B.E.

Fourth Semester Examination, Dec-2006

OBJECT ORIENTED PROGRAMMING USING C++

Note : Attempt any five questions. All questions carry equal marks.

Q. 1. (a) List the restrictions and limitations in overloading operators.

Ans. The following restrictions apply to operator overloading :

1. Invention of new operators is not allowed. For example :

   void operator @ (int); //illegal, @ is not a built-in operator or a type name.

2. Neither the precedence nor the number of arguments of an operator may be altered. An overloaded && for example, must have exactly two arguments-just like the built-in && operator.

3. The following operators cannot be overloaded :

Direct member access operator	*
De-reference pointer to class member operator	.
Scope resolution operator	::
Conditional operator	::
Size of operator	?:
Size of operator	size of

Similarly, and of the new casting operators : static_cast <>, dynamic_cast <>, reinterpret_cast<> and const_cast<>, as well as the # and ##preprocessor tokens, may not be overloaded.

Q. 1. (b) What is type conversion? Discuss following, using examples :

(i) Conversion from basic type to class type

(ii) Conversion from class type to basic type.

Ans. Answer type conversions :

An expression of a given type is implicitly converted in the following situations :

* The expression is used as an operand of an arithmetic or logical operation.
* The expression is used as a condition in a if statement or an iteration statement (such as a for loop). The expression will be converted to a Boolean (or an integer in C89).
* The expression is used in a switch statement. The expression will be converted to an integral type.
* The expression is used as an initialization. This includes the following :
* An assignment is made to an value that has a different type than the assigned value.
* A function is provided an argument value that has a different type than the parameter.

type of the function.

Use-defined conversions (C++ only) : User-defined conversions allow you to specify conversions with constructors how with conversion functions. User-defined conversions are implicitly used in addition to standard conversions for conversion of initializes, functions arguments, function return values, expression operands, expressions controlling iteration, section statements, and explicit type conversions.

There are two types of user-defined conversions :

* Conversion by constructor
* Conversion functions

You can define a member function of a class, called a conversion function, that converts from the type of its class to another specified type.

Conversion function syntax

```
>>- + ..... + ..... operator ... + ..... + ..... conversion_type.....>
'-class--::-'     +-const----+
                  '-volatile-'
>--+.....+...(..) .... + ..... + ..... >
|. ..... . |     '-{.....function_body..)-'
| V       -||
'....pointer_operator - +-'
```

A conversion function that belongs to a class X specifies a conversion from the class type X to the type specified by the conversion_type.

Q. 2. (a) What is a constructor? Explain the use of dynamic constructor, using suitable example.

Ans. A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the value of data members of the class. The construction is declared and defined as follows :

```
class integer
{
        int m, n;
        public ;
        inter (void);
        ....
        ....
};
integer : : integer (void)
{
```

```
    m = 0, n = 0;
}
```

The dynamic initialization of constructions is also possible. The initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization format using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Q. 2. (b) Differentiate between :

(i) Struct and class

(ii) Constructor and destructor.

Ans. (i) Struct and Class :

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

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

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

```
class class_name {
access_specifier_1;
    member 1;
access_specifier_2;
    member 2;
    ...
} object_names;
```

Structure : A structure is a collection of variables under a single name. These variables can be of different types and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.

Defining a structure : A structure type is usually defined near to the start of a file using a typedef statement.typedef defines and names a new type, allowing its use throughout the program.typedefs usually occur just after the #define and #include statements in a file.

Here is an example structure definition.

```
typedef struct {
char name [64];
char course [128];
int age;
```

```
int year;
} student;
```

(B) Constructors and Destructors : A class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void.

