B. E.

Fourth Semester Examination, Dec.-2007

OBJECT ORIENTED PROGRAMMING USING C++

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

Q. 1. (a) What are preprocessor directives. Why these-are needed?

Ans. Preprocessor directives are lines included in the code in programs that are not program statements but directives for the preprocessor. These lines are always preceded by a pound sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements. These preprocessor directives extend only across a single line of code. As soon as a new line character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

These are needed for following purpose :

1. Pragma directive #pragma) :

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you see. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with #pragma. If the compiler does not support a specific argument for #pragma, it is ignored–no error is generated.

2. Macro definitions (# define, #undef) :

To define preprocessor macros we can use #define. Its format is :

# define identifier replacement : When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block of simply anything. The preprocessor does not understand C++, it simply replaces any occurrence of identifier by replacement.

3. Conditional inclusions (#fdef, #fndef, #if, #endif, # else and #elif) :

These directives allow to include or discard part of the code of a program if a certain condition is met. # ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is.

4. Line control (#line) :

When we compile a program and some error happen during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error. The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is :

#line number "filename".

Q. 1. (b) What are header files & library files? Why these are needed?

Ans. The C++ library includes the same definitions as the C language library organized in the same structure of header files, with the following differences :

```
{public :
void SomeFunction (int &x) {x* = x;}
int SomeFunction (int x) {return x* x;}};
// In main ( )
SomeClass s;
        int x = 5;
        x = SomeFunction (x);
```

The compiler knows to call the second implementation of the method because we are assigning the return value from the function to x.

(ii) public : can be accessed from anywhere

private : only accessible from within the class

protected : private except classes that inherit from the class with the protected member can accesses it.

Protected members in C++ can only be accessed by the members of the class that original declared the members, friends of the class, classes derived with public or protected access from the class that declared the members or classes that directly derived from the class with the protected members. This means that in C++ if you have :

```
class A {
protected :
int a;};
class B : private A { };
class C : public B { };
```

Then B can access A::a, but C cannot.

Q. 3. (a) What is destructor? Where it has been used? Give example.

Ans. Destructors are usually used to decollate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted. A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example :

```
class X {
public :
//Constructor for class X
X ( );
//Destructor for class X
~ X ( ); }
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared const, volatile, const votalite or static. A destructor can be declared virtual or pure virtual. If no userdefined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an in line public member of its class.

- Each header file has the same name as the C language version but with a "c" prefix and no extension. For example, the C++ equivalent for the C language header file <stdlib.h> is <cstdlib>.
- Every element of the library is defined within the std namespace.

Nevertheless for compatibility with C, the traditional header names.h (like stdlib.h) are also provided with the same definitions within the global namespace. In the examples provided in this reference, this version is used so that the examples are fully C-compatible, although its use is deprecated in C++.

The are also certain specific changes in the C++ implementation :

- wchar_t is a fundamental type in C++ and therefore does not appear as a defined type in the corresponding header files where it appears in C. The same applies to several macros introduced by ammendment 1 to ISO C in the header file <iso646.h>, which are keywords in C++.
- The following functions have changes in their declarations related to the constness of their parameters : strchr, strpbrk, strrchr, strstr, memchr.
- The functions atexit, exit and abort, defined in <cstdlib> have additions to their behavior in C++.
- Overloaded versions of some functions are provided with additional types as parameters and the same semantics, like float and long double versions of the functions in the cmath header file or long versions for abs and div.

Q. 2. Differentiate between the following with examples :

(i) Overloading & Overriding,

(ii) Public, private & protected access type.

Ans. (i) In C++, overriding is a concept used in inheritance which involves a base class implementation of a method. Then in a subclass, you would make another implementation of the method. This is overriding. Here is a simple example.

