
>In Class we can initialize the variable during the declaration.
>Class can not be declared without a tag at the first time.
>Class support polymorphism.
Structure defines as:
> In C++ extended the structure to contain functions also. All declarations inside a structure are by default public.
> Structures contain only data while class binds both data and member functions.
> Structure doesn't support the polymorphism, inheritance and initialization.
> Structure is a collection of the different data type.
> Structure is overloaded.
> Structures are value type.
> Structure can be declared without a tag at the first time

b. Write a program that accepts the names and marks of n students in k subjects each. The program should print the total marks of each student. Use array of structures.

Answer:

```
#include<iostream.h>
struct student
{   int rollno;
   char name[50];
   float marks[5];
   float total,per;
}re[100];
int main()
{int a;
cout << "Enter Total student\n";
cin >> a;
for(int i=0; i<a; i++)
{   re[i].total=0;
   cout<<"\n\nEnter your roll no\n";
   cin >> re[i].rollno;
   cout <<"Enter your name\n";
   cin >> (re[i].name);
   cout <<"\n\nEnter your marks\n";
   for(int j=0; j<5; j++)
```

```
  {   cout<<"Enter marks "<<j+1 <<endl;
      cin >>re[i].marks[j];
      re[i].total+=re[i].marks[j];
  }
  re[i].per = re[i].total/5;
}
cout << "Result above 90%\n";
for(int i=0; i <a; i++)
{   if(re[i].per > 90)
  {
      cout << "Name :" << re[i].name<<endl;
      cout << "Roll no :" << re[i].rollno<<endl;
      cout << "Total marks :" << re[i].total<<endl;
      cout << "Percentage :" << re[i].per<<endl;
  }
}
}
```

c. What is the difference between global and local variables?

Answer:
Local
these variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Global
these variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

Q.4 a. What are static data members and static member functions? How do they differ from non static data members and non static member functions?

Answer:
Classes can contain static member data and member functions. When a data member is declared as **static**, only one copy of the data is maintained for all objects of the class. (For more information, see Static Member Functions.)
Static data members are not part of objects of a given class type; they are separate objects. As a result, the declaration of a static data member is not considered a definition. The data member is declared in class scope, but definition is performed at file scope.

b. Write the difference between call by reference and call by value.

Answer:
In C++, call by reference passes a reference to an object (an alias for the original object). Generally this will be implemented as the object's address, though that's not guaranteed.

Call by value means taking a value of some sort, and passing a copy of that value to the function.

The basic difference is that when you pass a parameter by value, the function receives only a *copy* of the original object, so it can't do anything to affect the original object. With pass by reference, it gets a reference to the original object, so it has access to the original object, not a copy of it -- unless it's a const reference, it can modify the original object.

Q.5. a. What is a friend function? What are the merits and demerits of using friend function? Write a program to swap private variables of two classes using friend function.

Answer:
A friend function is a special function in c++ which inspite of not being member fuction of a class has privalage to access private and protected data of a class.

A friend function is a non member function of a class, that is declared as a friend using the keyword "friend" inside the class. By declaring a function as a friend, all the access permissions are given to the function.

A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

Need for Friend Function:

when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

Q.6 a. With the help of an example, explain function overloading and operator overloading.

Answer
Function overloading: C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler.selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types. Operator overloading allows existing C++ operators to be redefined so that they work on objects of user-defined classes. Overloaded operators are syntactic sugar for equivalent function calls. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability and reduce maintenance costs

b. Write a program to add two distances comprising of feet and inches using operator overloading.

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

class distance
{
  int feet;
  float inches;
  public:

   distance()        //constructor1
   {feet=0;inches=0;}
   distance(int ft,float inch)      //constructor2
   {feet=ft;inches=inch;}

   void getdata()
   {   cout<<"Enter Feet and inches respectively: ";
   cin>>feet>>inches;
   }

   void display()
   { cout<<"Feet : "<<feet<<endl<<"Inches :"<<inches;}

   //Operator declaration in Class
   friend distance operator +(distance &ob1, distance &ob2);
   friend distance operator -(distance &ob1, distance &ob2);
   friend distance operator *(int d, distance &ob); //ob1= 2 * ob2
   friend intoperator ==(distance &ob1, distance &ob2);
   friend intoperator < (distance &ob1, distance &ob2);
   friend intoperator > (distance &ob1, distance &ob2);
   friend istream & operator >> (istream &din, distance &ob3);
   friend ostream & operator << (ostream &dout, distance &ob3);
};

distance operator +(distance &ob1, distance &ob2)
{
   distance temp;
   temp.feet   = ob1.feet   + ob2.feet;
   temp.inches = ob1.inches + ob2.inches;
   if(temp.inches > 12)
   {
     temp.inches -= 12;
     temp.feet++;
   }
   return(temp);
}

distance operator -(distance &ob1, distance &ob2)
{
   distance temp;
   float ob1inch,ob2inch;
   ob1inch = (ob1.feet * 12) + ob1.inches;
   ob2inch = (ob2.feet * 12) + ob2.inches;
   temp.inches = ob1inch - ob2inch;
   temp.feet   = temp.inches/12;
   temp.inches = temp.inches - (temp.feet * 12);
   return(temp);
}

distance operator *(int d, distance &ob)
{
   distance temp;
   float i;
   temp.feet   = d * ob.feet;
   temp.inches = d * ob.inches;
   i = temp.inches/12;
   temp.feet   = temp.feet + i;
   temp.inches = temp.inches-(i*12.0);
   return(temp);
}

intoperator ==(distance &ob1, distance &ob2)
{
  if(ob1.feet == ob2.feet && ob1.inches == ob2.inches)
     return(1);
  elsereturn(0);
}
```

```
intoperator < (distance &ob1, distance &ob2)
{
  if(ob1.feet < ob2.feet && ob1.inches < ob2.inches)
    return(1);
  elsereturn(0);
}

intoperator > (distance &ob1, distance &ob2)
{
  if(ob1.feet > ob2.feet && ob1.inches > ob2.inches)
    return(1);
  elsereturn(0);
}

istream & operator >> (istream &din, distance &ob3)
{
   cout<<"\nEnter Data for Object3\n";
   cout<<"Enter Feet   : ";
   din>>ob3.feet;
   cout<<"Enter Inches : ";
   din>>ob3.inches;
   return(din);
}

ostream & operator << (ostream &dout, distance &ob3)
{
   dout<<"\nData of OBJECT3\n";
   dout<<"\nFeet   :"<<ob3.feet;
   dout<<"\nInches :"<<ob3.inches;
   return(dout);
}


void main()
{
 clrscr();
 distance ob1,ob2,ob3;  //Invoked constructor1

 cout<<"\n=====Enter Data for OBJECT1=====\n";
 ob1.getdata();
 cout<<"\n\n=====Enter Data for OBJECT2=====\n";
 ob2.getdata();

 int choice,data;
 while(1)
 {
  up:
  clrscr();

  cout<<"=====Display for OBJECT1=====\n";
    ob1.display();

 cout<<"\n=====Display for OBJECT2=====\n";
    ob2.display();
 cout<<endl;

   cout<<"\nChose your choice\n";
   cout<<"1)  Addition          ( + )\n";
   cout<<"2)  Subtraction        ( - )\n";
   cout<<"3)  Multiplication     ( * )\n";
   cout<<"4)  Comparision        ( == )\n";
   cout<<"5)  Comparision        ( < )\n";
   cout<<"6)  Comparision        ( > )\n";
   cout<<"7)  Input             ( >> )\n";
   cout<<"8)  Output            ( << )\n";
   cout<<"Enter your choice:-";
   cin>>choice;
   cout<<endl<<endl;
   switch(choice)
   {
      case 1 :  ob3 = ob1 + ob2;
        break;
      case 2 :  ob3 = ob1 - ob2;
        break;
      case 3 :    cout<<"\nEnter integer to be multiplied:-";
          cin>>data;
          ob3 = data * ob1;
        break;
      case 4 :  if(ob1 == ob2)
          { cout<<"\nBoth Objects are equal or same value\n";}
        else
          { cout<<"\nThey are Unequal\n";}
        getch();
        goto up;
   case 5 :  if(ob1 < ob2)
          { cout<<"\nObject1 is Less than Object2\n";}
        else
          { cout<<"\nObject2 is Less than Object1\n";}
        getch();
        goto up;
      case 6 :  if(ob1 > ob2)
          { cout<<"\nObject1 is Greater than Object2\n";}
```

```
         else
           { cout<<"\nObject2 is Greater than Object1\n";}
         getch();
         goto up;
       case 7 :  cout<<"\nInputing Data in\n";
         cin>>ob3;
         break;
       case 8 : cout<<"\nOutputing Data out\n";
        cout<<ob3;
        break;
      default :  cout<<"\n\nHave a nice day....\n";
          getch();
          gotoout;
    }
    cout<<"\n\nResult in OBJECT3 as under\n";
    ob3.display();
    getch();
}
out:
}
```

Q7 a. How does the public derivation of a class differ from private and protected derivation?

Answer:
Inheritance is implemented in C++ through the mechanism of derivation. Derivation allows you to derive a class, called a *derived class*, from another class, called a *base class*.

In the declaration of a derived class, you list the base classes of the derived class. The derived class inherits its members from these base classes.

The *qualified_class_specifier* must be a class that has been previously declared in a class declaration.
An *access specifier* is one of `public`, `private`, or `protected`.
The `virtual` keyword can be used to declare virtual base classes.
When you derive a class, the derived class inherits class members of the base class. You can refer to inherited members (base class members) as if they were members of the derived class
The derived class can also add new class members and redefine existing base class members. In the above example, the two inherited members, `a` and `b`, of the derived class `d`, in addition to the derived class member `c`, are assigned values. If you redefine base class members in the derived class, you can still refer to the base class members by using the `:` (scope resolution) operator.

b. When do we make a virtual function "pure"? What are the implications of making a pure virtual function?

Answer:
Only class methods can be rendered as pure-virtual functions (or pure-virtual methods, to be precise). Non-member functions cannot be declared pure-virtual.

The implication of declaring a pure-virtual function is that the class to which it is a member becomes an abstract data type (or abstract base class). This means you cannot instantiate an object of that class, you can only derive classes from it. Moreover, the derived classes must also implement the pure-virtual functions or they, too, become abstract data types. Only concrete classes that contain a complete implementation can be instantiated, although they can inherit implementations from their base classes (but not from the class that initially declared the method).

Pure-virtual functions ensure that you do not instantiate base class objects that are not intended to be instantiated (they are conceptual rather than actual objects) and that derived objects provide a specific implementation. Typically, all methods of an abstract base class will be declared pure-virtual to ensure correct behaviour in derived classes through a common interface. Abstract base classes will also have few member variables, preferably none at all.

For example, a shape is a conceptual object whereas a rectangle, triangle or circle are actual objects. As such they can all be derived from shape. The shape base class need only declare the interface that is common to all shapes, such as Draw(), Rotate(), Mirror() and so on. But since a shape doesn't have enough information to implement these methods, they must be declared pure-virtual. This ensures that shape cannot be instantiated (since it would be meaningless) and that every object derived from shape not only has a common interface, but that each type of shape provides its own specific implementation of those methods. This is not unlike overriding standard virtual functions, however the difference is that you must override the pure-virtual methods; it is not optional unless the derived class is intended to become abstract itself.

Q8. a. Discuss throwing exceptions and catching exceptions in exception handling.

Answer:
Throwing Exceptions:
Exceptions can be thrown anywhere within a code block using throw statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
  if( b == 0 )
  {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions:
The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
  // protected code
}catch( ExceptionName e )
{
 // code to handle ExceptionName exception
}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try
{
  // protected code
}catch(...)
{
 // code to handle any exception
}
```

b. Discuss the difference between class template and function template

Answer:
C++ Function templates are those functions which can handle different data types without separate code for each of them. For a similar operation on several kinds of data types, a programmer need not write different versions by overloading a function. It is enough if we writes a C++ template based function

```
template <class T>
T Add(T a, T b) //C++ function template sample
{
return a+b;
}
```

now t can be int, foat or any other data type.
so using functionn template we can work with many datatypes in one function.
we can call the function by using int x=Add(2,4)
Class Template
a *class template* provides a specification for generating classes based on parameters.For example, the C++ standard library has a list container called `list`, which is a template. The statement `list<int>` designates or instantiates a linked-list of type `int`. The statement `list<string>` designates or instantiates a linked-list of type `string`

Q9 a Write a program that uses a "stock.dat" having item name, item code and cost. Perform the following operations on the file.

