CMR
INSTITUTE OF
TECHNOLOGY

USN

Internal Assesment Test III - January 2020

Sub:	Object Oriented Programming Using C++							Code:	18MCA11
Date:	13.01.2020	Duration:	90 mins	Max Marks:	50	Sem:	I	Branch:	MCA

No.		Question	Marks	OBE CO	RBT
		Part - I			
1.		Explain STL container classes.	[10]	CO4	L2
		OR			
2.		What is Exception? Explain the use of try, catch and throw with example. Write a program for the exception division by zero.	[10]	CO3	L2
		Part - II			
3.	(a)	Explain manipulators. Write a Program to create your own manipulators.	[6]	CO4	L3
	(b)	Explain setw and setfill manipulators with example.	[4]	CO4	L2
		OR			
4.		Explain how can we catch all exception, restrict exception and rethrow exception with example	[10]	CO3	L2
		Part –III			
5.		What is stream? Discuss the four streams which are automatically opened when a C++ Program begins execution.	[10]	CO3	L2
6.		What is iterator? Write a simple C++ program to store and display integer elements using STL Vector class.	[10]	CO4	L2
		Part – IV			
7.		What is STL? List and explain the three types of containers in STL.	[10]	CO4	L2
		OR			
8.	(a)	Write a C++ program to create a template function for Bubble Sort and demonstrate sorting of integers and doubles	[6]	CO3	L3
	(b)	How are the Object Oriented Programming different from procedure Oriented Programming?	[4]	CO1	L2
		` **Part - V**			
9.		Create an abstract base class EMPLOYEE with data members: Name, EmpID and BasicSal and a pure virtual function Cal_Sal().Create two derived classes MANAGER (with data members: DA and HRA and SALESMAN (with data members: DA, HRA and TA). Write appropriate constructors and member functions to initialize the data, read and write the data and to calculate the net salary. The main() function should create array of base class pointers/references to invoke overridden functions and hence to implement run	[10]	CO3	L3
		OR			
10.		Write a program to implement FILE I/O operations on characters. I/O operations includes inputting a string, Calculating length of the string, Storing the string in a file, fetching the stored characters from it, etc.	[10]	CO3	L3

Internal Assesment Test III – January 2020 – 13/01/2020
Answer Key

Sub:	Object Oriented Programming Using C++	Code: 18MCA11	

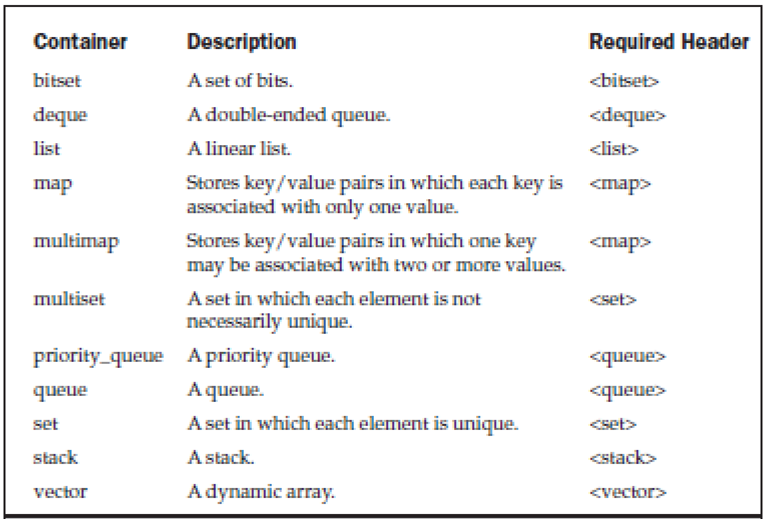
Part - I

1. Explain STL container classes. [10]

Containers

Containers are objects that hold other objects, and there are several different types. For example, the vector class defines a dynamic array, deque creates a double-ended queue, and list provides a linear list. These containers are called sequence containers

It also defines associative containers, which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys.
Each container class defines a set of functions that may be applied to the container.
For example, a list container includes functions that insert, delete, and merge elements.