```
Class Base
{public :
virtual void DoSomething {x = x+5;}
private :
        int x;};
        class Derived : public Base
        {public :
        virtual void DoSomething ( ) {y = y + 5; Base :: DoSomething ();}
        private :
        int y;};
```

Overloading is when you make multiple versions of a function. The compiler figures out which function to call by either :

1. The different parameters the function takes, or
2. The return type of the function. If you use the same function declaration, then you will get a compiler error because it will not know which function to use. Here is another simple example.

```
        class SomeClass
```

Ans. If we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space. The answer is dynamic memory, for which C++ integrates the operators new delete.

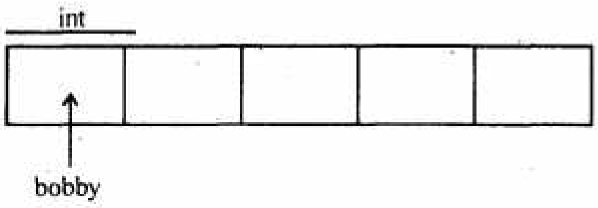
Operators new and new [] : In order to request dynamic memory we use the operator new. New is followed by a data type specifier and –if a sequence of more than one element is required–the number of these within brackets [ ]. It returns a pointer to the beginning of the new block of memory allocated. Its form is :

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to assign a block (an array) of elements of type type, where number_of_eleemnts is an integer value representing the amount of these. For example :

```
)bby;
= new
```

In this case, the system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby. Therefore, now, bobby points to a valid block of memory with space for five elements of type int.



Q. 3. (c) Explain, what is a friend class?

Ans. Friend classes :

As we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
class
<iostream>
namespace std;
Square;
Rectangle {
dth, heiht;
.... ( )
.... (with * height);}
convert (CSquare a);
Square {
set_side (int a)
= a;}
```

```
class CRectangle;
Rectangle::convert (CSquare a){
= a.side;
          = a.side;
          ( ) {
          are sqr;
          angle rect;
          _side (4);
          convert(sqr);
          <rect.area ( );
          0;.
```

Q. 4. (a) What are restrictions on operator overriding?

Ans. Overloading operators :

You cannot change the precedence of an operator.

- The associativity cannot be changed.
- You cannot use default arguments operator.
- You cannot change the number of arguments.
- You cannot create new operators.
- The meaning of how an operator works with built-in types, such as int, remains the same.
- Operators can be overloaded either for objects of the user-defined type or for a combination of objects of the user-defined type and objects of the built-in type.

Q. 4. (b) How the operators can be overloaded?

Ans. An overloaded operator is called an operator function. You declare an operator function with the keyword operator preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator. Consider the standard + (plus) operator. When this operator is used with operands of different standard types, the operators have slightly different meanings. For example, the addition of two integers is not implemented in the same way as the addition of two floating-point numbers. C++ allows you to define your own meanings for the standard C++ operators when they are applied to class types. In the following example, a class called complex is defined to model complex numbers, and the + (plus) operator is redefined in this class to add two complex numbers.

```
//This example illustrates overloading the plus (+) operator.
#include <iostream>
class complx
{         double real,
          imag;
public : complx (double real = 0., double imag = 0.); //constructor
complx operator + (const complx &) const; //operator + ( )};
```

```
// define constructor
complx::complx (double r, double i)
{ real = r; imag = i;}
// define overloaded + (plus) operator
complx complx::operator + (const complx &) const
{         complx result;
          result.real =(this->real + c.real);
          result.imag = (this->imag + c.imag);
          return result;}
          int main ( )
{         complx x (4, 4);
          complx y (6, 6);
          complx z = x+y; //calls complx::operator+()}.
```

Q. 5. (a) Explain Inheritance with example.

Ans. Creating or deriving a new class using another class as a base is called inheritance in C++. The new class created is called a Derived class and the old class used as a base is called a Base class in C++ inheritance terminology. The derived class will inherit all the features of the base class in C++ inheritance. The derived class can also add its own features, data etc., It can also override some of the features (functions) of the base class, if the function is declared as virtual in base class. C++ inheritance is very similar to a parent-child relationship. When a class is inherited all the functions and data member are inherited, although not all of them will be accessible by the member functions of the derived class. But there are some exceptions to it too.

