USN | | | | | | | | | | |

Internal Test II – April 2018

Sub: Object Oriented Programming Using C++ Sub Code: 16MCA22 Branch: MCA

Date: 16/04/18 Duration: 90 min's Max Marks: 50 Sem / Sec: A OBE

Answer any FIVE FULL Questions from each part.

| | | MARKS | CO | RBT |
|---|---|---|---|---|
| Part I-1 (a) | Explain the overloading of new & delete operator. | [10] | CO2 | L2 |
| 2 (a) | Explain the order of constructor and destructor called in multi level inheritance with example. | [5] | CO2 | L2 |
| (b) | Explain the concept of pointer to an object with an example. | [5] | 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 | L2 |
| (b) | Explain overloading of special operator [ ] . | [4] | CO2 | L2 |
| 4 (a) | Explain the dynamic memory allocation operator in C++. Explain proper syntax and example. | [6] | CO2 | L2 |
| (b) | Write a program to dynamically allocate memory for object (using constructor). | [4] | CO2 | L2 |
| Part III-5(a) | Which are the different access specifiers? Explain the effect of inheritance when deriving by different access specifiers with appropriate examples. | [10] | CO2 | L2 |
| 6(a) | Define a STUDENT class with USN, Name, and Marks in 3 tests of a subject. Declare an array of 10 STUDENT objects. Using appropriate functions, find the average of the two better marks for each student. Print the USN, Name and the average marks of all the students. | [10] | CO2 | L2 |
| Part IV-7(a) | Explain the usage of reference parameters with an example of swapping two numbers. | [5] | CO2 | L2 |
| (b) | Write a program to find the difference of two numbers using default arguments. | [5] | CO2 | L2 |
| 8(a) | 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 |
| Part V-9(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 |
| 10 | 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 |

**1.a) Explain the overloading of new & delete operator.**

It is possible to overload **new** and **delete**.

.
The skeletons for the functions that overload **new** and **delete** are shown here:

```
// Allocate an object.
void *operator new(size_t size)
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor called automatically. */
return pointer_to_memory;
}
// Delete an object.
void operator delete(void *p)
{
/* Free memory pointed to by p.
Destructor called automatically. */
}
```

The type **size_t** is a defined type capable of containing the largest single piece of memory that can be allocated. The overloaded **new** function must return a pointer to the memory that it allocates, orthrow a **bad_alloc** exception if an allocation error occurs.

The **delete** function receives a pointer to the region of memory to be freed. It then releases the previously allocated memory back to the system. When an object is deleted, its destructor is automatically called.

To overload the **new** and **delete** operators for a class, simply make the overloaded operator functions class members.
For example, here the **new** and **delete** operators are overloaded for the **loc** class**:**

```
#include <iostream>
#include <cstdlib>
#include <new>
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";
}
void *operator new(size_t size);
void operator delete(void *p);
};
// new overloaded relative to loc.
void *loc::operator new(size_t size)
{
void *p;
```

Output from this program is shown here.

```
In overloaded new.
In overloaded new.
10 20
```

-10 -20
In overloaded delete.
In overloaded delete.

When **new** and **delete** are for a specific class, the use of these operators on any other type of data causes the original **new** or **delete** to be employed. The overloaded operators are only applied to the types for which they are defined. This means that if you add this line to the **main( ),** the default **new** will be executed:

```
int *f = new float; // uses default new
```

You can overload **new** and **delete** globally by overloading these operators outside of any class declaration.

To see an example of overloading **new** and **delete** globally, examine this program:

```
#include <iostream>
#include <cstdlib>
#include <new>
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";
}
};
// Global new
void *operator new(size_t size)
{
void *p;
p = malloc(size);
if(!p) {
bad_alloc ba;
throw ba;
}
return p;
}
// Global delete
void operator delete(void *p)
{
free(p);
}
int main()
{
loc *p1, *p2;
float *f;
try {
p1 = new loc (10, 20);
} catch (bad_alloc xa) {
cout << "Allocation error for p1.\n";
return 1;;
}
```

```
try {
p2 = new loc (-10, -20);
} catch (bad_alloc xa) {
cout << "Allocation error for p2.\n";
return 1;
}
try {
f = new float; // uses overloaded new, too
} catch (bad_alloc xa) {
cout << "Allocation error for f.\n";
return 1;;
}
*f = 10.10F;
cout << *f << "\n";
p1->show();
p2->show();
delete p1;
delete p2;
delete f;
return 0;
}
```

Overloading new and delete for Arrays

To allocate and free arrays,you must use these forms of **new** and **delete**.

```
// Allocate an array of objects.
void *operator new[](size_t size)
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor for each element called automatically. */
return pointer_to_memory;
}
// Delete an array of objects.
void operator delete[](void *p)
{
/* Free memory pointed to by p.
Destructor for each element called automatically.
*/
}
```

Overloading the nothrow Version of new and delete

You can also create overloaded **nothrow** versions of **new** and **delete**.

```
// Nothrow version of new.
void *operator new(size_t size, const nothrow_t &n)
{
// Perform allocation.
if(success) return pointer_to_memory;
else return 0;
}
// Nothrow version of new for arrays. void *operator new[](size_t size, const nothrow_t &n)
{
// Perform allocation.
if(success) return pointer_to_memory;
else return 0;
}
void operator delete(void *p, const nothrow_t &n)
{
```

```
// free memory
}
void operator delete[](void *p, const nothrow_t &n)
{
// free memory
}
```

**2a) Explain the order of constructor and destructor called in multi level inheritance with example**

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
```

Q2 b: Explain the concept of pointer to an object with an example.

Ans:

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

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;

}
```

**3.b) Explain overloading of special operator [ ] .**

Ans: In C++, the **[ ]** is considered a binary operator when you are overloading it. Therefore, the general form of a member **operator[ ]( )** function is as shown here:
*type class-name*::operator[](int *i*)
{
// . . .
} Given an object called **O**, the expression
O[3]
translates into this call to the **operator[ ]( )** function:
O.operator[](3)

Example:

```
 #include <iostream>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i) { return a[i]; }
};
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] on left of =
cout << ob[1]; // now displays 25
return 0;
}
```

Q 4. a) **Explain the dynamic memory allocation operator in C++. Explain proper syntax and example.**

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.

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

**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;

}
```

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

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.

Q 6 a: **Define a STUDENT class with USN, Name, and Marks in 3 tests of a subject. Declare an array of 10 STUDENT objects. Using appropriate functions, find the average of the two better marks for each student. Print the USN, Name and the average marks of all the students.**

Ans:

```
#include <iostream>

using namespace std;

// Declare a class with appropriate member functions and member variables as in the problem statement

class Student

{

        char usn[11];

        char name[15];

        int m1,m2,m3;

        float avg;

        public :

                void readstudent();

                void calcavg();

                void display();

};

//Function to read the student details from the user

void Student :: readstudent()

{

        cout<<"Enter the usn\n";

        cin>>usn;

        cout<<"Enter the name\n";

        cin>>name;

        cout<<"enter the marks of 3 subjects\n";

        cin>>m1>>m2>>m3;

}

// Function to calculate better of two better marks and to find the average of them
```

```
void Student::calcavg()
{
	float small;
	small = ((m1<=m2)?((m1<=m3)?m1:m3):((m2<=m3)?m2:m3));
	avg = (float)((m1+m2+m3) - small)/2;
}
// Function to display the student details
void Student :: display()
{
	cout<<usn<<"\t"<<name<<"\t"<<avg<<endl;
}
int main()
{
	Student st[10];
	int i,n;
	cout<<"Enter the number of students\n";
	cin>>n;
	for(i=0;i<n;i++)
	{
		st[i].readstudent();
	}
	for(i=0;i<n;i++)
	{
		st[i].calcavg();
	}
	cout<<"USN \t Student Name \t Average \n";
	for(i=0;i<n;i++)
	{
```

```
            st[i].display();
        }
        return 0;
}
```

Q 7a: **Explain the usage of reference parameters with an example of swapping two numbers.**

Ans:

```
#include<iostream>
using namespace std;
template <class X> void swapargs(X &a, x &b)
{
    X temp;
    temp=a;
    a=b;
    b=temp;
}
int main()
{
    int i=10,j=20;
    double x=10.5,y=90.8;
    cout<<"Original i,j:"<<i<<' '<<j<<'\n';
    cout<<"Original x,y:"<<x<<' '<<y<<'\n';
    swapargs(i,j);
    swapargs(x,y);
    cout<<"Swapped  i,j:"<<i<<' '<<j<<'\n';
    cout<<"Swapped x,y:"<<x<<' '<<y<<'\n';
    return 0;
}
```

Q 7 b: **Write a program to find the difference of two numbers using default arguments.**

Ans:

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

  void diff(int a,int b= 6);
  int main( )
  {
     int a,b;
     cout<<"enter any two numbers\n";
     cin>>a>>b;
```

```
    diff(a) ; // sum of default values
    diff(a,b);
    diff(b);
    return 0;
 }
 void diff (int a1, int a2)
 {
    int temp;
    temp = a1 - a2;
    cout<<"a="<<a1<<endl;
    cout<<"b="<<a2<<endl;
    cout<<"Difference="<<temp<<endl;
 }
```

Q 8 a:**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.**

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;
}
```

Q 9 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.**

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( )**.

Q 9 b: **What is the need for a virtual base class? Show how a virtual base class eliminates ambiguity.**

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.

Q 10: **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).**

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;
}
```