Container	Description	Required Header
bitset	A set of bits.	<bitset>
deque	A double-ended queue.	<deque>
list	A linear list.	<list>
map	Stores key/value pairs in which each key is associated with only one value.	<map>
multimap	Stores key/value pairs in which one key may be associated with two or more values.	<map>
multiset	A set in which each element is not necessarily unique.	<set>
priority_queue	A priority queue.	<queue>
queue	A queue.	<queue>
set	A set in which each element is unique.	<set>
stack	A stack.	<stack>
vector	A dynamic array.	<vector>

Table 24-1. *The Containers Defined by the STL*

Ex:
```
#include <vector>
#include <iostream>
int main()
{
using namespace std;
 vector<int> vect;
    for (int nCount=0; nCount < 6; nCount++)
 vect.push_back(10 - nCount); // insert at end of array
  for (int nIndex=0; nIndex < vect.size(); nIndex++)
        cout << vect[nIndex] << " ";
  cout << endl;
}
```

2. **What is Exception? Explain the use of try, catch and throw with example. Write a program for the exception division by zero.** [10]

Exception means run time errors. Exception handling allows you to manage run-time errors in an orderly fashion. Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.
C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The

following discussion elaborates upon this general description.
Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.) Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown. The general form of **try** and **catch** are shown here.

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

The **try** can be as short as a few statements within one function or as allencompassing as enclosing the **main( )** function code within a **try** block (which effectively causes the entire program to be monitored).
When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement associated with a **try**. Which **catch** statement is used is determined by the type of the exception. That is, if the data type specified by a **catch** matches that of the exception, then that **catch** statement is executed (and all others are bypassed). When an exception is caught, *arg* will receive its value. Any type of data may be caught, including classes that you create. If no exception is thrown (that is, no error occurs within the **try** block), then no **catch** statement is executed.
The general form of the **throw** statement is shown here:
throw *exception*;
Example

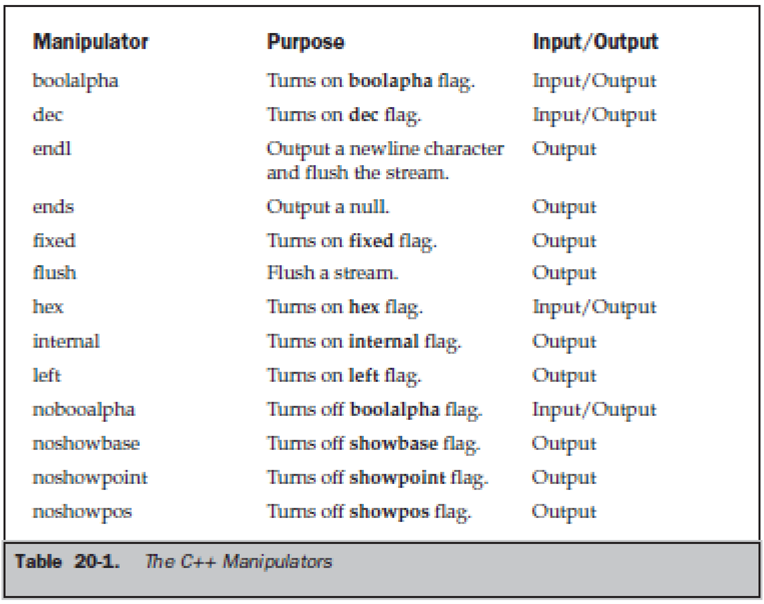
```
#include <iostream>
using namespace std;
void divide(double a, double b);
int main()
{
double i, j;
do {
cout << "Enter numerator (0 to stop): ";
cin >> i;
cout << "Enter denominator: ";
cin >> j;
divide(i, j);
} while(i != 0);
return 0;
}
void divide(double a, double b)
{
try {
if(!b) throw b; //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

Part - II

[6]

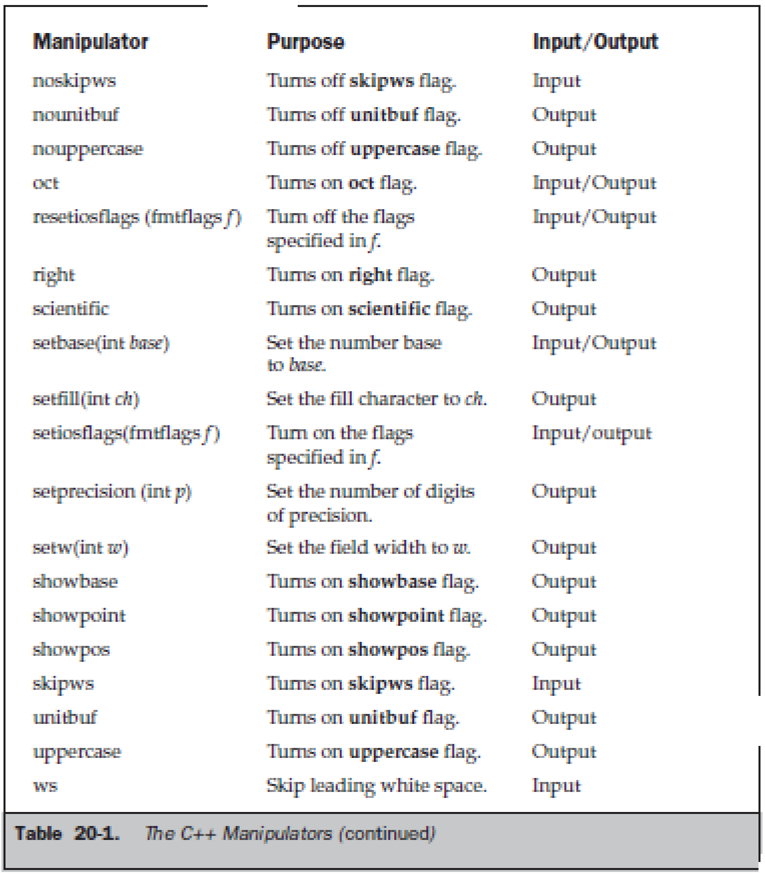
3. (a) Explain manipulators. Write a Program to create your own manipulators.

Manipulator functions are special stream functions that change certain characteristics of the input and output. They change the format flags and values for a stream. The main advantage of using manipulator functions is that they facilitate that formatting of input and output streams.
The following are the list of standard manipulator used in a C++ program. To carry out the operations of these manipulator functions in a user program, the header file input and output manipulator <iomanip.h> must be included.



Manipulator	Purpose	Input/Output
boolalpha	Turns on **boolapha** flag.	Input/Output
dec	Turns on **dec** flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on **fixed** flag.	Output
flush	Flush a stream.	Output
hex	Turns on **hex** flag.	Input/Output
internal	Turns on **internal** flag.	Output
left	Turns on **left** flag.	Output
nobooalpha	Turns off **boolalpha** flag.	Input/Output
noshowbase	Turns off **showbase** flag.	Output
noshowpoint	Turns off **showpoint** flag.	Output
noshowpos	Turns off **showpos** flag.	Output

Table 20-1. *The C++ Manipulators*



Manipulator	Purpose	Input/Output
noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

Table 20-1. *The C++ Manipulators (continued)*

Ex:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << hex << 100 << endl;
cout << setfill('?') << setw(10) << 2343.0;
return 0;
}
```

This displays

```
64
??????2343
```

Creating our own inserter:
All parameterless manipulator output functions have this skeleton:

```
ostream &manip-name(ostream &stream)
{
// your code here
return stream;
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;
// A simple output manipulator.
ostream &sethex(ostream &stream)
{
stream.setf(ios::showbase);
stream.setf(ios::hex, ios::basefield);
return stream;
}
int main()
{
cout << 256 << " " << sethex << 256;
return 0;
}
```

O/P : **256 0x100**

(b) Explain setw and setfill manipulators with example. [4]

setfill(int *ch*) Set the fill character to *ch .It is used to format an* Output.
setw(int *w*) Set the field width to *w. It is used to format an* Output.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << hex << 100 << endl;
cout << setfill('?') << setw(10) << 2343.0;
return 0;
}
```

This displays

```
64
??????2343
```

4. Explain how can we catch all exception, restrict exception and rethrow exception with example [10]

Catching All Exceptions

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**.

```
catch(...) {
// process all exceptions
}
```

Here, the ellipsis matches any type of data. The following program illustrates **catch(...)**.

```
// This example catches all exceptions.
#include <iostream>
using namespace std;
void Xhandler(int test)
{
try{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
catch(...) { // catch all exceptions
cout << "Caught One!\n";
}
}
int main()
{
cout << "Start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "End";
return 0;
}
```

This program displays the following output.