We are going to implement CRectangle including a constructor :

```
//example : class constructor
# include <iostream>
using namespace std;
class CRectangle {
int width, height;
public :
CRectangle (int, int);
int area ( ) {return (width *height);}
};
CRectangle :: CRectangle (int a, int b) {
width = a;
height = b;
}
int main ( ) [
CRectangle rect (3, 4);
CRectangle rectb (5,6);
cout << "rect area : "<<rect.area () << endl
cout << "rectb area : "<<rectb. area () <<endl;
return 0;
}
```

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (!) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// example on constructors and destructors
```

```
#include<iostream>
using namespace std;
class CRectangle {
int *width, * height;
public:
CRectangle (int, int);
~CRectangle ( );
int area ( ) {return (*width **height);}
};
CRectangle :: CRectangle (int a, int b)}
width = new int;
height = new int;
*width = a;
*height = b;
}
CRectangle : : ~ CRectangle ( ) {
delete width;
delete height;
}
int main ( ){
CRectangle rect (3, 4), rectb (5, 6);
cout << "rect area:" <<rect. area ( ) << endl;
cout << "rectb area : "<<rectb.area ( ) <<endl;
return 0;
}.
```

Q. 3. (a) Differentiate between procedural abstraction and data abstraction.

Ans. Procedural abstraction : One of the main purposes of using functions is to aid in the top down design of programs. During the design stage, as a problem is subdivided into tasks (and then into sub-tasks, sub-sub-tasks, etc.), the problem solver (programmer) should have to consider only what a function is to do and not be concerned about the details of the function. The function name and comments at the beginning of the function should be sufficient to inform the user as to what the function does. (Indeed, during the early stages or program development, experienced programmers often use simple "dummy" functions or stubs, which simply return an arbitrary value of the correct type, to best out the control flow of the main or higher level program component).

Developing functions in this manner is referred to as functional or procedural abstraction. This process is

aided by the use of value parameters and local variables declared within the body of a function. "Functions written in this manner can be regarded as "black boxes." As users of the function, we neither know nor care why they work.

Data Abstraction : It refers to the act of representing essential features without including the background details or explanation. Classes uses the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and function to operate these attributes. The function that operate on these data are called methods or member functions.

Q. 3. (b) What do you understand by an abstract class? Explain.

Ans. An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (=0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class :

```
class AB {
public :
virtual void f ( ) = 0;
};
```

Function AB :: f is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following :

```
struct A {
virtual void g ( ) { } = 0;
};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this :

```
struct A {
virtual void f ( ) = 0;
};
struct B : A [
virtual void f ( ) { }
};
// Error :
// Class A is an abstract class
// A g ( );
// Error :
```

```
// Class A is an abstract class
// void h (A);
A & i (A &);
int main ( ) {
// Error :
// Class A is an abstract
// A ;
A* pa;
B b;
// Error :
// Class A is an abstract class
// Static_cast <A> (b);
}.
```

Q. 4. (a) What do you understand by a header file? Discuss the use of iostream header file.

Ans. Header files are basically used to include the built in functions for the program. We have used the following # include directive is the program.

```
# include <iostream>
```

This directive cause the preprocessor to add the contents of the iostream file to program. It contains declaration for the identifier cout and operator <<. Some versions of C++ uses the header file called iostream.n. The header file iostream should be included at the beginning of all program that use input/output statement. Some implementations use isotream. hpp yet other uses iostream. h∧x.i .

Q. 4. (b) Differentiate between macro and function, using suitable examples.

Ans. Functions are segments of code that allow you to better organize your code. You can think of a function as a small program, and of a program as a collection of functions. I could have written a function for the "Hellow World" program :

```
# include < iostream.h>
void print_hellow ( ) {          // This line declares the function
cout << "Hellow World!\n";       // This is the body, which defines the function
}
void main ( ) [
print _hellow ( );               // This is how the function is called
}
```

* If the function returns a value then the type of that value must be specified in function-type. For the

moment this could be int, float or char. If the function does not return a value then the function-type must be void.

- The function-name follows the same rules of composition as identifiers.
- The parameter-list lists the formal parameters of the function together with their types.
- The local-definitions are definitions of variables that are used in the function-implementation. These variables have no meaning outside the function.
- The function-implementation consists of C++ executable statements that implement the effect of the function.

Macro : C++ offers new capabilities, some of which supplant those offered by the ANSIC preprocessor. These new capabilities enhance the type safety and predictability of the language :

- In C++, objects declared as const can be used in constant expressions. This allows programs to declare constraints that have type and value information, and enumerations that can be viewed symbolically with the debugger. Using the preprocessor #define directive to define constants is not as precise. No storage is allocated for a const object unless an expression that takes its address is found in the program.
- The C++ in line function capability supplants function_type macros. The advantages of using on-line functions over macros are :
- Type safety. In line functions are subject to the same type checking as normal functions. Macros are not type safe.
- Correct handling of arguments that have side effects. In line functions evaluate the expressions supplied as arguments prior to entering the function body. Therefore, there is no chance that an expression with side effects will be unsafe.

The #undef directive removes the definition of a macro. Once you have removed the definition, you can redefine the macro to a different value. The #define Directive and The # undef Directive discuss the #define and #undef directives, respectively.

Q. 5. (a) Differentiate between static and dynamic linking.

Ans. The overloaded member functions are selected for invoking by matching arguments both type and number. This information is known to the compiler at the compile time and therefore compiler is able to select the appropriate function for a particular call at compile time itself. This is called early binding or static binding or static linking. It is also called compile time polymorphism.

At run time, when it is known what class objects are under consideration, he appropriate version of function is invoked. Since the function is linked with particular class much later after the compilation. This process is termed as late binding. It is also called dynamic binding because the selection of appropriate function is done dynamically at runtime.

Q. 5. (b) What is a pure virtual function? Justify its use in an example.

Ans. A "do nothing" function may be defined as follows :

```
virtual void display = 0;
```

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. A class containing pure virtual function cannot be used to declare any objects of its own.

Q. 6. (a) Differentiate between put ( ) and write ( ) and get ( ) and read ( ).

Ans. The classes isotream and ostream define two member functions get ( ) and put ( ) to handle the single character input/output operations. There are two types of get ( ) functions. We can use both get (char*) and get (void) prototypes to fetch a character including the blank space, tab and newline character. The get (char*) version assigns the input character to its argument and get (void) returns the input character. The function put ( ) as member of ostream class can be used to output a line of text, character by character.

Eg.

```
. Cout. Put ('X')
"The write (line, size).
```

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display.

The write and read ( ) handle the data in binary form. This means that the values are stored in the disk file in same format in which they are stored in internal memory. The binary input and output functions takes following form.

```
infile.read ((char*) & V, size of (v));
Outfile.write ((char*) & V, size of (v));.
```

Q. 6. (b) Discuss the following file mode parameters : ios ::app, ios::noreplace, ios::trunk, ios::out.

Ans. ios::app : Append to end of file. It takes us to end of file when it is opened. It can be used with the files capable of output.

ios::no replace : opens the file if already exists.

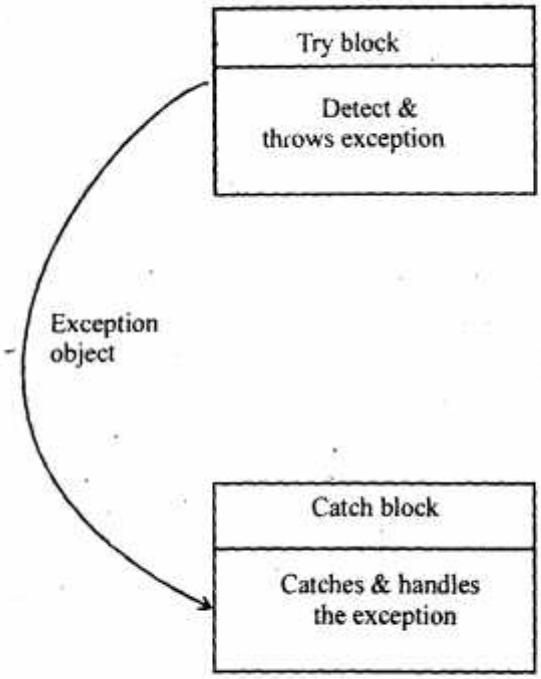
ios::trunk : Delete the contents of file if it exists.

ios::out : Open the file for writing only.

Q. 7. (a) What is an exception? How is an exception handled in C++?

Ans. C++ exception handling mechanism is basically built upon three keywords namely try, throw, and catch. The keyword try is used to preface a block of statement which may generate exceptions. This block of statements is known as try block. When exception is detected, it is thrown using a throw statement in the try block. A catch block defined by the keyword "catch" catches the exception thrown by the throw statement in the try block and handles it appropriately.

Eg.



Q. 7. (b) Write a program to demonstrate the concept of rethrowing an exception.

Ans.

```
# include <iostream.h>
void divide (double x, double y)
{
cout <<"inside function";
        try
{
        if (y = =0.0)
        throw y;
        else
        cout << "Division =" << x/y;
}
        catch (double)
```

```
{
        cout << "caught double inside function";
        throw;
}
        cout << "end of function";
}
        int main ( )
{
        cout <<"inside main";
        try
{
        divide (10.5, 2.0);
        divide (20.0, 0.0);
}
        catch (double)
{
        cout <<"caught double inside main";
}
        cout << "end of main";
        return 0;
}.
```

Q. 8. Discuss the effect of inheritance on the visibility of members in :

(a) Public derivation

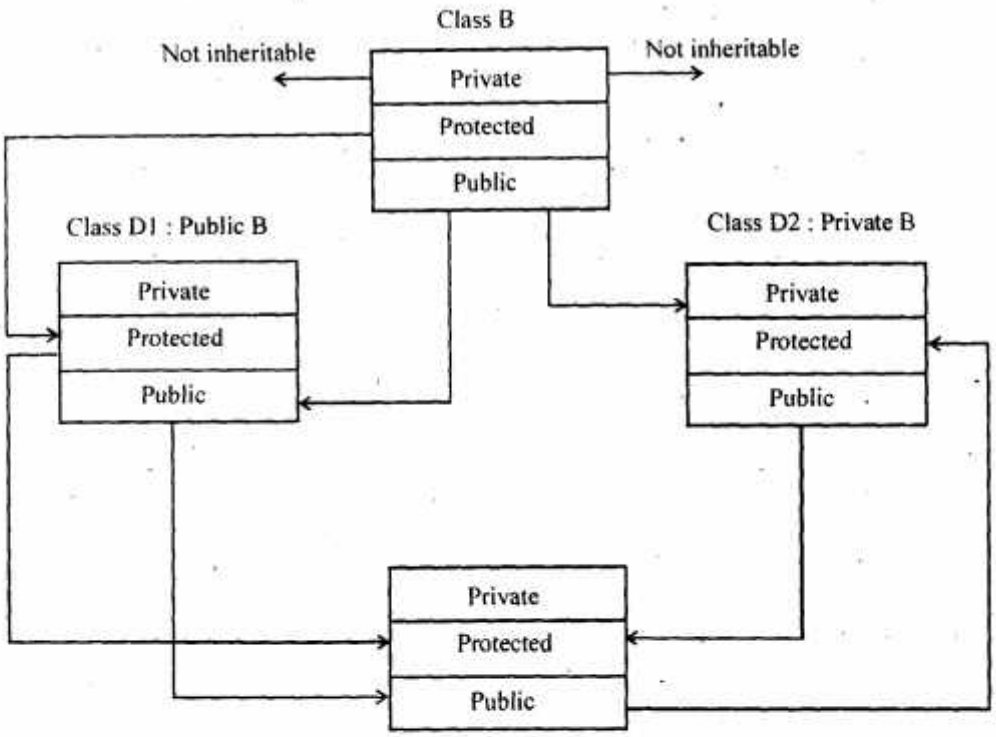
(b) Private derivation

(c) Protected derivation.

Ans.

The keywords private, protected and public may appear in any order and any number of times in the declaration of class eg.

```
Class beta
{
        protected :
        ..... .
        Public :
```



.....

Private :

.....

Public :

.....

};

is a valid class definition.

Base Class Visibility	Derived class visibility		
	Public derivation	Private	Protected
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected