SYLLABUS FROM ACADEMIC SESSION: 2022-2023
OBJECT-ORIENTED PROGRAMMING USING C++
(CIC-211)

INSTRUCTIONS TO PAPER SETTERS:

1. There should be 9 questions in the term end examinations question paper.
2. The first (1st) question should be compulsory and cover the entire syllabus. This question should be objective, single line answers or short answer type question of total 15 marks.
3. Apart from question 1 which is compulsory, rest of the paper shall consist of 4 units as per the syllabus. Every unit shall have two questions covering the corresponding unit of the syllabus. However, the student shall be asked to attempt only one of the two questions in the unit. Individual questions may contain upto 5 sub-parts / sub-questions. Each Unit shall have a marks weightage of 15.
4. The questions are to be framed keeping in view the learning outcomes of the course / paper. The standard / level of the questions to be asked should be at the level of the prescribed textbook.
5. The requirement of (scientific) calculators / log-tables / data – tables may be specified if required.

UNIT I

Object Oriented Programming Paradigm, Basic Concepts of Object Oriented Programming, Benefits of Object Oriented Programming, Object Oriented Languages, Applications of Object Oriented Programming, C++ Programming Language, Tokens, Keywords, Identifiers and Constants, Data Types, Type Compatibility, Variables, Operators in C++, Implicit Type Conversions, Operator Precedence, The Main Function, Function Prototyping, Call by Reference, Return by Reference, Inline Functions, Function Overloading, Friend Functions, default parameter value.

UNIT II

Specifying a class, Member Functions, Encapsulation, information hiding, abstract data types, objects & classes, Static Member Functions, Arrays of Objects, Constructors & Destructors, Parameterized Constructors, Copy Constructors, Dynamic Constructors, Destructors, identity and behaviour of an object, C++ garbage collection, dynamic memory allocation, Explicit Type Conversions, Operator Overloading.

UNIT III

Inheritance, inheritance methods, Class hierarchy, derivation – public, private & protected, aggregation, Inheritance Constructors, composition vs. classification hierarchies, Containership, Initialization List, Polymorphism, categorization of polymorphic techniques, polymorphism by parameter, parametric polymorphism, generic function – template function, function overriding, run time polymorphism, virtual functions.

UNIT – IV

Standard C++ classes, using multiple inheritance, persistant objects, streams and files, namespaces, exception handling, generic classes, standard template library: Library organization and containers, standard containers, algorithm and Function objects, iterators and allocators, strings, streams, manipulators, user defined manipulators, vectors.

SYLLABUS

ACADEMIC SESSION : 2015-2016

OBJECT ORIENTED PROGRAMMING (ETCS-210)

Instruction to Paper Setters: **Maximum Marks : 75**

1. *Question No. 1 should be compulsory and cover the entire syllabus. This question should have objective or short answer type questions. It should be of 25 marks.*
2. *Apart from Question No. 1, rest of the paper shall consist of four units as per the syllabus. Every unit should have two questions. However, student may be asked to attempt only 1 question from each unit. Each question should be 12.5 marks*

Objective: *To learn object oriented concepts to enhance programming skills.*

UNIT – I

Objects, relating to other paradigms (functional, data decomposition), basic terms and ideas (abstraction, encapsulation, inheritance, polymorphism). Review of C, difference between C and C++, cin, cout, new, delete operators.

[T1,T2][No. of hrs. 11]

UNIT – II

Encapsulation, information hiding, abstract data types, object & classes, attributes, methods. C++ class declaration, state identity and behavior of an object, constructors and destructors, instantiation of objects, default parameter value, object types, C++ garbage collection, dynamic memory allocation, metaclass/abstract classes.

[T1,T2][No. of hrs. 11]

UNIT – III

Inheritance, Class hierarchy, derivation – public, private & protected; aggregation, composition vs classification hierarchies, polymorphism, categorization of polymorphic techniques, method polymorphism, polymorphism by parameter, operator overloading, parametric polymorphism, generic function – template function, function name overloading, overriding inheritance methods, run time polymorphism.

[T1,T2][No. of hrs. 11]

UNIT – IV

Standard C++ classes, using multiple inheritance, persistant objects, streams and files, namespaces, exception handling, generic classes, standard template library: Library organization and containers, standard containers, algorithm and Function objects, iterators and allocators, strings, streams, manipulators, user defined manipulators, vectors, valarray, slice, generalized numeric algorithm.

[T1,T2][No. of hrs. 11]

NEW TOPICS ADDED FROM ACADEMIC SESSION [2022-23]
THIRD SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING USING C++ (CIC-211)

Q.1. What are the applications of Object Oriented Programming?

Ans. OOPs stands for Object-Oriented Programming. It is about creating objects that contain both data and functions. Object-Oriented programming has several advantages over procedural languages. As OOP is faster and easier to execute it becomes more powerful than procedural languages like C++. OOPs is the most important and flexible paradigm of modern programming. It is specifically useful in modeling real-world problems. Below are some applications of OOPs:

- **Real-Time System design:** Real-time system inherits complexities and makes it difficult to build them. OOP techniques make it easier to handle those complexities.
- **Hypertext and Hypermedia:** Hypertext is similar to regular text as it can be stored, searched, and edited easily. Hypermedia on the other hand is a superset of hypertext. OOP also helps in laying the framework for hypertext and hypermedia.
- **AI Expert System:** These are computer application that is developed to solve complex problems which are far beyond the human brain. OOP helps to develop such an AI expert System
- **Office automation System:** These include formal as well as informal electronic systems that primarily concerned with information sharing and communication to and from people inside and outside the organization. OOP also help in making office automation principle.
- **Neural networking and parallel programming:** It addresses the problem of prediction and approximation of complex-time varying systems. OOP simplifies the entire process by simplifying the approximation and prediction ability of the network.
- **Stimulation and modeling system:** It is difficult to model complex systems due to varying specifications of variables. Stimulating complex systems require modeling and understanding interaction explicitly. OOP provides an appropriate approach for simplifying these complex models.
- **Object-oriented database:** The databases try to maintain a direct correspondence between the real world and database object in order to let the object retain it identity and integrity.
- **Client-server system:** Object-oriented client-server system provides the IT infrastructure creating object-oriented server internet(OCSI) applications.
- **CIM/CAD/CAM systems:** OOP can also be used in manufacturing and designing applications as it allows people to reduce the efforts involved. For instance, it can be used while designing blueprints and flowcharts. So it makes it possible to produce these flowcharts and blueprint accurately.

Q.2. What are the benefits of Object Oriented Programming?

Ans. Benefits of OOP

- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
- OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).
- The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
- OOP systems can be easily upgraded from small to large systems.
- It is possible that multiple instances of objects co-exist without any interference,

- It is very easy to partition the work in a project based on objects.
- It is possible to map the objects in problem domain to those in the program.
- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes.
- Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
- The data-centered design approach enables us to capture more details of model in an implementable form.

Q.3. Explain the the different types of tokens in c++.

Ans. C++ Tokens are the smallest individual units of a program.

C++ is the superset of C and so most constructs of C are legal in C++ with their meaning and usage unchanged. So tokens, expressions, and data types are similar to that of C.

Following are the C++ tokens : (most of c++ tokens are basically similar to the C tokens)

(a) Keywords: Keywords are reserved words which have fixed meaning, and its meaning cannot be changed. The meaning and working of these keywords are already known to the compiler. C++ has more numbers of keyword than C, and those extra ones have special working capabilities.

There are 32 of these, and here they are:

auto const double float int short struct unsigned break continue else for long signed switch void case default enum goto register sizeof typedef volatile char do extern if return static union while

There are another 30 reserved words that were not in C, are therefore new to C++, and here they are:

asm dynamic_cast namespace reinterpret_cast try bool explicit new static_cast typeid catch false operator template typename class friend private this using const_cast inline public throw virtual

delete mutable protected true wchar_t

(b) Identifiers: Identifiers are names given to different entries such as variables, structures, and functions. Also, identifier names should have to be unique because these entities are used in the execution of the program.

Identifier naming conventions

- Only alphabetic characters, digits and underscores are permitted.
- First letter must be an alphabet or underscore (_).
- Identifiers are case sensitive.
- Reserved keywords can not be used as an identifier's name.

(c) Constants: Constants are like a variable, except that their value never changes during execution once defined.

There are two other different ways to define constants in C++. These are:

- By using const keyword
- By using #define preprocessor

Declaration of a constant :

const [data_type] [constant_name]=[value];

(d) Variable: A variable is a meaningful name of data storage location in computer memory. When using a variable you refer to memory address of computer.

Syntax to declare a variable

[data_type] [variable_name];

Example

```
#include <iostream.h>
int main() {
  int a,b;// a and b are integer variable
  cout<<" Enter first number :";
  cin>>a;
  cout<<" Enter the second number:";
  cin>>b;
  int sum;
  sum=a+b;
  cout<<" Sum is : "<<sum <<"\n";
  return 0;
}
```

(e) Operators: C++ operator is a symbol that is used to perform mathematical or logical manipulations.

Arithmetic Operator

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus

Increment and Decrement Operators

Operator	Description
++	Increment
– –	Decrement

Relational Operators

Operator	Description
==	Is equal to
!=	Is not equal to
>	Greater than
<	Less
>=	Greater than or equal to
<=	Less than or equal to

Logical Operators

Operator	Description
&&	And operator. Performs logical conjunction of two expressions. (if both expressions evaluate to True, result is True. If either expression evaluates to False, the result is False)
||	Or operator. Performs a logical disjunction on two expressions. (if either or both expressions evaluate to True, the result is True)
!	Not operator. Performs logical negation on an expression.

Bitwise Operators

Operator	Description
<<	Binary Left Shift Operator
!=	Is not equal to
>>	Binary Right Shift Operator
~	Binary One's Complement Operator
&	Binary AND Operator
^	Binary XOR Operator
|	Binary OR Operator

Assignment Operator

Operator	Description
=	Assign
+=	Increments, then assign
-=	Decrements, then assign
*=	Multiplies, then assign
/=	Divides, then assign
%=	Modulus, then assigns
<<=	Left shift and assigns
>>=	Right shift and assigns
&=	Bitwise AND assigns
^=	Bitwise exclusive OR and assigns
|=	Bitwise inclusive OR and assigns

Misc Operator

Operator	Description
,	Comma operator
sizeOf()	Returns the size of a memory location.
&	Returns the address of a memory location.

Operator	Description
*	Pointer to a variable.
? :	Conditional Expression

Q.4. What is the precedence of operators in c++?

Ans.

Precedence	Operator	Description	Associativity
1	::	Scope Resolution	Left to Right

2	a++ a-- type( ) type{ } a( ) a[ ] ->	Suffix/postfix increment Suffix/postfix decrement Function cast Function cast Function call Subscript Member access from an object Member access from object ptr	Left to Right
3	++a --a +a -a ! ~ (type) *a &a sizeof co_await new new[ ] delete delete[]	Prefix increment Prefix decrement Unary plus Unary minus Logical NOT Bitwise NOT C style cast Indirection (dereference) Address-of Size-of await-expression Dynamic memory allocation Dynamic memory deallocation	Right to Left
4	.* ->*	Member object selector Member pointer selector	Left to Right
5	a * b a / b a % b	Multiplication Division Modulus	Left to Right
6	a + b a - b	Addition Subtraction	Left to Right
7	<< >>	Bitwise left shift Bitwise right shift	Left to Right
8	<=<	Three-way comparison operator	Left to Right
9	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right
10	== !=	Equal to Not equal to	Left to Right
11	&	Bitwise AND	Left to Right
12	^	Bitwise XOR	Left to Right

13	|	Bitwise OR	Left to Right
14	&&	Logical AND	Left to Right
15	||	Logical OR	Left to Right
16	a ? b : c throw co_yield = += -= *= /= %= <<= >>= &= ^= |=	Ternary Conditional throw operator yield expression (C++ 20) Assignment Addition Assignment Subtraction Assignment Multiplication Assignment Division Assignment Modulus Assignment Bitwise Shift Left Assignment Bitwise Shift Right Assignment Bitwise AND Assignment Bitwise XOR Assignment Bitwise OR Assignment	Right to Left
17	,	Comma operator	Left to Right

Q.5. What is the difference between Call by value & Call by Reference?
Ans.

Call By Value	**Call By Reference**
While calling a function, we pass values of variables to it. Such functions are known as "Call By Values".	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function is known as "Call By Reference"
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.

// C program to illustrate
// call by value
 	
#include <stdio.h>
 	
// Function Prototype
void swapx(int x, int y);
 	
// Main function
int main()
{
int a = 10, b = 20;
 	
// Pass by Values
swapx(a, b);
 	
printf("a=%d b=%d\n", a, b);
 	
return 0;
}
 	
// Swap functions that swaps
// two values
void swapx(int x, int y)
{
int t;
t = x;
x = y;
y = t;
printf("x=%d y=%d\n", x, y);
}
Output:
x=20 y=10
a=10 b=20
	// C program to illustrate
	// Call by Reference
	
	#include <stdio.h>
	
	// Function Prototype
	void swapx(int*, int*);
	
	// Main function
	int main()
	{
	int a = 10, b = 20;
	
	// Pass reference
	swapx(&a, &b);
	
	printf("a=%d b=%d\n", a, b);
	
	return 0;
	}
	
	// Function to swap two variables
	// by references
	void swapx(int* x, int* y)
	{
	int t;
	t = *x;
	*x = *y;
	*y = t;
	printf("x=%d y=%d\n", *x, *y);
	}
	Output:
	x=20 y=10
	a=20 b=10

Thus actual values of a and b remain unchanged even after exchanging the values of x and y.	Thus actual values of a and b get changed after exchanging values of x and y.
In call-by-values, we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
Values of variables are passed by the Simple technique.	Pointer variables are necessary to define to store the address values of variables.

Q.6. What do you mean by return by reference? Explain with the help of

example.

Ans. Return by Reference

- In C++, Pointers and References held close relation with in another.
- The major difference is that the pointers can be operated on like adding values whereas references are just an alias for another variable.
- Functions in C++ can return a reference as it is return a pointer.
- When function returns a reference it means it returns a implicit pointer.
- In C++ Programming, not only can you pass values by reference to a function but you can also return a value by reference.
- Return by reference is very different from Call by reference.
- Functions behaves a very important role when variable or pointers are returned as reference.

Syntax

dataType & functionName(parameters);

where,

dataType is the return type of the function,

and parameters are the passed arguments to it.

Return by Reference by using Global Variable

```
#include <iostream>
using namespace std;
// Global variable
int number;
// Function declaration
int& retByRef(){
   return number;
}
int main(){
   // Function call for return by reference
   retByRef() = 2;
      // print number
   cout << number;
   return 0;
}
```

Output

2

Q.7. What do you mean by the Inline function?

Ans. C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

(1) If a function contains a loop. (for, while, do-while)

(2) If a function contains static variables.

(3) If a function is recursive.

(4) If a function return type is other than void, and the return statement doesn't exist in function body.

(5) If a function contains switch or goto statement.

Q.8. What do you mean by the Friend Function & Friend Class?

Ans. C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
friend data_type function_name(argument/s);          // syntax of friend function.
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword friend or scope resolution operator.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Example when the function is friendly to two classes.

```
#include <iostream>
using namespace std;
class B;        // forward declararion.
class A
{
  int x;
  public:
  void setdata(int i)
  {
    x=i;
  }
  friend void min(A,B);        // friend function.
};
class B
{
  int y;
  public:
  void setdata(int i)
  {
    y=i;
  }
  friend void min(A,B);                // friend function
```

```
};
void min(A a,B b)
{
   if(a.x<=b.y)
   std::cout << a.x << std::endl;
   else
   std::cout << b.y << std::endl;
}
  int main()
{
  A a;
  B b;
  a.setdata(10);
  b.setdata(20);
  min(a,b);
   return 0;
}
```

Output:

10

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

Example of a friend class.

```
#include <iostream>
 using namespace std;
 class A
{
   int x =5;
   friend class B;          // friend class.
};
class B
{
 public:
   void display(A &a)
   {
      cout<<"value of x is : "<<a.x;
   }
};
int main()
{
   A a;
   B b;
   b.display(a);
   return 0;
}
```

Output:

value of x is : 5

In the above example, class B is declared as a friend inside the class A. Therefore B is a friend of class A. Class B can access the private members of class A.

UNIT - I

Q.1 Refer to Q.1 (a),(b),(c),(d) of First Term Examination 2015 (Pg. No. 1-2015).
Q.2 Refer to Q.2 (a),(b) of First Term Examination 2015 (Pg. No. 2-2015).
Q.3 Refer to Q.4 (a) of First Term Examination 2015 (Pg. No. 5-2015).
Q.4 Refer to Q.1 (c) of Second Term Examination 2015 (Pg. No. 8-2015).
Q.5 Refer to Q.1 (a),(c),(e),(g),(i) of End Term Examination 2015 (Pg. No.14,15,16-2015).
Q.6 Refer to Q.2 of End Term Examination 2015 (Pg. No. 17-2015).
Q.7 Refer to Q.8 (a) of End Term Examination 2015 (Pg. No. 25-2015).
Q.8 Refer to Q.1 (a),(e),(f) of First Term Examination 2016 (Pg. No. 1,2-2016).
Q.9 Refer to Q.4 (a),(b),(c) of First Term Examination 2016 (Pg. No. 5,6,7-2016).
Q.10 Refer to Q.1 (a) of Second Term Examination 2016 (Pg. No. 8-2016).
Q.11 Refer to Q.1 (a),(d),(e),(g),(h),(j) of End Term Examination 2015 (Pg. No. 16,17,18,19-2015).
Q.12 Refer to Q.2 (b) of End Term Examination 2016 (Pg. No. 20-2016).
Q.13 Refer to Q.3 (a),(b) of End Term Examination 2016 (Pg. No. 20-2016).
Q.14 Refer to Q.4 (a) of End Term Examination 2016 (Pg. No. 21-2016)..
Q.15 Refer to Q.1 (a),(b) of First Term Examination 2017 (Pg. No. 1-2017).
Q.16 Refer to Q.3 (a) of First Term Examination 2017 (Pg. No. 4-2017).
Q.17 Refer to Q.4 (c),(d),(e) of First Term Examination 2017 (Pg. No. 6-2017).
Q.18 Refer to Q.1 (a) of End Term Examination 2017 (Pg. No. 8-2017).
Q.19 Refer to Q.2 (a) of End Term Examination 2017 (Pg. No. 12-2017).
Q.20 Refer to Q.3 (a) of End Term Examination 2017 (Pg. No. 13-2017).
Q.21 Refer to Q.6 (b) of End Term Examination 2017 (Pg. No. 19-2017).
Q.22 Refer to Q.1 (b) of First Term Examination 2018 (Pg. No. 1-2018).
Q.23 Refer to Q.2 (a) of First Term Examination 2018 (Pg. No. 3-2018).
Q.24 Refer to Q.3 (a),(b) of First Term Examination 2018 (Pg. No. 4-2018).
Q.25 Refer to Q.1 (c),(d) of End Term Examination 2018 (Pg. No. 7-2018).
Q.26 Refer to Q.2 (a),(b) of End Term Examination 2018 (Pg. No. 10-2018).
Q.27 Refer to Q.3 (b) of End Term Examination 2018 (Pg. No. 13-2018).
Q.28 Refer to Q.1 (a),(d) of First Term Examination 2019 (Pg. No. 1-2019).
Q.29 Refer to Q.4 (a) of First Term Examination 2019 (Pg. No. 3-2019).
Q.30 Refer to Q.1 (a),(b),(c),(d) of End Term Examination 2019 (Pg. No. 5-2019).