Inheritance Example :

```
class exforsys {
public :
exforsys (void) {x = 0;}
void f (int n1) {x = n1 * 5}
void output (void) {cout << x;}
          private ; int x;};
          class sample : public exforsys {
public :
sample (void) {s1 = 0;}
void f1 (int n1) {s1 = n1*10;}
void output (void) {exforsys : : output ( );
          cout <<s1;}
          private : int s1};
          int main (void) {sample s;
          s.f. (10);
          s.output ( ); s.f1 (20);
          s. output ( );
}.
```

Q. 5. (b) Differentiate between Composition & Inheritance.

Ans. Composition allows software to be developed by assembling existing components rather than creating new ones. Composition is also sometimes called as aggregation and defines the process of putting an object inside another object. It models the has-a relationship. E.g . a class employee can contain an object of type salary which itself is another type of object. e.g.

```
class Salary
{       private : double DA;
        double HRA;
        double travelAllowance;
        double basic salary;
        public :
//      member functions};
        Clasls Employee
{       private :
        int empno;
Salary salary; //composition : salary object is contained inside employee.
public :
//member functions};
```

Inheritance :

Creating or deriving a new class using another class as a base is called inheritance in C++. The new class created is called a Derived class and the old class used as a base is called Base class in C++ inheritance terminology. The derived class will inherit all the features of the base class in C++ inheritance. The derived class can also add its own features, data etc., It can also override some of the features (functions) of the base class, if the function is declared as virtual in base class.

Q. 6. (a) Write short note on virtual functions.

Ans. C++ virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. The whole function body can be replaced with a new set of implementation in the derived class. The concept of C++ virtual functions is different from C++ Function overloading. C++ Virtual Function–Properties :

C++ virtual function is,

- A member function of a class.
- Declared with virtual keyword.
- Usually has a different functionality in the derived class
- A function call is resolved at run-time.

The difference between a non-virtual C++ member function and a virtual member function is, the non-virtual member functions are resolved at compile time. This mechanism is called static binding. Where as the C++ virtual member functions are resolved during runt-time. This mechanism is known as dynamic binding. C++ Virtual function—Example :

```
Class Window //Base class for C++ virtual function example
{ public :
virtual void Create ( ) //virtual function for C++ virtual function example
{ cout <<"Base class Window"<<endl;} };
class CommandButton : public Window
{       public :
        void Create ( )
{       cout <<"Derived class Command Button–Overriden C++ virtual function"<endl;} };
        void main ( )
{       Window *x, *y;
        x = new Window ( );
        x - > Create ( );
        y = new CommandButton ( );
        y->Create ( );}.
```

Q. 6. (b) What are abstract base classes & concrete classes?

Ans. Abstract classes :

An abstract class or abstract base class (ABC), is a class that cannot be instantiated. Such a class is only meaningful if the language supports inheritnace. An abstract class is designed only as a parent class from which child classes maybe derived. Abstract classes are often used to represent abstract concepts or entities. The incomplete features of the abstract class are then shared by a group of sibling subclasses which add different variations of the missing pieces. Abstract classes are super classes which contain abstract methods and are defined such that concrete subclasses are to extend them by implementing the methods. The behaviors defined by such a class are "generic" and much of the class will be undefined and unimplemented. Before a class derived from an abstract class can become concrete, i.e., a class that can be instantiated, it must implement particular methods for all the abstract methods of its parent classes. When specifying an abstract class, the programmer is referring to a class which has elements that are meant to be implemented by inheritnace. The abstraction of the class methods to be implemented by the subclasses is meant to simplify software development. This also enables the programmer to focus on planning and design. Most object oriented programming languages allow the programmer to specify which classes are considered abstract and will not allow these to be instantiated. For example, in Java, the keyword abstract is used. In C++, an abstract class is a class having at least one abstract method.