```
Start
Caught One!
Caught One!
Caught One!
End
```

Rethrowing an Exception

If you wish to rethrow an expression from within an exception handler, you may do so by calling **throw**, by itself, with no exception. This causes the current exception to be passed on to an outer **try/catch** sequence. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a **catch** block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate outward to the next **catch** statement. The following program illustrates rethrowing an exception, in this case a **char *** exception.

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
void Xhandler()
{
try {
throw "hello"; // throw a char *
}
catch(const char *) { // catch a char *
cout << "Caught char * inside Xhandler\n";
throw ; // rethrow char * out of function
}
}
int main()
{
cout << "Start\n";
try{
Xhandler();
}
catch(const char *) {
cout << "Caught char * inside main\n";
}
cout << "End";
return 0;
}
```

This program displays this output:

```
Start
Caught char * inside Xhandler
Caught char * inside main
End
```

Restricting Exceptions

You can restrict the type of exceptions that a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a **throw** clause to a function definition. The general form of this is shown here:

ret-type func-name(*arg-list*) throw(*type-list*)
{
// ...
}

Here, only those data types contained in the comma-separated *type-list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw *any* exceptions, then use an empty list.

Attempting to throw an exception that is not supported by a function will cause the standard library function **unexpected( )** to be called. By default, this causes **abort( )** to be called, which causes abnormal program termination.

The following program shows how to restrict the types of exceptions that can be
thrown from a function.

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
int main()
{
cout << "start\n";
try{
Xhandler(0); // also, try passing 1 and 2 to Xhandler()
}
catch(int i) {
cout << "Caught an integer\n";
}
catch(char c) {
cout << "Caught char\n";
}
catch(double d) {
cout << "Caught double\n";
}
cout << "end";
return 0;
}
```

Part –III

5. What is stream? Discuss the four streams which are automatically opened when a C++ Program begins execution. [10]

A stream is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. Because all streams behave the same, the same I/O functions can operate on virtually any type of physical device.

Template Class	Character-based class	Wide-Character-based Class
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

When a C++ program begins execution, four built-in streams are automatically opened.

They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

By default, the standard streams are used to communicate with the console. However, in environments that support I/O redirection (such as DOS, Unix, OS/2, and Windows), the standard streams can be redirected to other devices or files.

The standard input stream (cin):

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >>

```
#include <iostream>

using namespace std;

int main( )
{
   char name[50];

   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;
```

}

The standard output stream (cout):

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as <<

Ex:

```
#include <iostream>
 using namespace std;
 int main( )
{
   char str[] = "Hello C++";
    cout << "Value of str is : " << str << endl;
}
```

The standard error stream (cerr):

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

```
#include <iostream>
using namespace std;
int main( )
{
   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;
}
```

The standard log stream (clog):

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed

```
#include <iostream>
 using namespace std;
 int main( )
{
   char str[] = "Unable to read....";

   clog << "Error message : " << str << endl;
}
```

6. What is iterator? Write a simple C++ program to store and display integer elements using STL Vector class. [10]

Iterators are objects that act, more or less, like pointers. They give you the ability to
cycle through the contents of a container in much the same way that you would use
a pointer to cycle through an array. There are five types of iterators:
Iterator Access Allowed

- Random Access Store and retrieve values. Elements may be accessed randomly.
- Bidirectional Store and retrieve values. Forward and backward moving.
- Forward Store and retrieve values. Forward moving only.
- Input Retrieve, but not store values. Forward moving only.
- Output Store, but not retrieve values. Forward moving only.

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

int main() {
  // create a vector to store int
  vector<int> vec;
  int i;

  // display the original size of vec
  cout << "vector size = " << vec.size() << endl;

  // push 5 values into the vector
  for(i = 0; i < 5; i++){
    vec.push_back(i);
  }

  // display extended size of vec
  cout << "extended vector size = " << vec.size() << endl;

  // access 5 values from the vector
  for(i = 0; i < 5; i++){
    cout << "value of vec [" << i << "] = " << vec[i] << endl;
  }
return 0;
}
```

Part – IV

7. What is STL? List and explain the three types of containers in STL. [10]

At the core of the standard template library are three foundational items: *containers*, *algorithms*, and *iterators*. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers

Containers are objects that hold other objects, and there are several different types. For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called *sequence containers* because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines *associative containers*, which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key. Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

Algorithms act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of

containers. Many algorithms operate on a *range* of elements within a container.
Iterators
Iterators are objects that act, more or less, like pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array.
There are five types of iterators:

Iterator	Access	Allowed
Random Access	Store and retrieve values.	Elements may be accessed randomly.
Bidirectional	Store and retrieve values.	Forward and backward moving.
Forward	Store and retrieve values.	Forward moving only.
Input	Retrieve, but not store values.	Forward moving only.
Output	Store, but not retrieve values.	Forward moving only.

Container Classes:
Vectors
Perhaps the most general-purpose of the containers is **vector**. The **vector** class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be adjusted at run time to accommodate changing program conditions. A vector solves
this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements. The template specification for **vector** is shown here:
template <class T, class Allocator = allocator<T> > class vector

Some of the most commonly used member functions are **size( )**, **begin( )**, **end( )**, **push_back( )**, **insert( )**, and **erase( )**. The **size( )** function returns the current size of the vector. This function is quite useful because it allows you to determine the size of a vector at run time. Remember, vectors will increase in size as needed, so the size of a vector must be determined during execution, not during compilation. The **begin( )** function returns an iterator to the start of the vector. The **end( )** function returns an iterator to the end of the vector. As explained, iterators are similar to pointers, and it is through the use of the **begin( )** and **end( )** functions that you obtain an iterator to the beginning and end of a vector. The **push_back( )** function puts a value onto the end of the vector. If necessary, the vector is increased in length to accommodate the new element. You can also add
elements to the middle using **insert( )**. A vector can also be initialized. In any event, once a vector contains elements, you can use array subscripting to access or modify those elements. You can remove elements from a vector using **erase( )**

List
The list class supports a bidirectional, linear list. Unlike a vector, which supports random access, a list can be accessed sequentially only. Since lists are bidirectional, they may be accessed front to back or back to front.
A list has this template specification:
template <class T, class Allocator = allocator<T> > class list
Here, T is the type of data stored in the list. The allocator is specified by Allocator, which defaults to the standard allocator. It has the following constructors:
explicit list(const Allocator &a = Allocator( ));
explicit list(size_type num, const T &val = T ( ),
const Allocator &a = Allocator( ));
list(const list<T, Allocator> &ob);
template <class InIter>list(InIter start, InIter end, const Allocator &a = Allocator( ));
The first form constructs an empty list. The second form constructs a list that has num elements with the value val, which can be allowed to default. The third form constructs a list that contains the same elements as ob. The fourth form constructs a list that contains the elements in the range specified by the iterators start and end.

Maps
The map class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus, in its most general sense, a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone number as its value. Associative containers are becoming more popular in programming.
As mentioned, a map can hold only unique keys. Duplicate keys are not allowed. To create a map that

allows nonunique keys, use multimap. The map container has the following template specification:
template <class Key, class T, class Comp = less<Key>, class Allocator = allocator<pair<const key, T> >
class map

8. (a) Write a C++ program to create a template function for Bubble Sort and demonstrate sorting of integers and doubles [6]

```
#include<iostream>
using namespace std;
#define Max 100
template <class T>
void sort(T a[],int n)
{
int i,j;
for(i=0;i<n-1;i++)
{
                for(j=0;j<n-i-1;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                        T temp=a[j];
                                        a[j]=a[j+1];
                                        a[j+1]=temp;
                        }
                }

        }

}

int main()

{

        int a[Max],i,n;
        double d[Max];
        cout<<"enter array size\n\n";
        cin>>n;
        cout<<"enter array integer elements\n\n";
        for(i=0;i<n;i++)
                cin>>a[i];
                cout<<"enter array double elements\n\n";

        for(i=0;i<n;i++)
                cin>>d[i];
        cout<<"integer part\n\n";
        sort(a,n);
        for(i=0;i<n;i++)

                cout<< a[i]<<"\n";
        cout<<"double part\n\n";
        sort(d,n);
        for(i=0;i<n;i++)

                cout<<d[i]<<"\n";
        return 0;
}
```

(b) How are the Object Oriented Programming different from procedure Oriented Programming? [4]

Definition

OOP stands for Object-oriented programming and is a programming approach that focuses on data rather than the algorithm, whereas POP, short for Procedure-oriented programming, focuses on procedural abstractions.

Programs

In OOP, the program is divided into small chunks called objects which are instances of classes, whereas in POP, the main program is divided into small parts based on the functions.

Accessing Mode

Three accessing modes are used in OOP to access attributes or functions – 'Private', 'Public', and 'Protected'. In POP, on the other hand, no such accessing mode is required to access attributes or functions of a particular program.

Focus

The main focus is on the data associated with the program in case of OOP while POP relies on functions or algorithms of the program.

Execution

In OOP, various functions can work simultaneously while POP follows a systematic step-by-step approach to execute methods and functions.

Data Control

In OOP, the data and functions of an object act like a single entity so accessibility is limited to the member functions of the same class. In POP, on the other hand, data can move freely because each function contains different data.

Security

OOP is more secure than POP, thanks to the data hiding feature which limits the access of data to the member function of the same class, while there is no such way of data hiding in POP, thus making it less secure.

Ease of Modification

New data objects can be created easily from existing objects making object-oriented programs easy to modify, while there's no simple process to add data in POP, at least not without revising the whole program.

Process

OOP follows a bottom-up approach for designing a program, while POP takes a top-down approach to design a program.

Examples

Commonly used OOP languages are C++, Java, VB.NET, etc. Pascal and Fortran are used by POP.

` **Part - V**

9. Create an abstract base class EMPLOYEE with data members: Name, EmpID and BasicSal and a pure virtual function Cal_Sal().Create two derived classes MANAGER (with data members: DA and HRA and SALESMAN (with data members: DA, HRA and TA). Write appropriate constructors and member functions to initialize the data, read and write the data and to calculate the net salary. The main() function should create array of base class pointers/references to invoke overridden functions and hence to implement run [10]

```
#include<iostream>
#define Max 20
using namespace std;

class EMPLOYEE {
            public:
            char name[Max];
            int empid;
            float basic;
            EMPLOYEE()
            {

            }
            void read()
            {
                        cout<<"\n enter employee number:";
                        cin>>empid;
                        cout<<endl<<"Enter employee name:";
                        cin>>name;
                        cout<<endl<<"Enter the basic salary:";
                        cin>>basic;
            }
            void putinfo()
            {
                        cout << " Employee number is : " << empid;
                        cout << "\nName : " << name;
                        cout << "\nbasic  : " << basic << "\n";
            }
            virtual void calsal() = 0; // Pure virtual function ie why EMPLOYEE becomes a abstract
base class
};

class MANAGER:public EMPLOYEE
{
           float da,it,hra,netsal,gross;
           public:
                      MANAGER()
                      {
                      da=it=hra=netsal=gross=0;
                      }
                      void calsal()
                      {
                                 da=basic*0.52;
                                 hra=basic*0.65;
                                 gross=basic+hra+da;
                                 it=gross*0.3;
                                 netsal=gross-it;
                                 cout<<"Manager Details:"<<endl;
                                 EMPLOYEE::putinfo();
                                 cout << "DA : " << da;
                                 cout << "\nHRA : " << hra;
                                 cout << "\nGROSS  : " << gross ;
                                 cout<<"\n IT :" <<it;
                                 cout<<"\n NETSAL :"<<netsal;
                   }
};
class SALESMAN: public EMPLOYEE
```

```
{
        float da,it,hra,netsal,gross,ta;
        public:
                SALESMAN()
                 {
                da=it=ta=hra=netsal=gross=0;
                 }
                void calsal()
                {
                        da=basic*0.42;
                        hra=basic*0.52;
                        ta=basic*0.32;
                        gross=basic+da+hra+ta;
                        it=gross*0.3;
                        netsal=gross-it;
                        cout<<" Salesman Details:"<<endl;
                        EMPLOYEE::putinfo();
                        cout << "DA : " << da;
                        cout << "\nHRA : " << hra;
                        cout<<"\n TA :"<<ta;
                        cout << "\nGROSS : " << gross ;
                        cout<<"\n IT :" <<it;
                        cout<<"\n NETSAL :"<<netsal;
                }
};

int main()
        {
                 // base class pointer
                MANAGER m;
                SALESMAN s;
                EMPLOYEE *p[]={
                                &m,&s
                                };

                cout<<"\nEnter manager details\n";
                p[0]->read();

                cout<<"\nEnter salesman details\n";
                p[1]->read();

                for(int i=0;i<2;i++)
                        p[i]->calsal();
                        return 0;
        }
```

10. Write a program to implement FILE I/O operations on characters. I/O operations includes inputting a string, Calculating length of the string, Storing the string in a file, fetching the stored characters from it, etc. [10]

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

int main() {
  // create a vector to store int
  vector<int> vec;
  int i;
```

```
   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;

   // push 5 values into the vector
   for(i = 0; i < 5; i++){
      vec.push_back(i);
   }

   // display extended size of vec
   cout << "extended vector size = " << vec.size() << endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++){
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }
return 0;
}
```

O/P:

```
vector size = 0

extended vector size = 5

value of vec [0] = 0

value of vec [1] = 1

value of vec [2] = 2

value of vec [3] = 3

value of vec [4] = 4
```