UNIT - II

Q.1 Refer to Q.3 (a),(b) of First Term Examination 2015 (Pg. No. 3-2015).
Q.2 Refer to Q.4 (b) of First Term Examination 2015 (Pg. No. 5-2015).
Q.3 Refer to Q.2 (a),(b) of Second Term Examination 2015 (Pg. No. 8-2015).
Q.4 Refer to Q.1 (f),(h) of End Term Examination 2015 (Pg. No. 15,16-2015).
Q.5 Refer to Q.5 of End Term Examination 2015 (Pg. No. 22-2015).
Q.6 Refer to Q.6 of End Term Examination 2015 (Pg. No. 24-2015).
Q.7 Refer to Q.1 (b),(c),(d) of First Term Examination 2016 (Pg. No. 1,2-2016).
Q.8 Refer to Q.2 (b) of First Term Examination 2016 (Pg. No. 3-2016).
Q.9 Refer to Q.3 (a),(b) of First Term Examination 2016 (Pg. No. 3-2016).
Q.10 Refer to Q.3 (i) of Second Term Examination 2016 (Pg. No. 12-2016).
Q.11 Refer to Q.1 (b),(i) of End Term Examination 2016 (Pg. No. 16,18-2016).
Q.12 Refer to Q.2 (a) of End Term Examination 2016 (Pg. No. 19-2016).
Q.13 Refer to Q.4 (b) of End Term Examination 2016 (Pg. No. 22-2016).
Q.14 Refer to Q.5 (a),(b) of End Term Examination 2016 (Pg. No. 23-2016).
Q.15 Refer to Q.1 (c),(d),(e) of First Term Examination 2017 (Pg. No. 1-2017).
Q.16 Refer to Q.2 (a),(b) of First Term Examination 2017 (Pg. No. 2-2017).
Q.17 Refer to Q.3 (b) of First Term Examination 2017 (Pg. No. 5-2017).
Q.18 Refer to Q.4 (a),(b) of First Term Examination 2017 (Pg. No. 6-2017).
Q.19 Refer to Q.1 (c),(d) of End Term Examination 2017 (Pg. No. 8-2017).
Q.20 Refer to Q.1 (g),(h),(i) of End Term Examination 2017 (Pg. No. 10,11-2017).
Q.21 Refer to Q.2 (b) of End Term Examination 2017 (Pg. No. 13-2017).
Q.22 Refer to Q.3 (b) of End Term Examination 2017 (Pg. No. 14-2017).
Q.23 Refer to Q.4 (a),(b) of End Term Examination 2017 (Pg. No. 15-2017).
Q.24 Refer to Q.7 (a) of End Term Examination 2017 (Pg. No. 21-2017).
Q.25 Refer to Q.1 (a),(d) of First Term Examination 2018 (Pg. No. 1,2-2018).
Q.26 Refer to Q.2 (b) of First Term Examination 2018 (Pg. No. 3-2018).
Q.27 Refer to Q.4 (a) of First Term Examination 2018 (Pg. No. 5-2018).
Q.28 Refer to Q.1 (f),(i) of End Term Examination 2018 (Pg. No. 8,9-2018).
Q.29 Refer to Q.4 (a),(b) of End Term Examination 2018 (Pg. No. 14-2018).
Q.30 Refer to Q.5 (a),(b) of End Term Examination 2018 (Pg. No. 17-2018).
Q.31 Refer to Q.1 (b) of First Term Examination 2019 (Pg. No. 1-2019).
Q.32 Refer to Q.2 (a) of First Term Examination 2019 (Pg. No. 2-2019).
Q.33 Refer to Q.3 (a) of First Term Examination 2019 (Pg. No. 3-2019).
Q.34 Refer to Q.4 (b) of First Term Examination 2019 (Pg. No. 4-2019).
Q.35 Refer to Q.1 (e) of End Term Examination 2019 (Pg. No. 6-2019).
Q.36 Refer to Q.2 (b) of End Term Examination 2019 (Pg. No. 11-2019).
Q.37 Refer to Q.3 (a),(b) of End Term Examination 2019 (Pg. No. 12-2019).
Q.38 Refer to Q.4 (b) of End Term Examination 2019 (Pg. No. 17-2019).
Q.39 Refer to Q.8 (b) of End Term Examination 2019 (Pg. No. 24-2019).

UNIT - III

Q.1 Refer to Q.1 (a),(b) of Second Term Examination 2015 (Pg. No. 7-2015).
Q.2 Refer to Q.3 (a),(b) of Second Term Examination 2015 (Pg. No. 9-2015).
Q.3 Refer to Q.4 (a),(b) of Second Term Examination 2015 (Pg. No. 12-2015).
Q.4 Refer to Q.1 (b),(d),(j) of End Term Examination 2015 (Pg. No. 14,15,16-2015).
Q.5 Refer to Q.3 of End Term Examination 2015 (Pg. No. 18-2015).
Q.6 Refer to Q.4 of End Term Examination 2015 (Pg. No. 20-2015).
Q.7 Refer to Q.9 (a) of End Term Examination 2015 (Pg. No. 27-2015).
Q.8 Refer to Q.1 (b),(c) of Second Term Examination 2016 (Pg. No. 8-2016).
Q.9 Refer to Q.2 (i),(ii) of Second Term Examination 2016 (Pg. No. 10-2016).
Q.10 Refer to Q.3 (ii) of Second Term Examination 2016 (Pg. No. 13-2016).
Q.11 Refer to Q.4 (ii) of Second Term Examination 2016 (Pg. No. 15-2016).
Q.12 Refer to Q.1 (c),(f) of End Term Examination 2016 (Pg. No. 16,17-2016).
Q.13 Refer to Q.6 of End Term Examination 2016 (Pg. No. 25-2016).
Q.14 Refer to Q.7 (a),(b) of End Term Examination 2016 (Pg. No. 28-2016).
Q.15 Refer to Q.1 (e),(f),(j) of End Term Examination 2017 (Pg. No. 9,11-2017).
Q.16 Refer to Q.5 (a),(b) of End Term Examination 2017 (Pg. No. 17-2017).
Q.17 Refer to Q.6 (a) of End Term Examination 2017 (Pg. No. 19-2017).
Q.18 Refer to Q.7 (b) of End Term Examination 2017 (Pg. No. 21-2017).
Q.19 Refer to Q.1 (c),(e) of First Term Examination 2018 (Pg. No. 1,2-2018).
Q.20 Refer to Q.4 (b) of First Term Examination 2018 (Pg. No. 6-2018).
Q.21 Refer to Q.1 (e),(g),(h),(k),(l) of End Term Examination 2018 (Pg. No. 8-2018).
Q.22 Refer to Q.3 (a) of End Term Examination 2018 (Pg. No. 12-2018).
Q.23 Refer to Q.7 (a) of End Term Examination 2018 (Pg. No. 20-2018).
Q.24 Refer to Q.8 (a) of End Term Examination 2018 (Pg. No. 23-2018).
Q.25 Refer to Q.1 (c),(e) of First Term Examination 2019 (Pg. No. 1-2019).
Q.26 Refer to Q.1 (f),(h),(i),(j) of End Term Examination 2019 (Pg. No. 7,8-2019).
Q.27 Refer to Q.2 (a) of End Term Examination 2019 (Pg. No. 9-2019).
Q.28 Refer to Q.4 (a) of End Term Examination 2019 (Pg. No. 15-2019).
Q.29 Refer to Q.5 (b) of End Term Examination 2019 (Pg. No. 18-2019).
Q.30 Refer to Q.6 (a),(b) of End Term Examination 2019 (Pg. No. 18-2019).
Q.31 Refer to Q.7 (b) of End Term Examination 2019 (Pg. No. 21-2019).

UNIT - IV

Q.1 Refer to Q.1 (d) of Second Term Exam 2015 (Pg. No. 8-2015)

Q.2 Refer to Q.7 of End Term Exam 2015 (Pg. No. 24-2015)

Q.3 Refer to Q.8 (b) of End Term Exam 2015 (Pg. No. 26-2015)

Q.4 Refer to Q.9 (b) of End Term Exam 2015 (Pg. No. 28-2015)

Q.5 Refer to Q.1 (d) of Second Term Exam 2016 (Pg. No. 9-2016)

Q.6 Refer to Q.4 (i) of Second Term Exam 2016 (Pg. No. 14-2016)

Q.7 Refer to Q.8 (a),(b) of End Term Exam 2016 (Pg. No. 30-2016)

Q.8 Refer to Q.9 of End Term Exam 2016 (Pg. No. 31-2016)

Q.9 Refer to Q.8 (a),(b) of End Term Exam 2017 (Pg. No. 22-2017)

Q.10 Refer to Q.1 (j) of End Term Exam 2018 (Pg. No. 9-2018)

Q.11 Refer to Q.7 (b) of End Term Exam 2018 (Pg. No. 22-2018)

Q.12 Refer to Q.8 (b) of End Term Exam 2018 (Pg. No. 24-2018)

Q.13 Refer to Q.7 (a) of End Term Exam 2019 (Pg. No. 20-2019)

Q.14 Refer to Q.8 (a) of End Term Exam 2019 (Pg. No. 22-2019)

SECOND TERM EXAMINATION [APRIL-2015]
FOURTH SEMESTER [B. TECH]
OBJECT ORIENTED PROGRAMMING [ETCS-210]

MM : 30

Time: 1 Hr

Note: *Question No.1 is compulsory. Attempt any two more questions from the rest.*

Q.1. (a) Explain templates? (2.5)

Ans. Template is one of the feature added to C+ recently. A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array etc. Similarly we can define a template for a function say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.

Syntax for class template :

```
Template <class T>
Class  classname
{
};
```

Syntax for function template :

```
Template <class T>
Returntype functionname (arguments of type T)
{
}
```

Q.1. (b) Difference between Function Overloading and Function Overriding. (2.5)

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
Functions name and signatures must be same.	Having same Function name with different **Signatures**.
Overriding is the concept of **runtime polymorphism**	Overloading is the concept of **compile time polymorphism.**
When a function of base class is re-defined in the derived class called as Overriding	Two functions having same name and return type, but with different type and/or number of arguments is called as Overloading.
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types
Function should be public.	Function can be different access specifies

Syntax of new is:

p_var = new type name: Where p_var is a previously declared pointer of type typename. typename can be any basic data type.

New can also create an array:

p_var = new type [size]: In this case, size specifies the length of one-dimensional array to create.

Example 1:

```
int *p; p=new int;
```

It allocates memory space for an integer variable. And allocated memory can be released using following statement,

```
delete p;
```

Example 2:

```
int *a; a = new int[100];
```

It creates a memory space for an array of 100 integers. And to release the memory following statement can be used,

```
delete []a;
```

Q.2. (a) What are Friend Function? Explain with example. **[5 × 2 = 10]**

Ans. Friend Functions:

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

```
#include <iostream>
using namespace std;
class Rectangle {
                int width, height;
public:
Rectangle (int w = 1, int h = 1): width(w),height(h){}
    friend void display(Rectangle &);
};
void display(Rectangle &r) {
    cout << r.width * r.height << endl;
}
int main () {
            Rectangle rect(5,10);
            display(rect);
 return 0;
}
```

We make a function a friend to a class by declaring a prototype of this external function within the class, and preceding it with the keyword **friend.**

```
friend void display (Rectangle &);
```

The friend function **display(rect)** has an access to the private member of the Rectangle class object though it's not a member function. It gets the width and height using dot: **r.width** and **r.height**. If we do this inside main, we get an error because they

are private members and we can't access them outside of the class. But friend function to the class can access the private members.

Q.2. (b) What are Inline Functions? Explain with example.

Ans. C++ **Inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Example : Use of inline function to return max of two numbers:

```
#include <iostream>
inline int Max(int x, int y)
{
  return (x > y)? x : y;
}
int main( )
{
  cout << "Max (20,10): " << Max(20,10) << endl;
  cout << "Max (0,200): " << Max(0,200) << endl;
  cout << "Max (100,1010): " << Max(100,1010) << endl;
  return 0;
}
```

Q.3. (a) What are copy constructors? Explain with example. **[5 × 2 = 10]**

Ans. The copy constructor is a special kind of constructor which creates a new object which is a copy of an existing one, and does it efficiently.The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it. The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it from scratch. A copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

Example of copy constructor.

```
#include<iostream>
class Point
{
private:
  int x, y;
public:
  Point(int x1, int y1) {x = x1; y = y1;}
// Copy constructor
Point(const Point and p2) {x = p2.x; y = p2.y;}
int getX()      { return x; }
int getY()      {return y; }
};
```

```
int main()
{
Point p1(10, 15); // Normal constructor is called here
Point p2 = p1; // Copy constructor is called here
cout << "p1.x = " << p1. getX() << ", p1.y = " << p1.getY();
cout << "\np2.x = " << p2. getX() << ", p2.y = " << p2.getY();
return 0;
}
```

Q. 3. (b) What are static data member? Explain with example.

Ans. A data member of a class can be qualified as static.The properties of a static member variable are similar to that of a *C* static variable. A static member variable has certain special characteristics.

1. It is initialized to zero when the first object of its class is created.

2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

3. It is visible by within the class, but its lifetime is the entire program.

Example:

```
#include<iostream.h>
using namespace std;
class item
{
static int count;
int num;
public:
void getdata(int a)
{
number=a; count++;
}
void getcount()
{
Cout<<count<<"\n";
}
};
int item : : count;
int main()
{
item a,b,c;
a.getcount();
b.getcount();
c.getcount();
a.getdata(100);
b.getdata(200);
c.getdata(300);
```

```
a.getcount();
b.getcount();
c.getcount();
return 0;
}
```

Output is : 0,0,0 and 3,3,3

Q. 4. (a) Difference between C and C++. **[5 × 2 = 10]**

Ans.

C	C++
1. C is Procedural Language.	1. C++ is non Procedural i.e Object oriented Language.
2. No virtual Functions are present in C	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	3. The concept of polymorphism is used in C++.Polymorphism is the most Important Feature of OOPS.
4. Operator overloading is not possible . in C	4. Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.
7. Multiple Declaration of global variables are allowed.	7. Multiple Declaration of global variables are not allowed.
8. In C • scan f () Function used for Input. • printf () Function used for output.	8. In C++. • Cin>> Function used for Input. • Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	9. Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other functions	10. In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	11. C++ allows the declaration of variable anywhere in the scope *i.e.* at time of its First use.
12. No inheritance is possible in C.	12. Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for Memory Deallocating.	13. In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	14. It support both built-in and user define data types.
15. In C, Exception handling is not present.	15. In C++, exception handling is done with Try and Catch block.

Q.4. (b) Write a program using class and objects to add two complex numbers to show the concepts of returning objects.

Ans.

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

```
class complex
{
float x;
float y;
public:
void input(float real, float imag)
{
X=real; y=imag; }
friend complex sum(complex, complex);
void show (complex);
};
complex sum(complex c1, complex c2)
{
complex c3;
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return (c3);
}
void complex : : show(complex c)
{
cout<<c.x<<"+ j"<<c.y<<"\n";
}
int main()
{
complex a,b,c;
a.input(3.1, 5.65);
b.input(2.75 ^ 2);
c=sum(a, b),
cout<<a.show(a);
cout<<b.show(b);
cout<<c.show(c);
return 0;
}
```

SECOND TERM EXAMINATION [APRIL-2015]
FOURTH SEMESTER [B. TECH]
OBJECT ORIENTED PROGRAMMING [ETCS-210]

Time: 1 Hr MM : 30

Note: Question No.1 is compulsory. Attempt any two more questions from the rest.

Q.1. (a) Explain templates? (2.5)

Ans. Template is one of the feature added to C++ recently. A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array etc. Similarly we can define a template for a function say mul( ), that would help us create various versions of mul() for multiplying int, float and double type values.

Syntax for class template :

```
Template <class T>
Class  classname
{
};
```

Syntax for function template :

```
Template <class T>
Returntype functionname (arguments of type T)
{
}
```

Q.1. (b) Difference between Function Overloading and Function Overriding. (2.5)

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
Functions name and signatures must be same.	Having same Function name with different Signatures.
Overriding is the concept of **runtime polymorphism**	Overloading is the concept of **compile time polymorphism.**
When a function of base class is re-defined in the derived class called as Overriding	Two functions having same name and return type, but with different type and/or number of arguments is called as Overloading.
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types
Function should be public.	Function can be different access specifies

Q1. (c) Difference between Early and Later binding? (2.5)

Ans. Early Binding: Events occurring at compile time are known as early binding. In the process of early binding all information which is required for a function call is known at compile time. Early binding is a fast and efficient process. Examples of early binding: function calls, overloaded function calls, and overloaded operators.

Late Binding: In this process all information which is required for a function call is not known at compile time. Hence, objects and functions are not linked at run time. Late Binding is a slow process. However, it provides flexibility to the code. Example of Late Binding: Virtual functions.

Q1. (*d*) Explain Exception handling? (2.5)

Ans. An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **Throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **Catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **Try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Q.2.(a) Explain the concept of Class to Basic Type Conversion with example? (5)

Ans. Class to Basic Type Conversions: The constructor handles the task of converting basic types to class types very well. But you cannot use constructors for converting class types to basic data types. Instead, you can define an overloaded *casting operator* that can be used to convert a class data type into a basic data type. The general form of an *overloaded casting operator* function is shown below. This function is also known as a *conversion function.*

```
Operator typename()
{
----------
Function body
----------
}
```

The above function converts the class type into the *typename* mentioned in the function. For example, **operator float( )** converts a class type to **float**, **operator int()** converts a class type to int, and so on.

Consider the following conversion function:

```
Vector :: operator double()
{
double sum=0;
for(int i=0;i<size;i++)
sum=sum + v[i] * u[i];
return sqrt(sum);
}
```

The above function converts a vector object into its corresponding scalar magnitude, in other words, it converts a vector type into a double.

Q.2. (b) Explain operator overloading? Write a program to overload binanry '+' operator. (5)

Ans. C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

You can redefine or overload most of the built-in operators available in C++. Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Program : Overload the binary + operator.

```
#include<iostream>
using namespace std;
class complex{
  float x1,y1;
public:
        void disp(){cout<<x1<<"+"<<y1<<"i"<<"\n";//to represent in complex form }
        complex(float a,float b)//constructor
        { x1=a; y1=b; }
        complex operator+(complex cc)
{ return complex(x1+cc.x1,y1+cc.y1);//operator overloading defined here }
};
int main(){
        complex c1(2.5,-0.3);//invokes constructor
        complex c2(4.5,7);
        complex c3=c1+c2;//invokes operator +
        c3.disp();// displays final answer
        return 0;
}
```

Q.3. (a) Explain Inheritance and type of Inheritance? (5)

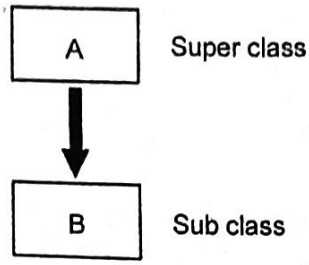
Ans. One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

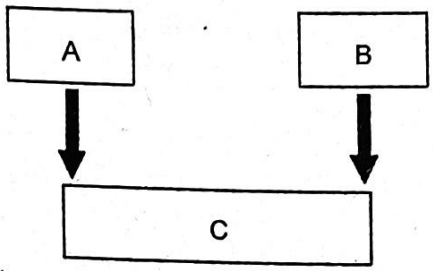
In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

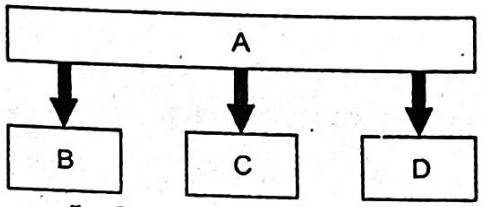
Single Inheritance: In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



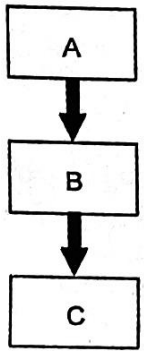
Multiple Inheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.



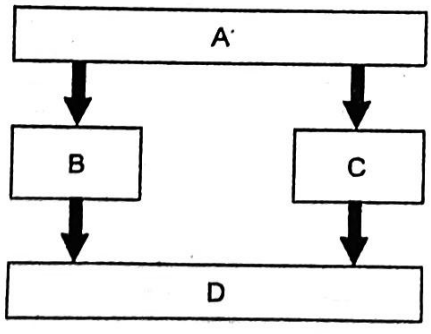
Hierarchical Inheritance: In this type of inheritance, multiple derived classes inherits from a single base class.



Multilevel Inheritance: In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid Inheritance: Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



Q.3. (b) Design 3 classes Student, Exam and Result. Such that Exam is inherited from Student class, Result is inherited from Exam class. Write a program to model this relationship. What type of inheritance does this model belong to. (5)

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student//base class
{
private:
 int 1;
 char nm[20];
public:
  void read();
  void display();
};
class exam: public student//dervivec from student
{
protected:
 int s1;
 int s2;
 int s3;
public:
  void getmarks();
  void putmarks();
};
class result : public exam //derived  from marks
{
private:
  int t;
  float p;
  public:
  void process();
  void printresult();
};
void student::read()
 {
cout<<"enter Roll no and Name "<<endl;
in>>1>>nm;
}
void student:: display()
{
cout <<"Roll NO:"<<1<<endl;
```

```
cout<<"name : "<<nm<<endl;
}
void exam ::getmarks()
{
cout<<"enter three subject marks  "<<endl;
cin>>s1>>s2>>s3;
}
void exam ::putmarks()
{
cout <<"subject 1:"<<s1<<endl;
cout<<" subject 2 : "<<s2<<endl;
cout <<"subject 3:"<<s3<<endl;
}
void result::process()
{
  t= s1+s2+s3;
  p = t/3.0;
}
void result::printresult()
{
cout<<"total = "<<t<<endl;
cout<<"per  = "<<p<<endl;
}
void main()
{
result x;
clrscr();
x.read();
x.getmarks();
x.process();
x.display();
x.putmarks();
x.printresult();
  getch();
 }
```

Q.4. (*a*) When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function? Explain with example. (5)

Ans. When should pure virtual functions be used in:

In C++, a regular, "non-pure" virtual function provides a definition, which means that the class in which that virtual function is defined does not need to be declared abstract. You would want to create a pure virtual function when it doesn't make sense to provide a definition for a virtual function in the base class itself, within the context of inheritance.

An example of when pure virtual functions are necessary: For example, let's say that you have a base class called Figure. The Figure class has a function called **draw**. And, other classes like Circle and Square derive from the Figure class. In the Figure class, it doesn't make sense to actually provide a definition for the draw function, because of the simple and obvious fact that a "Figure" has no specific shape. It is simply meant to act as a base class. Of course, in the Circle and Square classes it would be obvious what should happen in the draw function they should just draw out either a Circle or Square (respectively) on the page. But, in the Figure class it makes no sense to provide a definition for the draw function. And this is exactly when a pure virtual function should be used – the draw function in the Figure class should be a pure virtual function.

Q. 4. (*b*) Write a function template for finding the minimum value contained in an array. **(5)**

Ans.

