**Q 2 (a)** What do you mean by scope of variable? How variables can be initialized? Explain with example.

Answer

 Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that *a*, *b*, and *result* were   of   type   *int*.
A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.
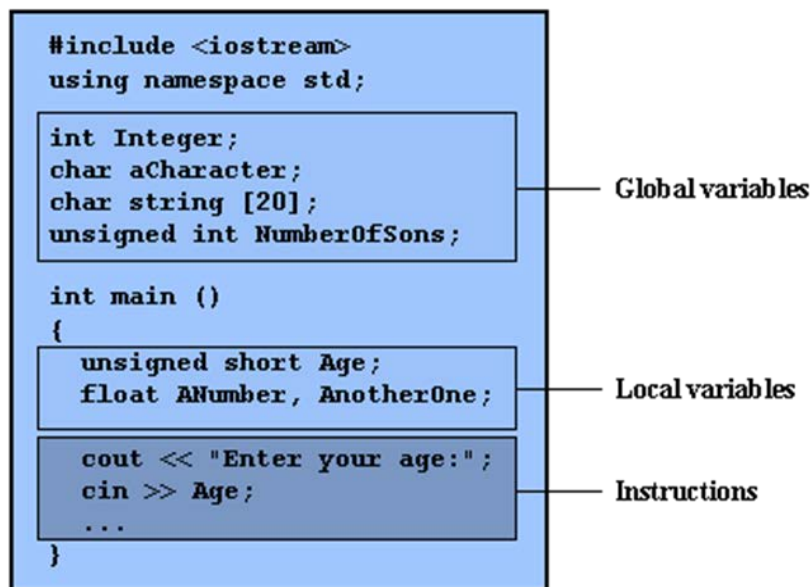
Global variables can be referred from anywhere in the code, even inside functions, whenever   it   is   after   its   declaration.
The scope of local variables is limited to the block enclosed in braces ({}) where they are
declared. For example, if they are declared at the beginning of the body of a function (like in function *main*) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to *main*, the local variables declared in *main* could not be accessed from the other function and vice versa.

Initialization of variables-

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like initialization, is done by appending an equal sign followed by the value to which the variable will be initialized:

type identifier = initial_value ;

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()):

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables
#include <iostream>
using namespace std;
int main ()
{
  int a=5;          // initial value = 5
  int b(2);         // initial value = 2
  int result;       // initial value undetermined
  a = a + 3;
  result = a - b;
  cout << result;
  return 0;
}
```

Q 2 (b) What are the two types of comments that can be added to a C++ program?

Answer Page Number 14 of the Textbook

Q 2 (c) Write a program that reads the name of the user and prints a greeting message to the user. For example, if the name entered by the user is 'Ram', then program should print Hello Ram!

Answer Page Number 15 of the Textbook.

Q 3 (a) Differentiate classes and structures with examples and write code segment for both.

Answer

  Classes

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.
An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.
Classes are generally declared using the keyword class, with the following format:

```
class class_name {
  access_specifier_1:
```

```
   member1;
 access_specifier_2:

   member2;
  ...
  } object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members that can be either data or function declarations, or optionally access specifiers.All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called access specifier. An access specifier is one of the following three keywords: private, public or protected.

These specifiers modify the access rights that the members following them acquire: private members of a class are accessible only from within other members of the same class or from their friends. Protected members are accessible from members of their same class and from their friends, but also from members of their derived classes. Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access.

For example:

```
class CRectangle {
   int x, y;
 public:
   void set_values (int,int);
   int area (void);
  } rect;
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

```
 int a;
```

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);
```

myarea = rect.area();
The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

structures are declared in C++ using the following syntax:

```
struct structure_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

where structure_name is a name for the structure type, object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name.

The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure_name is created and can be used in the rest of the program as if it was any other type.

For example:

```
struct product {
  int weight;
  float price;
} ;
product apple;
product banana, melon;
```

We have first declared a structure type called product with two members: weight and price, each of a different fundamental type. We have then used this name of the structure type (product) to declare three objects of that type: apple, banana and melon as we would have done with any fundamental data type.

Once declared, product has become a new valid type name like the fundamental ones int, char or short and from that point on we are able to declare objects (variables) of this compound new type, like we have done with apple, banana and melon.

The only difference between a structure and a class is that, in a class, the member data or function are private by default whereas, in a structure, they are public by default.

The following code segment:

```
Class demo
{
private:
int iNum1;
public:
void func(void);
};
```

can be written as:

```
class demo
{
```

```
int iNum1;
public:
void func(void);
};
```
The keyword private need not be mentioned because, in a class, the members are private by default.
The code segment can be modified using a structure in the following way:
```
Class demo
{
int iNum1;
public:
void func(void);
};
```
The keyword private need not be mentioned because, in a class, the members are private by default.
The code segment can be modified using a structure in the following way:
```
struct demo
{
void func(void);
private:
int iNum1;
};
```
The keyword public need not be mentioned because the structure members are public by default.

Q 3 (b) In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address (using arrays). Justify with example.

Answer

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.
In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function:
```
  void procedure (int arg[])
```
accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:
```
  int myarray [40];
```
it would be enough to write a call like this:
```
 procedure (myarray);
```
A complete example:
```
// arrays as parameters
#include <iostream>
using namespace std;
```

```
void printarray (int arg[], int length) {
 for (int n=0; n<length; n++)
   cout << arg[n] << " ";
 cout << "\n";
}
int main ()
{
 int firstarray[] = {5, 10, 15};
 int secondarray[] = {2, 4, 6, 8, 10}
 printarray (firstarray,3);
 printarray (secondarray,5);
 return 0;
}
```
 5 10 15
2 4 6 8 10
 As you can see, the first parameter (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:
   base_type[][depth][depth]
for example, a function with a multidimensional array as argument could be:
   void procedure (int myarray[][3][4])
Notice that the first brackets [] are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.

 Arrays, both simple or multidimensional, passed as function parameters are a quite
         common source of errors for novice programmers.

Q 3 (c) Which arithmetic operations can be performed on pointers? Explain with suitable examples

   Answer
   There are only two arithmetic operations that can be performed on pointers such as addition and subtraction. The integer value can be added or subtracted from the pointer. The result of addition and subtraction is an address. The difference of the two memory addresses results an integer and not the memory address. When a pointer is incremented it points to the memory location of the next element of its base type and when it is decremented it points to the memory location of the previous element of the same base type. For example,
         int *x;
         int *p;
         p=x++;
here x and p are pointers of integer type. Pointer x is incremented by 1. Now variable p points to the memory location next to the memory location of the pointer x.

Suppose memory address of x is 2000 and as a result p will contain memory address 2004 because integer type takes four bytes so the memory address is incremented by 4. Incrementing the pointer results in incrementing the memory address by the number of bytes occupied by the base type. For example,

    double *x;
    double *p;
    p=x++;

variables x and p are pointers of double type. Pointer x is incremented by 1. Now if the memory address of x was 2000 and after incrementing p will contain memory address 2008 as double takes 8 bytes. Decrementing the pointer results in decrementing the memory address by the number of bytes occupied by the base type. You cannot add two pointers. No multiplication and division can be performed on pointers. Here is a program which illustrates the working of pointer arithmetic.

```
#include<iostream>
using namespace std;
int main ()
{
        int *x;
        int *p,*q;
        int c=100,a;
        x=&c;
        p=x+2;
        q=x-2;
        a=p-q;
        cout << "The address of x : " << x << endl;
        cout << "The address of p after incrementing x by 2 : " << p << endl;
        cout << "The address of q after deremening  x by 2 : " << q << endl;
        cout << " The no of elements between p and q :" << a << endl;
        return(0);
}
```

The result of the program is:-
In the program x, p and q are pointers of integer type. The statement
        p=x+2;
 makes p to point to the memory address which is next two memory locations apart from the location of x. The statement
  q=x-2;
makes q to point to memory address which is previous two memory locations apart from the location of x. The statement
        a=p-q;
computes the no of memory locations between p and q will come out to be 4. The statement
   cout << "The address of x : " << x << endl;
prints the address of the memory location of x which is 0012FF70. The statement
        cout << "The address of p after incrementing x by 2 : " << p << endl;

prints the memory address of p which comes out to be 0012FF78. Each memory location occupies 4 bytes. Therefore after incrementing by 2, memory address is incremented by 8. The statement

cout << "The address of q after decrementing x by 2 : " << q << endl;

prints the memory address of q which is 0012FF68. After decrementing, the memory address is decremented by 8. The statement

cout << " The no of elements between p and q :" << a << endl;

prints the no of memory locations between p and q which comes out to be 4 as p points to next two memory locations of x and q points to the previous two memory locations of x.

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

Q 4 (a) Class can hold both data and functions? Explain your answer with suitable example.

Answer

   An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, or optionally access specifiers.All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called access specifier. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

private members of a class are accessible only from within other members of the same class or from their friends.

protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.

Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

class CRectangle {
    int x, y;
  public:

```
      void set_values (int,int);
      int area (void);
   } rect;
```
Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

```
   int a;
```
where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
 rect.set_values (3,4);
myarea = rect.area();
```
The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Q 4 (b) Write a program which invokes a function by value and explain it.

Answer

The following program illustrates the invoking of a function by value:
```
//Program
//This function swaps the value of two variable
#include<iostream.h>
void swap(int, int);
void main()
{
int iVar1, iVar2;
cout<<"Input two numbers "<<endl;
cin>>iVar1;
cin>>iVar2;
swap(iVar1, iVar2);
cout<<"In main "<<iVar1<<" "<<iVar2<<endl;
}
void swap(int iNum1, int iNum2)
{
int iTemp;
iTemp = iNum1;
iNum1 = iNum2;
```

iNum2 = iTemp;

cout<<"In swap "<<iNum1<<" "<<iNum2<<endl;

}

The sample output of Program is

Input two numbers

15

25

In swap 25 15

In main 15 25

In the above Program, values entered for the variables iVar1 and iVar2 are passed
to the function swap (). When the function swap () is invoked, these values
get copied into the memory locations of the parameters iNum1 and iNum2,
respectively where as in the Call by Value method, the called function
creates new variables to store the value of the arguments passed to it.

Q 4 (c) Explain advantages of friend function in C++ programming.

Answer

Friend function is a function, which have a direct access to the class data members(either
private or public) from outside the class.
It is declared within the class with a prefix
" friend ".

* The keyword friend is placed only in the function declaration of the
friend
function and not in the function definition.

* When a class is
declared as a friend, the friend class has access to the
private data of the class that
made this a friend.

friend function can access the private data member of a single or more than two
classes. It has no restriction while using the private member of the classes. With the
help of friend function we can share the private data of two classes. A friend function
can be declared in the public or private area. Friend function is a special type of
function which allows direct access to the data members and member functions which
may be either private or public of any class remaining outside of the class. It not a
member functions but has full access to the class.

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.

A friend function is a nonmember function which has access to class private
members. it is like allowing access to one's personal belongings to a friend. 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.

In addition to that it may be said that 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.

The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument. The keyword friend is placed only in the function declaration of the friend function and not in the function definition. It is possible to declare a function as friend in any number of classes.

When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend. A friend function, even though it is not a member function, would have the rights to access the private members of the class.

It is possible to declare the friend function as either private or public.

Q 5 (a) What is operator overloading ? Write a programe to demonstrate the use of overloading of addition operator?

Answer Page Number 135 of the textbook.

Q 5 (b) What is the use of constructor? Write general syntax for constructor.

> **Answer**
> The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.
> General Syntax of Constructor
> A constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:
> { arguments};
> The default constructor for a class X has the form
> X::X()
> In the above example, the arguments are optional.
> The constructor is automatically named when an object is created. A constructor is named whenever an object is defined or dynamically allocated using the "new" operator.

Q 5 (c) What will be output of following program?

```
#include <iostream>
using namespace std;
class IETE
{
private:
int a;
public:
IETE()
{ }
IETE(int w)
{
a=w;
}
```

```
                    IETE(IETE& e)
                    {
                    a=e.a;
                    cout << " Example of Copy Constructor";
                    }
                    void result()
                    {
                    cout<< a;
                    }
                    };
                    void main()
                    {
                    IETE e1(50);
                    IETE e3(e1);
                    cout<< "\ne3=";e3.result();
                    }
```

Answer

In the above the copy constructor takes one argument an object of type IETE which is passed by reference. The output of the above program is example of copy costrutors
e3=50

Q 6 (a) What is inherited from the base class? Explain with an example.