(i) Add a new item to the file.
(ii) Modify the details of an item
(iii) Display the contents of the file

Answer:
The following program illustrates a number of operations on a file:
- File creation.
- Modification of records. (In this case the file position indicator is set to the beginning of a
record and the new record is written.)

```
/* Program L10Ex1.c */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <ctype.h>
#include <string.h>
#include <sys/stat.h>
typedef struct
{
char name[60];
float media;
} StudentT;
typedef union
```

```
{
StudentT stud;
char st[sizeof(StudentT)];
} BufferT;
typedef struct
{
int nb;
float media;
} ElementT;
typedef enum {FALSE, TRUE} BooleanT;
void sort(const char filename[], const char sorted_filename[])
{
ElementT el, t[100];
char msgBuf[201];
int i, j, n, fd1, fd2;
BooleanT flag;
BufferT stu;
/* read file to array t */
j = 0;
fd1 = open(filename, O_RDONLY);
while ( read(fd1, stu.st, sizeof(BufferT)) > 0 && j < 100)
{
t[j].nb = j;
t[j].media = stu.stud.media;
j++;
}
n = j;
/* sort array t by media */
j = 0;
do
{
flag = TRUE;
for (i = 1; i < n - j ; i++)
if (t[i-1].media < t[i].media)
{
memcpy(&el, &t[i-1], sizeof(ElementT));
memcpy(&t[i-1], &t[i], sizeof(ElementT));
memcpy(&t[i], &el, sizeof(ElementT));
flag = FALSE;
}
} while (flag == FALSE);
/* write sorted output */
if ((fd2 = open(sorted_filename, O_CREAT | O_WRONLY | O_TRUNC, S_IREAD |
S_IWRITE)) < 0)
{
sprintf(msgBuf, "\nCannot create %s. ", sorted_filename);
perror(msgBuf);
getchar();
exit(1);
}
for (i = 0; i < n; i++)
{
lseek(fd1, (long)(t[i].nb *sizeof(BufferT)), SEEK_SET);
read(fd1, stu.st, sizeof(BufferT));
write(fd2, stu.st, sizeof(BufferT));
}
close(fd1); close(fd2);
}
void show_students(char filename[])
{
BufferT stu;
char s[41];
int j, fd1;
if ((fd1 = open(filename, O_RDONLY)) < 0)
{
sprintf(s, "Cannot open %s for reading. ", filename);
perror(s);
getchar();
exit(3);
}
memset(&stu, '\0', sizeof(BufferT));
j = 0;
while ( read(fd1, stu.st, sizeof(BufferT)) > 0)
{
printf("\n%d. %-60s %7.2f", j++, stu.stud.name, stu.stud.media);
memset(&stu, '\0', sizeof(BufferT));
}
close(fd1);
}
int main(int argc, char *argv[])
{
unsigned int i, n;
int fd1;
long l;

char ch;
BufferT stu;
float avgAux;
char nameAux[sizeof(stu.stud.name)];
char filename[] = "Group.dat";
char sorted_filename[] = "SortedGroup.dat";
printf("\nNumber of students in the group= "); scanf("%u%*c", &n);
```

```
/* create group file */
if ((fd1 = open(filename, O_CREAT | O_WRONLY | O_TRUNC, S_IREAD |
S_IWRITE)) < 0)
{
perror("\nCannot create Group.dat. ");
getchar();
exit(1);
}
```

b.What does the Standard C++ library contain?

Answer:
A1: The Standard C++ library provides an extensible framework and contains components for language support, diagnostics, general utilities, strings, locales, standard template library (containers, iterators, algorithms, and numerics), and input/output.

The Standard C++ library can be divided into the following categories:

1. Standard Template Library (STL) components provide a C++ program with access to a subset of the most widely-used algorithms and data structures. STL headers can be grouped into three major organizing concepts:
   - Containers: template classes that support common ways to organize data, such as <vector>, <list>, <deque>, <stack>, <queue>, <set>, and <map>.
   - Algorithms: template functions for performing common operations on sequences of objects, such as <functional>, <algorithm>, and <numeric>.
   - Iterators: the glue that pastes algorithms and containers together, such as <utility>, <iterator>, and <memory>.

2. Input/Output includes components for forward declarations of iostreams (<iosfwd>), predefined iostreams objects (<iostream>), base iostreams classes (<ios>), stream buffering (<streambuf>), stream formatting and manipulators (<iosmanip>, <istream>, <ostream>), string streams (<sstream>), and file streams (<fstream>).

<u>Text Book</u>

C++ and Object-Oriented Programming Paradigm, Debasish Jana, 2nd Edition, PHI, 2005.