```
#include<iostream.h>
template<class t>
void minarr(t a[])
{
t i,min;
min=a[0];
for(i=0;i<5;i++)
{
if(a[i]<min)
min=a[i];
}
cout<<min;
}
void main()
{
int a[5]={10,20,30,40,50};
char b[5]={'a','b','c','d','e'};
minarr(a);
minarr(b);
}
```

END TERM EXAMINATION [JUNE-2015]
FOURTH SEMESTER [B. TECH]
OBJECT ORIENTED PROGRAMMING [ETCS-210]

Time: 3 Hrs. **MM : 75**

Note: *Attempt any five questions including Q.No. 1 which is compulsory. Select one question from each unit.*

Q.1. Attempt the following questions: **(10 × 2.5 = 25)**

(a) Illustrate the use of delete operator in C++.

Ans. The delete Operator. Once the memory is allocated using new operator, it should released to the operating system. If the program uses large amount of memory using new, system may crash because there will be no memory available for operating system. The following expression returns memory to the operating system.

delete [] ptr;

The brackets [] indicates that, array is deleted. If you need to delete a single object then, you don't need to use brackets.

delete ptr;

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete.

Q.1. (b) Can you say that destructor in C++ may be overloaded? Justify your answer.

Ans. A destructor can never be overloaded. An overloaded destructor would mean that the destructor has taken arguments. Since a destructor does not take arguments, it can never be overloaded. An object that is going to be destroyed needs only one way of cleaning itself up.

Destructor has no arguments and no return type to enforce that once it is called, the object is no longer useable.

Q.1. (c) Abstraction and Encapsulation may be used interchangeable in the context of object orientation/justify your answer.

Ans.

Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the details internal or mechanics of how an object does something.
4. **Abstraction**: Outer layout, used in terms of design. **For Example:** Outer Look of a Mobile, Phonelike it has a display screen and keypad buttons to dial a number.	4. **Encapsulation**: Inner layout, used in terms of implementation. **For Example:** Inner Implementation detail of a Mobile Phone, how keypad button and display screen are connect with each other using circuits.

Q. 1. (*d*) Differentiate between 'array pointer' and 'pointer array' by writing single line C++ statement in each case.

Array of pointers : int *array of pointers[4]; We know that pointer holds a address of int data type variable. So the above mentioned array_of_pointers can store four different or same int data type address *i.e.* array_of_pointers[0], array_of_pointers [1], array_of_pointers[2] array_of_pointers[3] each can hold a int data type address.

Pointer to array :

In simple language the pointer_to_array is a single variable that holds the address of the array.

Example:

```
int simple_array[4];
Int *p;
Now we can assign
 p = simple_array;
```

Q.1. (*e*) What is the significance of enum data type in C++? Exlain with illustration.

Ans. An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword enum is used to defined enumerated data type.

enum type_name{ value1, value2,...,valueN };

Here, *type_name* is the name of enumerated data type or tag. And *value1*, *value2*,....,*valueN* are values of type *type_name*.

By default, *value*1 will be equal to 0, *value*2 will be 1 and so on but, the programmer can change the default value.

Changing the default value of enum elements

```
enum suit{
   club=0;
   diamonds=10;
   hearts=20;
   spades=3;
};
```

Q.1. (*f*) How static data members are different from non static data members?

Ans : Static data mamber: A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a static variable. A static member variable has certain special characeristics.

1. It is initialized to 0 (zero) when the first object of this class is created. No other initialization is permitted.

2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, No matter how many objects are created.

3. It is visible only within the class but its lifetime is in the entire program.

The type and scope of each static member variable must be defined outside the class definition.This is necessary because the ststic data member are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any object. They are also known as class variable.

Data member: Data memeber is the simple variable of a class which is initialized by member function of same class. By default data member is private and external functions can not access it.

Q.1. (g) What do you mean be compile time polymmorhism.?

Ans. Polymorphism is the ability of an object or reference to take many different forms at different instances. These are of two types one is the "compile time polymorphism" and other one is the "run-time polymorphism".

Compile time polymorphism: In this method object is bound to the function call at the compile time itself or we can say that events occurring at compile time are known as compile time polymorphism or early binding or static binding. In the process of early binding all info which is required for a function call is known at compile time. Early binding is a fast and efficient process. Examples of early binding: function calls, overloaded function calls, and overloaded operators.

Q.1. (h) Interpret the C++ statement friend return type op++ (x, y);

Ans. Friend return-type op++(*x*, *y*)

This statement belongs to the declaration of the friend function. In this statement the unary operator++ is overloaded an arguments x and y with the help of friend function. To overload the unary operators with the help of friend function, minimum one argument is required and to overload the binary operators with the help of friend function, minimum two arguments are required.

The keyword friend procedes function prototype declaration. It must be written inside the class. This function can be defined inside or outside the class. The arguments used in friend function are generally objects of the friend class. Friend function can access private member of the class through the objects.

Q.1. (i) Define the term 'type casting'.

"**Type Casting**" is method using which a variable of one datatype is converted to another datatype, to do it simple the datatype is specified using parenthesis in front of the value.

Example:

```
#include <iostream.h> using namespace std; int main()
{
short x=40;
int y;
y = (int)x;
cout << "Value of y is:: "<< y <<"\nSize is::"<< sizeof(y);
return 0;
}
```

Result:

Value of y is:: 40

Size is::4

In the above example the short data type value of x is type cast to an integer data type, which occupies "4" bytes.

Type casting can also done using some typecast operators available in C++. Following are the typecast operators used in C++.

- static_cast

- const_cast
- dynamic_cast
- reinterpret_cast
- typeid

Q.1. (j) What is the purpose of virtual function?

Ans. Virtual function is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. In other words, the purpose of virtual functions is to allow customization of derived class implementations.

If there are member functions with same name in base class and derived class, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context. This feature in C++ programming is known as polymorphism which is one of the important feature of OOP.

If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable.

UNIT-I

Q.2. Write a program to read records of students such as student name, rollno, total marks through a reading function that expect an array o strucute as input from the main function and also display the name of those students who got more than 60% marks through display function. (12.5)

Ans.

```
#include<iostream.h>
#include<conio.h>
struct stud
{
int rno;
char name[20];
int total;
}s[5];
void getdata(struct stud st[])
{
int i;
cout<<"Enter the Roll no, Name and Total marks of 5 students :-"<<"\n";
for(i=0;i<5;i++)
cin>>st[i].rno>>st[i].name>>st[i].total;
}
void main()
{
int i;
clrscr();
getdata(s);
for(i=0;i<5;i++)
{
```

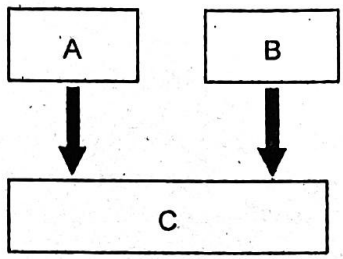
```
if(s[i].total>60)
cout<<s[i].rno<<"\t"<<s[i].name<<"\t"<<s[i].total<<"\n";
}
getch();
}
```

Q.3. Illustrate the difference between multiple inheritance, multilevel inheritance and hybrid inheritance through small working programs. (12.5)

Ans. One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

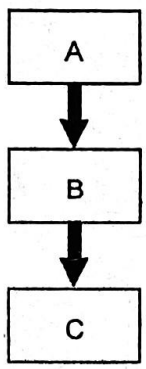
Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



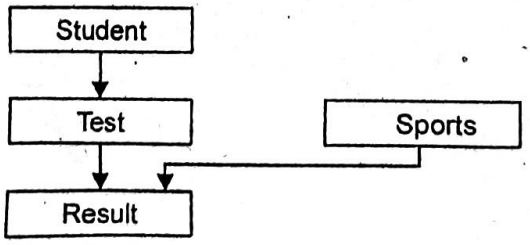
Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid Inheritance

Hybrid Inheritance is combination of two or more Inheritance such as multilevel and multiple.



Program to implement the hybrid inheritance :

```
#include<iostream.h>
#include<conio.h>
```

```
class stu
{
protected:
            int rno;
public:
  void get_no(int a) { rno=a; }
void put_no(void) { cout<<"Roll no"<<rno<<"\n"; }
};
class test:public stu
{
        protected:
        float part1,part2;
public:
     void get_mark(float x,float y)
       { part1=x; part2=y;  }
void put_marks()
{
 cout<<"Marks obtained:"<<"part1="<<part1<<"\n"<<"part2="<<part2<<"\n";
}
};
class sports
{
     protected:
     float score;
public:
 void getscore(float s)  { score=s; }
void putscore(void)
{
  cout<<"sports:"<<score<<"\n";
}
};
class result: public test, public sports
{
   float total;
public:
   void display(void);
};
void result::display(void)
{
   total=part1+part2+score;
   put_no();
   put_marks();
```

```
    putscore();
    cout<<"Total Score="<<total<<"\n";
}
int main()
{
    clrscr();
    result stu;
    stu.get_no(123);
    stu.get_mark(27.5,33.0);
    stu.getscore(6.0);
    stu.display();
    return 0;
}
```

UNIT–II

Q.4. What is the use of various kinds of constructors and destructors in C++? Explain with illustrations in each case (12.5)

Ans. Constructor: It is a member function having same name as it's class and which is used to initialize the objects of that class type with a legel initial value. Constructor is automatically called when object is created.

Types of Constructor

Default Constructor: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

```
Circle :: Circle()
{
    radius = 0;
}
```

Parameterized Constructor: A constructor that receives arguments/parameters, is called parameterized constructor.

```
Circle :: Circle(double r)
{
    radius = r;
}
```

Copy Constructor : A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
Circle :: Circle(Circle &t)
{
    radius = t.radius;
}
```

There can be multiple constructors of the same class, provided they have different signatures.

Destructor: A destructor is a member function having same name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.

```
~Circle() {}
```

Example: In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructor in class so this example also illustrates the concept of constructor overloading

```
#include<iostream>
using namespace std;
class Circle //specify a class
{
private :
double radius; //class data members
public:
  Circle() //default constructor
  {
      radius = 0;
  }
  Circle(double r) //parameterized constructor
  {
      radius = r;
  }
  Circle(Circle &t) //copy constructor
  {
    radius = t.radius;
  }
  void setRadius(double r) //function to set data
  {
    radius = r;
  }
  double getArea()
  {
      return 3.14 * radius * radius;
  }
  ~Circle() //destructor
    {}
  };
int main()
{
  Circle c1; //defalut constructor invoked
  Circle c2(2.5); //parmeterized constructor invoked
  Circle c3(c2); //copy constructor invoked
  cout << c1.getArea()<<endl;
  cout << c2.getArea()<<endl;
  cout << c3.getArea()<<endl;
  return 0;
}
```

Q.5. Design a class to represent a bank account having data members as customer name, account number, account type, balance amount, etc. and methods as to assign initial values, to deposit an amount, to withdraw an amount and to display the name of the of those customers who have balance more than Rs. 50000. (12.5)

Ans.

```
#include<iostream.h>
#include<conio.h>
class bank
{
char name[20];
char type[10];
long int amount;
public:
long int acn;
void init();
void deposit();
void withdraw();
void display();
};
void bank :: init()
{
cout<<"Enter the Account No., Name of Customer, A/c Type and the Balance Amou
:"<<"\n";
cin>>acn>>name>>type>>amount;
}
void bank :: deposit()
{
int amt;
cout<<"Enter amount which you want to deposit:"<<"\n";
cin>>amt;
amount=amount+amt;
}
void bank :: withdraw()
{
int amt;
cout<<"Enter amount which you want to withdraw:"<<"\n";
cin>>amt;
amount=amount-amt;
}
void bank :: display()
{
if(amount>50000)
cout<<acn<<"\t"<<name<<"\t"<<type<<"\t"<<amount<<"\n";
```

```
}
void main()
{
clrscr();
bank obj[5];
int i,ch;
char choice;
long int acno;
cout<<"Enter details for 5 customers:"<<"\n";
for(i=0;i<5;i++)
obj[i].init();
do
{
cout<<"(1)Deposit an Amount"<<"\n";
cout<<"(2)Withdraw an Amount"<<"\n";
cout<<"(3)Display the Record"<<"\n";
cout<<"What do you want to do;-"<<"\n";
cin>>ch;
switch(ch)
{
case 1: cout<<"Enter an account no. in which you want to deposit:"<<"\n";
        cin>>acno;
        for(i=0;i<5;i++)
        {
        if(acno==obj[i].acn)
        obj[i].deposit();
        }
        break;
case 2: cout<<"Enter an account no. in which you want to withdraw:"<<"\n";
        cin>>acno;
        for(i=0;i<5;i++)
        {
        if(acno==obj[i].acn)
        obj[i].withdraw();
        }
        break;
case 3: for(i=0;i<5;i++)
        obj[i].display();
        break;
default:cout<<"Wrong choice";
}
cout<<"Do you want to continue (y/n)"<<"\n";
```

```
oid numberoflines();
nt main(){
    string line;
    ifstream myfile("texte/sample.txt");
  if(myfile.is_open()){
    while(!myfile.eof()){
      getline(myfile,line);
      cout<< line << endl;
      number_of_lines++;
    }
     myfile.close();
  }
   numberoflines();
}
void numberoflines(){
   number_of_lines--;
   cout<<"number of lines in text file: " << number_of_lines << endl;
}
```

UNIT-IV

Q.8.(a) What do you understandd by the concept of recursion? What is its advantages and illustrate the use by making your own power function through recursion. (6.5)

Ans. Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Advantages of Recursion: 1. Using recursion we can avoid unnecessary calling of functions.

2. Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex. Ex : Tower of Hanoi. You reduce size of the code when you use recursive call.

3. Recursion is used to divide the problem into same problem of subtypes and hence replaces complex nesting code.

4. A recursive definition defines an object in simpler cases of itself reducing nested looping complexity.

5. Recursive functions can be effectively used to solve problems where the solution is expressed in terms of applying the same solution.

Power function using recursion :

```
#include<iostream.h>
int main()
{
int pow,num;
long int res;
long int power(int,int);
```

```
cin>>choice;
}
while(choice=='y' || choice=='Y');
getch();
}
```

UNIT–III

Q.6. When do you need operator overloading? Illustrate bianry operator overloading through a working a working program which shows multiplication of two complex numbers. (12.5

Ans. C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.

You can redefine or overload most of the built-in operators available in C++. Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Program: Overload the binary * operator.

```
#include<iostream.h>
using namespace std;
class complex{
  float x1,y1;
      public:
            void disp(){cout<<x1<<"+"<<y1<<"i"<<"\n";}
            complex(float a,float b)
            { x1=a; y1=b; }
            complex operator*(complex cc)
{ return complex(x1*cc.x1, y1*cc.y1);}
};
int main(){
     complex c1(2.5,-0.3);
     complex c2(4.5,7);
     complex c3=c1*c2;
     c3.disp();
     return 0;
}
```

Q.7. Write a program that counts the number of lines in a file considering a text file consisting of data that is passed to a function for counting the lines. (12.5

Ans.

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

int number_of_lines = 0;
```

```
void numberoflines();
int main(){
    string line;
    ifstream myfile("textexample.txt");
  if(myfile.is_open()){
    while(!myfile.eof()){
      getline(myfile,line);
      cout<< line << endl;
      number_of_lines++;
    }
     myfile.close();
  }
   numberoflines();
}
void numberoflines(){
   number_of_lines—;
   cout<<"number of lines in text file: " << number_of_lines << endl;
}
```

UNIT–IV

Q.8.(a) What do you understandd by the concept of recursion? What is its advantages and illustrate the use by making your own power function through recursion. **(6.5)**

Ans. Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Advantages of Recursion: 1. Using recursion we can avoid unnecessary calling of functions.

2. Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex. Ex : Tower of Hanoi. You reduce size of the code when you use recursive call.

3. Recursion is used to divide the problem into same problem of subtypes and hence replaces complex nesting code.

4. A recursive definition defines an object in simpler cases of itself reducing nested looping complexity.

5. Recursive functions can be effectively used to solve problems where the solution is expressed in terms of applying the same solution.

Power function using recursion :

```
#include<iostream.h>
int main()
{
int pow,num;
long int res;
long int power(int,int);
```

```
void numberoflines();
int main(){
    string line;
    ifstream myfile("textexample.txt");
  if(myfile.is_open()){
    while(!myfile.eof()){
      getline(myfile,line);
      cout<< line << endl;
      number_of_lines++;
    }
     myfile.close();
  }
   numberoflines();
}
void numberoflines(){
   number_of_lines--;
   cout<<"number of lines in text file: " << number_of_lines << endl;
}
```

UNIT–IV

Q.8.(a) What do you understandd by the concept of recursion? What is its advantages and illustrate the use by making your own power function through recursion. **(6.5)**

Ans. Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Advantages of Recursion: 1. Using recursion we can avoid unnecessary calling of functions.

2. Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex. Ex : Tower of Hanoi. You reduce size of the code when you use recursive call.

3. Recursion is used to divide the problem into same problem of subtypes and hence replaces complex nesting code.

4. A recursive definition defines an object in simpler cases of itself reducing nested looping complexity.

5. Recursive functions can be effectively used to solve problems where the solution is expressed in terms of applying the same solution.

Power function using recursion :

```
#include<iostream.h>
int main()
{
int pow,num;
long int res;
long int power(int,int);
```

```
cout<<"Enter a number and a power:";
cin>>num>>pow;
res=power(num,pow);
cout<<"Result is"<<res;
return 0;
}
int i=1;
long int sum=1;
long int power(int num,int pow){
    if(i<=pow){
      sum=sum*num;
      power(num,pow-1);
    }
    else
    return sum;
}
```

Q.8. (b) What do you mean by exception handling? Explain various constracts supported by it through illustrations. (6)

Ans. An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **Throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **Catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **Try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

The following is an example, which throws a division by zero exception and we catch it in catch block.

```
#include <iostream>
using namespace std;
double division(int a, int b)
{
  if( b == 0 )
  {
     throw "Division by zero condition!";
  }
  return (a/b);
}
int main ()
{
  int x = 50;
  int y = 0;
```

```
    double z = 0;
    try {
       z = division(x, y);
       cout << z << endl;
    }catch (const char* msg) {
       cerr << msg << endl;
      }
       return 0;
}
```

Q.9 (a) Differentiate between function overloading and function overriding with the help of small working programs. (6)

Ans: Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Function Overloading

```
/*Calling overloaded function test() with different arguments.*/
#include <iostream>
using namespace std;
void test(int);
void test(float);
void test(int, float);
int main() {
   int a = 5;
   float b = 5.5;
   test(a);
   test(b);
   test(a, b);
   return 0;
}
void test(int var) {
     cout<<"Integer number: "<<var<<endl;
}
void test(float var){
     cout<<"Float number: "<<var<<endl;
}
void test(int var1, float var2) {
  cout<<"Integer number: "<<var1;
  cout<<" And float number:"<<var2;
}
```

Function Overriding

```
class A
{
 int a;
public:
 A()   {       a = 10;   }
```

```
    void show()
    {          cout << a;        }
};
class B: public A
{
    int b;
    public:
    B()      {      b = 20;   }
    void show()
    {            cout << b;       }
};
int main()
{
    A ob1;
    B ob2;
    A::show();
    B::show();
    return 0;
}
```

Q.9. (b) What is standard template library? How is it different from the C++ standard library? Why is it gaining importance among the programmers? (6)

Ans. The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components:

Component	Description
Containers	Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
Algorithms	Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
Iterators	Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Difference between STL and C++ Standard Library: The standard C++ library is a collection of functions, constants, classes, objects and templates that extends the C++ language providing basic functionality to perform several tasks, like classes to interact with the operating system, data containers, manipulators to operate with them and algorithms commonly needed.

The Standard Template Library (STL), part of the C++ Standard Library, offers collections of algorithms, containers, iterators, and other fundamental components implemented as templates, classes, and functions essential to extend functionality and standardization to C++. STL main focus is to provide improvements implementation standardization with emphasis in performance and correctness.

FIRST TERM EXAMINATION [MARCH–2016]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

M.M. : 30

Time : 1½ hrs.

Note: *Question No.1 is compulsory. Attempt any two more Questions from the rest.*

Q.1. (a) Determine the output for the following code. (2)

```
int main()
{ int x = 10, y = 20;
int * ptr = &x;
int &ref = y;
* ptr ++;
ref ++;
cout <<x <<"" < < y;
return 0;
}
```

Ans. Output :10 21 (2)

Q.1. (b) Determine the output for the following code.

```
class A
{
int x = 10;
Public:
Void display()
{
cout<<" The value of X = "<<x<<endl;"
}
};
void main()
{
A Obj;
Obj. display();
}
```

Ans. Output: Error in Line no. 3. "Can Not Initialize a Class Member Here". (2)

Q.1. (c) Determine the output for the following code.

```
class Test
{
static int i;
int j;
}:
int Test::i;
int main()
{
```

```
cout<< size of (Test):
return 0:
}
```

Ans. Output:2

Q.1. (d) Determine the output for the following code. (2)

```
class A
{
int i, j;
Public:
Void setdata()
{
i = 10;
j = 20;
}
void getdata (int i, int j)
i = i;
j=j;
cout<< "Value of I is  = <<i<<endl;
cout<<"Value of J is = " <<j <<endl;
}};
void main()
{
A Obj;
Obj.setdata();
Obj.getdata(2,3);
}
```

Ans. Output : Value of I is =2
Value of J is =3

Q.1. (e) Describe the use of scope resolution operator and referece operator. (1)

Ans. Scope resolution operator (::): Scope resolution operator(::) is used to define a function outside a class or when we want to use a global variable but also has a local variable with same name.

Reference operator (&): A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

Q.1. (f) What is the role of 'new' operator in C++? (1)

Ans. New Operator: You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.

For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the new operator with the following statements:

```
double* pvalue = NULL; // Pointer initialized with null
pvalue = new double; // Request memory for the variable
```

Q.2.(a) Explain the various features of object oriented programming. (5)

Ans. Refer Q.3(b) of End Term Examination 2016.

Q.2. (b) What is class? Write a program to create a class called employee which consist name, desg, ecode and salary as a data member and read, write as a function member, using this class to read and print 10 employee information. (5)

Class: A class in C++ is a user defined type or data structure declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a class is private).

Program:

```
#include<iostream.h>
#include<string.h>
class employee
{
public:
char name[20];
char desg[20];
int ecode;
long int sal;
void read()
{
cout<<"Enter the name and designation of employee";
gets(name);
gets(desg);
cout<<"Enter the code and salary of employee";
cin>>ecode>>sal;
}
void print()
{
cout<<"Name is "<<name<<"\t"<<"Designation is "<<desg<<"\t"<<"Code is "<<code<<"\t"<<"Salary is "<<sal<<"\n";
}
};
void main()
{
employee e[10];
int i;
cout<<"Enter the record of 10 employees :"
for(i=0;i<10;i++)
e[i].read();
for(i=0;i<10;i++)
e[i].print();
}
```

Q.3. (a) What is constructor? Mention its type. Explain copy constructor with example. (5)

Ans. Constructor: It is a member function having same name as it's class and which is used to initialize the objects of that class type with a legel initial value. Constructor is automatically called when object is created.

Types of Constructor

Default Constructor: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

```
Circle :: Circle()
{
   radius = 0;
}
```

Parameterized Constructor : A constructor that receives arguments/parameters is called parameterized constructor.

```
Circle :: Circle(double r)
{
   radius = r;
}
```

Copy Constructor-: A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
Circle :: Circle(Circle &t)
{
   radius = t.radius;
}
```

There can be multiple constructors of the same class, provided they have different signatures.

Example :

```
class Circle //specify a class
{
   private :
      double radius; //class data members
   public:
 Circle() //default constructor
 {
    radius = 0;
 }
 Circle(double r) //parameterized constructor
 {
    radius = r;
 }
 Circle(Circle &t) //copy constructor
 {
    radius = t.radius;
 }
 void setRadius(double r) //function to set data
 {
   radius = r;
   }
      double getArea()
```

```
  {
    return 3.14 * radius * radius;
  }
 };

int main()
{
   Circle c1; //defalut constructor invoked
   Circle c2(2.5); //parmeterized constructor invoked
   Circle c3(c2); //copy constructor invoked
   cout << c1.getArea()<<endl;
   cout << c2.getArea()<<endl;
   cout << c3.getArea()<<endl;
   return 0;
}
```

Q.3. (b) Write a C++ program to keep track of the number of objects created a particular class without using extern variable. (5)

Ans.

```
#include<iostream.h>
class A
{
static int count;
public:
A() //constructor
{
cout<<(++count)<<"Objects have been created";
}
};
int A::count=0; //initialize
int main()
{
A ob1,ob2,ob3;
return 0;
}
```

Q.4. (a) Why friend function is required? Write a program to add two complex number using friend function. (5)

Ans. A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class Box
{
   double width;
public:
```

```
    double length;
    friend void printWidth( Box box );
    void setWidth( double wid );
};
```

C++ Program for add two complex no using friend function.

```
#include<iostream.h>
#include<conio.h>
class Cmplx1
{
int real,imagin;
public :
void get()
{
cout<<"\n\n\tENTER THE REAL PART : ";
cin>>real;
cout<<"\n\n\tENTER THE IMAGINARY PART : ";
cin>>imagin;
}
friend void sum(Cmplx1,Cmplx1);
};
void sum(Cmplx1 c1,Cmplx1 c2)
{
cout<<"\n\t\tRESULT : ";
cout<<"\n\n\t["<<c1.real<<" + i "<<c1.imagin;
cout<<" ] + [ "<<c2.real<<" + i "<<c2.imagin;
cout<<" ] = "<<c1.real+c2.real<<" + i "<<c1.imagin+c2.imagin;
}
void main()
{
Cmplx1 op1,op2;
clrscr();
cout<<"\n\n\tADDITION OF TWO COMPLEX NUMBERS USING FRIEND
FUNCTIONS\n\n";
cout<<"\n\tINPUT\n\n\t\tOPERAND 1";
op1.get();
cout<<"\n\n\t\tOPERAND 2";
op2.get();
sum(op1,op2);
getch();
}
```

Q.4.(b) Define function overloading. Demonstrate with C++ program. (5)

Ans. You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Function Overloading

```
/*Calling overloaded function test() with different arguments.*/
#include <iostream>
using namespace std;
void test(int);
void test(float);
void test(int, float);
int main() {
        int a = 5;
        float b = 5.5;
    test(a);
    test(b);
    test(a, b);
    return 0;
}
void test(int var) {
    cout<<"Integer number: "<<var<<endl;
}
void test(float var){
    cout<<"Float number: "<<var<<endl;
}
void test(int var1, float var2) {
    cout<<"Integer number: "<<var1;
    cout<<" And float number:"<<var2;
}
```

Q.4 (e) What are the difference between class and structure in C++?

Ans. In C++; the only difference between a struct and a class is that struct members are public by default, and class members are private by default.

However, as a matter of style, it's best to use the struct keyword for something that could reasonably be a struct in C (more or less POD types), and the class keyword if it uses C++-specific features such as inheritance and member functions.

The members and base classes of a struct are public by default, while in class, they default to private. Note: you should make your base classes *explicitly* public, private, or protected, rather than relying on the defaults.

A struct simply *feels* like an open pile of bits with very little in the way of encapsulation or functionality. A class *feels* like a living and responsible member of society with intelligent services, a strong encapsulation barrier, and a well defined interface. Since that's the connotation most people already have, you should probably use the struct keyword if you have a class that has very few methods and has public data (such things *do* exist in well designed systems!), but otherwise you should probably use the class keyword.

SECOND TERM EXAMINATION [APRIL-2016]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 1½ hrs. **M.M. : 30**

Note: *Question No. 1 is compulsory. Attempt any two more Questions from the rest.*

Q.1.(a) Differentiate between early and Late binding. (2)

Ans. Early Binding: Events occurring at compile time are known as early binding. In the process of early binding all info which is required for a function call is known at compile time. Early binding is a fast and efficient process. Examples of early binding function calls, overloaded function calls, and overloaded operators.

Late Binding: In this process all info which is required for a function call is not known at compile time. Hence, objects and functions are not linked at run time. Late Binding is a slow process. However, it provides flexibility to the code. Example of Late Binding: Virtual functions.

Q.1.(b) How to properties of the following two derived classes differ? (2)

(i) class D1 : private B { } (ii) class D1 : public B { }

Ans. (*i*) Class D1: private B{}: In this statement the Class D1 is privately derived from Base Class B.

(*ii*) Class D1 : Public B { }: In this statement the Class D1 is publically derived from Base Class B.

Q.1.(c) Explain hybrid inheritance with example. (2)

Ans. Hybrid Inheritance:

Hybrid Inheritance is a method where one or more types of inheritance are combined together and used.

Program:

```
#include<iostream.h>
#include<conio.h>
class arithmetic
{
protected:
int num1, num2;
public:
void getdata()
{
cout<<"For Addition:";
cout<<"\nEnter the first number: ";
cin>>num1;
cout<<"\nEnter the second number: ";
cin>>num2;
}
};
class plus:public arithmetic
{
protected:
int sum;
```

```
public:
void add()
{
sum=num1+num2;
}
};
class minus
{
protected:
int n1,n2,diff;
public:
void sub()
{
cout<<"\nFor Subtraction:";
cout<<"\nEnter the first number: ";
cin>>n1;
cout<<"\nEnter the second number: ";
cin>>n2;
diff=n1-n2;
}
};
class result:public plus, public minus
{
public:
void display()
{
cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
}
};
void main()
{
clrscr();
result z;
z.getdata();
z.add();
z.sub();
z.display();
getch();
}
```

Q.1. (d) What are containers? Describe various types of containers. (2)

Ans. Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc. The containers are class templates; when you declare a container variable, you specify the type of the elements that the container will hold. Containers can be constructed with initializer lists. They have member functions for adding and removing elements and performing other operations.

Two basic types of containers:

- Sequences
  - User controls the order of elements.

- vector, list, deque
- Associative containers
  - The container controls the position of elements within it.
  - Elements can be accessed using a *key*.
  - set, multiset, map, multimap

Q.1. (e) What is difference between a class and an abstract class in C++? (2)

Ans. Abstract Class: Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Class: A class in C++ is a user defined type or data structure declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a class is private).

Q.2.(i) Create three classes namely Student, Exam and Result. Student class represents student information. Exam class, inherited from student class represents marks scored in five subjects. Result class is derived from exam class, calculating total marks. Write an interactive program to represents this inheritance. Also the type of inheritance depicted by this program. (6)

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student    // base class
{
private:
 int rl;
 char nm[20];
public:
  void read();
  void display();
};
class exam : public student // dervivec from student
{
protected:
 int s1;
 int s2;
 int s3;
public:
  void getmarks();
  void putmarks();
};
class result : public exam // derived  from marks
{
private:
  int t;
  float p;
  public:
   void process();
   void printresult();
```

```
};
 void student::read()
 {
  cout<<"enter Roll no and Name "<<endl;
  cin>>rl>>nm;
 }
 void student:: display()
 {
 cout <<"Roll NO:"<<rl<<endl;
 cout<<"name : "<<nm<<endl;
 }
 void exam ::getmarks()
 {
 cout<<"enter three subject marks  "<<endl;
  cin>>s1>>s2>>s3;
 }
 void exam ::putmarks()
 {
  cout <<"subject 1:"<<s1<<endl;
  cout<<" subject 2 : "<<s2<<endl;
  cout <<"subject 3:"<<s3<<endl;
 }

 void result::process()
 {
   t= s1+s2+s3;
   p = t/3.0;
   }
 void result::printresult()
 {
  cout<<"total = "<<t<<endl;
  cout<<"per  = "<<p<<endl;
  }
 void main()
 {
  result x;
  clrscr();
  x.read();
  x.getmarks();
  x.process();
  x.display();
  x.putmarks();
  x.printresult();
 getch();
  }
```

This the the program of Multilevel Inheritance.

Q.2.(ii) Distinguish between virtual function and pure virtual functions with example? (4)

Ans. Virtual Function and Pure Virtual Function defers in declaration. Virtual Function is declared with keyword 'virtual' at the start of declaration.

Example : virtual return_type function_name(function arguments);

While Pure Virtual Function is declared as

Example : virtual return_type function_name(function arguments) = 0;

Sometime it has been written that pure virtual function has no function body. But this is not always true. Having a default definition of pure virtual function is not common but its possible.

Example : virtual return_type function_name(function arguments) = 0; //Declaration

```
return_type class_name::function_name(function argument) //Definition
{
.
.
.
}
```

Virtual Function and Pure Virtual Function defers in use also. Virtual Function makes its class a polymorphic base class. Derived classes may override the virtual function and redefine the behavior. Virtual Function called from base class pointer/ reference will be decided on run time. Which is called run time binding.

Pure Virtual Function provides the similar functionality as Virtual Function. Apart from that Pure Virtual Function makes class a Abstract Class. It means, if a class have Pure Virtual Function, class can't be instantiated. Only reference or pointers can be created of such class.

Pure Virtual Function in base classes enforce derived classes to define the pure virtual function. Else derived class will also be Abstract Class.

Q.3. (i) Wirte a program to overload the pre-increment and post-increment++ operator. (5)

Ans.

```
#include <iostream.h>
using namespace std;
class Integer {
  private:
    int value;
  public:
    Integer(int v) : value(v) { }
    Integer operator++();
    Integer operator++(int);
    int get Value() {
      return value;
    }
};
// Pre-increment Operator
Integer Integer::operator++()
{
  value++;
  return *this;
```

```
// Post-increment Operator
Integer Integer::operator++(int)
{
  const Integer old(*this);
  ++(*this);
  return old;
}
int main()
{
  Integer i(10);
   cout << "Post Increment Operator" << endl;
  cout << "Integer++ : " << (i++).getValue() << endl;
  cout << "Pre Increment Operator" << endl;
  cout << "++Integer : " << (++i).getValue() << endl;
}
```

Q.3. (*ii*) Differentiate between Function Overloading and Function Overriding with example? (5)

Ans.

Overriding	Overloading
Functions name and signatures must be same.	Having same Function name with different Signatures.
Overriding is the concept of runtime polymorphism.	Overloading is the concept of compile time polymorphism.
When a function of base class is re-defined in the derived class called as Overriding.	Two functions having same name and return type, but with different type and/or number of arguments is called as Over-loading.
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types.
Function should be public.	Function can be different access specifies.

Function Overloading

```
/*Calling overloaded function test() with different arguments.*/
#include <iostream.h>
using namespace std;
void test(int);
void test(float);
void test(int, float);
int main() {
  int a = 5;
  float b = 5.5;
  test(a);
  test(b);
  test(a, b);
  return 0;
}
void test(int var) {
```

```
    cout<<"Integer number: "<<var<<endl;
}
void test(float var){
    cout<<"Float number: "<<var<<endl;
}
void test(int var1, float var2) {
    cout<<"Integer number: "<<var1;
    cout<<" And float number:"<<var2;
}
```

Function Overriding

```
class A
{
      int a;
      public:
       A()  {          a = 10;       }
       void show()
       {          cout << a;       }
};
class B: public A
{
      int b;
      public:
      B()       {          b = 20;       }
      void show()
      {          cout << b;       }
};
int main()
{
      A ob1;
      B ob2;
      A::show();
      B::show();
      return 0;
}
```

Q.4.(*i*) Write a program to demonstrate the use of multiple catch block. (5)

Ans.

```
#include<iostream.h>
#include<conio.h>
void test(int x)
{
 try
 {
        if(x>0)
          throw x;
    else
          throw 'x';
 }

 catch(int x)
 {
```

```
        cout<<"Catch a integer and that integer is:"<<x;
    }

    catch(char x)
    {
        cout<<"Catch a character and that character is:"<<x;
    }
}

void main()
{
  clrscr();
  cout<<"Testing multiple catches\n:";
  test(10);
  test(0);
  getch();
}
```

Q.4. (*ii*) What is generic programming? How is it implemented in C++? (5)

Ans. Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.

You can use templates to define functions as well as classes, let us see how do they work:

Function Template:

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

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

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

Class Template:

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name {
.
.
.
}
```

Here, type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

END TERM EXAMINATION [MAY-JUNE 2016]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q.No. 1 which is compulsory. Select one question from each unit.*

Q.1. (a) How does new operator in C++ is different from dynamic allocation functions in C? **(2.5×10=25)**

Ans. 1.Operator new constructs an object (calls constructor of object), malloc does not. Hence new invokes the constructor (and delete invokes the destructor) This is the most important difference.

2. Operator new is an operator, malloc is a function.

3. Operator new can be overloaded, malloc cannot be overloaded.

4. Operator new throws an exception if there is not enough memory, malloc returns a NULL.

5.Operator new[] requires you to specify the number of objects to allocate, malloc requires you to specify the total number of bytes to allocate.

6. Operator new/new[] must be matched with operator delete/delete[] to deallocate memory, malloc() must be matched with free() to deallocate memory.

Q.1. (b) How static member is different from a non static member?

Ans. STATIC DATA MEMBER: A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a static variable. A static member variable has certain special characeristics.

(1) It is initialized to 0(zero) when the first object of this class is created. No other initialization is permitted.

(2) Only one copy of that member is created for the entire class and is shared by all the objects of that class, No matter how many objects are created.

(3) It is visible only within the class but its lifetime is in the entire program.

The type and scope of each static member variable must be defined outside the class definition.This is necessary because the static data member are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any object. They are also known as class variable.

NON STATIC DATA MEMBER: Data member is the simple variable of a class which is initialized by member function of same class. By default data member is private and external functions can not access it.

Q.1. (c) What do you mean by an abstract class?

Ans. Abstract Class: Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class:

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.

2. Abstract class can have normal functions and variables along with a pure virtual function.

3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Q.1. (d) What is Inline function? How is it different from a macro?

Ans. Inline function: C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time. To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function.

```
// The use of the 'inline' keyword

inline int sum(int a, int b)
{
   return (a+b);
}
```

The major difference between inline functions and macros is the way they are handled. Inline functions are parsed by the compiler, whereas macros are expanded by the C++ preprocessor. Inline follows strict parameter type checking, macros do not. Debugging of macros is also difficult. This is because the preprocessor does the textual replacement for macros, but that textual replacement is not visible in the source code itself. Because of all this, it's generally considered a good idea to use inline functions over macros.

Q.1.(e) How does C++ support data abstraction?

Ans. Data Abstraction : Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

While defining a class, both member data and member functions are described. However while using an object (that is an instance of a class) the built in data types and the members in the class are ignored. This is known as **data abstraction**. In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class iostream to stream data to standard output like this:

```
#include <iostream.h>
int main( )
{
  cout << "Hello C++" <<endl;
  return 0;
}
```

Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

Q.1. (f) What do you mean by a generic data type?

Ans. Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes.

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

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

The general form of a generic class declaration is shown here:

```
template <class type> class class-name {
.
.
.
}
```

Q.1. (g) Explain the role of reference variable with suitable example?

Ans. Reference variable : A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

For example, suppose we have the following example:

```
int   i = 17;
```

We can declare reference variables for i as follows.

```
int&   r = i;
```

References are usually used for function argument lists and function return values.

Q.1. (h) What do you mean by the name space concept?

Ans. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name {
// code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
name::code;  // code could be variable or function.
```

Q.1. (i) What do you mean by base address of an array? Can you access the content of an array by incrementing the base address value?

Ans. An array is a sequence of objects, each of the same data type. The starting address of this array of objects, i.e. the address of the first object in the array is called **the base address** of the array. The address of each successive element of the array is offset from the base by the size of the array type, e.g. for each successive element of an integer array, the address is offset by the size of an integer type object.

Yes we can access the content of an array by incrementing the base address value

Q.1.(i) What is type casting and when is it used?

Ans. "Type Casting" is method using which a variable of one datatype is converted to another datatype, to do it simple the datatype is specified using parenthesis in front of the value.

Example:

```
#include <iostream.h> using namespace std; int main()
{
short x=40;
int y;
y = (int)x;
cout << "Value of y is:: "<< y <<"\nSize is::"<< sizeof(y);
return 0;
}
```

Result:

Value of y is:: 40

Size is::4

In the above example the short data type value of x is type cast to an integer data type, which occupies "4" bytes.

Type casting can also done using some typecast operators available in C++. following are the typecast operators used in C++.

- static_cast
- const_cast
- dynamic_cast
- reinterpret_cast
- typeid

UNIT-I

Q.2. (a) Wirte a C++ code to store 6 integer numbers using dynamic memory allocation approach. (6)

Ans.

```
#include <iostream.h>
using namespace std;
int main()
{
int n, *pointer, c;
pointer = new int[6];
cout <<"Input 6 integers\n";
for (c = 0; c <6; c++)
cin >> pointer[c];
cout <<"Elements entered by you are\n";
for (c = 0; c <6; c++)
cout << pointer[c] << endl;
delete[] pointer;
return 0;
}
```

Q.2. (b) Write a program to print the following output. (6.5)

```
1
2 2
3 3 3
4 4 4 4
.....
```

Ans.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i, j, n;
cout<<"Enter the number of rows";
cin>>n;
for(i=1;i<=n; i++)
{
for(j=1; j <=i; j++)
{
cout<<i;
}
cout<<"\n";
}
getch();
}
```

Q.3. (a) Write a function for multiplication of two matrices. (6)

Ans.

```
void multiplication(int a[][10],int b[][10],int mult[][10],int r1,int c1,int r2,int c2)
{
  int i,j,k;
/* Initializing elements of matrix mult to 0.*/
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  {
    mult[i][j]=0;
  }
/* Multiplying matrix a and b and storing in array mult. */
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  for(k=0; k<c1; ++k)
  {
    mult[i][j]+=a[i][k]*b[k][j];
  }
}
void display(int mult[][10], int r1, int c2)
{
```

```
int i, j;
cout << endl << "Output Matrix: " << endl;
for(i=0; i<r1; ++i)
for(j=0; j<c2; ++j)
{
  cout << " " << mult[i][j];
  if(j==c2-1)
    cout << endl;
}
}
```

Q.3. (b) Explain various object oriented features. **(6.5)**

Ans. Features of OOPs :-

Object: This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class: When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction: Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation: Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance: One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism: The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading: The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

UNIT-II

Q.4. (a) What is a friend function? Write its merits and demerits. **(6)**

Ans. A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

```
class Point
{
private:
  int x, y;
public:
```

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class class_name
{
  ..... .... ........
  friend return_type function_name(argument/s);
  ..... .... ........
}
```

Merits:

1. We can able to access the other class members in our class if, we use friend keyword.
2. We can access the members without inheriting the class.

Demerits:

1.Maximum size of the memory will occupied by objects according to the size of friend members.

2.We can't do any run time ploymorphism concepts in those members.

Q.4.(b) Write a program that will add two time values measured in terms of hours and minutes. (6.5)

Ans.

```
#include <iostream.h>
using namespace std;
class time {
private:
  int hr;
  int minute;
public:
  void get();
  void disp();
  class time add(class time t1, class time t2);
};
void time:: get(){
  cout << "Enter Hours: " ;
  cin >> hr;
  cout << endl;
  cout << "Enter Minutes:" ;
  cin >> minute;
  cout << endl;
}
void time:: disp(){
  cout << "Hours ------> " << hr << "hrs" << endl;

  cout << "Minutes ----> " << minute <<"mins"<< endl;
}
class time time:: add(class time rt1, class time rt2)
{
```

```
    class time rt3;
    rt3.minute = rt1.minute + rt2.minute;
    rt3.hr = rt1.hr + rt2.hr;
      if(rt3.minute >= 60){
      rt3.hr = rt3.hr + ((rt3.minute)/60);
      rt3.minute = rt3.minute % 60;
    }
    return rt3;
}
int main(){
    class time t1,t2,t3,t;
    t1.get();
    t2.get();
    t1.disp();
    cout<<"\n";
    t2.disp();
    cout<<"\n";
    cout << "Sum of times:"<< endl << endl;
    t3 = t.add(t1,t2);
    t3.disp();
}
```

Q.5.(a) What is a copy constructor? Explain with suitable example. **(6)**

Ans. The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
  // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object.

Following is a simple example of copy constructor.

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

class Point
{
private:
   int x, y;
public:
```

```
    Point(int x1, int y1) { x = x1; y = y1; }
    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()      { return x; }
    int getY()      { return y; }
};

int main()
{
  Point p1(10, 15); // Normal constructor is called here
  Point p2 = p1; // Copy constructor is called here

  // Let us access values assigned by constructors
  cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
  cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

  return 0;
}
```

Q.5. (b) Write a program that overloads and unary minus operator. (6

Ans.

```
#include <iostream.h>
using namespace std;
class Distance
{
  private:
    int feet;          // 0 to infinite
    int inches;        // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance
    void displayDistance()
    {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
    // overloaded minus (-) operator
    Distance operator- ()
    {
```

```
    feet = -feet;
    inches = -inches;
    return Distance(feet, inches);
  }
};
int main()
{
  Distance D1(11, 10), D2(-5, 11);

  -D1;              // apply negation
  D1.displayDistance();  // display D1

  -D2;              // apply negation
  D2.displayDistance();  // display D2

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

F: -11 I:-10

F: 5 I:-11

UNIT–III

Q.6. What do you mean by polymorphism? Explain various types of polymorphism with suitable example. (12.5)

Ans. Polymorphism: Polymorphism is the ability to use an operator or function in different ways. Polymorphism gives different meanings or functions to the operators or functions. Poly, referring too many, signifies the many uses of these operators and functions. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

Types of Polymorphism:

C++ provides two different types of polymorphism.

- run-time
- compile-time

Run-time:

The appropriate member function could be selected while the programming is running. This is known as run-time polymorphism. The *run-time* polymorphism is implemented with inheritance and virtual functions.

- Virtual functions

Virtual functions

A function qualified by the virtual keyword. When a virtual function is called via a pointer, the class of the object pointed to determines which function definition will be used. Virtual functions implement polymorphism, whereby objects belonging to different classes can respond to the same message in different ways.

Example of Virtual Function :

```
#include<iostream.h>
#include<conio.h>
```

```
class base
{
  public:
   virtual void show()
   {
        cout<<"\n  Base class show:";
   }
   void display()
   {
        cout<<"\n  Base class display:" ;
   }
};
class drive:public base
{
  public:
   void display()
   {
        cout<<"\n  Drive class display:";
   }
   void show()
   {
        cout<<"\n  Drive class show:";
   }
};
void main()
{
  clrscr();
  base obj1;
  base *p;
  cout<<"\n\t P points to base:\n" ;

  p=&obj1;
  p->display();
  p->show();
  cout<<"\n\n\t P points to drive:\n";
  drive obj2;
  p=&obj2;
  p->display();
  p->show();
  getch();
}
```

Output:

P points to Base

Base class display

Base class show

P points to Drive

Base class Display
Drive class Show

Compile-time:

The compiler is able to select the appropriate function for a particular call at compile-time itself. This is known as compile-time polymorphism. The *compile-time* polymorphism is implemented with templates.

- Function name overloading
- Operator overloading

1. Operator Overloading: The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

The general form of an operator function is:

```
return type classname: : operator(op-arglist)
{
   Function body
}
```

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op () in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.

2. Function Overloading: Using a single function name to perform different types of tasks is known as function overloading.Using the concept of function overloading, design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

Example of Function Overloading :

```
#include <iostream.h>
using namespace std;
class printData
{
  public:
    void print(int i) {
     cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
     cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
     cout << "Printing character: " << c << endl;
    }
};
```

```
int main(void)
{
  printData pd;
  // Call print to print integer
  pd.print(5);
  // Call print to print float
  pd.print(500.263);
  // Call print to print character
  pd.print("Hello C++");
  return 0;
}
```

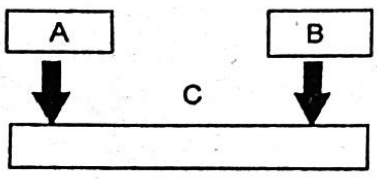
Q.7.(a) Write a template function that finds maximum value in an ar which is passed to it as an argument.

```
#include<iostream.h>
template<class t>
void maxarr(t a[])
{
t i,max;
max=a[0];
for(i=0;i<5;i++)
{
if(a[i]>max)
max=a[i];
}
cout<<max;
}
void main()
{
int a[5]={10,20,30,40,50};
char b[5]={'a','b','c','d','e'};
maxarr(a);
maxarr(b);
}
```

Q.7. (b) Explain multiple inheritance with suitable example. (t

Ans. Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more th two base classes.




```
#include<iostream.h>
#include<conio.h>
```

```
class student
{
  protected:
    int rno,m1,m2;
  public:
        void get()
        {
                cout<<"Enter the Roll no :";
                cin>>rno;
                cout<<"Enter the two marks  :";
                cin>>m1>>m2;
        }
};
class sports
{
  protected:
    int sm;              // sm = Sports mark
  public:
         void getsm()
        {
          cout<<"\nEnter the sports mark :";
          cin>>sm;

        }
};
class statement:public student,public sports
{
  int tot,avg;
  public:
  void display()
        {
          tot=(m1+m2+sm);
          avg=tot/3;
          cout<<"\n\n\tRoll No   : "<<rno<<"\n\tTotal    : "<<tot;
        cout<<"\n\tAverage   : "<<avg;
        }
};
void main()
{
  clrscr();
  statement obj;
  obj.get();
  obj.getsm();
  obj.display();
  getch();
}
```

In this example class statement is derived from the class student and class sp... So there is multiple inheritance.

UNIT–IV

Q.8. (a) How exception handling is performed in C++?

Ans. An exception is a problem that arises during the execution of a program. A ... exception is a response to an exceptional circumstance that arises while a progra... running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to anoth... C++ exception handling is built upon three keywords: try, catch, and throw.

- **throw:** A program throws an exception when a problem shows up. This is d... using a throw keyword.
- **catch:** A program catches an exception with an exception handler at the plac... a program where you want to handle the problem. The catch keyword indicates t... catching of an exception.
- **try:** A try block identifies a block of code for which particular exceptions will... activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using ... combination of the try and catch keywords. A try/catch block is placed around the c... that might generate an exception. Code within a try/catch block is referred to as protect... code, and the syntax for using try/catch looks like the following:

```
try
{
  // protected code
}catch( ExceptionName e1 )
{
  // catch block
}catch( ExceptionName e2 )
{
  // catch block
}catch( ExceptionName eN )
{
  // catch block
}
```

Q.8. (b) Write a C++ program that displays the content of a file.

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
    clrscr();
    ifstream ifile;
    char s[100], fname[20];
    cout<<"Enter file name to read and display its content (like file.txt) : ";
    cin>>fname;
```

```
ifile.open(fname);
if(!ifile)
{
     cout<<"Error in opening file..!!";
     getch();
     exit(0);
}
while(ifile.eof()==0)
{
 ifile>>s;
 cout<<s<<" ";
}
     cout<<"\n";
     ifile.close();
     getch();
}
```

Q.9. Write a program that reads Guru Gobind Singh Indraprastha University from the keboard into five separate string objects and then concatenates them into a new string using append() function. (12.5)

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
void main()
{
char str1[20],str2[20], str3[20], str4[20], str5[20];
char append(char[],char [],char [],char [],char []);
cout<<"Enter 5 strings";
gets(str1);
gets(str2);
gets(str3);
gets(str4);
gets(str5);
append(str1, str2, str3, str4, str5);
getch();
}
char append(char str1[],char str2[], char str3[], char str4[], char str5[])
{
char cnct[100];
int i,j;
for(i=0; str1[i]!='\0'; i++)
cnct[i]=str1[i];
cnct[i]=NULL;
for(i=0;cnct[i]!='\0'; i++);
for(j=0; str2[j]!='\0'; j++)
```

```
    {
    cnct[i]=str2[j];
    i++;
    }
    cnct[i]=NULL;
    for(i=0;cnct[i]!='\0'; i++);
    for(j=0;str3[j]!='\0'; j++)
    {
    cnct[i]=str3[j];
    i++;
    }
    cnct[i]=NULL;
    for(i=0; cnct[i]!='\0'; i++);
    for(j=0; str4[j]!='\0'; j++)
    {
    cnct[i]=str4[j];
    i++;
    }
    cnct[i]=NULL;
    for(i=0; cnct[i]!='\0'; i++);
    for(j=0;str5[j]!='\0';j++)
    {
    cnct[i]=str5[j];
    i++;
    }
    cnct[i]=NULL;
    puts(cnct);
    return(0);
    }
```

FIRST TERM EXAMINATION [FEB. 2017]
FOURTH SEMESTER [B.TECH.]
OBJECT ORITENTED PROGRAMMING USING C++ [ETCS–210]

Time : 1½ hrs. M.M. : 30

Note: *Q.No. 1 is compulsory. Attempt any two more question from the rest.*

Q.1. (a) State the characteristics of Object Oritented Programming. (2)

Ans. Class definitions: Basic building blocks OOP and a single entity which has data and operations on data together.

Objects: The instances of a class which are used in real functionality-its variables and operations.

Abstraction: Specifying what to do but not how to do, a flexible feature for having a overall view of an object's functionality.

Encapsulation: Binding data and operations of data together in a single unit.

Inheritance and class hierarchy: Reusability and extension of existing classes.

Polymorphism: Multiple definitions for a single name-functions with same name with different functionality.

Message passing: Objects communicates through invoking methods and sending data to them. This feature of sending and receiving information among objects through function parameters is known as Message Passing.

Q.1. (b) What is the difference between reference variables and normal variable? (2)

Ans. Normal variables carry the specified data where as the pointer variable carried the address of the specified data, *e.g.* if we give int x =10; ptr p = *x; Where x is the normal variable carries 10 and pointer variable is p which carried address of the integer variable x. Normal variable contains the value of variable either int or float whereas pointer variable contains the address of another variable. "Pointer variable holds memory address or physical address of any variable value means it indirectly points any variable, we can perform all operations +,*,–,and / through it but normal variable directly points variable value".

Q.1. (c) What are empty classes? Can instances of empty class be created? (2)

Ans. We can declare empty classes, but objects of such types still have nonzero size. The memory allocated for such objects is of nonzero size; therefore, the objects have different addresses. Having different addresses makes it possible to compare pointers to objects for identity. Also, in arrays, each member array must have a distinct address. An empty base class typically contributes zero bytes to the size of a derived class.

Q.1. (d) What are the differences between default and parameterized constructors? (2)

Ans. Default Constructor :

A constructor that has no parameter is called the default constructor. The default constructor for class A is A::A(), If no such constructor is defined, then the compiler supplies a default constructor.Therefore a statement such as

A a;

Invokes the default constructor of the compiler to create the object a.

Parameterized constructor :

The constructor that can take arguments are called the parameterized constructors. In the parameterized constructors we passed the arguments when the object of class are created.The constructor integer() may be modified to take arguments as shown below.

```
class integer
{
int m,n;
public:
integer(int x,int y)                    //parameterized constructor
{
m=x;
n=y;
}
};
main()
{
integer int(100, 200);                  //invoke the parameterized constructor
}
```

Q.1. (e) Define const member function and const objects. (2)

Ans. const member function

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows :

```
void mul(int, int) const;
```

The qualifier const is appended to the function prototypes. The compiler will generate an error message if such function try to alter the data values.

const Objects

We may create and use constant objects using const keyword before object declaration. For example, we may create X as a constant object of the class matrix as follows :

```
Const matrix X(m, n);                   //object X is constant
```

Any attempt to modify the values of m and n will generate compile time error.Further a constant object can call only const member function.

Q.2. (a) Declare a class to represent Time. Time is expressed in hours, minutes and seconds. Time class includes member functions to input and display time. Use constructor and destructor is program. (5)

Ans.

```
#include<iostream.h>
#include<conio.h>
class time
{
int hour;
int min;
int sec;
public:
time(int h,int m,int s)          //parameterized constructor declaration
```

```
{
hour=h;
min=m;
sec=s;
}
~time()                    //destructor declaration
{
cout<<"Destructor invoked";
}
void display()             //display function
{
cout<<"Time is "<<hour<<" Hour:"<<min<<" Minutes:"<<sec<<" Seconds"<<"\n";
}
};
void main()
{
time t1(5,35,20);          //constructor invoked
clrscr();
t1.display();
getch();
}                          //destructor invoked
```

Q.2. (b) With the help of an example, explain how an object can be passed and returned from a member function of a class. (5)

Ans.

```
#include<iostream.h>
#include<conio.h>
class complex
{
float x;
float y;
public:
void input(float real, float imag)
{
x=real;
y=imag;
}
friend complex sum(complex, complex); //declaration with objects as arguments
void show(complex);
};
complex sum(complex c1, complex c2)   //c1, c2 are the objects
{
complex c3;                           //object c3 is created
```

```
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return(c3);          // returns object c3
}
void complex :: show(complex c)
{
cout<<c.x<<"+j"<<c.y<<"\n";
}
void main()
{
complex a,b,c;
clrscr();
a.input(3.1, 5.65);
b.input(2.75, 1.2);
c=sum(a,b);                          //c+a+b
cout<<"a= "; a.show(a);
cout<<"b= "; b.show(b);
cout<<"c= "; c.show(c);
getch();
}
```

Q.3. (a) What is function overloading? Write overloaded functions for computing area of a circle a square and a rectangle. Also discuss ambiguity in these functions. **(5)**

Ans. Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Program to calculate the area of circle, square and triangle :

```
#include<iostream.h>
#include<conio.h>
const float pi=3.14;
float area (float n, float b, float h)
{
float ar;
ar=n*b*h;
return ar;
}
float area (float r)
{
float ar;
ar=pi*r*r;
```

```
return ar;
}
float area (float s)
{
float ar;
ar=s*s;
return ar;
}
void main()
{
float b,h,r,l;
float result;
clrscr();,
cout<<"\nEnter the Base & Height of Triangle;\n";
cin>>b>>h;
result=area(0.5,b,h);
cout<<"\nArea of Triangle: "<<result<<endl;
cout<<"\nEnter the Radius of Circle: \n";
cin>>r;
result=area(r);
cout<<"\nArea of Circle: "<<result<<endl;
cout<<"\nEnter the Side of Square: \n";
cin>>s;
result=area(s);
cout<<"\nArea of Square: "<<result<<endl;
getch();
}
```

Q.3. (*b*) Discuss the memory requirement, data members, member functions, static and non-static data members. **(5)**

Ans. During runtime, however... in C++, classes define types but (unless you activate RTTI which allows limited introspection into classes) don't generally occupy any memory themselves 1—they're just the frameworks for the construction and destruction of objects. Their *methods,* however—the constructors, destructors, instance methods, and class methods, occupy some portion of executable memory, but compilers can and do optimize away any such methods that are not used in the program.

Instances of types (that is, objects as well as primitives like int variables) occupy the bulk of memory in C++, but for their member functions they refer back to their classes. Exactly how much memory an instance of a particular class uses is entirely and utterly an implementation detail, and you should generally not need to care about it.

The C++ standard doesn't explicitly state when static memory is allocated, as long as it is available on first use. That said, it is most likely allocated during program initialization, thus guaranteeing its presence as soon as it is required, without needing special-case code to detect and perform allocation on access.

Q.4 Write short notes on the following:

Q.4. (a) Copy Constructor (2)

Ans. The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to :

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here "

```
classname (const classname &obj) {
  // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object.

Q.4. (b) Static Members. (2)

Ans. We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Q.4. (c) Inline Functions. (2)

Ans. C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Q.4. (d) Data Hiding. (2)

Ans. Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private.

Q.4. (e) Scope Resolution Operator.

Ans. Scope resolution operator in C++

✓ In C++ language the scope resolution operator is written ": :".

✓ C++ supports to the global variable from a function, Local variable is to define the same function name.

✓ Identify variables with use of scope resolution operator when we use the same name for local variable and global variable (or in class or namespace)

✓ Resolution operator is placed between the front of the variable name then the global variable is affected.If no resolution operator is placed between the local variable is affected.

END TERM EXAMINATION [MAY-JUNE 2017]
FOURTH SEMESTER [B.TECH.]
OBJECT ORITENTED PROGRAMMING USING C++ [ETCS–210]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q.No.1 which is compulsory.*

Q.1. Attempt the following:

Q.1. (a) Differentiate between pointer and reference variables. (2.5)

Ans. Difference Between Pointer & Reference Variable :

A pointer is a variable which stores the address of another variable.

A reference is a variable which refers to another variable.

For example :-

```
int i = 3;
int *ptr = &i;
int &ref = i;
```

The first line simply defines a variable. The second defines a pointer to that variable's memory address. The third defines a reference to the first variable.

Not only are the operators different, but you use the differently as well. With pointers must use the * operator to dereference it. With a reference no operator is required. It is understood that you are intending to work with the referred variable.

The following two lines will both change the value of i to 13.

```
*ptr = 13;
ref = 13;
```

Q.1. (b) What are empty classes? Can instances of empty class be created? (2.5)

Ans. Refer to Q.1. (c) First Term Examination 2017.

Q.1. (c) Differentiate between default and parameterized constructors. (2.5)

Ans. Default Constructor :

A constructor that has no parameter is called the default constructor. The default constructor for **class A is A::A()**, If no such constructor is defined, then the compiler supplies a default constructor.Therefore a statement such as

```
A a;
```

Invokes the default constructor of the compiler to create the object a.

Parameterized constructor :

The constructor that can take arguments are called the parameterized constructors. In the parameterized constructors we passed the arguments when the object of class are created.The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
int m,n;
public:
integer(int x,int y)                    //parameterized constructor
```

```
{
m=x;
n=y;
}
};
main()
{
integer int(100, 200);          //invoke the parameterized constructor
}
```

Q.1. (d) What is Garbage Collection in C++. **(2.5)**

Ans. Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and can thus have significant influence on performance.

In computer science, **garbage collection** (GC) is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Resources other than memory, such as network sockets, database handles, user interaction windows, and file and device descriptors, are not typically handled by garbage collection. Methods used to manage such resources, particularly destructors, may suffice to manage memory as well, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming.

Q.1. (e) Why are virtual functions used? **(2.5)**

Ans. Use of Virtual Function:

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raise Salary()*, *transfer()*, *promote()* etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

Q.1. (f) What is containership? Explain with an example. **(2.5)**

Ans. When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax for the declaration of another class is:

```
Class class_name1
{
_____
_____
};

Class class_name2
{
_____
_____
};

Class class_name3
{
Class_name1 obj1;          // object of class_name1
Class_name2 obj2;          // object of class_name2
_____
_____
};
```

Q.1. (g) Define static objects with example. (2,5)

Ans. Static Object :

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized using constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc
{
 int i;
 public:
 Abc()
 {
  i=0;
  cout << "constructor";
 }
 ~Abc()
 {
  cout << "destructor";
 }
};

void f()
{
static Abc obj;
}
```

```
int main()
{
int x=0;
if(x==0)
{
f();
}
cout << "END";
}
```

Output :
constructor END destructor

You must be thinking, why was destructor not called upon the end of the scope of if condition. This is because object was static, which has scope till the program lifetime, hence destructor for this object was called when main() exits.

Q.1. (*h*) How constructors and destructors are executed in multilevel inheritance. **(2.5)**

Ans. C++ constructor call order will be from top to down that is from base class to derived class and c++ destructor call order will be in reverse order.

Below is the example of constructor destructor call order for multi-level inheritance, in which Device class is the base class and Mobile class is derived from Device base class and then Android class is derived from Mobile class as a base class.

When we create the object of Android class, order of invocation of constructor destructor will be as below

Constructor : Device
Constructor : Mobile
Constructor : Android
Constructor : Android
Constructor : Mobile
Constructor : Device

Q.1. (*i*) Define Reusability, how C++ supports Reusability? **(2.5)**

Ans. Reusability :

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or subclass. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Q.1. (*j*) Differentiate function overloading and function overriding. **(2.5)**

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
Functions name and signatures must be same.	Having same Function name with different **Signatures**.
Overriding is the concept of **runtime polymorphism**	Overloading is the concept of **compile time polymorphism**
When a function of base class is re-defined in the derived class called as Overriding	Two functions having same name and return type but with different type and/or number of arguments is called as Overloading
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types
Function should be public.	Function can be different access specifies

Q.2. (a) Explain the characteristic of Object-oriented language, with appropriate examples. (8)

Ans. Characteristics of OOPs:

Class

Here we can take **Human Being** as a class. A class is a blueprint for any functional entity which defines its properties and its functions. Like Human Being, having body parts, and performing various actions.

Inheritance

Considering HumanBeing a class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc. Male and Female are also classes, but most of the properties and functions are included in HumanBeing, hence they can inherit everything from class HumanBeing using the concept of **Inheritance**.

Objects

My name is Abhishek, and I am an **instance/object** of class Male. When we say, Human Being, Male or Female, we just mean a kind, you, your friend, me we are the forms of these classes. We have a physical existence while a class is just a logical definition. We are the objects.

Abstraction

Abstraction means, showcasing only the required things to the outside world while hiding the details. Continuing our example, **Human Being's** can talk, walk, hear, eat, but the details are hidden from the outside world. We can take our skin as the Abstraction factor in our case, hiding the inside mechanism.

Encapsulation

This concept is a little tricky to explain with our example. Our Legs are binded to help us walk. Our hands, help us hold things. This binding of the properties to functions is called Encapsulation.

Polymorphism

Polymorphism is a concept, which allows us to redefine the way something works, by either changing how it is done or by changing the parts using which it is done. Both the ways have different terms for them.

If we walk using our hands, and not legs, here we will change the parts used to perform something. Hence this is called **Overloading**.

And if there is a defined way of walking, but I wish to walk differently, but using my legs, like everyone else. Then I can walk like I want, this will be called as **Overriding**.

Q.2. (b) Explain the use of copy constructor with example program. (4.5)

Ans. A The copy constructor is a special kind of constructor which creates a new object which is a copy of an existing one, and does it efficiently.The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it. The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it from scratch. A copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

Example of copy constructor.

```
#include<iostream>
class Point
{
private:
  int x, y;
public:
  Point(int x1, int y1) { x = x1; y = y1; }

  // Copy constructor
  Point(const Point &p2) {x = p2.x; y = p2.y; }
  int getX()      { return x; }
  int getY()      { return y; }
};
int main()
{
  Point p1(10, 15); // Normal constructor is called here
  Point p2 = p1; // Copy constructor is called here
  cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
  cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
  return 0;
}
```

Q.3. (a) Write a program to show the use of friend function and friend class. (7.5)

Ans. Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

```
class Node
{
private:
int key;
Node *next;
/* Other members of Node Class*/
friend class LinkedList; //Now class LinkedList can
// access private members of Node
};
```

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

```
public:
void showB(B&);
};
class B
private:
int b;
public:
B() {b = 0;}
friend void A:: shows B(B& x); // Friend Function
};
void A:: shows B(B &x)
{
//Since show() is friend of B, it can
// access private members of B
std:: cout <<"B::b ="<<x. b;
}
int main()
{
A a;
B x;
a. shows B(x);
return 0;
}
Run on IDE
Output:
B::b = 0
```

Q.3. (b) What are Destructors? Write a program to show the order in which local objects are destructed. (5)

Ans. The Class Destructor

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class perfixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing before coming out of the program like closing files, releasing memories etc.

Example :

```
class A
{
public:
~ A(); // This is the destructor
};
```

Order in which objects are Destructed:

C++ constructor call order will be from top to down that is from base class to derived class and c++ destructor call order will be in resverse order.

Below is the example of constructor destructor call order for multi-level inheritance, in which Device class is the base class and Mobile class is derived from Deivice base class and then Android class is derived from Mobile class as a base class.

```
#include
```

```
using namespace std;
//base class
class Device{
public:
Device(){cout<<"Constructor: Device\n";}
~Device (){cout<<" Destructor : Device\n";}
};
//derived class
class Mobile: public Device{
public
Mobile(){cout<<"Constructor: Mobile\n";}
~Mobile(){cout<<"Destructor : Mobile\n";}
};
//derived class
class Android: public Mobile {
public:
Android(){cout<<"Constructor : Android\n";}
~Android(){cout<<"Destructor : Android\n";}
};
//----------------------TEST----------------------
int main ()
{
Android _android;
// create the object that will call required constructors
return 0;
}
```

When we create the object of Android class, oder of invocation of constructor and destructor will be as below:

```
Constructor : Device
Constructor : Mobile
Constructor : Android
Destructor : Android
Destructor : Mobile
Destructor : Device
```

Q.4. (a) Create a class, which keeps track of the number of its instances, use static data member, constructors and destructors to maintain updated information about active objects. **(7.5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
class man
{
static int no;
char name;
int age;
public:
man()
{
no++;
cout<<"\n Number of objects exists:"<<no;
}
```

```
~ man ()
{
--no;
cout<<"\ n Number of objects exists:"<<no;
}
};
int man : : no = 0;
void main ()
{
clrscr ();
man A, B, C;
cout <<"\n Press any key to destroy object";
getch ();
}
```

OUTPUT

```
Number of Objects exists: 1
Number of Objects exists: 2
Number of Objects exists: 3
Press any key to destory object
Number of objects exists: 2
Number of objects exists: 1
Number of objects exists: 0
```

Explanation : In this program, the class man has one static data member no. The static data member is initialized to zero. Only one copy of static data member is created and all objects share the same copy of static data memory.

In function main(), objects A, B, and C are declared. When objects are declared, constructor is executed and static data member no is increased with one. The constructor also displays the value of no on the screen. The value of static member shows us the number of objects present. When the user presses a key, destructor is executed, which destroys the object. The value of static variable shows the number of existing objects.

Q.4. (*b*) How to achieve dynamic memory allocation in C++? Explain with a program. (5)

Ans. Dynamic memory allocation:

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack**.

new operator:

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator: To allocate memory of any data type, the syntax is:

- pointer-variable = **new** data-type;

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
#include<iostream.h>
    #include<conio.h>

    void main()
    {
```

```
int size,i;
int *ptr;

cout<<"\n\tEnter size of Array : ";
cin>>size;

ptr = new int[size];
//Creating memory at run-time and return first byte of address to ptr.

for(i=0;i<5;i++)      //Input arrray from user.
{
      cout<<"\nEnter any number : ";
      cin>>ptr[i];
}

for(i=0;i<5;i++)      //Output arrray to console.
cout<<ptr[i]<<", ";

delete[] ptr;
//deallocating all the memory created by new operator
}
```

Output :

```
Enter size of Array : 5

Enter any number : 78
Enter any number : 45
Enter any number : 12
Enter any number : 89
Enter any number : 56

78, 45, 12, 89, 56,
```

Q.5. (a) How base class member functions can be involved in a derived class if the derived class also has member function with the same name? Explain with example. (8.5)

Ans. A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as virtual and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class. The following example shows a base class that provides an implementation of the PrintBalance function and two derived classes

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
```

```
class Account {
public:
   Account( double d) {_balance = d;}
   virtual double GetBalance( ) {return _balance;}
   virtual void PrintBalance( ) {cerr << "Error. Balance not available for base type"
 <<endl;}
private:
   }; double _balance;
class CheckingAccount : public Account
{ public:
   CheckingAccount(double d) : Account(d) {}
   void PrintBalance( ) { cout << "Checking account balance:" << GetBalance( )
endl;}
};
class SavingsAccount : public Account {
public:
   SavingsAccount( double d) : Account(d) {}
   void PrintBalance( ) { cout << "Savings account balance: " << GetBalance( )
};
int main() {
   // Create objects of type CheckingAccount and SavingsAccount.
   CheckingAccount *pChecking = new CheckingAccount( 100.00 ) ;
   SavingsAccount *pSavings = new SavingsAccount( 1000.00 );
   // Call PrintBalance using a pointer to Account.
   Account *pAccount = pChecking;
   pAccount->PrintBalance();
   // Call PrintBalance using a pointer to Account.
   pAccount = pSavings;
   pAccount->PrintBalance();
   }
```

Q.5. (*b*) Differentiate public, protected and private access specifiers.

Ans. Access specifiers defines the access rights for the statements or functions that follows it until another access specifier or till the end of a class. The three types of access specifiers are "private", "public", "protected".

private:

The members declared as "private" can be accessed only within the same class, not from outside the class.

```
class PrivateAccess
{
private:  // private access specifier
int x;        // Data Member Declaration
```

```
void display();  // Member Function decaration
}
```

public:

The members declared as "public" are accessible within the class as well as from outside the class.

```
class PublicAccess
{
public:  // public access specifier
int x;        // Data Member Declaration
void display();  // Member Function decaration
}
```

protected:

The members declared as "protected" cannot be accessed from outside the class, but can be accessed from a derived class. This is used when inheritance is applied to the members of a class.

```
class ProtectedAccess
{
protected:  // protected access specifier
int x;        // Data Member Declaration
void display();  // Member Function decaration
}
```

Q.6. (a) What is generic programming? Write its advantages? **(5)**

Ans. Generic Programming:

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes.

Advantages of generic programming :

• Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.

• Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.

• Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

•You can also use class templates to develop a set of typesafe classes.

Q.6. (b) What is the difference between C & C++. Show and explain the usage of new and delete keyword. **(7.5)**

Ans. Difference between C and C++.

C	C++
1. C is Procedural Language.	1. C++ is non Procedural i.e Object oriented Language.
2. No virtual Functions are present in C.	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	3. The concept of polymorphism is used Important Feature of OOPS.
4. Operator overloading is not possible in C.	4. Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.
7. Multiple Declaration of global variables are allowed.	7. Multiple Declaration of global varioables are not allowed.
8. In C. • scanf() Function used for Input. • printf() Function used for output.	8. In C++. • Cin>> Function used for Input. • Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	9. Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other functions	10. In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	11. C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
12. No inheritance is possible in C.	12. Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	13. In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	14. It support both built-in and user define data types.
15. In C, Exception Handling is not present.	15. In C++, Exception Handling is done with Try and Catch block.

New & delete Keywords :

new and delete operators are provided by C++ for runtime memory management. They are used for dynamic allocation and freeing of memory while a program is running.

The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.

Syntax of new is:

p_var = new type name;

where p_var is a previously declared pointer of type typename. typename can be any basic data type.

new can also create an array:

p_var = new type [size];

In this case, size specifies the length of one-dimensional array to create.

Example 1:

int *p; p=new int;

It allocates memory space for an integer variable. And allocated memory can be released using following statement,

delete p;

Example 2:

int *a; a = new int[100];

It creates a memory space for an array of 100 integers. And to release the memory following statement can be used,

delete []a;

Q.7. (a) What are Abstract classes? (3)

Ans. Abstract Class

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;
};
```

Q.7. (b) Write a program having student as an abstract class and create many derived classes such as engineering, science, medical, etc. from the student class. Create their objects and process them. (9.5)

Ans.

```
# include <isotream>
using namespace std;
class student
{
   protected:
      int roll-number;
   public:
      void get-number (int a)
      roll-number = a;
}
void put-number (void)
{
cout << "Roll. No.:" <<roll-number <<"\n";
}
};
class engineering: virtual public student
{
   proteced:
      float part 1, part 2;
   public:
      void get_marks (float x, float y)
```

```
    {
      part 1 = x;  part 2 = y;
      }
    void put_marks (void)
    {
      cout << "Marks obtained:" << "\n";
          <<"part 1 = "<<part 1<<"\n"
          <<"part 2 = "<<part 2<<"\n"
    }
  };
class medical : virtual public student
{
  protected:
    float part 1, part 2;
public:
    void get_mark (Heat x, float y)
  {
    part 1 = x; part 2 = y;
  }
  void put-marks (void)
  cout <<"marks obtained :" << "\n"
    <<"part 1 =" << part 1 << "\n"
    <<"part 2 = << part << "\n"
```

Q.8. (*a*) What are exceptions? How reliability is affected by exception handling? (4)

Ans. An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Q.8. (*b*) Write an interactive program to compute the square root of a number. The input value must be tested for validity. If it is negative, the user defined function my_sqrt() should raise an exception. (8.5)

Ans.

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

 double SqrtNumber(double num)
 {
     double lower_bound=0;
```

```
    double temp=0;              /* ek edited this line */

    int nCount = 50;

 while(nCount  != 0)
 {
     temp=(lower_bound+upper_bound)/2;
     if(temp*temp==num)
     {
          return temp;
     }
     else if(temp*temp > num)

     {
          upper_bound = temp;
     }
     else
     {
          lower_bound = temp;
     }
  nCount - -
 }
  return temp;
 }

 int main()
 {
 double num;
 cout<<"Enter the number\n";
 cin>>num;

 if(num < 0)
 {
 cout<<"Error: Negative number!";
 return 0;
 }

 cout<<"Square roots are: +"<<sqrtnum(num) and <<" and -"<<sqrtnum(num);
 return 0;
 }
```

FIRST TERM EXAMINATION [FEB. 2018]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 1.30 hrs. M.M. : 30

Note: *Q. 1. is compulsory. Attempt any two questions from rest.*

Q.1. (a) What is an Abstract Data Type? (2)

Ans. Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type.

Q. 1.(b) What is the difference between cout and printf()? (2)

Ans. • Both cout and printf are used for outflow of data i.e. output.

- cout is a output stream while printf is a function.
- cout requires header file iostream while printf requires header file stdio.h
- cout is used only by c++ while printf can be used by both c and c++.
- cout example : cout<<"Message"; while printf example : printf("%d",sum);
- cout doesn't require format specifier while C does require.

Q. 1. (c) Define wild pointer? (2)

Ans. Wild Pointer:-

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave badly.

```
int main()
{
int *p; /* wild pointer */
/* Some unknown memory location is being corrupted.
This should never be done. */
*p = 12;
}
```

Run on IDE

Please note that if a pointer p points to a known variable then it's not a wild pointer. In the below program, p is a wild pointer till this points to a.

```
int main()
{
int *p; /* wild pointer */
int a = 10;
p = &a; /* p is not a wild pointer now*/
*p = 12; /* This is fine. Value of a is changed */
}
```

Q. 1. (d) Differentiate between constructor and destructor? **(2)**

Ans.

Purpose	**Constructor** Constructor is used to **initialize the instance of a class.**	**Destructor** Destructor **destroys the objects** when they are no longer needed.
When Called	Constructor is Called when new instance of a class is created.	Destructor is called when instance of a class is deleted or released.
Memory Management Arguments	Constructor **allocates the memory.** Constructors can have arguments.	Destructor **releases the memory.** Destructor can not have any **arguments.**
Overloading	**Overloading** of constructor is possible.	Overloading of Destructor is not possible.
Name	Constructor has the same name as class name.	Destructor also has the same name as class name but with **(~) tiled operator.**

Syntex	ClassName(Arguments)
	{
	/ Body of Constructor
	}
		~ ClassName()
		{
		}

Q. 1. (e) What is Containership? **(2)**

Ans. When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax for the declaration of another class is:

```
Class class_name1
{
_____
_____
};

Class class_name2
{
_____
_____
};
```

```
Class class_name3
{
Class_name1 obj1;          // object of class_name1
Class_name2 obj2;          // object of class_name2
_____
_____
};
```

Q. 2. (a) What are difference between reference variable and normal variable? Why cannot a constant value be initialized to variable of reference type? (4)

Ans. Difference between reference variable and normal variable:-

1. References may not be null whereas normal variable may be null.
2. Normal variable contains the value of variable either int or float whereas pointer variable contains the address of the variable.
3. It is necessary to initialize the reference variable at the time of declaration. It is not necessary to initialize the normal variable at the time of declaration.

A constant value can not be initialize to a variable of reference type because reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

Q. 2. (b) Consider a class MyArray having pointer to integer as its data member. Its objects must appear like arrays but they must be dynamically resizable. Write a program to illustrate the use of constructors in my array object. (6)

Ans.

```
#include<iostream.h>
#include<conio.h>
class myarray
{
int *p;
int a;
public:
myarray()
{
cout<<"Enter value for a:-\n";
cin>>a;
p=&a;
}
void disp()
{
cout<<"Address of a = "<<p<<" and value of a ="<<*p<<"\n";
}
};
void main()
{
int n,i;
clrscr();
cout<<"How many objects you want to create:-\n";
cin>>n;
```

```
myarray *obj=new myarray[n];
for(i=0;i<n;i++)
{
obj->disp();
obj++;
}
getch();
}
```

Q. 3. (a) What are friend functions and friend classes? Write a normal function which adds object of the complex number class. Declare this normal function as friend of Complex class. **(6)**

Ans. Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

```
class Node
{
private:
 int key;
 Node *next;
 /* Other members of Node Class */
 friend class LinkedList; // Now class  LinkedList can
                          // access private members of Node
};
```

Program:-

```
/* C++ Program to Implement Complex Numbers using Classes
*/
#include <iostream>
using namespace std;
class Complex
{
  private:
    int real;
    int imag;
  public:
    Complex(int r = 0, int i = 0): real(r), imag(i) {};
    void setComplex(void)
    {
      cout << "Enter the real and imaginary parts :";
      cin >> this->real;
      cin >> this->imag;
    }
    Complex add(const Complex& c)
    {
      Complex comp;
      comp.real = this->real + c.real;
      comp.imag = this->imag + c.imag;
      return comp;
```

```
    }
    void printComplex(void)
    {
      cout << "Real :" << this->real << endl
        << "Imaginary :" << this->imag << endl;
    }
};
int main()
{
  Complex a, b, c, d;
   cout << "Setting first complex number" << endl;
  a.setComplex();
  cout << "Setting second complex number" << endl;
  b.setComplex();
  /* Adding two complex numbers */
  cout << "Addition of a and b:" << endl;
  c = a.add(b);
  c.printComplex();
}
```

Q. 3. (b) What are the programming paradigms of Object Oriented Programming? **(4)**

Ans. Object Oriented Paradigm:-

Object-Oriented Analysis

The primary tasks in object-oriented analysis (OOA) are–

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

Object-Oriented Design

The implementation details generally include–

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

Object-Oriented Programming

The important features of object–oriented programming are–

- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Q. 4. (a) Discuss memory requirements for classes, objects, data members, member functions, static and non-static data members. **(5)**

Ans. During runtime, however... in C++, classes define types but (unless you activate RTTI which allows limited introspection into classes) don't generally occupy any memory themselves 1—they're just the frameworks for the construction and destruction of objects.

Their *methods*, however—the constructors, destructors, instance methods, and class methods, occupy some portion of executable memory, but compilers can and do optimize away any such methods that are not used in the program.

Instances of types (that is, objects as well as primitives like int variables) occupy the bulk of memory in C++, but for their member functions they refer back to their classes. Exactly how much memory an instance of a particular class uses is entirely and utterly an implementation detail, and you should generally not need to care about it.

The C++ standard doesn't explicitly state when static memory is allocated, as long as it is available on first use. That said, it is most likely allocated during program initialization, thus guaranteeing its presence as soon as it is required, without needing special-case code to detect and perform allocation on access.

Q. 4. (b) What is this pointer? What is your reaction to this statement:
delete this;
Write a program demonstrating the use of this pointer. **(5)**

Ans. this pointer:-

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

```
class ClassName {
    private:
     int dataMember;
    public:
      method(int a) {
  // this pointer stores the address of object obj and access dataMember
         this->dataMember = a;
      }
}
int main() {
   ClassName obj;
   obj.method(5);
 }
```

"delete this" in C++

1. delete operator works only for objects allocated using operator new. If the object is created using new, then we can do delete this, otherwise behavior is undefined.

2. Once *delete this* is done, any member of the deleted object should not be accessed after deletion.

END TERM EXAMINATION [MAY-JUNE 2018]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 3 hrs. M.M. : 75

Note: Attempt any five question including Q. no. 1 which is compulsory.

Q.1. (a) What is copy constructor? (2.5)

Ans. Refer Q. 4. (a) of First Term Exam 2017.

Q. 1. (b) Differentiate between a class and a structure. (2.5)

Ans. In C++, the only difference between a struct and a class is that struct member are public by default, and class members are private by default.

However, as a matter of style, it's best to use the struct keyword for something that could reasonably be a struct in C (more or less POD types), and the class keyword if it uses C++- specific features such as inheritance and member functions.

The members and base classes of a struct are public by default, while in class, they default to private. Note: you should make your base classes *explicitly* public, private, or protected, rather than relying on the defaults.

A struct simply *feels* like an open pile of bits with very little in the way of encapsulation or functionality. A class *feels* like a living and responsible member of society with intelligent services, a strong encapsulation barrier, and a well defined interface. Since that's the connotation most people already have, you should probably use the struct keyword if you have a class that has very few methods and has public data (such things *do* exist in well designed systems!), but otherwise you should probably use the class keyword.

Q. 1. (c) What is the difference between call by value and call by reference for a function? (2.5)

Ans. Difference between Call by Value and Call by Reference

call by value	call by reference
In call by value, a copy of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.	In call by reference, the location (address) of actual arguments is passed to formal arguments of the called function. This means by accessing the addresses of actual arguments we can alter them within from the called function.
In call by value, actual arguments will remain safe, they cannot be modified accidentally.	In call by reference, alteration to actual arguments is possible within from called function; therefore the code must handle arguments carefully else you get unexpected results.

Q. 1. (d) What do you mean by namespace? (2.5)

Ans. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which name are defined. In eassence, a namespace defines a scope.

A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace namespace_name {
// code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
name::code; // code could be variable or function.
```

Q. 1. (e) Explain the concept of virtual class. **(2.5)**

Ans. An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.

C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

Q.1.(f) Can an empty class be created? If yes, what is the significance of it? **(2.5)**

Ans. C++ allows creating an Empty class, yes. We can declare an empty class and its object. The declaration of Empty class and its object are same as normal class and object declaration.

An Empty class's object will take only **one byte** in the memory; since class doesn't have any data member it will take minimum memory storage. **One byte is the minimum memory amount that could be occupied.**

Let's consider the following program

```
#include <iostream>
using namespace std;
class Example
{
};
int main()
{
        Example objEx;
        cout<<"Size of objEx is:"<<sizeof(objEx)<<endl;

        return 0;
}
```

Size of objEx is: 1

Here Example is an empty class which does not has any data member and member function, and objex is the object of class Example. See the output **"Size of objEx is: 1"**.

Q. 1. (g) What is the difference between function overloading and function overriding. **(2.5)**

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
(1) Functions name and signatures must be same.	(1) Having same Function name with different **Signatures.**
(2) Overriding is the concept **of runtime polymorphism**	(2) Overloading is the concept **of compile time polymorphism.**
(3) When a function of base class is re-defined in the derived class called as Overriding.	(3) Two functions having same name and return type, but with different type and/or number of arguments is called as, Overloading.
(4) It needs inheritance.	(4) It doesn't need inheritance.
(5) Functions should have same data type.	(5) Functions can have different data types
(6) Function should be public.	(6) Function can be different access specifies

Q. 1.(h) What is code reusability? Explain its different types. **(2.5)**

Ans. Reusability

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or subclass. A derived class includes all features of the generic base class.

Polymorphism and Inheritance are the types of code reusability.

Q. 1.(i) What do you mean by persistent objects? **(2.5)**

Ans. A persistent object can live after the program which created it has stopped. Persistent objects can even outlive different versions of the creating program, can outlive the disk system, the operating system, or even the hardware on which the OS was running when they were created.

The challenge with persistent objects is to effectively store their member function code out on secondary storage along with their data bits (and the data bits and member function code of all member objects, and of all their member objects and base classes, etc). This is non-trivial when you have to do it yourself. In C++, you have to do it yourself. C++/OO databases can help hide the mechanism for all this.

Q. 1. (j) What is exception handling? Explain its use. **(2.5)**

Ans. An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is runining, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.

- **Throw:** A program throws an exception when a problem shows up. This is done using throw keyword.

• **Catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

• **Try:** A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Q. 1. (k) How does a compile time polymorphism differ from run time polymorphism? **(2.5)**

Ans.

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by **function** overloading and **operator** overloading.	It is achieved by **virtual functions** and **pointers**.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

Q. 1. (*l*) What is a template class and how is it useful. **(2.5)**

Ans. Template is one of the feature added to C+ recently. A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array etc. Similarly we can define a template for a function say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.

Syntax for class template :

```
Template <class T>
Class classname
{
};
```

Syntax for function template :

```
Template <class T>
Returntype functionname (arguments of type T)
{
}
```

Q. 2. (a) What do mean by an inline function? Write a small program using inline function. **(6)**

Ans. C++ **Inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition *is* an inline function definition, even without the use of the **inline** specifier.

Example : Use of inline function to return max of two numbers:

```
#include <iostream>
inline int Max(int x, int y)
{
  return (x > y)? x : y;
int main ()
{
cout << "Max (20,10):" << Max (20,10) << endl;
cout << "Max (0,200):" << Max (0,200) << endl;
cout << "Max (100,1010):" << Max (100,1010) << endl;
return 0;
}
```

Q. 2. (b) Is function overloading a type of polymorphism? Write a program in C++ that computes area of a triangle. Square and a rectangle using this concept. **(6.5)**

Ans. Yes function overloading is the type of polymorphism.

Program to compute the area:-

```
using namespace std;
int area(int);
int area(int,int);
float_area(float);
float area(float,float);
int main()
{
    int s,l,b;
    float r,bs,ht;
    cout<<"Enter side of a square:";
    cin>>s;
    cout<<"Enter length and breadth of rectangle:";
    cin>>l>>b;
    cout<<"Enter radius of circle:";
    cin>>r;
    cout<<"Enter base and height of triangle:";
    cin>>bs>>ht;
    cout<<"Area of square is"<<area(s);
    cout<<"\nArea of rectangle is"<<area(l,b);
    cout<<"\nArea of circle is"<<area(r);
    cout<<"\nArea of triangle is"<<area(bs,ht);
```

```
}
int area(int s)
{
return(s*s);
}
int area(int l,int b)
{
return(l*b);
}
float area(float r)
{
return(3.14*r*r);
}
float area(float bs,float ht)
{
return((bs*ht)/2);
}
```

Sample Input

Enter side of a square:2

Enter length and breadth of rectangle:3 6

Enter radius of circle:3

Enter base and height of triangle:4 4

Sample Output

Area of square is 4

Area of rectangle is 18

Area of circle is 28.26

Area of triangle is 8

Q. 3. (a) What is dereferencing operator? Write a small program using pointer to members of a class and explain the use of this operator. (6)

Ans. In computer programming, a dereference operator, also known as an indirection operator, operates on a pointer variable, and returns the location-value, or l-value that it points to in memory. In the C programming language, the deference operator is denoted with an asterisk (*).

For example, in C, we can declare a variable x, that holds an integer value, and a variable p, which holds a pointer to an integer value in memory:

```
int x; int *p;
```

Here, the asterisk tells the compiler, "p is not an integer, but rather a pointer to a location in memory which holds an integer." Here, it is not a dereference, but part of a pointer declaration.

Pointer to Data Members of class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

datatype class_name :: *pointer_name ;

Syntax for Assignment :

pointer_name = &class_name :: datamember_name ;

Both declaration and assignment can be done in a single statement too.

datatype class_name::*pointer_name = &class_name::datamember_name ;

Lets take an example, to understand the complete concept.

```
class Data
{
public:
int a;
void print() { cout << "a is"<< a; }
};
int main()
{
Data d, *dp;
dp = &d;    // pointer to object
int Data::*ptr=&Data::a;  // pointer to data member 'a'
d.*ptr=10;
d.print();
dp->*ptr=20;
dp->print();
}
```

Output :

a is 10 a is 20

The syntax is very tough, hence they are only used under special circumstances.

Q. 3. (b) What is a friend function? Write any three features of it and explain its functionality using suitable example. Why friend functions should be avoided? **(6.5)**

Ans. A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows

```
class Temperature
{
  int celsius;
  public:
    Temperature()
    {
      celsius = 0;
```

```
    }
    friend int temp( Temperature )    //declaring friend function
};
```

Let's see an example:-

```
#include <iostream>
using namespace std;
class Temperature
{
        int celsius;
        public:
                Temperature()
                {
                        celsius = 0;
                }
                friend int temp( Temperature );  // declaring friend function
};
int temp( Temperature t )    // friend function definition
{
        t.celsius = 40;
        return t.celsius;
}
int main()
{
        Temperature tm;
        cout << "Temperature in celsius :" << temp( tm ) << endl;
        return 0;
}
```

Output

Here we declared a function 'temp' as the friend function of the class 'Temperature'. In the friend function, we directly accessed the private member **celsius** of the class 'Temperature'.

When the first statement of the main function created an object 'tm' of the class 'Temperature' thus calling its constructor and assigning a value 0 to its data member celsius.

The second statement of the main function called the function 'temp' which assigned a value 40 to celsius.

Features of Friend Function:-

1. It is called like normal functions in C or C++.

2. Private member can be accessed inside friend function using object name and dot(.) operator.

3. It can take multiple objects as parameter as required.

Q. 4. (a) What is parameterized constructor? Explain with suitable example. **(6)**

Ans. Default Constructor :

A constructor that has no parameter is called the default constructor. The default constructor for class A is A::A(), If no such constructor is defined, then the compiler supplies a default constructor.Therefore a statement such as

A a;

Invokes the default constructor of the compiler to create the object a.

Parameterized constructor :

The constructor that can take arguments are called the parameterized constructors. In the parameterized constructors we passed the arguments when the object of class are created.The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
int m,n;
public:
integer(int x,int y)        //parameterized constructor
{
m=x;
n=y;
}
};
main()
{
integer int(100, 200);   //invoke the parameterized constructor
}
```

Q. 4. (b) What do you mean by dynamic object initialization and how is it achieved? (6.5)

Ans. The Dynamic Initialization of Objects means to initialize the data members of the class while creating the object. When we want to provide initial or default values to the data members while creating of object - we need to use dynamic initialization of objects.

It can be implemented by using parameterized constructors in C++.

Example:

Here, we have a class named **"Student"** which contains two private data members 1) rNo - to store roll number and 2) perc - to store percentage.

Program:

```
#include <iostream>
using namespace std;
//structure definition with private and public members
struct Student
{
   private:
          int rNo;
          float perc;
          public:
```

```
        //constructor
        Student(int r, float p)
        {
            rNo = r;
            perc = p;
        }
        //function to read details
        void read(void)
        {
            cout<<"Enter roll number:";
            cin>>rNo;
            cout<<"Enter percentage:";
            cin>>perc;
        }
        //function to print details
        void print(void)
        {
            cout<<endl;
            cout<<"Roll number:"<<rNo<<endl;
            cout<<"Pecentage:" <<perc<<"%"<<endl;
        }
};
//Main code
int main()
{
    //reading roll number and percentage to initialize
    //the members while creating object
    cout<<"Enter roll number to initialize the object:";
    int roll_number;
    cin>>roll_number;
    cout<<"Enter percentage to initialize the object:";
    float percentage;
    cin>>percentage;
    //declaring and initialize the object
    struct Student std(roll_number, percentage);
    //print the value
    cout<<"After initializing the object, the values are...."<<endl;
    std.print();
    //reading and printing student details
    //by calling public member functions of the structure
    std.read();
    std.print();
```

```
    return 0;
}
```

Output

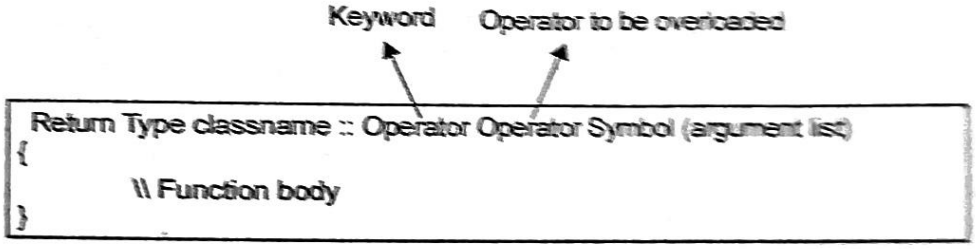
```
Enter roll number to initialize the object: 101
Enter percentage to initialize the object: 84.02
After initializing the object, the values are...
Roll number: 101
Pecentage: 84.02%
Enter roll number: 102
Enter percentage: 87.95
Roll number: 102
Pecentage: 87.95%
```

Q. 5. (a) What is operator overloading? Write a program that adds and subtract two complex number using operators overloading. (5)

Ans. Operator Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

Operator Overloading Syntax



C++ program to perform arithmetic operations on complex numbers using operator overloading:-

```
#include<iostream.h>
#include<conio.h>
class complex
{
      float x,y;
      public:
        complex() {}
        complex(float real,float img)
        {
            x=real; y=img;
        }
complex operator+(complex);
complex operator-(complex);
void display()
```

```
{
    cout<<x<<" +"<<y<<"i"<<endl;
}
};
complex complex::operator+(complex c)
{
    complex temp;
    temp.x=x+c.x;
    temp.y=y+c.y;
    return(temp);
}
complex complex::operator-(complex d)
{
    complex temp;
    temp.x=x-d.x;
    temp.y=y-d.y;
    return(temp);
}
void main()
{
   clrscr();
   complex c1(5,3),c2(3,2),c3=c1+c2,c4=c1-c2,
   c1.display();
   c2.display();
   cout<<"Addition"<<endl;
   c3.display();
   cout<<"Subtraction"<<endl;
   c4.display();
   getch();
}
```

OUTPUT:

```
5 + 3i
3 + 2i
Addition
8 + 5i
Subtraction
2 + 1i
```

Q. 5. (b) Is conversion from one class type to another class type possible in C++? If yes, how is it achived? **(6.5)**

Ans. Yes the conversion from one class type to another class type is possible.

In this type of conversion both the type that is source type and the destination type are of class type. Means the source type is of class type and the destination type is also of the class type. In other words, one class data type is converted into the another class type.

For example we have two classes one for "computer" and another for "mobile". Suppose if we wish to assign "price" of computer to mobile then it can be achieved by the statement below which is the example of the conversion from one class to another class type.

mob = comp ; // where mob and comp are the objects of mobile and computer classes respectively.

Here the assignment will be done by converting *"comp"* object which is of class type into the *"mob"* which is another class data type.

Conversion from one class to another class can be performed either by using the constructor or type conversion function.

Program to convert one class to another class:-

```
#include "iostream.h"
#include "conio.h"
#include "iomanip.h"
class Time
{
        int hrs,min;
        public:
                Time(int h,int m)
                {
                        hrs=h;
                        min=m;
                }
                Time()
                {
                        cout<<"\n Time's Object Created";
                }
                int getMinutes()
                {
                        int tot_min = ( hrs * 60 ) + min ;
              return tot_min;
                }
                void display()
                {
                        cout<<"Hours:"<<hrs<<endl ;
              cout<<" Minutes ::"<<min <<endl ;
                }
};
class Minute
{
      int min;
      public:
    Minute()
```

```
    {
      min = 0;
    }
    void operator=(Time T)
    {
            min=T.getMinutes();
    }
    void display()
    {
            cout<<"\n Total Minutes :" <<min<<endl;
    }
};
void main()
{
    clrscr();
    Time t1(2,30);
    t1.display();
    Minute m1;
    m1.display();
    m1 = t1;   // conversion from Time to Minute
    t1.display();
    m1.display();
    getch();
}
```

In the below program we have two classes *"Time"* and *"Minute"* respectively in which we have tried to convert from *Time* class to *Minute* class.

Q. 6. (a) Explain with suitable example that how a base class member function can be accessed in a derived class if derived class has also a member function with the same name as that of the base class member function. **(8)**

Ans. Refer Q. 5. (a) of End Term Exam 2017.

Q. 6. (b) Differentiate between different types of access specifiers in C++. **(4.5)**

Ans. Refer Q. 5. (b) of End Term Exam 2017.

Q. 7. (a) What is the role of this pointer? Explain with suitable example. **(6)**

Ans. Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is a constant pointer that holds the memory address of the current object. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

Example of this pointer

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Student
{
   int Roll;
   char Name[25];
   float Marks;
   public:
   Student(int R,float Mks,char Nm[])        //Constructor 1
   {
                    Roll = R;
          strcpy(Name,Nm);
               Marks = Mks;
   }
   Student(char Name[],float Marks,int Roll)    //Constructor 2
   {
                  Roll = Roll;
      strcpy(Name,Name);
            Marks = Marks;
   }
   Student(int Roll,char Name[],float Marks)    //Constructor 3
   {
          this->Roll = Roll;
   strcpy(this->Name,Name);
     this->Marks = Marks;
   }
   void Display()
   {
   cout<<"\n\tRoll :" <<Roll;
   cout<<"\n\tName :" <<Name;
   cout<<"\n\tMarks :" <<Marks;
   }
};
void main()
{
   Student S1(1,89.63, "Sumit");
   Student S2("Kumar",78.53,2);
   Student S3(3,"Gaurav",68.94);
   cout<<"\n\n\tDetails of Student 1 :";
   S1.Display();
```

```
        cout<<"\n\n\tDetails of Student 2 :";
        S2.Display();
        cout<<"\n\n\tDetails of Student 3 :";
        S3.Display();
    }
```

In constructor 1,variables declared in argument list different from variables declared as class data members. When compiler doesn't find Roll, Name, Marks as local variable, then, it will find Roll, Name, Marks in class scope and assign values to them.

But Constructor 2 will not initialize class data members. When we pass values to constructor 2, it will initialize values to itself local variables b'coz variables declared in argument list and variable declared as data members are of same name.

In this situation, we use 'this' pointer to differentiate local variable and class data members as shown in constructor 3.

Q. 7. (b) Write a C++ program that displays the content of a file at console. **(6.5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
    clrscr();
    ifstream ifile;
    char s[100], fname[20];
    cout<<"Enter file name to read and display its content (like file. txt):";
    cin>>fname;
    ifile. open(fname);
    if(!ifile)
    }
        cout<<"Error in opening file...!!";
        getch();
        exit(0);
    }
    while(ifile.eof()==0)
    }
    ifile>>s;
    cout<<s<<" ";
    }
        cout<<"\n";
        ifile.close();
        getch();
    }
```

Q. 8. (a) What do you mean by a template member function? Explain with suitable example. (6)

Ans. Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how they work

Function Template

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

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

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
inline T const& Max (T const& a, T const& b) {
  return a < b ? b:a;
}
int main () {
  int i = 39;
  int j = 20;
  cout << "Max(i, j):" << Max(i, j) << endl;
  double f1 = 13.5;
  double f2 = 20.7;
  cout << "Max(f1, f2):" << Max(f1, f2) << endl;
  string s1 = "Hello";
  string s2 = "World";
  cout << "Max(s1, s2):" << Max(s1, s2) << endl;
  return 0;
}
```

If we compile and run above code, this would produce the following result –

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Q. 8. (b) Explain with suitable example, the multiple catching in exception handing. **(6.5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
#void test(int x)
{
  try
  {
    if(x>0)
    throw x;
  else
    throw 'x';
  }

  catch(int x)
  {
    cout<<"Catch a integer and that integer is."<<x;
  }

  catch(char x)
  }
      cout<<"Catch a character and that character is."<<x;
  }
}

void main()
{
  clrscr();
  cout<<"Testing multiple catches \n:";
  test(10);
  test(0)
  getch();
}
```

FIRST TERM EXAMINATION [FEB. 2019]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 1.30 hrs. **M.M. : 30**

Note :- *Attempt in all three questions. Q. No. 1 which is compulsory. Assume, missing data if any.*

Q.1. Define the following terms related to object oriented proframming paradigm.

Q.1. (a) Encapsulation (2)

Ans. Encapsulation: This concept is a little tricky to explain with our example. Our Legs are binded to help us walk. Our hands, help us hold things. This binding of the properties to functions is called Encapsulation.

Q.1. (b) Abstract Data Type (2)

Ans. An **abstract data type** (or ADT) is a class that has a defined set of operations and values. In other words, you can create the starter motor as an entire abstract data type, protecting all of the inner code from the user. When the user wants to start the car, they can just execute the start() function. In programming, an ADT has the following features:

- An ADT doesn't state how data is organized, and
- It provides only what's needed to execute its operations

An ADT is a prime example of how you can make full use of data abstraction and data hiding. This means that an abstract data type is a huge component of object-oriented programming methodologies: enforcing abstraction, allowing data hiding (protecting information from other parts of the program), and encapsulation (combining elements into a single unit, such as a class).

Q.1. (c) Inheritance (2)

Ans. Inheritance: Considering Human Being a class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc. Male and Female are also classes, but most of the properties and functions are included in Human Being, hence they can inherit everything from class Human Being using the concept of **Inheritance**.

Q.1. (d) Polymorphism. (2)

Ans. Polymorphism: Polymorphism is a concept, which allows us to redefine the way something works, by either changing how it is done or by changing the parts using which it is done. Both the ways have different terms for them.

If we walk using our hands, and not legs, here we will change the parts used to perform something. Hence this is called **Overloading**.

And if there is a defined way of walking, but I wish to walk differently, but using my legs, like everyone else. Then I can walk like I want, this will be called as **Overriding**.

Q.1. (e) Containership and Genericity. (2)

Ans. Containership: We can create an object of one class into another and that object will be a member of the class. This type of relationship between classes is known as containership or has a relationship as one class contain the object of another class. And the class which contains the object and members of another class in this kind of relationship is called a container class.

The object that is part of another object is called contained object, whereas object that contains another object as its part or attribute is called container object.

Genericity: It is one of the most powerfull means for obtaining flexibility in programing with statically typed programming languages.

Q.2. (a) What is the difference between member functions defined inside and outside the body of a class? How are inline member function defined outside the body of a class. **(5)**

Ans. Member function inside the class: A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

Member function outside the class: As the name suggests, here the functions are defined outside the class however they are declared inside the class.

Functions should be declared inside the class to bound it to the class and indicate it as it's member but they can be defined outside of the class.

To define a function outside of a class, scope resolution operator :: is used.

Inline member function define outside the class : If we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
class Cube
{
  public:
  int side;
  int getVolume();
}
// member function defined outside class definition
int Cube :: getVolume()
{
  return side*side*side;
}
```

The main function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot (.) operator.

Q.2. (b) What are friend functions and friend classes? Write a normal function which adds objects of the complex number class. Declare this normal function as a friend of Complex class. **(5)**

Ans. Refer Q.3. (a) of First Term Examination 2018. (Pg. No. 4-2018)

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

(a) A method of another class

(b) A global function

```
class Node
{
private:
  int key;
  Node *next;

  /* Other members of Node Class */
```

```
friend int LinkedList::search(); // Only search() of linkedList
                                 // can access internal members );
```

Q.3.(a) Consider a class called MyArray having pointer to integers as its data member. Its objects must appear like arrays, but they must be dynamically re-sizable. Write a program to illustrate the use of constructors in MyArray class. (5)

Ans.

```
#include<iostream.h>
#include<conio.h>
class myarray
{
int *p;
int a;
public:
myarray()
{
cout<<"Enter value for a:-\n";
cin>>a;
p=&a;
}
void disp()
{
cout<<"Address of a = "<<p<<" and value of a =" <<*p<< " \n";
}
};
void main()
{
int n,i;
clrscr();
cout<<"How many objects you want to create:-\n";
cin>>n;
myarray *obj=new myarray[n];
for(i=0;i<n;i++)
{
obj->disp();
obj++;
}
getch();
}
```

Q.3. (b) Create a class, which keeps track of the number of its instances. Using static data member, constructors and destructors to maintain updated information about active objects. (5)

Ans. Refer Q.4. (a) of End Term Examination 2017. (pg. 15-2017)

Q.4. (a) What is name mangling and explain its needs? Is this transparent to user? (5)

Ans. Name Mangling: C++ supports function overloading, i.e., there can be more than one functions with same name and differences in parameters. How does C++ compiler distinguishes between different functions when it generates object code – it

changes names by adding information about arguments. This technique of adding additional information to function names is called Name Mangling. C++ standard doesn't specify any particular technique for name mangling, so different compilers may append different information to function names.

Here is an example of Name Mangling. The overloaded functions are named as func(), and another function my_function() is there.

Example

```
int func(int x) {
  return x*x;
}
double func(double x) {
  return x*x;
}
void my_function(void) {
  int x = func(2); //integer
  double y = func(2.58); //double
}
```

Q.4. (b) What is the difference between stack based and heap based objects. How memory allocation failure can be handled in C++? **(5)**

Ans. Stack Allocation : The allocation happens on contiguous blocks of memory. We call it stack memory allocation because the allocation happens in function call stack. The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. This all happens using some predefined routines in compiler. Programmer does not have to worry about memory allocation and deallocation of stack variables.

```
int main()
{
  // All these variables get memory
  // allocated on stack
  int a;
  int b[10];
  int n = 20;
  int c[n];
}
```

Heap Allocation : The memory is allocated during execution of instructions written by programmers. Note that the name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate. If a programmer does not handle this memory well, memory leak can happen in the program.

```
int main()
{
  // This memory for 10 integers
  // is allocated on heap.
  int *ptr  = new int[10];
}
```

END TERM EXAMINATION [MAY 2019]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 3 hrs. **M.M. : 75**

Note :- *Attempt any five questions Including Q. no. 1 which is compulsory.*

Q.1. Attempt the following:- **(2.5 × 10 = 25)**

Q.1. (a) Explain the advantages of new operator over malloc 0

Ans. Differences between new operator and malloc() function in C++ are:-

1. New is an operator whereas malloc() is a library function.
2. New allocates memory and calls constructor for object initialization. But malloc() allocates memory and does not call constructor.
3. Return type of new is exact data type while malloc() returns void*.
4. New is faster than malloc() because an operator is always faster than a function.

Q.1. (b) Differentiate between C and C++.

Ans. Difference between C and C++

C	C++
1. C is Procedural Language.	1. C++ is non Procedural i.e Object oriented Language.
2. No virtual Functions are present in C.	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible	3. The concept of polymorphism is used in C++.Polymorphism is the most important feature of OOPS.
4. Operator overloading is not possible in C.	4. Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.
7. In C. • scanf() Function used for Input. • printf() Function used for Output.	7. In C++. • Cin>> Function used for Input. • Cout<< Function used for output.
8. No inheritance is possible in C.	8. Inheritance is possible in C++
9. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	9. In C++, new and delete operators are used for Memory Allocating and Deallocating.
10. In C, Exception Handling is not present.	10. In C++, Exception Handling is done with Try and Catch block.

Q.1. (c) What is the use of inline member functions?

Ans. C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Q.1. (d) Show the use of Mutable keyword with example.

Ans. Mutable data members are those members whose values can be changed in runtime even if the object is of constant type. It is just opposite to constant.

Sometimes logic required to use only one or two data member as a variable and another one as a constant to handle the data. In that situation, mutability is very helpful concept to manage classes.

Example

```
#include <iostream>
using namespace std;
class Test {
  public:
    int a;
  mutable int b;
  Test(int x=0, int y=0) {
    a=x;
    b=y;
  }
  void seta(int x=0) {
    a = x;
  }
  void setb(int y=0) {
    b = y;
  }
  void disp() {
    cout<<endl<<"a:" <<a<<" b:" <<b<<endl;
  }
};
int main() {
  const Test t(10,20);
  cout<<t.a<<" " <<t.b<<"\n";
  // t.a=30; //Error occurs because a cannot be changed, because object is constant.
  t.b=100; //b still can be changed, because b is mutable.
  cout<<t.a<<" "<<t.b<<"\n";
  return 0;
}
```

Q.1. (e) Why virtual destructors are used?

Ans. Use of Virtual Destructors: Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
//A program with virtual destructor
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
  };
  int main(void)
  {
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
  }
```

Output:

Constructing base
Constructing derived
Destructing derived
Destructing base

Q.1. (f) What is the main difference between array of pointers and pointer to an array?

Ans. Array of pointers:

int *array_of_pointers[4];

We know that pointer holds a address of int data type variable. So the above mentioned array_of_pointers can store four different or same int data type address i.e. array_of_pointers[0], array_of_pointers[1], array_of_pointers[2] array_of_pointers[3] each can hold a int data type address.

Pointer to array: In simple language the pointer_to_array is a single variable that holds the address of the array.

Example: int simple_array[4];

int *p;

Now we can assign

p = simple_array;

Q.1. (g) Define static objects with example.

Ans. Refer Q.1.(g) of End Term Examination 2017. (Pg.No. 10-2017)

Q.1. (h) Differentiate between static and dynamic binding.

Ans.

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by **function** overloading and **operator** overloading.	It is achieved by **virtual functions** and **pointers**.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

Q.1. (i) Define Reusability, how C++ supports Reusability?

Ans. Reusability: C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or subclass. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Q.1. (j) Explain the use of this pointer with example.

Ans. this pointer: Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

```
class ClassName {
    private:
      int dataMember;
    public:
       method(int a) {
 // this pointer stores the address of object obj and access dataMember
          this->dataMember = a;
         ... .. ...
       }
}
int main() {
   ClassName obj;
   obj.method(5);
  ... .. ...
}
```

Q.2. (a) What is inheritance? Give its various types and access mechanisms. What are the advantages of scope resolutor & referencing? (8.5)

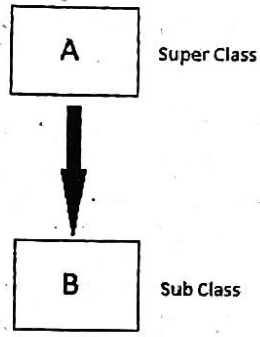
Ans. One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

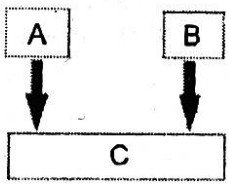
In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

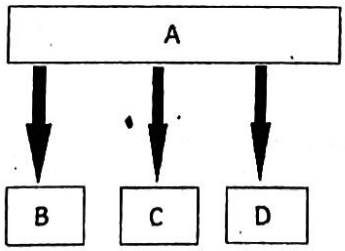
Single Inheritance: In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



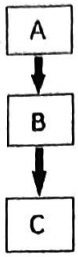
Multiple Inheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.



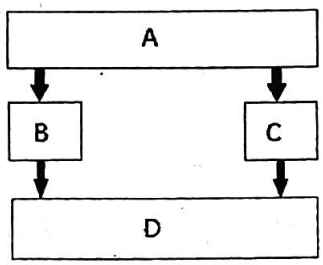
Hierarchical Inheritance: In this type of inheritance, multiple derived classes inherits from a single base class.



Multilevel Inheritance: In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid Inheritance: Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



Advantage of Scope Resolution Operator (::)

1. To access a global variable when there is a local variable with same name.
2. To define a function outside a class.
3. To access a class's static variables.
4. In case of multiple Inheritance:

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

5. Refer to a class inside another class:

If a class exists inside another class we can use the nesting class to refer the nested class using the scope resolution operator.

Referencing (&)

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

Example:

```
#include<iostream>
using namespace std;
int main()
{
  int x = 10;
  // ref is a reference to x.
  int& ref = x;

  // Value of x is now changed to 20
  ref = 20;
  cout << "x =" << x << endl ;

  // Value of x is now changed to 30
  x = 30;
  cout << "ref =" << ref << endl ;
```

```
  return 0;
}
```

Output:

x = 20

ref = 30

Q.2. (b) Create a class A with data members B and C respectively. Include a function swap () to swap the values in the data members. (4)

Ans. Swapping of 2 numbers using class:

```
#include<iostream.h>
#include<conio.h>
class A
{
int B,C;
public:
void getdata();
void swap();
void display();
};
void swap::getdata()
{
cout<<"Enter two numbers:";
cin>>B>>C;
}
void A::swap()
{
B=B+C;
C=B-C;
B=B-C;
}
void A::display()
{
cout<<"B="<<B<<"C="<<C;
}
int main()
{
clrscr();
A s;
s.getdata();
cout<<"\nBefore swap:\n";
s.display();
s.swap();
cout<<"\n\nAfter swap:\n";
```

```
s.display();
getch();
return 0;
}
```

Q.3. (a) What do you mean by an array of objects? Explain how members of objects can be accessed in array of objects with the help of C++ program. (7.5)

Ans. Array of Objects in c++

1. Like array of other user-defined data types, an array of type class can also be created.
2. The array of type class contains the objects of the class as its individual elements.
3. Thus, an array of a class type is also known as an array of objects.
4. An array of objects is declared in the same way as an array of any built-in data type.

Example:

```
#include<iostream>
using namespace std;
class A
{
private:
string name;
int age;
public:
void getName()
{
        cout<<"Your name is :" << name << "\n";
}

void setName()
{
        cout<<"Enter your name :" ;
        cin>>name;
}

int setAge()
{
        cout<<"Enter your age :" ;
        cin>>age;
}

void getAge()
{
        cout<<"Your age is :"  << age << "\n";
}
```

```
};

int main()
{
int size = 4;
A array[size]; //Array holding 4 object references of type, A.

//Setting the name and age of properties of each object of A class, stored in an array
for(int i=0; i<size; i++) //accessing array elements using length variable in for-loop
{
        array[i].setName();
        array[i].setAge();
}

//Getting the name and age of properties of each object of A class, stored in an array
for(int i=0; i<size; i++) //accessing array elements using length variable in for-loop
{
        array[i].getName();
        array[i].getAge();
}
}
```

Output-

```
Enter the name : Raj
Enter the age : 28
Enter the name : Tom
Enter the age : 25
Enter the name : Tobiasz
Enter the age : 31
Enter the name : Julia
Enter the age : 27
Enter the name : Steven
Enter the age : 32
The name is : Raj
The age is : 28
The name is : Tom
The age is : 25
The name is : Tobiasz
The age is : 31
The name is : Julia
The age is : 27
The name is : Steven
The age is : 32
```

Q.3. (b) What are Destructors? Write a program to show the order in which objects are destructed. **(5)**

Ans. The Class Destructor

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Example :

```
class A
{
 public:
 ~A();          //This is the destructor.
};
```

The Order in which Objects are Destructed:

C++ constructor call order will be from top to down that is from base class to derived class and c++ destructor call order will be in reverse order.

Below is the example of constructor destructor call order for multi-level inheritance, in which Device class is the base class and Mobile class is derived from Device base class and then Android class is derived from Mobile class as a base class.

Example :

```
#include
using namespace std;

//base class
class Device{

public:
Device(){cout<<"Constructor: Device\n";}
~Device(){cout<<"Destructor : Device\n";}
};

//derived class
class Mobile:public Device{

public:

Mobile(){cout<<"Constructor: Mobile\n";}
~Mobile(){cout<<"Destructor : Mobile\n";}
};
//derived class
```

```
class Android:public Mobile{
 public:

Android(){cout<<"Constructor: Android\n";}
~Android(){cout<<"Destructor : Android\n";}
};

//----------------------------TEST----------------------------
int main()
{
Android _android; // create the object that will call required constructors

return 0;
}
```

When we create the object of Android class, order of invocation of constructor and destructor will be as below:

Constructor: Device
Constructor: Mobile
Constructor: Android
Destructor : Android
Destructor : Mobile
Destructor : Device

Q.4. (a) What is the difference between overloading and overriding of a function? Write a program in C++ to overload== operator and compare two objects using the operator. **(7.5)**

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
Functions name and signatures must be same.	Having same Function name with different Signatures.
Overriding is the concept of runtime polymorphism	Overloading is the concept of compile time polymorphism
When a function of base class is re-defined in the derived class called as Overriding	Two functions having same name and return type, but with different type and/or number of arguments is called as Overloading
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types
Function should be public.	Function can be different access specifies

Program :

1. ** C++ Program to overload == operator in circle class*

```
2. #include <iostream>
3. using namespace std;
4.
5. class Circle {
6.    private:
7.       float radius;
8.    public:
9.       Circle(float r = 0) : radius(r) { }
10.      void changeRadius(int radius) { this->radius = radius; }
11.      float getRadius() { return radius; }
12.      float getArea() const { return 3.14 * radius * radius; }
13.      bool operator==(Circle a);
14.};
15.
16.inline bool Circle::operator==(Circle a)
17.{
18.   return this->radius == a.radius;
19.}
20.
21.int main()
22.{
23.   Circle A(10), B(20), C(10);
24.
25.   cout << "Circle A : Radius = " << A.getRadius() << " units, Area = "
26.        << A.getArea() << " sq. units\n";
27.   cout << "Circle B : Radius = "<< B.getRadius() << " units, Area = "
28.        << B.getArea() << " sq. units\n";
29.   cout << "Circle C : Radius = " << C.getRadius() << " units, Area = "
30.        << C.getArea() << " sq. units\n";
31.   cout << "A == B : ";
32.   if (A == B)
33.      cout << "Circles are equal.\n";
34.   else
35.      cout << "Circles are not equal.\n";
36.   cout << "A == C :";
37.   if (A == C)
38.      cout << "Circles are equal.\n";
39.   else
40.      cout << "Circles are not equal.\n";
41.}
```

Output:

Circle A : Radius = 10 units, Area = 314 sq. units

Circle B : Radius = 20 units, Area = 1256 sq. units

Circle C : Radius = 10 units, Area = 314 sq. units

A == B : Circles are not equal.

A == C : Circles are equal.

Q.4. (b) How to achieve dynamic memory allocation in C++? Explain with a program. 5

Ans. Dynamic memory allocation:

Dynamic memory allocation in C/C++ refers to performing memory allocations manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack**.

new operator:

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator: To allocate memory of any data type, the syntax is:

- pointer-variable = **new** data-type;

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
#include<iostream.h>
 #include<conio.h>

 void main()
 {

  int size,i;
  int *ptr;

  cout<<"\n\tEnter size of Array :" ;
  cin>>size;

  ptr = new int[size];
  //Creating memory at run-time and return first byte of address to ptr.


 for(i=0;i<5;i++)      //Input arrray from user.
 {
  cout<<"\nEnter any number :";
  cin>>ptr[i];
 }

 for(i=0;i<5;i++)      //Output arrray to console.
 cout<<ptr[i]<<", ";
```

```
delete[] ptr;
//deallocating all the memory created by new operator

}
```

Output :

```
Enter size of Array : 5

Enter any number : 78
Enter any number : 45
Enter any number : 12
Enter any number : 89
Enter any number : 56

78, 45, 12, 89, 56
```

Q.5. (a) How base class member functions can be invoked in a derived class if the derived class also has a member function with the same name? Explain with example. **(8.5)**

Ans. Refer Q. 5(a) of End Term Exam 2017. (Pg. 17.-2017)

Q.5. (b) Why do we need virtual function? When do we make a virtual function pure? **(4)**

Ans. Need of Virtual Function:

In C++, a regular, "non-pure" virtual function provides a definition, which means that the class in which that virtual function is defined does not need to be declared abstract. You would want to create a pure virtual function when it doesn't make sense to provide a definition for a virtual function in the base class itself, within the context of inheritance.

An example of when pure virtual functions are necessary :

For example, let's say that you have a base class called Figure. The Figure class has a function called **draw**. And, other classes like Circle and Square derive from the Figure class. In the Figure class, it doesn't make sense to actually provide a definition for the draw function, because of the simple and obvious fact that a "Figure" has no specific shape. It is simply meant to act as a base class. Of course, in the Circle and Square classes it would be obvious what should happen in the draw function – they should just draw out either a Circle or Square (respectively) on the page. But, in the Figure class it makes no sense to provide a definition for the draw function. And this is exactly when a pure virtual function should be used – the draw function in the Figure class should be a pure virtual function.

Q.6. (a) What is generic programming? What are its advantages? **(5)**

Ans. Generic Programming:

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**

You can use templates to define functions as well as classes

Advantages of generic programming :

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur
- You can also use class templates to develop a set of typesafe classes.

Q.6. (b) Write a template based program for adding objects of the Vector class. Use dynamic data members instead for storing vector elements. (7.5)

Ans.

```
#include <iostream>
using namespace std;
template<class T>
class vec {
public:
 vec(T f1, T f2) {
  x=f1;
  y=f2;
 }
 vec() {
 }
 T x;
 T y;
 vec operator+(const vec& v1) {
  vec result;
  result.x = v1.x+this->x;
  result.y = v1.y+this->y;
  return result;
 };
};
template<class T>
ostream &operator <<(ostream &stream, vec<T> v)
 {
  cout << "(" << v.x << ", "<< v.y <<")";
  return stream;
};
int main() {
 vec<int> v1(3,6);
 vec<int> v2(2,-2);
 vec<int> v3=v1+v2;
```

```
cout << "v3 =" << v3 << endl;
vec<float> v4(3.9,6.7);
vec<float> v5(2.0,-2.2);
vec<float> v6=v4+v5;
cout << "v6 =" << v6 << endl;
}
```

Q.7.(a) Write a program to compute the square root of a number. The input values must be tested for validity. If it is negative, the user defined function my_sqrt0 should raise an exception. **(7.5)**

Ans.

```
#include<iostream>
#include<cmath>
using namespace std;
class number
{
private:
  double num;
public:
  class NEG {}; //exception class
  void readnum()
  {
    cout<<"Enter number:";
    cin>>num;
  }
  double my_sqrt()
  {
    if(num<0)
      throw NEG();
    else
      return (sqrt(num));
  }
};
int main()
{
  number n1;
  double result;
  n1.readnum();
  try
  {
    cout<<"\n Trying to find square root . . .";
    result=n1. my_sqrt(); //interacts with class; may cause error
    cout<<"\n Square root is: " <<result<<endl;
```

```
    cout<<"Success . . . Exception is not raised."<<endl;
  }
  catch(number::NEG)
  {
    cout<<"\n Square root of negative number not possible!"<<endl;
  }
  return 0;
}
```

Here, if my_sqrt() detects the error and raises an exception by passing a nameless object of type class NEG. And important point to note is that runtime system searches catch block to detect the handler.

So, the output of the program is:

Enter number: 4
Trying to find square root . . .
Square root is: 2
Success . . . Exception is not raised.
And if you input negative number, then output is:
Enter number: -4
Trying to find square root . . .
Square root of negative number not possible!

Q.7. (b) Write a program having STUDENT as an abstract class and create derived classes such as ENGINEERING, SCIENCE, MEDICAL, etc. from the STUDENT class. Create their objects and process them. (5)

Ans.

```
# include <isotream>
using namespace std;
class student
{
  protected:
    int roll-number;
  public:
    void get-number (int a)
    roll-number = a;
}
void put-number (void)
{
cout << "Roll. No.:" <<roll-number <<"\n";
}
};
class engineering: virtual public student
{
  proteced:
    float part 1, part 2;
  public:
```

```
    void get_marks (float x, float y)
    {
       part 1 = x;  part 2 = y;
       }
    void put_marks (void)
    {
       cout << "Marks obtained:" << "\n";
            <<"part 1 = "<<part 1<<"\n"
            <<"part 2 = "<<part 2<<"\n"
    }
};
class medical : virtual public student
{
  protected:
    float part 1, part 2;
public:
    void get_mark (Heat x, float y)
  {
    part 1 = x; part 2 = y;
  }
  void put-marks (void)
  cout <<"marks obtained :" << "\n"
    <<"part 1 =" << part 1 << "\n"
    <<"part 2 = << part << "\n"
```

Q.8. (a) Explain seekg(), tellg(), seekp(), tellp() functions in file handling with sample program. **(7.5)**

Ans.

- tellp() - It tells the current position of the put pointer.

Syntax: filepointer.tellp()

- tellg() - It tells the current position of the get pointer.

Syntax: filepointer.tellg()

- seekp() - It moves the put pointer to mentioned location.

Syntax: filepointer.seekp(no of bytes,reference mode)

- seekg() - It moves get pointer(input) to a specified location.

Syntax: filepointer.seekg((no of bytes,reference point)

For seekp and seekg three reference points are passed:

ios::beg - beginning of the file

ios::cur - current position in the file

ios::end - end of the file

Below is a program to show importance of tellp, tellg, seekp and seekg:

```
#include <iostream>
#include<conio>
#include <fstream>
```

```
using namespace std;
int main()
{
  fstream st; // Creating object of fstream class
  st.open("E:\studytonight.txt",ios::out); // Creating new file
  if(!st) // Checking whether file exist
  {
    cout<<"File creation failed";
  }
  else
  {
    cout<<"New file created"<<endl;
    st<<"Hello Friends"; //Writing to file

    // Checking the file pointer position
    cout<<"File Pointer Position is" <<st.tellp()<<endl;
    st.seekp(-1, ios::cur); // Go one position back from current position

    //Checking the file pointer position
    cout<<"As per tellp File Pointer Position is" <<st.tellp()<<endl;

    st.close(); // closing file
  }
  st.open("E:\studytonight.txt",ios::in);  // Opening file in read mode
  if(!st) //Checking whether file exist
  {
    cout<<"No such file";
  }
  else
  {
    char ch;
    st.seekg(5, ios::beg);  // Go to position 5 from begning.
    cout<<"As per tellg File Pointer Position is" <<st.tellg()<<endl; //Checking file
pointer position
    cout<<endl;
    st.seekg(1, ios::cur); //Go to position 1 from beginning.
    cout<<"As per tellg File Pointer Position is"<<st.tellg()<<endl; //Checking file
pointer position
    st.close(); //Closing file
  }
  getch();
  return 0;
}
```

Output :

New file created

File Pointer Position is 13

As per tellp File Pointer Position is 12

As per tellg File Pointer Position is 5

As per tellg File Pointer Position is 6

Q.8.(b) How garbage collection is handled in C++? **(5)**

Ans. Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and can thus have significant influence on performance.

In computer science, **garbage collection** (**GC**) is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Resources other than memory, such as network sockets, database handles, user interaction windows, and file and device descriptors, are not typically handled by garbage collection. Methods used to manage such resources, particularly destructors, may suffice to manage memory as well, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming.