Q2 (a) What is meant by Programming Paradigm? Discuss four main programming paradigms.

Answer

Programming paradigm: A pattern that serves as a *school of thoughts* for programming of computers.

Four basic programming paradigms are as follows:

1. Imperative paradigm

Characteristics:

- Discipline and idea
- Digital hardware technology and the ideas of Von Neumann
- Incremental *change of the program state* as a function of *time*.
- Execution of computational steps in an order governed by *control structures*
- We call the steps for *commands*
- Straightforward abstractions of the way a traditional Von Neumann computer
- works
- Similar to descriptions of everyday routines, such as food recipes and car repair
- Typical commands offered by imperative languages
- Fortran, Algol, Pascal, Basic, C
- The natural abstraction is the procedure
- Abstracts one or more actions to a procedure, which can be called as a
- Single command.
- "Procedural programming"

2. Functional paradigm

Functional programming is in many respects a simpler and more clean programming paradigm than the imperative one.

Characteristics:

- Discipline and idea
- Mathematics and the theory of functions
- The values produced are *non-mutable*
- Impossible to change any constituent of a composite value
- As a remedy, it is possible to make a revised copy of composite value
- Abstracts a single expression to a function which can be evaluated as an expression
- Functions are first class values
- Functions are full- fledged data just like numbers, lists, ...

3. Logic paradigm

The logic paradigm is dramatically different from the other three main programming paradigms.

The logic paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. The logical paradigm seems less natural in the more general areas of computation.

Characteristics:

- Discipline and idea
- Automatic proofs within artificial intelligence
- Based on axioms, inference rules, and queries.
- Program execution becomes a systematic search in a set of facts, making use of a set of inference rules

4. object-oriented paradigm

The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is undoubtedly the strong support of encapsulation and the logical grouping of program aspects.

Characteristics:

- Discipline and idea
- The theory of concepts, and models of human interaction with real world phenomena
- Data as well as operations are encapsulated in objects
- Information hiding is used to protect internal properties of an object
- Objects interact by means of message passing
- A metaphor for applying an operation on an object
- In most object-oriented languages objects are grouped in classes
- Objects in classes are similar enough to allow programming of the classes, as opposed to programming of the individual objects
- Classes represent concepts whereas objects represent phenomena
- Classes are organized in inheritance hierarchies
  Provides for class extension or specialization

Q3 (a) Write a program in C++ that print a pattern similar to the following pattern using a for loop.

```
*****
****
***
**
*
```

Answer

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int size,j,i;
cout << "\n\nEnter the size of the series : - ";
cin >> size;
;
for(i=0;i<size;i++)
{
cout << "\n\t\t\t   ";
```

```
for(j=i;j<size;j++)
{
cout << "*";
}
}
getch();
}
```

Q3 (b) How a multidimensional array can be initialized in C++? Explain various methods by giving suitable examples.

Answer

You can initialize a multidimensional array using any of the following techniques:

- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by increasing the subscript of the last dimension fastest. This form of a multidimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array month_days:

```
static month_days[2][12] =
{
 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
```

- Using braces to group the values of the elements you want initialized. You can put braces around each element, or around any nesting level of elements. The following definition contains two elements in the first dimension (you can consider these elements as rows). The initialization contains braces around each of these two elements:

```
static int month_days[2][12] =
{
 { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
 { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
    };
```

- Using nested braces to initialize dimensions and elements in a dimension selectively. In the following example, only the first eight elements of the array grid are explicitly initialized. The remaining four elements that are not explicitly initialized are automatically initialized to zero.

```
static short grid[3] [4] = {8, 6, 4, 1, 9, 3, 1, 1};
```

The initial values of grid are:

Element	Value	Element	Value
grid[0] [0]	8	grid[1] [2]	1
grid[0] [1]	6	grid[1] [3]	1
grid[0] [2]	4	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0

Element	Value	Element	Value
grid[1] [0]	9	grid[2] [2]	0
grid[1] [1]	3	grid[2] [3]	0

- Using *designated* initializers. The following example uses designated initializers to explicitly initialize only the last four elements of the array. The first eight elements that are not explicitly initialized are automatically initialized to zero.
- static short grid[3] [4] = { [2][0] = 8, [2][1] = 6,
  [2][2] = 4, [2][3] = 1 };

  The initial values of grid are:

Element	Value	Element	Value
grid[0] [0]	0	grid[1] [2]	0
grid[0] [1]	0	grid[1] [3]	0
grid[0] [2]	0	grid[2] [0]	8
grid[0] [3]	0	grid[2] [1]	6
grid[1] [0]	0	grid[2] [2]	4
grid[1] [1]	0	grid[2] [3]	1

Q3 (c) List four most common conditions that invalidates a pointer value or memory location of a valid item.

Answer

(i) Pointer is a null pointer.
(ii) The pointer specifies address of an item that no longer exists.
(iii)The pointer specifies an address not used by the executing program.

The pointer specifies an address that is inappropriately aligned for the type of item pointed to.

Q4 (a) Explain function declaration, function definition and function cell using a suitable example. What is function prototype?

Answer Page Number 127 –129 of Textbook

Q4 (b) What do you mean by function overloading? When do we use this concept? Illustrate the concept by writing a C++ program.

Answer

It's creation of multiple functions in a same class with same name but different number of arguments or different type of arguments. It's a method of implementing Polymorphism in C++, the basic advantage of function overloading is that it creates comfort for the user as he has to remember only one name of the function

#include<iostream.h>

```
class shape
{
int length,breadth,height,radius,ar;
public:
void area(int m, int n, int o)
{
length=m;
height=n;
breadth=o;
ar=length*breadt*height;
}
void area(int x)
{
radius=x;

ar=(3.142*r)*r;
}
void result()
{
cout<<"area is = "<<ar;
}
};
void main()
{
Shape Rectangle;
Shape Circle;
Rectangle.area(10,12,5);
Circle.area(5);
Rectangle.result();
Circle.result();
}
```

Q4 (c) Explain the following:
(i) Return by reference (ii) Pointer to function

Answer Page Number 146 –148 of Textbook

Q5 (a) List any three restrictions that apply to class members.

Answer

- A non-**static** member variable cannot have an initializer.
- No member can be an object of the class that is being declared.
- No member can be declared as **auto**, **extern**, or **register**.

Q5 (b) Is it possible for one class to be a friend of another class? Demonstrate this using a suitable C++ program.

Answer

Is it possible for one class to be a **friend** of another class? Demonstrate this concept using a suitable C++ program.

```
// Using a friend class.
#include <iostream>
using namespace std;
class TwoValues {
int a;
int b;
public:
TwoValues(int i, int j) { a = i; b = j; }
friend class Min;
};
class Min {
public:
int min(TwoValues x);
};
int Min::min(TwoValues x)
{
return x.a < x.b ? x.a : x.b;
}
int main()
{
TwoValues ob(10, 20);
Min m;
cout << m.min(ob);
return 0;
}
```

In this example, class **Min** has access to the private variables **a** and **b** declared within the **TwoValues** class. It is critical to understand that when one class is a **friend** of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the **friend** class.

Q5 (c) Why a destructor function in a derived class is executed before the destructor in the base? Write a C++ program that illustrates the order in which constructors and destructors are executed. Also discuss the output.

Answer

A destructor function in a derived class is executed before the destructor in the base. Since the destruction of a base class object implies the destruction of the derived class object, the derived object's destructor must be executed before the base object is destroyed. This program illustrates the order in which constructors and destructors are executed:

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

```
class Base {
public:
Base() { cout << "\nBase created\n"; }
~Base() { cout << "Base destroyed\n\n"; }
};
class D_class1 : public Base {
public:
D_class1() { cout << "D_class1 created\n"; }
~D_class1() { cout << "D_class1 destroyed\n"; }
};
int main()
{
D_class1 d1;
cout << "\n";
return 0;
}
```

This program produces the following output:

```
Base created
D_class1 created
D_class1 destroyed
Base destroyed
```

Q6 (a) Write a C++ program that creates a class called Loc, which stores longitude and latitude values. Overload the '+' operator using a friend function, assignment '=' operator and unary operator '++' relative to this class.

Answer

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
friend loc operator+(loc op1, loc op2); // now a friend
loc operator=(loc op2);
loc operator++();
};
// Now, + is overloaded using friend function.
```

```
loc operator+(loc op1, loc op2)
{
loc temp;
temp.longitude = op1.longitude + op2.longitude;
temp.latitude = op1.latitude + op2.latitude;
return temp;
}

// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1 = ob1 + ob2;
ob1.show();
return 0;
}
```

Q6 (b) Write a program to illustrate user-defined conversions in operator overloading.

Answer Page Number 246 of Textbook

Q6 (c) Give the syntax of operator overloading for:
(i) Pre-increment (ii) Post increment

Answer Page Number 247 –256 of Textbook

Q7 (b) Is it possible to inherit a base class as protected? When this is done, what happens to all public and protected members of the base class become protected members of the derived class? Write a suitable C++ program to demonstrate.

Answer

It is possible to inherit a base class as **protected.** When this is done, all public and protected members of the base class become protected members of the derived class.

