CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | |

Internal Assesment Test II - December 2019

Sub:	Object Oriented Programming Using C++							Code:	18MCA11
Date:	16.12.2019	Duration:	90 mins	Max Marks:	50	Sem:	I	Branch:	MCA

		Marks	OBE CO	RBT
	Part - I			
1.	What are virtual functions? With an example demonstrate the use of virtual functions.	[10]	CO2	L2
	OR			
2.	What are inserter and extractor? Explain how to create your own inserter and extractor with an example.	[10]	CO2	L1
	Part - II			
3 (a)	Discuss how to overload an operator using friend and write a program to overload ++ operator using friend	[6]	CO2	L3
(b)	Explain overloading of special operator ()	[4]	CO2	L2
	OR			
4 (a)	Create a class called MATRIX using two-dimensional array of integers. Implement the following operations by overloading the operator = = which checks the compatibility of two matrices to be subtracted. Overload the operator ‘-‘for matrix subtraction as m3 = m1- m2 when (m1= =m2).	[10]	CO2	L3
(b)	**Part –III** Which are the different access specifiers? Explain the effect of inheritance when deriving by different access specifiers with appropriate examples	[4]	CO2	L2
6 (a)	Explain the order of constructor and destructor called in multi level inheritance with example	[4]	CO2	L2
(b)	Write a cpp program witch shows how a virtual function is called through a base class reference	[6]	CO2	L3
	Part – IV			
7. (a)	Write a program to dynamically allocate memory for object (using constructor).	[4]	CO2	L3
(b)	Explain the dynamic memory allocation operator in c++. Explain proper syntax with example	[6]	CO2	L2
	OR			
8 (a)	Create a base class with two protected data members i and j and two public methods setij() and showij() to set the values of I and j and display the values of i and j respectively. Create a derived class which inherits base class as protected. Show how derived class object sets the values of i and j and display their values.	[6]	CO2	L3
(b)	What is the need for a virtual base class? Show how a virtual base class eliminates ambiguity	[4]	CO2	L2
	` **Part - V**			
9 (a)	What are pure virtual functions? Discuss its significance.	[6]	CO2	L1
(b)	Differentiate between early binding and late binding	[4]	CO2	L2
	OR			
10.	Create a class called complex which has two data members real part, imaginary part. Implement a friend function for overloading ‘+’ operator which can compute the sum of two complex numbers and the function returns the complex object.	[10]	CO2	L3

Internal Assessment Test 2– December 2019
Answer Key

Sub:	Object Oriented Programming Using C++							**Code:**	18MCA11
Date:	16/12/2019	Duration:	90mins	Max Marks:	50	**Sem:**	I	**Branch:**	MCA

Note: Answer Any One FULL Question from each part.

1. **What are virtual functions? With an example demonstrate the use of virtual functions. (10)**

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function,we have to precede the function's declaration in the base class with the keyword virtual.

When a class containing a virtual function is inherited,the derived class redefines the virtual function to fit its own needs. Virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class.The redefinition creates a specific method.

It supports run-time polymorphism as they behave differently when accessed via a pointer. A base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon the type of object pointed to by the pointer. And this determination is made at run time. Thus, when different objects are pointed to, different versions of the virtual function are executed.

Syntax:

```
        virtual return-type function_name()
        {
                //body of the function
        }
```

Ex:

```
#include <iostream>
using namespace std;
class base
{
public:
        virtual void vfunc()
        {
                cout << "This is base's vfunc().\n";
        }
};

class derived1 : public base
{
public:
        void vfunc()
        {
                cout << "This is derived1's vfunc().\n";
        }
};

class derived2 : public base
 {
public:

        void vfunc()
         {
```

```
                    cout << "This is derived2's vfunc().\n";

          }

};
int main()
{
          base *p, b;
          derived1 d1;
          derived2 d2;
          // point to base
          p = &b;
          p->vfunc(); // access base's vfunc()
          // point to derived1
          p = &d1;
          p->vfunc(); // access derived1's vfunc()

          // point to derived2
          p = &d2;
          p->vfunc(); // access derived2's vfunc()
          return 0;
}
```

2. **What are inserter and extractor? Explain how to create your own inserter and extractor with an example. (10)**

Ans: In the language of C++, the << output operator is referred to as the *insertion operator* because it inserts characters into a stream. Likewise, the >> input operator is called the *extraction operator* because it extracts characters from a stream.

Creating Your Own Inserters
It is quite simple to create an inserter for a class that you create. All inserter functions have this general form:

```
ostream &operator<<(ostream &stream, class_type obj)
{
    // body of inserter
     return stream;
}
```

the function returns a reference to a stream of type ostream. Further, the first parameter to the function is a reference to the output stream. The second parameter is the object being inserted.

Creating Your Own Extractors
Extractors are the complement of inserters. The general form of an extractor function is

```
istream &operator>>(istream &stream, class_type &obj)
{
// body of extractor
return stream;
}
```

Extractors return a reference to a stream of type istream, which is an input stream.The first parameter must also be a reference to a stream of type istream. Notice that the second parameter must be a reference to an object of the class for which the extractor is overloaded. This is so the object can be modified by the input (extraction) operation.

```
#include <iostream>
#include <cstring>
using namespace std;
class phonebook
{
          char name[80];
```

```
	int areacode;
	int prefix;
	int num;
	public:
	phonebook() { };
	phonebook(char *n, int a, int p, int nm)
	{
	strcpy(name, n);
	areacode = a;
	prefix = p;
	num = nm;
	}
	friend ostream &operator<<(ostream &stream, phonebook o);
	friend istream &operator>>(istream &stream, phonebook &o);
};
// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
	stream << o.name << " ";
	stream << "(" << o.areacode << ") ";
	stream << o.prefix << "-" << o.num << "\n";
	return stream; // must return stream
}
// Input name and telephone number.
istream &operator>>(istream &stream, phonebook &o)
{
	cout << "Enter name: ";
	stream >> o.name;
	cout << "Enter area code: ";
	stream >> o.areacode;
	cout << "Enter prefix: ";
	stream >> o.prefix;
	cout << "Enter number: ";
	stream >> o.num;
	cout << "\n";
	return stream;
}
int main()
{
	phonebook a;
	cin >> a;
	cout << a;
	return 0;
}
```

3 (a) **Discuss how to overload an operator using friend and write a program to overload ++ operator using friend** **(6)**

Ans: If we want to use a friend function to overload the increment or decrement operators, we must pass the operand as a reference parameter. This is because friend functions do not have this pointers. If we overload these operators by using a friend, then the operand is passed by value as a parameter. This means that a friend operator function has no way to modify the operand. Since the friend operator function is not passed a this pointer to the operand, but rather a copy of the operand, no changes made to that parameter affect the operand that generated the call. But we can do by specifying the parameter to the friend operator function as a reference parameter. This causes any changes made to the parameter inside the function to affect the operand that generated the call.

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt)
```

```
            longitude = lg;
            latitude = lt;
}
void show()
{
            cout << longitude << " ";
            cout << latitude << "\n";
}

loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
            longitude = op2.longitude;
            latitude = op2.latitude;
            return *this; // i.e., return object that generated call
}

// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
            op.longitude++;
            op.latitude++;
            return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
            op.longitude--;
            op.latitude--;
}
int main()
{
            loc ob1(10, 20), ob2;
            ob1.show();
            ++ob1;
            ob1.show(); // displays 11 21
            ob2 = ++ob1;
            ob2.show(); // displays 12 22
            --ob2;
            ob2.show(); // displays 11 21
            return 0;
}
```

(b) **Explain overloading of special operator ()** **(4)**

Ans: When you overload the ( ) function call operator, we are creating an operator function that can be passed an arbitrary number of parameters.
In general, when you overload the ( ) operator, you define the parameters that you want to pass to that function. When you use the ( ) operator in your program, the arguments you specify are copied to those parameters. As always, the object that generates the call (O in this example) is pointed to by the this pointer. Here is an example of overloading ( ) for the loc class. It assigns the value of its two arguments to the longitude and latitude of the object to which it is applied.

```
#include <iostream>
using namespace std;
class loc {
            int longitude, latitude;
            public:
```

```
        loc() {}
        loc(int lg, int lt)
        {
        longitude = lg;
        latitude = lt;
        }
        void show()
        {
                cout << longitude << " ";
                cout << latitude << "\n";
        }
        loc operator+(loc op2);
        loc operator()(int i, int j);
};
// Overload ( ) for loc.
loc loc::operator()(int i, int j)
{
        longitude = i;
        latitude = j;
        return *this;
}
// Overload + for loc.
loc loc::operator+(loc op2)
{
        loc temp;
        temp.longitude = op2.longitude + longitude;
        temp.latitude = op2.latitude + latitude;
        return temp;
}
int main()
{
        loc ob1(10, 20), ob2(1, 1);
        ob1.show();
        ob1(7, 8); // can be executed by itself
        ob1.show();
        ob1 = ob2 + ob1(10, 10); // can be used in expressions
        ob1.show();
        return 0;
}
```

The output produced by the program is shown here.

```
10 20
7 8
11 11
```

4 (a) **Create a class called MATRIX using two-dimensional array of integers. Implement the following operations by overloading the operator = = which checks the compatibility of two matrices to be subtracted. Overload the operator '-'for matrix subtraction as m3 = m1- m2 when (m1= =m2). (10)**

Ans:
```
#include<iostream>
#define Max 20
using namespace std;
class Matrix
{
        public:
        int a[Max][Max];
        int r,c;
        void getorder();
        void getdata();
        Matrix operator -(Matrix);
        friend ostream& operator <<(ostream &, Matrix);
        int operator==(Matrix);
```

```
};
void Matrix::getorder()
{
	cout<<"enter the number of rows\n";
	cin>>r;
	cout<<"enter the number of columns\n";
	cin>>c;
}
void Matrix::getdata()
{
	int i,j;
	for(i=0;i<r;i++)
	{
		for(j=0;j<c;j++)
		{
			cin>>a[i][j];
		}
	}
}
Matrix Matrix::operator -(Matrix m2)
{
	Matrix m4;
	int i,j;
	for(i=0;i<r;i++)
	{
		for(j=0;j<c;j++)
		{
			m4.a[i][j] = a[i][j] - m2.a[i][j];
		}
	}
	m4.r = r;
	m4.c = c;
	return m4;
}
ostream & operator <<(ostream & out, Matrix m)
{
	int i,j;
	for(i=0;i<m.r;i++)
	{
		for(j=0;j<m.c;j++)
		{
		out<<m.a[i][j]<<"\t";
		}
		out<<endl;
	}
	return out;
}
int Matrix::operator==(Matrix m2)
{
	if((r==m2.r) && (c==m2.c))
		return 1;
	else
		return 0;
}
int main()
{
	Matrix m1,m2,m4;
	cout<<"enter the order of the first matrix\n";
	m1.getorder();
	cout<<"enter the order of the second matrix\n";
	m2.getorder();
	if(m1 == m2)
	{
		cout<<"enter the elements of the first matrix\n";
		m1.getdata();
		cout<<"enter the elements of the second matrix\n";
		m2.getdata();
		m4 = m1 - m2;
		cout<<"Difference of matrices is \n";
		cout<<m4<<endl;
	}
	else
	{
		cout<<"Order of the matrices is not same";
	}
	return 0;
}
```

5. **Which are the different access specifiers? Explain the effect of inheritance when deriving by different access specifiers with appropriate examples ((10))**

Ans : Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled public, private, and protected sections within the class body. The keywords public, private, and protected are called access specifiers.

A class can have multiple public, protected, or private labeled sections.

When the access specifier for a base class is public, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class.

When the base class is inherited by using the private access specifier, all public and protected members of the base class become private members of the derived class.

When the base class is inherited by using the protected access specifier, all public and protected members of the base class become protected members of the derived class.

Ex:

```
#include <iostream>
using namespace std;
class base
{
	protected:
	int i, j; // private to base, but accessible by derived
	public:
	void set(int a, int b) { i=a; j=b; }
	void show() { cout << i << " " << j << "\n"; }
};

class derived : public base
{
	int k;
	public:
	// derived may access base's i and j
	void setk() { k=i*j; }
	void showk() { cout << k << "\n"; }
};

int main()
{
	derived ob;
	ob.set(2, 3); // OK, known to derived
	ob.show(); // OK, known to derived
	ob.setk();
	ob.showk();
	return 0;
}
```

In this example, because base is inherited by derived as public and because i and j are declared as protected, derived's function setk( ) may access them. If i and j had been declared as private by base, then derived would not have access to them, and the program would not compile.

6 (a) **Explain the order of constructor and destructor called in multi level inheritance with example (4)**

It is possible for a base class, a derived class, or both to contain constructors and/or destructors .In case of multi level inheritance, the constructors are called in the order of derivation and destructors are called in reverse order.
EX:

```
#include <iostream>
using namespace std;
class base
{
	public:
	base() { cout << "Constructing base\n"; }
	~base() { cout << "Destructing base\n"; }
};
class derived1 : public base
{
	public:
	derived1() { cout << "Constructing derived1\n"; }
	~derived1() { cout << "Destructing derived1\n"; }
};

class derived2: public derived1
{
	public:
	derived2() { cout << "Constructing derived2\n"; }
	~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
	derived2 ob;
	// construct and destruct ob
	return 0;
}
```

The above program yields the following output

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

(b) **Write a cpp program which shows how a virtual function is called through a base class reference (6)**

```
#include <iostream>
using namespace std;
class base
{
	public:
	virtual void vfunc() {
	cout << "This is base's vfunc().\n";
```

```
        }
};

class derived1 : public base
{
        public:
        void vfunc()
        {

        cout << "This is derived1's vfunc().\n";
        }
};

class derived2 : public base
{
public:
        void vfunc()
        {
              cout << "This is derived2's vfunc().\n";
        }
};

// Use a base class reference parameter.

void f(base &r)
{
    r.vfunc();
}

int main()
{
        base b;
        derived1 d1;
        derived2 d2;
        f(b); // pass a base object to f()
        f(d1); // pass a derived1 object to f()
        f(d2); // pass a derived2 object to f()
        return 0;
}
```

In this example, the function f( ) defines a reference parameter of type base. Inside main( ), the function is called using objects of type base, derived1, and derived2. Inside f( ), the specific version of vfunc( ) that is called is determined by the type of object being referenced when the function is called.

7. (a) **Write a program to dynamically allocate memory for object (using constructor). (4)**

Ans:

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance
{
        double cur_bal;
        char name[80];
        public:
        balance(double n, char *s)
        {
                cur_bal = n;
                strcpy(name, s);
        }
        ~balance()
```

```
            {
                        cout << "Destructing ";
                        cout << name << "\n";
            }
            void get_bal(double &n, char *s)
            {
                        n = cur_bal;
                        strcpy(s, name);
            }
};

int main()
{
            balance *p;
            char s[80];
            double n;
            // this version uses an initializer
            try
            {
                    p = new balance (12387.87, "Ralph Wilson");
            }
            catch (bad_alloc xa)
            {
                    cout << "Allocation Failure\n";
                     return 1;
            }
            p->get_bal(n, s);
            cout << s << "'s balance is: " << n;
            cout << "\n";
            delete p;
            return 0;
}
```

(b) **Explain the dynamic memory allocation operator in c++. Explain proper syntax with example (6)**

Ans: Dynamic memory allocation operators: new and delete.

New: To allocate the memory.

Syntax:

```
        Ptr_var = new vartype;
```

e.g: ptr = new int;

```
        Ptr_var = new vartype(initial_value);
```

The type of initial value should be same as the vartype;

e.g: ptr = new int(100);

Delete: To free the memory.

```
        delete ptr_var;
```

If there is insufficient memory then the exception bad_alloc will be raised. This exception is defined in the header <new>. It is available in standard C++.

Advantages of new and delete:

new and delete operators are like malloc() and free() in C Language. But they have more advantages.

1. new automatically allocates enough memory to hold the object.

   (No need to use sizeof operator).

2. new automatically returns the pointer to the specified type.

   It is not needed to typecast it explicitly.

Allocating Arrays:

ptrvar = new arrtype[size];

Delete [ ] ptrvar;

** The initial values can't be given during the array allocation.

8 (a) **Create a base class with two protected data members i and j and two public methods setij() and showij() to set the values of I and j and display the values of i and j respectively. Create a derived class which inherits base class as protected. Show how derived class object sets the values of i and j and display their values. (6)**

Ans: 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;
}
```

As you can see by reading the comments, 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( ).

b) **What is the need for a virtual base class? Show how a virtual base class eliminates ambiguity (4)**

Ans: When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as virtual when it is inherited. You accomplish this by preceding the base class' name with the keyword virtual when it is inherited. For example, here is another version of the example program in which derived3 contains only one copy of base:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;
```

```
class base {
public:
int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2
{
	public:
	int sum;
};
int main()
{
	derived3 ob;
	 ob.i = 10; // now unambiguous
	ob.j = 20;
	ob.k = 30;
	// unambiguous
	ob.sum = ob.i + ob.j + ob.k;
	// unambiguous
	cout << ob.i << " ";
	cout << ob.j << " " << ob.k << " ";
	cout << ob.sum;
	return 0;
}
```

As you can see, the keyword virtual precedes the rest of the inherited class' specification. Now that both derived1 and derived2 have inherited base as virtual, any multiple inheritance involving them will cause only one copy of base to be present. Therefore, in derived3, there is only one copy of base and ob.i = 10 is perfectly valid and unambiguous. One further point to keep in mind: Even though both derived1 and derived2 specify base as virtual, base is still present in objects of either type. For example, the following sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;
myclass.i = 88;
```

The only difference between a normal base class and a virtual one is what occurs when an object inherits the base more than once. If virtual base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

9 (a) **What are pure virtual functions? Discuss its significance.** **(6)**

When a virtual function is not redefined by a derived class, the version defined in the base class will be used. When there is no meaningful definition of a virtual function within a base class ie., when a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created.
Thus we can ensure that all derived classes override a virtual function by using pure virtual function.
A pure virtual function is a virtual function that has no definition within the base class.
To declare a pure virtual function, use this general form:

```
	virtual type func-name(parameter-list) = 0;
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

Ex:

```
#include <iostream>
```

```
using namespace std;
class number
{
protected:
            int val;
public:

            void setval(int I)
             {
                        val = i;
            }
            // show() is a pure virtual function
            virtual void show() = 0;
};

class hextype : public number
 {
public:
            void show()
            {
                        cout << hex << val << "\n";
            }
};
class dectype : public number
{
public:
void show()
             {
                        cout << val << "\n";
            }
};

class octtype : public number
{
public:
            void show()
            {
                        cout << oct << val << "\n";
            }
};
int main()
{
            dectype d;
            hextype h;
            octtype o;
            d.setval(20);
            d.show(); // displays 20 - decimal
            h.setval(20);
            h.show(); // displays 14 – hexadecimal
            o.setval(20);
            o.show(); // displays 24 - octal
            return 0;

}
```

In the above example,a base class may not be able to meaningfully define a virtual function. In number class simply provides the common interface for the derived types to use. There is no reason to define show( ) inside number since the base of the number is undefined. We can always create a placeholder definition of a virtual function. By making show( ) as pure also ensures that all derived classes will redefine it to meet their own needs.

b) **Differentiate between early binding and late binding** **(4)**

Early Binding	Late Binding
1.Early binding refers to events that occur at compile time	1.Late binding refers to events that occur at run time
2.The information needed to call a function is known at compile time	2.The information needed to call a function is not known until run time
3.It is more efficient	3.It is more flexible
4.It is fast	4.It is slow
5.Example:function overloading	5.Example: virtual functions

10. **Create a class called complex which has two data members real part, imaginary part. Implement a friend function for overloading '+' operator which can compute the sum of two complex numbers and the function returns the complex object. (10)**

Ans :
```
#include<iostream>
using namespace std;
class complex

{
        int real;
        int imag;
   public:
        void read()
        {
                cout<<"enter real and imaginary";
                cin>>real>>imag;
        }
        void display()
        {
                        cout<<real<<"+"<<imag<<"i"<<endl;
        }
        friend complex operator +(complex, complex);
};

complex operator +(complex a1,complex a2)
        {
                complex temp1;
                temp1.real=a1.real+a2.real;
                temp1.imag=a1.imag+a2.imag;
                return temp1;
        }
int main()
{
        int a;
        complex s1,s2,s3;
        s1.read();
        s2.read();
        cout<<"First Complex number";
        s1.display();
        cout<<"Second Complex number";
        s2.display();
        s3=s1+s2;
        cout<<"addition of 2 complex number\n"<<endl;
        s3.display();
        return 0;
}
```