Answer
In principle, a derived class inherits every member of a base class except:
its constructor and its destructor
its operator=() members
its friends
Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:
derived_constructor_name (parameters): base_constructor_name (parameters) {...}
For example:
```
// constructors and derived classes
#include <iostream>
using namespace std;
class mother {
 public:
   mother ()
     { cout << "mother: no parameters\n"; }
   mother (int a)
     { cout << "mother: int parameter\n"; }
```

```
    };
    class daughter : public mother {
     public:
       daughter (int a)
         { cout << "daughter: int parameter\n\n"; }
    };
    class son : public mother {
     public:
       son (int a) : mother (a)
         { cout << "son: int parameter\n\n"; }
    };
    int main () {
      daughter cynthia (0);
      son daniel(0);
      return 0;
    }
```

mother: no parameters
daughter: int parameter
mother: int parameter
son: int parameter
Notice the difference between which mother's constructor is called when a new
daughter object is created and which when it is a son object. The difference is because
the constructor declaration of daughter and son:

daughter (int a)   // nothing specified: call default
son (int a) : mother (a) // constructor specified: call this

Q 6 (b) What is Inheritance? Write advantages of it.

Answer
Inheritance is the process by which new classes called derived classes are created
from existing classes called base classes. The derived classes have all the features of
the base class and the programmer can choose to add new features specific to the
newly created derived class.
For example, a programmer can create a base class named fruit and define derived
classes as mango, orange, banana, etc. Each of these derived classes, (mango, orange,
banana, etc.) has all the features of the base class (fruit) with additional attributes or
features specific to these newly created derived classes. Mango would have its own
defined features, orange would have its own defined features, banana would have its
own defined features, etc.
This concept of Inheritance leads to the concept of polymorphism.
Features or Advantages of Inheritance:
Reusability:
Inheritance helps the code to be reused in many situations. The base class is defined
and once it is compiled, it need not be reworked. Using the concept of inheritance, the
programmer can create as many derived classes from the base class as needed while
adding specific features to each derived class as needed.

Saves Time and Effort:
The above concept of reusability achieved by inheritance saves the programmer time and effort. Since the main code written can be reused in various situations as needed. Increases Program Structure which results in greater reliability.
General Format for implementing the concept of Inheritance:
class derived_classname: access specifier baseclassname
For example, if the base class is IETE and the derived class is sample it is specified as:
Sample Code
class sample: public IETE
The above makes sample have access to both public and protected variables of base class IETE. Reminder about public, private and protected access specifiers:
If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
Public members and variables are accessible from outside the class.
Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.
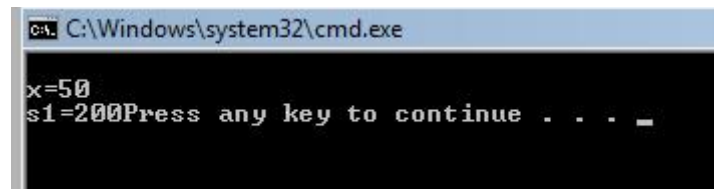
Q 6 (c) Write the output of the following program?

```
#include <iostream>
using namespace std;
class IETE
{
public:
    IETE( ) { x=0; }
    void f(int n1)
    {
        x= n1*5;
    }
    void output(void) { cout << "\n" << "x=" << x; }
private:
    int x;
};
class sample: public IETE
{
public:
    sample( ) { s1=0; }
    void f1(int n1)
    {
        s1=n1*10;
    }
     void output(void)
    {
         IETE::output();
```

```
                cout << "\n" << "s1=" << s1;
            }
        private:
            int s1;
        };
        int main(void)
        {
            sample s;
            s.f(10);
            s.f1(20);
            s.output();
        }
```

Answer

Out put of the program-

Q 7 (a) What are virtual functions? Why virtual functions are needed? What are properties of virtual functions? Explain and write syntax for virtual function.

Answer

Virtual, as the name implies, is something that exists in effect but not in reality. The concept of virtual function is the same as a function, but it does not really exist although it appears in needed places in a program. The object-oriented programming language C++ implements the concept of virtual function as a simple member function, like all member functions of the class.

The functionality of virtual functions can be overridden in its derived classes. The programmer must pay attention not to confuse this concept with function overloading. Function overloading is a different concept and will be explained in later sections of this tutorial. Virtual function is a mechanism to implement the concept of polymorphism (the ability to give different meanings to one function).

Need for Virtual Function:

The vital reason for having a virtual function is to implement a different functionality in the derived class.

or example: a Make function in a class Vehicle may have to make a Vehicle with red color. A class called FourWheeler, derived or inherited from Vehicle, may have to use a blue background and 4 tires as wheels. For this scenario, the Make function for FourWheeler should now have a different functionality from the one at the class called Vehicle. This concept is called Virtual Function.

Properties of Virtual Functions:
Dynamic Binding Property:
Virtual Functions are resolved during run-time or dynamic binding. Virtual functions are also simple member functions. The main difference between a non-virtual C++ member function and a virtual member function is in the way they are both resolved. A non-virtual C++ member function is resolved during compile time or static binding. Virtual Functions are resolved during run-time or dynamic binding
Virtual functions are member functions of a class.
Virtual functions are declared with the keyword virtual, detailed in an example below.
Virtual function takes a different functionality in the derived class.
Declaration of Virtual Function:
Virtual functions are member functions declared with the keyword virtual.
The general syntax to declare a Virtual Function uses:
class class_name //This denotes the base class of C++ virtual function
{
public:
virtual void member_function_name() //This denotes the C++ virtual function
{
....
....
}
};

Q 7   (b) Differentiate between static and dynamic polymorphism.

Answer

Static Polymorphism
    Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called early binding. The term early binding stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

    The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:
        void add(int , int);
        void add(float, float);
 When the add() function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.
 Dynamic Polymorphism
 Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the

program is executed. The term late binding refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Static Vs Dynamic Polymorphism

Static polymorphism is considered more efficient, and dynamic polymorphism more flexible.

Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.

As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change

in requirements requires a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.

To the programmer, efficiency and performance would probably be a primary concern, but to the user, flexibility or maintainability may be much more important. The decision is thus a trade-off between efficiency and flexibility.

Q 7 (c) Write a program showing use of exception handling.

Answer

Exception handling is a construct designed to handle the occurrence of exceptions that is special conditions that changes the normal flow of program execution. Since when designing a programming task (a class or even a function), one cannot always assume that application/task will run or be completed correctly (exit with the result it was intended to). It may be the case that it will be just inappropriate for that given task to report an error message (return an error code) or just exit. To handle these types of cases, C++ supports the use of language constructs to separate error handling and reporting code from ordinary code, that is, constructs that can deal with these exceptions (errors and abnormalities) and so we call this global approach that adds uniformity to program design the exception handling.

An exception is said to be thrown at the place where some error or abnormal condition is detected. The throwing will cause the normal program flow to be aborted, in a raised exception. An exception is thrown programmatic, the programmer specifies the conditions of a throw.

In handled exceptions, execution of the program will resume at a designated block of code, called a catch block, which encloses the point of throwing in terms of program execution. The catch block can be, and usually is, located in a different function/method than the point of throwing. In this way, C++ supports non-local error handling. Along with altering the program flow, throwing of an exception passes an object to the catch block. This object can provide data which is necessary for the handling code to decide in which way it should react on the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class X { };
class Y { };
class A { };
// pfv type is pointer to function returning void
typedef void (*pfv)();
void my_terminate() {
 cout << "Call to my terminate" << endl;
 abort();
}
void my_unexpected() {
 cout << "Call to my_unexpected()" << endl;
 throw;
}
void f() throw(X,Y, bad_exception) {
 throw A();
}
void g() throw(X,Y) {
 throw A();
}
int main()
{
 pfv old_term = set_terminate(my_terminate);
 pfv old_unex = set_unexpected(my_unexpected);
 try {
  cout << "In first try block" << endl;
  f();
 }
 catch(X) {
  cout << "Caught X" << endl;
 }
 catch(Y) {
  cout << "Caught Y" << endl;
 }
 catch (bad_exception& e1) {
  cout << "Caught bad_exception" << endl;
 }
 catch (...) {
  cout << "Caught some exception" << endl;
 }
 cout << endl;
 try {
  cout << "In second try block" << endl;
  g();
```

```
   }
   catch(X) {
     cout << "Caught X" << endl;
   }
   catch(Y) {
     cout << "Caught Y" << endl;
   }
   catch (bad_exception& e2) {
     cout << "Caught bad_exception" << endl;
   }
   catch (...) {
     cout << "Caught some exception" << endl;
   }
}
```

The following is the output of the above example:
In first try block
Call to my_unexpected()
Caught bad_exception
In second try block
Call to my_unexpected()
Call to my terminateThe output also includes a memory dump in the core file in the current directory because of a call to the abort() function.

Q 8 (a) With the help of an example, explain the function template and class template.

Answer

Function templates

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

template <class identifier> function_declaration;

template <typename identifier> function_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

template <class myType>

```
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
 int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
     values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
 mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
 mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates
#include <iostream>
using namespace std;
template <class T>
class mypair {
```

```
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};
template <class T>
T mypair<T>::getmax ()
{
 T retval;
 retval = a>b? a : b;
 return retval;
}
int main () {
 mypair <int> myobject (100, 75);
 cout << myobject.getmax();
 return 0;
}
```
The syntax of the definition of member function getmax:
```
template <class T>
T mypair<T>::getmax ()
```
Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter

Q 8 (b) Write the syntax for following and explain its uses -
      (i) Template specialization
      (ii) Parameter values for templates

Answer
(i) Template specialization
   A template specialization allows a template to make specific implementations when the pattern is of a determined type. For example, suppose that our class template pair included a function to return the result of the module operation between the objects contained in it, but we only want it to work when the contained type is int. For the rest of the types we want this function to return 0. This can be done the following way:
```
// Template specialization
#include <iostream.h>
template <class T>
class pair {
    T value1, value2;
  public:
```

```
   pair (T first, T second)
     {value1=first; value2=second;}
   T module () {return 0;}
};
template <>
class pair <int> {
   int value1, value2;
  public:
   pair (int first, int second)
     {value1=first; value2=second;}
   int module ();
};
template <>
int pair<int>::module() {
  return value1%value2;
}
int main () {
  pair <int> myints (100,75);
  pair <float> myfloats (100.0,75.0);
  cout << myints.module() << '\n';
  cout << myfloats.module() << '\n';
  return 0;
}    25
0
```

As you can see in the code the specialization is defined this way:

template <> class class_name <type>

The specialization is part of a template, for that reason we must begin the declaration with template <>. And indeed because it is a specialization for a concrete type, the generic type cannot be used in it and the first angle-brackets <> must appear empty. After the class name we must include the type that is being specialized enclosed between angle-brackets <>.

When we specialize a type of a template we must also define all the members equating them to the specialization (if one pays attention, in the example above we have had to include its own constructor, although it is identical to the one in the generic template). The reason is that no member is "inherited" from the generic template to the specialized one.

(ii)   Parameter values for templates

Besides the template arguments preceded by the class or typename keywords that represent a type, function templates and class templates can include other parameters that are not types whenever they are also constant values, like for example values of fundamental types. As an example see this class template that serves to store arrays:

```
// array template
#include <iostream.h>
```

```
template <class T, int N>
class array {
    T memblock [N];
  public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
array<T,N>::setmember (int x, T value) {
  memblock[x]=value;
}

template <class T, int N>
T array<T,N>::getmember (int x) {
  return memblock[x];
}

int main () {
  array <int,5> myints;
  array <float,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3,3.1416);
  cout << myints.getmember(0) << '\n';
  cout << myfloats.getmember(3) << '\n';
  return 0;
}
```
    It is also possible to set default values for any template parameter just as it is done with function parameters.

Some possible template examples seen above:

```
template <class T>            // The most usual: one class parameter.
template <class T, class U>   // Two class parameters.
template <class T, int N>     // A class and an integer.
template <class T = char>     // With a default value.
```

Q 9 (a) Write short note on :
          (i)   Standard input and output
          (ii) File I/O
          (iii) I/O Parameters

Answer

Standard input and output

You can write to the standard output using code like this:

```
cout << x << y << endl;
```

(Note: this is like calling *System.out.println* in Java.) The code is evaluated from left to right, so the value of x is written first, then the value of y, then a newline. Therefore, the code given above is equivalent to:

```
cout << x;
```

```
cout << y;
cout << endl;
```
Similarly, you can read from the standard input using code like this:
```
cin >> x >> y;
```
which is equivalent to:
```
cin >> x;
cin >> y;
```
(Again, the code is evaluated from left to right, so variable x will be set to the first input value and variable y will be set to the second input value.)

Note:

1. Remember that you must *#include <iostream>* in order to use cin, cout, or endl.

2. To remember which is the input operator and which is the output operator, think of the angle brackets as pointing in the direction the values are flowing; for example, when you write the value of x using "cout << x", the brackets point to cout (so the value is going from x to cout). When you read a value into x you use "cin >> x"; this time, the brackets point to x (so the value is going from cin to x).

3. Both the output operator << and the input operator >> are *overloaded*; they can be used to write/read any of the primitive types. However, the input operator >> skips all whitespace in the input, so you cannot use it to read a whitespace character (e.g., a space, tab, or newline).

4. It is up to you as the programmer to know what kind of data you are going to read. If the data in the input does not match the type of the variable you are reading into, you will either get a runtime error, or the value you read will be garbage. For example, if you run the following program:

```
#include <iostream>
    int main() {
              int x;
  cout << "enter a number: ";
  cin >> x;
              return(0);
}
```
and you type a letter instead of a number, you will either get a runtime error, or the value of x will be garbage

(ii)   File I/O o read from a file you must use a variable of type **ifstream**. To write to a file you must use a variable of type **ofstream**. In both cases, you must open the file before you can read or write.

For example, here's how to open the file named "input.dat" for reading:
```
#include <fstream>
  ifstream inFile;
inFile.open("input.dat");
```

```
   if (inFile.fail()) {
      cerr << "unable to open file input.dat for reading" << endl;
      exit(1);
   }
```

Note that to use files you must *#include <fstream>* (including iostream is not good enough). Also note that this code writes its error message to **cerr**; that is the **standard error**, and should generally be used for error messages instead of the standard output. Once inFile has successfully been opened for reading, you can use the usual input operator to read values:

```
   int n, sum = 0;
   while (inFile >> n) {
      sum += n;
   }
```

In this example, each time the while loop condition is evaluated; the next integer in the input file is read into variable x. The while loop condition will evaluate to false when all of the values in the input file have been read.
If you prefer to read one character at a time (including whitespace characters), you can use the **get** operation:

```
   char ch;
   while (inFile.get(c)) {
      ...
   }
```

In this example, each time the while loop condition is evaluated, the next character in the input file is read into variable ch. As in the previous example, the condition will evaluate to false when there are no more characters in the input.

(iii)   I/O Parameters

It is often useful to write a function that takes an input stream or an output stream as a parameter, and reads from or writes to the given stream (without worrying about whether it is using the standard input/output or a particular file). To do this, you should use parameters of type **istream** (for input), and **ostream** (for output). For example:

```
   void f( istream & input, ostream & output ) {
   int n;
      output << "enter a number: ";
      input >> n;
   }
```

Note:
"input" and "output" are the names used in the example for the two parameters, but there is nothing special about those names; as usual, it is up to the programmer to choose parameter names.
The ampersands in front of the parameter names mean that the two parameters are passed by **reference**. The difference between value parameters and reference parameters will be discussed in another set of notes. For now, just remember that

input and output streams must always be passed by reference (or you will get a not-very-clear compile-time error)
A call to function f can pass either the standard input or a file (an ifstream) as the first parameter, and can pass either the standard output, the standard error, or a file (an ofstream) as the second parameter.

Text Book

Object-oriented Programming with C++, Poornachandra Sarang, PHI, 2004