For example,

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void setij(int a, int b) { i=a; j=b; }
void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
class derived : protected base{
int k;
public:
// derived may access base's i and j and setij().
void setk() { setij(10, 12); k = i*j; }
// may access showij() here
void showall() { cout << k << " "; showij(); }
};
int main()
{
derived ob;
// ob.setij(2, 3); // illegal, setij() is
// protected member of derived
ob.setk(); // OK, public member of derived
ob.showall(); // OK, public member of derived
// ob.showij(); // illegal, showij() is protected
// member of derived
return 0;
}
```

Even though **setij()** and **showij()** are public members of **base,** they become protected members of **derived** when it is inherited using the **protected** access specifier. This means that they will not be accessible inside **main()**

Q8 (a) What is an exception? When do they occur? Illustrate using an example how to provide your own exception handler.

Answer

A generic function defines a general set of operations that will be applied to various types of data. Using this mechanism, the same general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quick sort sorting algorithm is the same whether it is applied to an array of integers or an array of **float**s.

It is just that the type of the data being sorted is different. By creating a generic functio you can define, independent of any data, the nature of the algorithm. Once this is done, the compiler automatically generates the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself. A generic function is created with the keyword **template**. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details, as needed. The general form of a **template** function definition is shown here:

```
// Function template example.
#include <iostream>
using namespace std;
// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
int main()
{
int i=10, j=20;
float x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << endl;
cout << "Original x, y: " << x << ' ' << y << endl;
cout << "Original a, b: " << a << ' ' << b << endl;
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j << endl;
cout << "Swapped x, y: " << x << ' ' << y << endl;
cout << "Swapped a, b: " << a << ' ' << b << endl;
return 0;
}
```

Q8 (b) Can you restrict the types of exception that a function can throw? Can you also prevent that function from throwing any exceptions whatsoever? Explain the concept giving a small C++ routine.

Answer

When a function is called from within a **try** block, you can restrict what type of exceptions that function can throw. In fact, you can also prevent that function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a **throw** clause to a function definition. The general form of this is shown here.

ret-type func-name(*arg-list*) throw(*type-list*)
{
// ...
}

Here, only those data types contained in the comma-separated *type-list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw *any* exceptions, then use an empty list.

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
int main()
{
cout << "start\n";
try{
Xhandler(0); // also, try passing 1 and 2 to Xhandler()
}
catch(int i) {
cout << "Caught an integer\n";
}
catch(char c) {
cout << "Caught char\n";
}
catch(double d) {
cout << "Caught double\n";
}
cout << "end";
return 0;
}
```

In this program, the function **Xhandler( )** may throw only **int**, **char**, and **double** exceptions. If it attempts to throw any other type of exception, then an abnormal program termination will occur.

Q9 (a) Define Standard Streams and file streams. Differentiate between two types of stream.

Answer

The I/O system provides a level of abstraction between the programmer and th hardware. This abstraction is called a *stream;* the actual device is called a *file*.
There are two types of streams: text and binary.

Text Streams

A *text stream* is a sequence of characters. Standard C states that a text stream is organized into lines terminated by a newline character. However, the newline character is optional on the last line. In a text stream, certain character translations may occur as required by the host environment. For example, a newline may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written or read and those on the external device. Also, because of possible translations, the number of characters written or read may not be the same as the number that is stored on the external device.

Binary Streams

A *binary stream* is a sequence of bytes that have a one-to-one correspondence to those on the external device. That is, no character translations occur. Also, the number of bytes written or read is the same as the number on the external device. However, an implementation-defined number of null bytes may be appended to a binary stream. These null bytes might be used to pad the information so that it fills a sector on a disk.

Q9 (b) Write a program in C++ that inputs characters from the keyboard and prints them in reverse case. That is, uppercase prints as lowercase, and lowercase as uppercase. The program halts when a period is typed.

Answer

```
 /* Case Switcher */
#include <conio.h>
#include <stdio.h>
#include <ctype.h>
int main(void) {
char ch;
do {
ch = getche();
if(islower(ch)) putchar(toupper(ch));
else putchar(tolower(ch));
} while (ch!='.'); /* use a period to stop*/
return 0;
}
```

Q9 (c) What do you mean by Containers? Define Sequence and Associative containers.

Answer

Containers are objects that hold other objects. This is one of the three foundation elements of STL.

The **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called *sequence containers* because in STL terminology, a sequence is a linear list.

In addition to the basic containers, the STL also defines *associative containers* that allow efficient retrieval of values according to keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key.

Text Book

C++ and Object-Oriented Programming Paradigm, Debasish Jana, Second Edition, PHI, 2005