CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | |

Internal Assesment Test - II

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

Date: 08.05.2017 Duration: 90 mins Max Marks: 50 Sem: II Branch: MCA

Answer Any FIVE FULL Questions

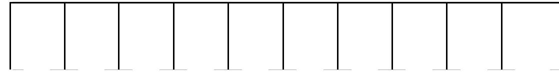
		Marks	OBE CO	OBE RBT
1(a)	What are virtual functions? With an example demonstrate the use of virtual functions.	[10]	CO2	L2
2(a)	Explain the overloading of new & delete operator.	[10]	CO2	L2
3(a)	**What are pure virtual functions? Discuss its significance.**	[5]	CO2	L2
(b)	Differentiate between early binding and late binding	[5]	CO2	L2
4(a)	) Explain the order of constructor and destructor called in multi level inheritance with example.	[4]	CO2	L2
(b)	Write a cpp program which shows how a virtual function is called through a base class reference.	[6]	CO2	L2
5(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
6(a)	Which are the different access specifiers? Explain the effect of inheritance when deriving by different access specifiers with appropriate examples.	[10]	CO2	L2
7(a)	What is template? Explain template function and template class with example.	[5]	CO2	L2
(b)	Create a class stack using template. Use this class to create a stack with integer and double elements.	[5]	CO2	L3
8(a)	What is Exception? Explain the use of try, catch and throw with example.	[6]	CO2	L2
(b)	Write a program for the exception division by zero.	[4]	CO2	L2

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8
CO1:	Differentiate between object oriented programming and procedure oriented programming & Disseminate the importance of Object oriented programming	1	1	-	-	-	-	3	3
CO2:	Apply C++ features such as Classes, objects, constructors, destructors, inheritance, operator overloading, and Polymorphism, Template and exception handling in program design and implementation.	2	2	3	-	-	-	2	3
CO3:	Use C++ to demonstrate practical experience in developing object-oriented solutions.	1	3	3	1	-	-	3	3
CO4:	Analyze a problem description and build object-oriented software using good coding practices and techniques.	1	2	3	2	-	-	3	3
CO5:	Implement an achievable practical application and analyze issues related to object-oriented techniques in the C++ programming language.	1	1	2	-	-	-	3	3

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1 - Apply *knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 - team work; PO5 - *Ethics*; PO6 - Communication; PO7- *Business Solution*; PO8 – Life-long learning;

CMR
INSTITUTE OF
USN
TECHNOLOGY



Internal Assessment Test 2 –April 2016

Sub:	Object Oriented Programming Using C++							Code:	16MCA 22
Date:	*08/05/2017*	*Duration:*	*90 mins*	*Max Marks:*	*50*	*Sem:*	*2*	*Branch:*	*MCA*

Answer any five of the following. 5X10=50M

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

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.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
}
```

3.a) What are pure virtual functions? Discuss its significance. 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 owndefinition. 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 (5M)

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

4a) **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

4b. Write a cpp program which shows how a virtual function is called through a base class reference. (10M)

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";

}

};

// 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, thefunction 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 specificversion of vfunc( ) that is called is determined by the type of object being referenced when the function is called.

5a)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;
```

}

5.b) Explain overloading of special operator [ ] .

6a.Which are the different access specifiers? Explain the effect of inheritance when deriving by different access specifiers with appropriate examples.

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.

7.a)What is template? Explain template function and template class with example.

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.

The general form of a template function definition is shown here:

template <class *Ttype*> *ret-type func-name*(*parameter list*)
{
// *body of function*
}

Here, *Ttype* is a placeholder name for a data type used by the function. This name may be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function.

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;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
return 0;
}
```

In addition to generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

```
/* This example uses two generic data types in a
class definition.
*/
#include <iostream>
using namespace std;
template <class Type1, class Type2> class myclass
{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b) { i = a; j = b; }
void show() { cout << i << ' ' << j << '\n'; }
};
int main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X', "Templates add power.");

ob1.show(); // show int, double
ob2.show(); // show char, char *
return 0;
}
```

This program produces the following output:

10 0.23
X Templates add power.

b)Create a class stack using template. Use this class to create a stack with integer and double elements.

```
#include <iostream>
using namespace std;
const int SIZE = 10;
```

```
// Create a generic stack class
template <class StackType> class stack {
StackType stck[SIZE]; // holds the stack
int tos; // index of top-of-stack
public:
stack() { tos = 0; } // initialize stack
void push(StackType ob); // push object on stack
StackType pop(); // pop object from stack
};
// Push an object.
template <class StackType> void stack<StackType>::push(StackType ob)
{
if(tos==SIZE) {
cout << "Stack is full.\n";
return;
}
stck[tos] = ob;
tos++;
}
// Pop an object.
template <class StackType> StackType stack<StackType>::pop()
{
if(tos==0) {
cout << "Stack is empty.\n";
return 0; // return null on empty stack
}
tos--;
return stck[tos];
}
int main()
{
// Demonstrate character stacks.
stack<char> s1, s2; // create two character stacks
int i;
s1.push('a');
s2.push('x');
s1.push('b');
s2.push('y');
s1.push('c');
s2.push('z');
for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";
// demonstrate double stacks
stack<double> ds1, ds2; // create two double stacks
ds1.push(1.1);
ds2.push(2.2);
ds1.push(3.3);
ds2.push(4.4);
ds1.push(5.5);
ds2.push(6.6);
for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";
return 0;
}
```

8.a)What is Exception? Explain the use of try, catch and throw with example.

Exception is run-time errors in program. Using exception handling,
your program can automatically invoke an error-handling routine when an error occurs.
Exception Handling Fundamentals
C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The following discussion elaborates upon this general description.
Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.) Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown. The general form of **try** and **catch** are shown here.

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}...
catch (typeN arg) {
// catch block
}

int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";

return 0;
}
```

This program displays the following output:
Start
Inside try block
Caught an exception -- value is: 100
End

b)Write a program for the exception division by zero.

```
#include <iostream>
```

```
using namespace std;
void divide(double a, double b);
int main()
{
double i, j;
do {
cout << "Enter numerator (0 to stop): ";
cin >> i;
cout << "Enter denominator: ";
cin >> j;
divide(i, j);
} while(i != 0);
return 0;
}
void divide(double a, double b)
{
try {
if(!b) throw b; // check for divide-by-zero
cout << "Result: " << a/b << endl;
}
catch (double b) {
cout << "Can't divide by zero.\n";
}
}
```