Q. 7. (a) How data is read from a sequential access file?

Ans. char*fgets (char* str, int num, FILE* stream);

Get string from stream :

Reads characters from stream and stores them as a C string into str until (num-1) characters have been read or either a newline or a the End-of-File is reached, whichever comes first. A newline character makes fgets stop reading, but it is considered a valid character and therefore it is included in the string copied to str. A null character is automatically appended in str after the characters read to signal the end of the C string.

Parameters :

str : Pointer to an array of chars where the string read is stored.

Num : Maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as str is used.

Stream : Pointer to a FILE object that identifies the stream where characters are read from. To read from the standard input, stdin can be used for this parameter.

Return value : On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned. If an error occurs, a null pointer is returned. Use either ferror or feof to check whether an error happened or the End-of-File was reached.

Example :

```
          / * fgets example */
          # include <stdio.h>
          int main ( )
          { file*pFile;
     char string [100];
          pfile = fopen ("myfile.txt", "r");
     if (pfile == NULL) perror ("Error opening
          else {
          fgets (string, 100, pFile);
          puts (string);
          fclose (pFile); }
          return 0.}.
```

This example reads the first line of myfile.txt or the first 100 characters, whichever comes first and prints them on the screen.

Q. 7. (b) How data is written to a random access file?

Ans. size_t fwrite (const void * ptr, size_t size, size_t count FILE * stream);

Write block of data to stream : Write an array of count elements, each one with a size of size bytes, from the block of memory pointed by ptr to the current position in the stream. The position indicator of the stream is advanced by the total amount of bytes written. The total amount of bytes written is (size * count).

Parameters.

ptr : Pointer to the array of elements to be written.

Size : Size in bytes of each element to be written.

Count : Number of elements, each one with a size of size bytes.

Stream : Pointer to a FILE object that specifies an output stream.

Return value : The total number of elements successfully written is returned as a size_t object, which is an integral data type. If this number differs from the count parameter, it indicates an error.

Example :

```
*/ fwrite example
write buffer */
# include <stdio.h>
int main ( ) {
FILE * pFile;
char buffer [ ] = {'x', 'y', 'z'};
        pfile = fopen ("myfile.bin", "wb"
fwrite (buffer, 1, sizeof (buffer),
fclose (pFile);
        return 0;}
```

A file called myfile.bin is created and the content of the buffer is stored into it. For simplicity, the buffer contains char elements but it can contain any other type. sizeof (buffer) is the length of the array in bytes (in this case it is three, because the array has three elements of one byte each).

Q. 7. (c) What are stream manipulators?

Ans. C++ provides various stream manipulators that perform formatting tasks :

- Stream manipulators are defined in <iomanip>
- These manipulators provide capabilities for

  - setting field widths,
  - setting precision,
  - setting and unsetting format flags,
  - flushing streams,
  - inserting a "newline" and flushing output stream.
  - *skipping whitespace in input stream.*

setprecision ( ) .

- Select output precision. i.e., number of significant digits to be printed.

- Example :

```
cout<<setprecision (2); //two significant digits
```

setw ()

- Specify the field width (Can be used on input or output, but only applies to next insertion or extraction).

- Example :

```
cout<<setw (4);                                   //field is four positions wide
```

.eof ( );

- Tests for end-of-file condition.

- Example :

```
cin.eof ( ); //true if eof encountered
```

.fail ( );

- Tests if a stream operation has failed.

- Example :

```
cin.fail ( );                                     //true if a format error occurred
! cin;                                            //same as above; true if format error.
```

Q. 8. (a) What are functions templates?

Ans. Function templates :

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type. In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument : just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type. The format for declaring function templates with type parameters is :

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way. For example, to create a template function that returns the greater one of two objects we could use :

```
template <class my Type>
my Type GetMax (myType a, my Type b)
return (a>b? a:b);}
```