Standard Template Library

Unit IV

To help C++ users in generic programming, Alexander and Meng Lee developed set of general purpose templatized classes (data structures) and functions (algorithms) that could be used as standard approach for storing and processing of data. The collection of these generic classes and functions is called Standard Template Library (STL).

- powerful set of C++ template classes.

Components of STL (3 core components)

1. Containers
2. Algorithms
3. Iterators

These three components work in conjunction with one another to provide support.

Algorithms employ iterators to perform operations stored in containers.

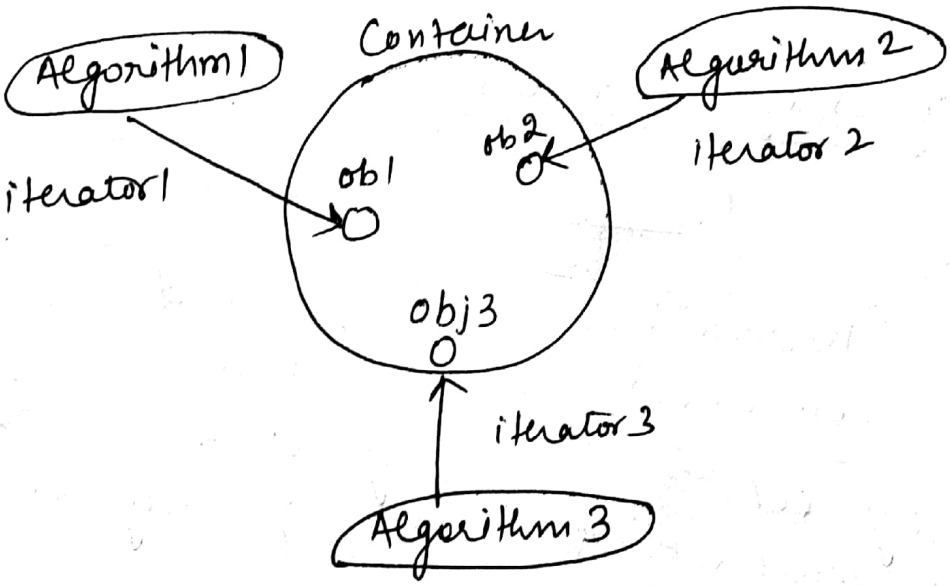


fig: Relationship between STL components

Scanned by CamScanner

① Container

↳ It is an object that actually stores data.

↳ It is a way data is organized in memory.

STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.

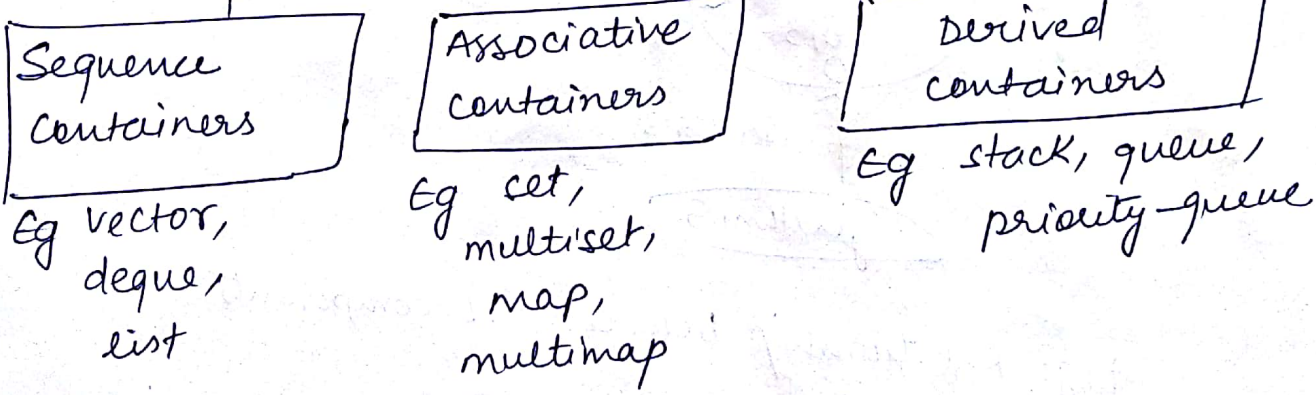
↳ Container library in STL provide containers that are used to create data structures like arrays, linked list, trees etc.

↳ These containers are generic, hence they can hold elements of its boo different data types.

Containers Example

① <vector>
② <list>
③ <deque>
④ <queue>
⑤ <stack>
⑥ <set>
⑦ <map>
⑧ <multimap>
⑨ <priority queue>

Types of containers



Eg vector, deque, list

Eg set, multiset, map, multimap

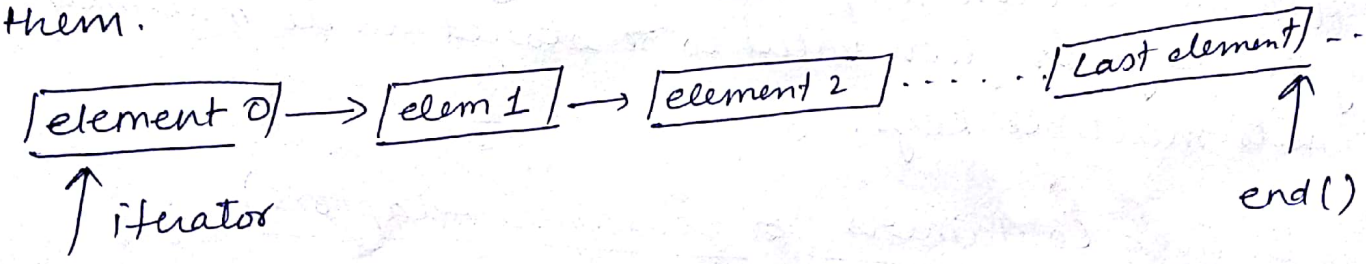
Eg stack, queue, priority-queue

Scanned by CamScanner

A container class defines set of functions that can be used manipulate its contents. Eg vector container defines functions for inserting elements, erasing the contents and swapping the contents.

① Sequence containers

↳ It stores elements in linear sequence like a line. Each element is related to other elements by its position along the line. They all expand themselves to allow insertion of elements and all of them support number of operations on them.



Eg :- Vector, List, Deque.

↳ Elements in all these containers can be accessed using an iterator.

↳ Difference between them is related to only their performance.

② Associative containers

↳ Designed to support direct access to elements using keys.

↳ they are not sequential.

Eg. Set, multiset, Map, Multimap

↳ All these containers store data in a structure called tree which facilitates fast searching, deletion and insertion. However, they are slow for random acess, inefficient for sorting.

Scanned by CamScanner

↳ Set and multiset can store number of items and operations for manipulating them using values as keys. Eg a 'set' might store objects of student class which are ordered alphabetically using names as keys. We can search for desired student using his name as key.

↳ Main difference between set and multiset is that multiset allows duplicate items while set does not.

↳ Map and multimap are used to store pairs of items, one called the key and other called the value.

Difference between map and multimap is that map allows only one key for given value to be stored while multimap permits multiple keys.

(3) Derived Containers (or Container Adaptors)

↳ provides 3 derived containers — stack, queue, priority-queue

↳ stacks, Queue, Priority queue can be created from different sequence containers.

↳ Derived containers do not support iterators and therefore we cannot use them for data manipulation

↳ they support 2 member functions — pop(), push() for implementing deleting and inserting operations.

II) Algorithm

↳ Algorithms are the functions that can be used across variety of containers for processing their contents.

↳ Algorithms act on containers. They provide the means by which you will perform initializing, sorting, searching, and transforming of the contents of containers.

Scanned by CamScanner

An a
specif
least
ec

* Two most common types of bugs are logical errors and semantic. errors.
algorithm reinforce the philosphy of reusability
Algorithm provides abstraction.
↳ STL algorithm based on the nature of operations they perform, may be categorized under:-
1) Retrieve or non-mutating algorithms
2) Mutating Algo
3) Sorting "
4) Set "
5) Relational "

III Iterators

↳ Iterators behave like pointers, used to acess container elements. They are often used to traverse from one element to another, a process is known as iterating through the container
↳ It actually act as a bridge between containers and algorithms.
↳ There are 5 typis of iterators :-

1. Input
2. Output
3. Forward
4. Bidirectional
5. Rational

Scanned by CamScanner

two most comm.
syntactic .

Vector

↳ Widely used container.
↳ It stores elements in contiguous memory locations & enable direct access to any element using subscript operator [].
↳ It can change its size dynamically and therefore allocates memory as needed at run time.

Eg
```
vector<int> V1;          // zero length int vector
vector<double> V2(10);   // 10-element double vector
vector<int> V3(V4);      // creates V3 from V4
vector<int> V(5,2);      // 5-element vector containing
                            all 2's.
```

Imp member function of vector

① at() : gives reference to an element
back() : gives reference to last element
begin() : " " " first "
capacity() : gives current capacity of vector
clear() : deletes all elements
empty() : determine if vector is empty or not
end() : gives reference to the end of vector
erase() : deletes specified elements
insert() : inserts elements in vector
pop-back() : deletes last element
push-back() : adds an element to the end
size() : gives number of elements

Scanned by CamScanner

Using vectors

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

int main()
{
    vector<int> v1{10,20,30,40,50};
    vector<char> v2(5);
    vector<int> v3(5,10);
    vector<string> v4(3,"welcome");

    cout<<v4[0]<<endl;          // "welcome"
    for(int i=0; i<=4; i++)
    {
        cout<<v1[i]<<endl;
    }
    v1.push_back(60);
    v1.push-back(70);
    v1.push-back(80);    // v1.pop_back(80);
    for(int i=0; i<=7; i++)
    {
        cout<<v1[i]<<endl;
    }
    vector<int> v5;
    for(int i=0; i<=9; i++)
    {
        v5.push-back(10*(i+1));
    }
    cout<<"current capacity"<<v5.capacity()<<endl;  // 16
    v5.pop-back(); v5.pop-back();
    v5.pop-back(); v5.pop_back();
```

Scanned by CamScanner

Abstract class

An abstract class is a cl... specifically used as ... least one pur... We can ... s...

```
cout << "current capacity" << vs.capacity() << endl; //6
cout << "size of vs vector : " << vs.size() << endl; //6
vs.pop_back();
cout << "size of vs vector : " << vs.size() << endl; //5

return 0;
}
```

Note :- Capacity remains unchanged after deleting some elements from vector.

Scanned by CamScanner

Templates

Unit IV

* Templates are powerful features of C++ which allows us to write generic programs.
* Simply, we can write a single function or a class to work with different data types using templates.
* Templates enable us to define generic classes and functions and thus provides support for generic programming.
Generic programming is an approach where generic types are used as parameters in algorithms so that they work for variety of suitable data types and data structures.
* Templates can be used to create family of classes or functions.
* Template can be considered as kind of macro. When an object of a specific type is defined for actual use, template definition for that class is substituted with required data type. Since, template is defined with parameter that would be replaced by specific data type at the time of actual use of the class or function, templates are sometimes called parameterized classes or functions.
* The concept of Templates can be used in 2 different ways

- Function Templates
- Class Templates

Scanned by CamScanner

Standard :-

① Function templates

* Single function templates can work with different types at once but, a single normal function can only work with one set of data types.

Syntax :-

```
template <class T>
returntype functionname (arguments of type T)
{
        // .... Body of function
        // with type T
        // wherever appropriate
        // ....
}
```

Eg 1 finding minimum

```
#include<iostream.h>
using namespace std;
template <class type1, class type2>
T or type min(type1 a, type2 b)
{
    if (a<b)
    return a;
    else
    return b;
}
```

```
int main()
{
    cout<<"Min no is"<< min(10, 20);
    cout<<"Min no is"<< min(10.5, 30.6);
    cout<<"Min char is"<< min('A', 'B');

    return 0;
}
```

Eg.2 TO swap

```
using namespace std;
template <class T>
void swap(T &x, T &y)
{
    T temp;
    temp = x;
    x = y;
    y = temp;
}
void func(int m, int n, float a, float b)
{
    cout<<"m & n before swap:"<<m<<" "<<n<<"\n";
    swap(m,n);
    cout<<"m & n after swap:"<<m<<" "<<n<<"\n";
    cout<<"a & b before swap:"<<a<<" "<<b<<"\n";
    swap(a,b);
    cout<<"a, b after swap:"<<a<<" "<<b<<"\n";
}
```

Scanned by CamScanner

```
int main()
{
    func(100, 200, 11.22, 33.44);
    return 0;
}
```

(2) Class Templates

* Like function templates, we can also create class templates for generic class operations
* Sometimes, we need a class implementation that is same for all classes, only the data types used are different.
* class templates make it easy to reuse the same code for all data types.

Eg Syntax :-

```
template <class T>
class classname
{
    // ... class member specification
    // with anonymous type T
    // wherever appropriate
    // ....
};
```

fig: Rela

Scanned by CamScanner

```
template <class T>
class classname
{
    ....
    public:
    T var;
    T somefunction (T arg);
    .....
};
```

How to create class template object?

- To create an object, we need to define data type inside <> when creation.

```
className <dataType> classObject;
```

Eg:-

```
className <int> classObject;
className <float> classObject;
className <string> classObject;
```

Eg Demonstration of class templates

```
using namespace std;
template <class T>
class Arith
{ private;
    T n1, n2;
  public:
    Arith (T n3, T n4)
    {   n1 = n3;
        n2 = n4;
    }
```

Scanned by CamScanner

```
void display()
{ cout<<" Numbers are "<< n1<<" and "<< n2 <<"."<<
  cout<<" Addition is : "<< add()<< endl;
  cout<<" Subtraction is : "<< subtract()<< endl;
}
T add()
{ return n1+n2;
}
T subtract()
{ return n1-n2;
}
};
int main()
{
  Arith <int> ar(10, 5);
  cout<<" Int Results :"<<endl;
  ar. display();
  Arith <float> arf (8.8, 2.2);
  cout<<" Float Results :"<<endl;
  arf. display();
  return 0;
}
```

Scanned by CamScanner

Overloading of Template functions

```
#include<iostream>
using namespace std;
template <class T>
void display(T x)   // overloaded template function display
{
 cout<<"Overloaded Template Display1 :"<<x<<"\n";
}
template <class T, class T1>
void display(T x, T1 y)
{
 cout<<"Overloaded Template Display2 :"<<x<<","<<y<<"\n";
}
void display(int x)
{
 cout<<"Explicit display:"<<x<<"\n";
}
int main()
{
   display(100);
   display(12.34);
   display(100, 12.34);
   display('C');
   return 0;
}
```

O/P :-

```
Explicit display : 100
Overloaded Template Display1 : 12.34
Overloaded Template Display2 : 100, 12.34
Overloaded Template Display1 : C
```

Note: display(100) invokes the ordinary version of display() and not template version

Scanned by CamScanner

Unit III

designed to be

Exception Handling Unit IV

* Two most common types of bugs are logical errors and syntactic errors.

Logical error :- occur due to poor understanding of the problem and solution procedure.

Syntactic error: arise due to poor understanding of the language itself.

↳ There are some peculiar problems other than logic or syntax errors, they are known as exceptions.

↳ Exceptions are run-time anomalies or unusual conditions that a program may encounter while executing.

Eg. Division by zero, access to an array outside its bounds or running out of memory or disk space.

↳ When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively.

↳ Exceptions are of two kinds :-

Synchronous Exceptions	Asynchronous Exceptions
↳ Eg. 'out of range index' errors and 'over-flow' errors	– Errors caused by events beyond the control of the program (such as keyboard interrupts).
↳ Proposed exception handling designed to handle only Synchronous exceptions.	
↳ If event occurs at same place every time, the program executed with same data, m/m application, event is synchronous.	

↳ Purpose of exception handling mechanism is to provide means to detect and report an 'exceptional circumstance' so that appropriate action can be taken.

Scanned by CamScanner

echanism suggests separate error handling code that p
ollowing tasks:-

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the excepti
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

↳ Error handling code consists of two segments :-
1) To detect error & to throw exceptions
2) To catch the exceptions and to take appropriate actions

Exception Handling Mechanism

↳ consists of 3 keywords :- try, throw, catch

Try :- used to preface block of statements which may generate exceptions. This block of statements is known as try block.

Throw :- when an exception is detected, it is thrown using 'throw' statement in the try block.

Catch : A catch block defined by keyword 'catch' catches the exception 'thrown' by the throw statement in try block and handles it appropriately.

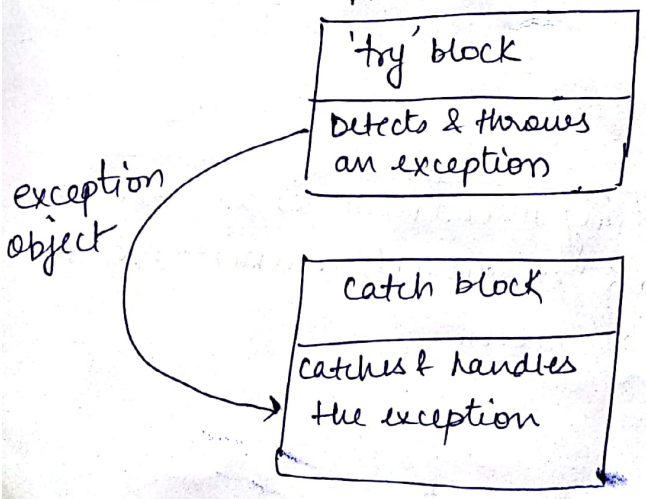


Fig:
Block - throwing exception

Scanned by CamScanner

atch block that catches an exception must immediately
llow the try block that throws the exception.

Syntax :-

```
...
...
try
{
    ...
    ...
    throw exception;      // Block of statements which
    ...                   // detects and throws an
    ...                   //exception
}
catch ( type arg)         // catches exception
{
    ...                   // Block of statements that
    ...                   // handles the exception
}
```

Eg

```
int main()
{
int a,b;
cout<<" Enter value of a & b";
cin>> a >> b;
int x = a-b;
try
{
    if (x!=0)
    { cout<<"Result(a/x) =" << a/x << "\n";
    }
    else                  // There is an exception
    {
      throw(x);           // Throws an object
    }
```

Scanned by CamScanner

```
catch (int i) // catches exception
{
 cout << "Exception caught : Divide by zero";
}
cout << "END";
return 0;
}
```

O/P

Enter value of a and b

20 20

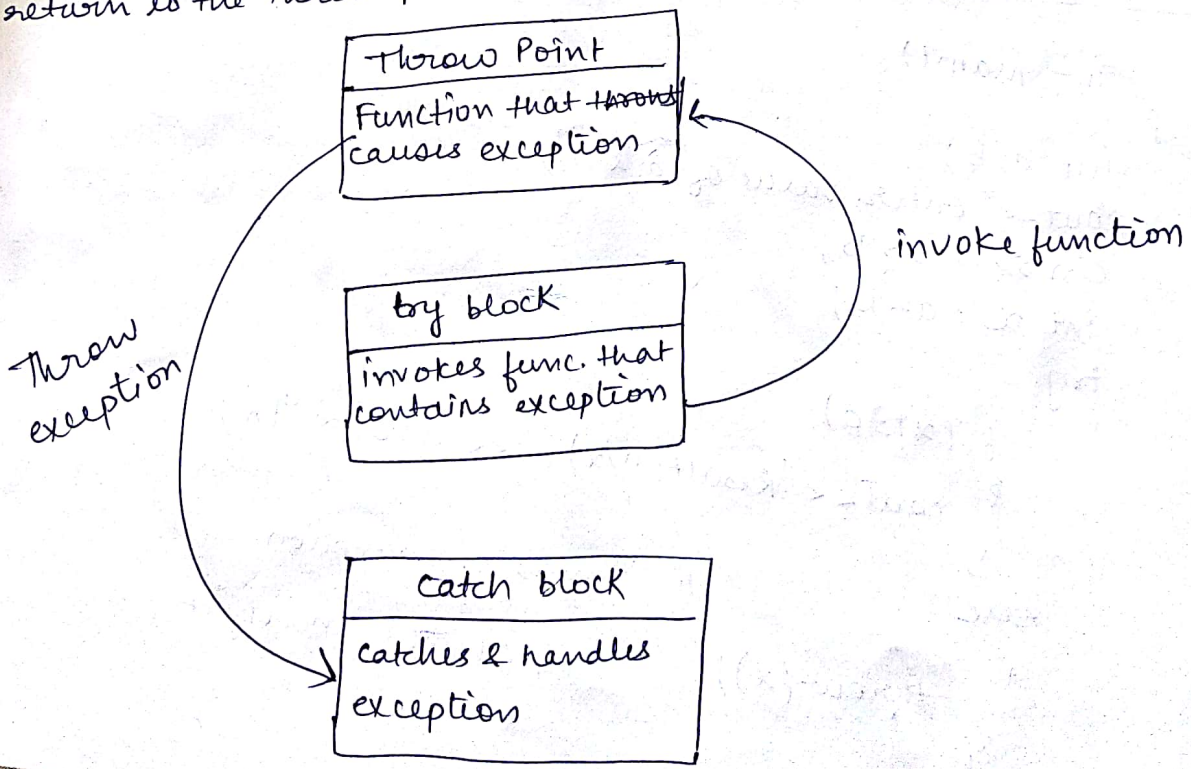
Exception caught : DIVIDE BY zero

END

Invoking function that generate Exception

↳ The point at which the throw is executed is called throw point.

↳ Once an exception is thrown to the catch block, control cannot return to the throw point.



Scanned by CamScanner

at :-

```
type function(arg list)          // function with exception
{
    ...
    ...
    throw(object);               // throws exception
    ...
    ...
}
try
{
    : // invoke function
}
catch (type arg)
{
    : // handles exception
}
```

Eg:-

```
void divide( int x, int y, int z )
{
  cout << "Inside function";
  if((x-y) != 0)                 // No problem
  {
     int r = z/(x-y);
     cout << "Result = " << r << "\n";
  }
  else                           // There is problem
  {
     throw(x-y);                 // Throws point
  }
}
int main()
{
    try
    {
       cout << "Inside try block \n";
```

Scanned by CamScanner

```
    divide(10,20,30); // Invokes divide()
    divide(10,10,20); //  "
}
catch(int i)
{
    cout<<"Caught exception\n";
}
return 0;
}
```

O/P :-

```
Inside try block
Inside function
Result = -3
Inside function
Caught exception
```

Throwing Mechanism

↳ When exception that is desired to be handled is detected, it is thrown using throw statement.

```
throw(exception);
throw exception;
throw;          // used for rethrowing exception
```

Scanned by CamScanner

Unit III

Catching Mechanism

Code for handling exceptions is included in catch blocks.
A catch block looks like a function definition :-

```
Catch (type arg)
{
    // statements for managing
    // exceptions
}
```

type → indicates type of exception that catch block handles.

Eg

```
void test (int x)
{ try { if (x>0)
            throw x;
        else
            throw 'x';
}
catch (int x)
{ cout << "Catch integer, that int is" << x;
}
catch (char x)
{
cout << "Catch char, that char is" << x;
}
void main()
{ test(10); test(0); }
```

O/P

```
10
x
```

Multiple Catch

```
try
{
    // try block
}
catch ( type1 arg)
{
    // catch block 1
}
catch (type2 arg)
{
    // catch block 2
}
.
.
.
catch (typeN arg)
{
    // catch block N
}
```

→ If catch block can't handle particular exception it has caught, we can rethrow the exception. Rethrow expression causes originally thrown object to be rethrown

Rethrowing Exception :→ A handler decide to rethrow the exception caught without processing it. We simply invoke throw without any argument :-

```
throw;
```

This causes current exception to be thrown to next enclosing try/catch sequence & is caught by catch statement listed after enclosing try block

Scanned by CamScanner

Ch
Managing Console I/O operations

C++ Streams

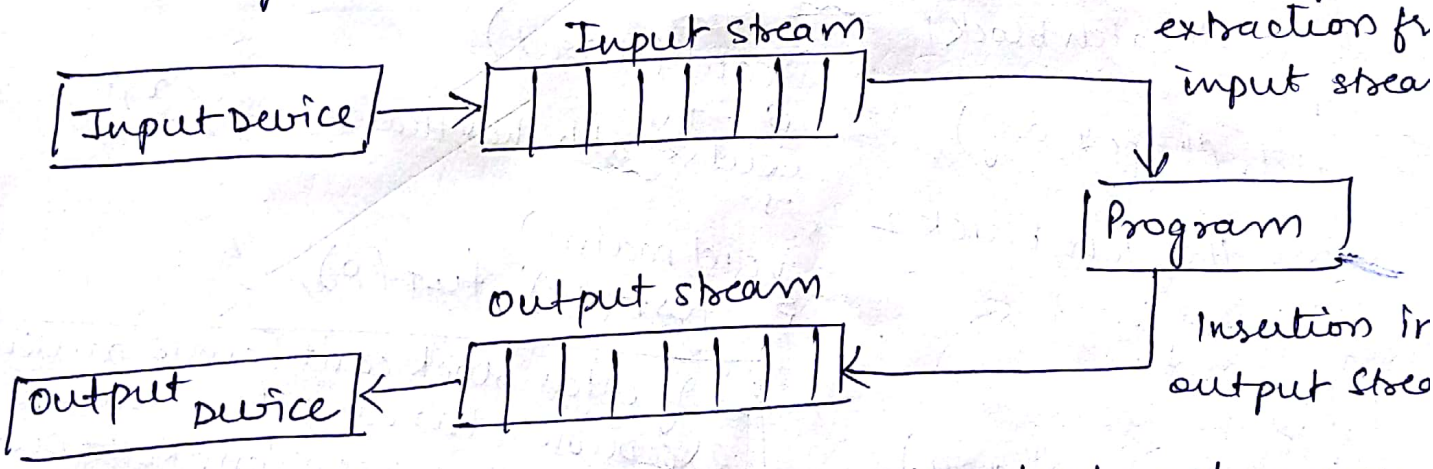
↳ I/o system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is called as stream.

Definition:-

↳ Streams is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.

↳ The source stream that provides data to the program is called input stream and destination stream that receives output from the program is called output stream.

↳ Program extracts the bytes from an input stream and inserts bytes into an output stream.



↳ Data in the input stream can come from keyboard or any other storage device. Similarly, data in the output stream c go to the screen or any other storage device.

↳ Stream act as interface between program and input/output device.

Scanned by CamScanner

C++ Stream classes

↳ C++ I/O system contains hierarchy of classes are used to define various streams to deal with both console and disk files these classes are called stream classes.

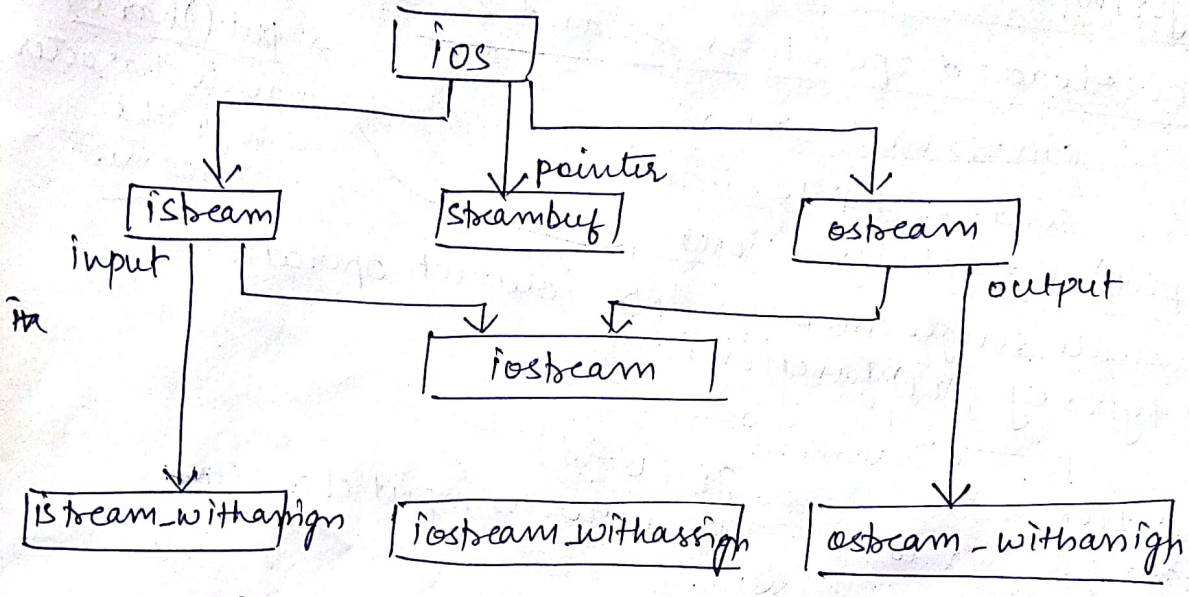


fig: Stream classes for console I/O operations

1) ios (general input/output stream class)
- contains basic facilities used by all other input, output classes.
- also contain pointer to buffer object (streambuf)

2) istream (input stream)
- inherits properties of ios
- declare input function → get(), getline(), read()
- contain overloaded extraction operator >>.

3) ostream (output stream)
- inherits properties of ios
- declare output functions put() and write(), <<

4) iostream (input/output stream)
- inherits properties of ios stream and ostream, istream through multiple inheritance and thus contains all the input, output functions.

Scanned by CamScanner

5) Streambuf

- provides an interface to physical device through buffer
- act as base for filebuf class used ios files.

Unformatted I/O operations → transfers internal binary representation of data as is

① overloaded operators >> and << b/w m/m & file

cin >> var1 >> var2 ....

cout << item1 << item2 ....

② put() and get() functions → put(char c) inserts character c into the stream.

↳ to handle single character input/output operations.

↳ 2 types of get() functions

get(char *) ↳ assign i/p char to its argument

get(void) ↳ returns input character

③ getline() and write() function

→ getline() reads whole line of text that ends with new line character.

cin.getline(line, size);

→ write() displays an entire line :-

cout.write(line, size);

Formatted I/O operations → converts internal binary representation of data to ASCII char w/c r written to file

↳ C++ supports no. of features used for formatting the output. Features :-

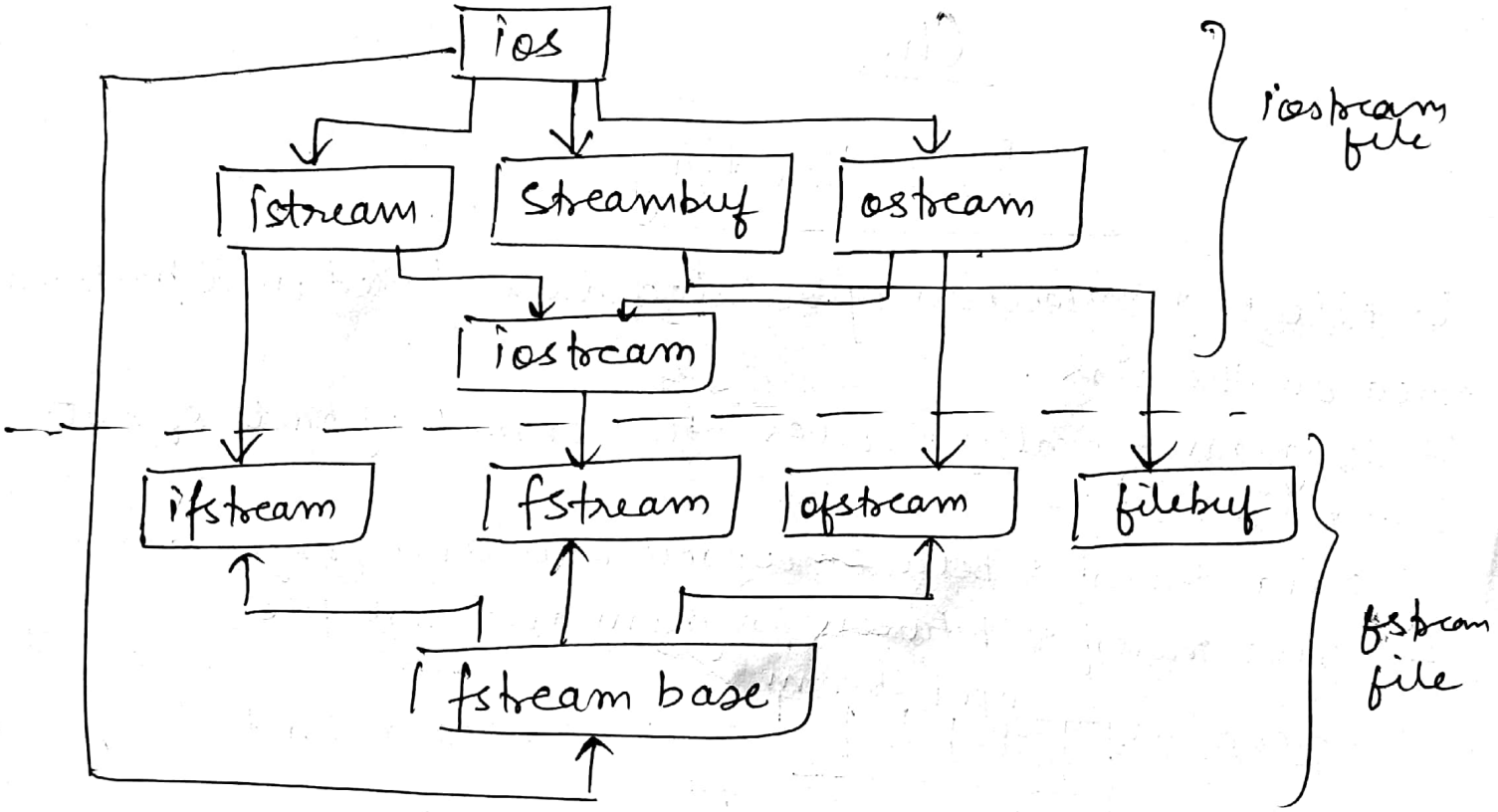
① ios class function and flags
② Manipulators
③ user-defined output function

Scanned by CamScanner

Classes for file stream Operations

I/O system of C++ contains set of classes that define the file handling methods. These include ifstream, ofstream, fstream. These classes are derived from fstreambase and from corresponding iostream class.

- These classes designed to manage disk files, declared in fstream and therefore, we must include this file in any program that uses files.



Details of file stream classes

① **filebuf** :- Purpose to set the file buffers to read, write.
- contains Openprot constant used in open() of file stream classes. Also contain close(), open().

② **fstreambase** :- provide operations common to file streams
- contains open(), close() functions

Scanned by CamScanner

Designing Own Manipulators

```
ostream & manipulator ( ostream & output)
{
    . . .
    . . . (code)
    return output;
}
```

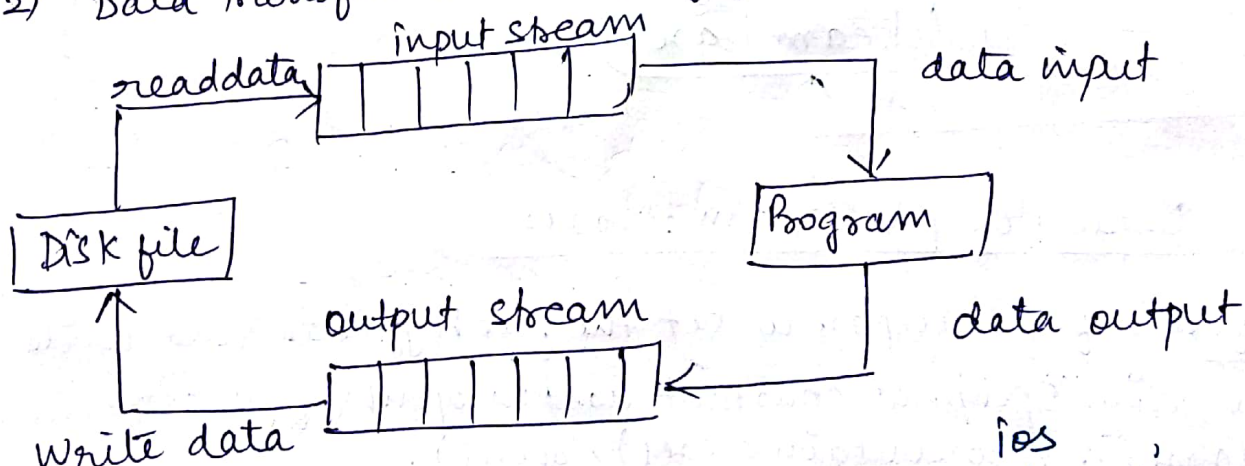
Ch.

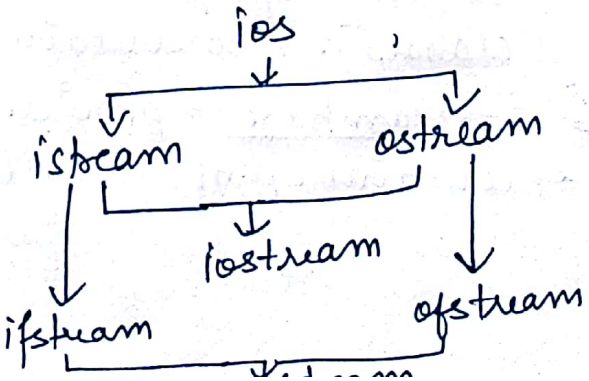
File Handling

↳ File is a collection of related data stored in a particular area on the disk.

↳ Program typically involves either or both of kinds of data communication :-

1) Data transfer between console unit and program
2) Data transfer between program and disk file





Scanned by CamScanner

setf() :- To specify format flags that can control the form of output display (such as left justification and right justification)

→ setf() (stands for set flags).

```
cout.setf(arg1, arg2)
```

eg: `cout.setf(ios::left, ios::adjustfield);`

eg2:
```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(15);
cout<<"TABLE 1"<<"\n";
```

T	A	B	L	E		1	*	*	*	*	*	*	*	*

Managing output with Manipulators

→ Header file 'iomanip' provides set of functions called manipulators which can be used to manipulate output formats.

Manipulator	Meaning	Equivalent
• setw(int w)	• set field width to w	• width()
• setprecision(int d)	• set floating point precision to d	• precision()
• setfill(int c)	• set the fill character to c	• fill()
• setiosflags(long f)	• set the normal flag f.	• setf()
• resetiosflags(long f)	• clear flag specified by f.	• unsetf()
• endl	• insert new line, flush stream	• "\n"

Scanned by CamScanner

ios format functions

1. **width()** :- To specify required field size for display an output value.

```
cout.width(w);
```

w :- field width (no. of columns)

```
cout.width(5);
cout << 543;
```

		5	4	3

② **precision()** :- To specify the no. of digits to be displayed after the decimal point of a float value.

- By default, floating point nos. are printed with 6 digits after decimal point. However, we can specify no. of digits to be displayed after decimal point while printing the floating point nos.

```
cout.precision(d);
```

eg. `cout.precision(3);`
`cout << sqrt(2) << "\n";`

O/P :
1.141

→ Precision retains the setting in effect until it is reset, but width() does not.

③ **Fill()** :- To specify a character that is used to fill the unused portion of a field.

```
cout.fill(ch);
```

eg
```
cout.fill('*');
cout.width(10);
cout << 5250 << "\n";
```

O/P

*	*	*	*	*	*	5	2	5	0

Scanned by CamScanner

ifstream :- Provides input operations.
- inherits get(), getline(), read(), seekg(), tellg(), from istream

④ ofstream : Provides output operations
- contains open(), inherits put(), seekp(), tellp(), write(), from ostream

⑤ fstream: Provides supports for simultaneous input and output operations.
- Inherits all functions from istream, ostream classes through iostream.

Opening and closing a file

A file can be opened in 2 ways:-
① Using constructor function of class (useful for one file)
② using member function open() of class (useful to manage multiple files using 1 stream)

① Using constructor

2 steps:-
a) create file stream object to manage stream using appropriate class. ie to say, class ofstream used to create output stream and file ifstream to create input stream.
b) initialize the file object with desired filename.

```
ofstream outfile("results.txt");
```

This statement opens the file 'results' and attaches it to the output stream outfile.
Similarly,

```
ifstream infile("data.txt");
```

this statement declares infile as an ifstream object, attaches it to the file 'data' for reading (input)

Scanned by CamScanner

outfile.close();

disconnects the file salary from output stream outfile. Object outfile still exists and salary file may again be connected to outfile.

② ~~Using~~ Opening files using open()

```
file-stream-class stream-object;
stream-object.open("filename");
```

Syntax:

Detecting end-of-file

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from file.

while(fin) — while(!fin.eof()); // if not at end of file, continue reading

An ifstream object, such as fin, returns value of 0 if any error occurs in the file operation including end-of-file condition.

Another approach:-

```
if(fin1.eof()!=0) {exit(1); }
```

It returns a non-zero value if the End-of-File (EOF) condition is encountered.

OPEN(): FILE MODES

① ios::out :→ open file for writing only
ios::app :→ append to end-of-file
ios::ate :→ goto end-of-file opening
ios::binary :→ Binary file
ios::in :→ open file for reading only
ios::nocreate :→ open fails if file doesn't exist
ios::noreplace :→ open fails if file already exists
ios::trunc :→ Delete content of file if it exists

Scanned by CamScanner

File Handling

ofstream: Data type represents the output file stream and is used to create files and to write information to files.

ifstream: This data type represents input file stream and is used to read information from files.

fstream: This data type represents the file stream generally, and has capabilities of both ofstream and ifstream which means it can create files, write informations to files, and read information from files

① Writing data into file

```
int main()
{
    char arr[100];
    cout << "Enter name and age" << endl;
    cin.getline(arr, 100);
    ofstream myfile("xyz.txt");
    myfile << arr;
    myfile.close();
    cout << "File write operation performed successfully" <<
        endl;

    return 0;
}
```

Scanned by CamScanner

Object Oriented Programming with C++ Balagurusamy Mc Graw Hill Education

(2) Reading content from file

```
int main()
{  char arr1[100];
   ifstream obj("xyz.txt");
   obj.getline(arr1, 100);
   cout<<"array content :"<< arr1 << endl;
   cout<<" File read operation completed "<< endl;
obj.close();
return 0;
}
```

or

```
int main()
{
    ifstream fin;
    fin.open("out.txt");
    char ch;
    while ( !fin.eof() )
      {
         fin.get(ch);
         cout<<ch;
      }
    fin.close();
    return 0;
}
```

// eof :- Returns true (non zero) if end of file is encountered while reading; otherwise return false (zero)

Scanned by CamScanner

Program to count no. of characters from file

```
#include<fstream.h>
#include<iostream.h>
using namespace std;
int main()
{
    ifstream fin;
    fin.open("out.txt");
     int count = 0;
     char ch;
     while(!fin.eof())
      {
        fin. get(ch);
        count ++;
      }
cout<<"No. of char in files :"<< count ;
  fin. close();
  return 0;
}
```

Prog:- Program to copy contents from one file to another file

```
int main()
{  ifstream fin;
   fin. open("out.txt");
   ofstream fout;
   fout. open("sam.txt");
   char ch;
   while( !fin.eof())
    {  fin. get(ch);
       fout<< ch;
    }
fin. close();
fout. close(); return 0; }
```

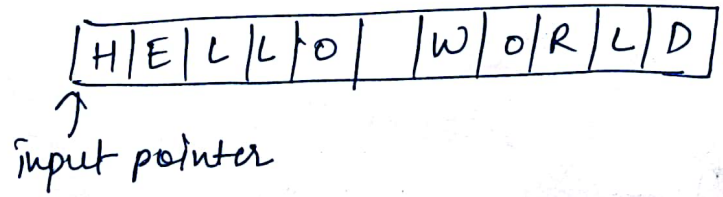
Scanned by CamScanner

File pointers and manipulations

- Each file has 2 associated pointers called as file pointers. One is called input pointers (or get pointer) and other output pointer (or put pointer).
- We can use these pointers to move through the files while reading or writing.
- Input pointer is used for reading content of given file loc^n and output pointer is used for writing to a given file location.
- Each time an input or output operations take place, the appropriate pointer is automatically advanced.

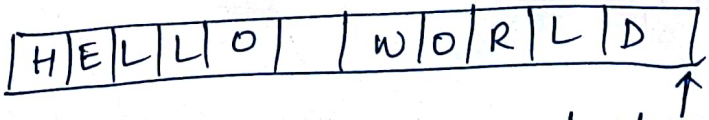
Default Actions

- When we open file in read-only mode, the input pointer is automatically set at beginning so that we can read file from start
- Similarly, when we open a file in write-only mode, existing contents are deleted and output pointer is set at beginning.
- In case, we want to open an existing file to add more data, file is opened in 'append' mode. This move the output pointer to end of file.



input pointer

open for reading only.



output pointer

open in append mode (for writing new data)

output pointer

open for writing data

Scanned by CamScanner

Poly mor...

— means man...

...d to be

...tains at

...uctions

① seekg() : Moves get pointer (input) to specified location (ifstream)

② seekp() : Moves put pointer (output) to a specified location. (ofstream)

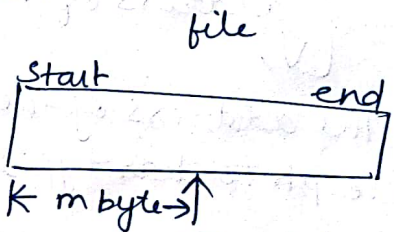
③ tellg() : gives the current position of get pointer (ifstream)

④ tellp() : gives the current position of the put pointer (ofstream)

Eg: infile.seekg(10);

↳ moves the file pointer to byte no. 10.

```
seekg(offset, refposition);
seekp(offset, refposition);
```



offset : no. of bytes the file pointer is to be moved from loc^n specified by parameter refposition.

refposition :- takes ones of the following 3 constants

ios::beg → start of file

ios::cur → current position of pointer

ios::end → end of file

Scanned by CamScanner

Unit III

Sequential I/O operations

① put(), get() :- designed for handling single character at time.

② write(), read() :- designed to write blocks of binary data.

```
infile.read((char *) &V, sizeof(V));
outfile.write((char *) &V, sizeof(V));
```

[V, address of variable, length of that variable]

The address of the variable must be cast to type char* (ie ptr to char type).

Error Handling

Functions	Return value & meaning
① eof()	Returns true (non zero value) if end of file is encountered while reading; otherwise returns false (zero)
② fail()	Returns true when an input or output operation has failed
③ bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred. However if it is false, it may be possible to recover from any other error reported & continue operation
④ good()	Returns true, if no error has occurred. This means, all the above functions are false. If it is true, we can perform I/O operations.

Scanned by CamScanner

ract class

abstract valarray

cifically, C++98 introduced special container called valarray to hold and provide mathematical operations on array efficiently.

* It supports element wise mathematical operations on array efficiently. Various forms of generalized subscript operators, slicing and indirect access.

* As compare to vectors, valarray are efficient in certain mathematical operations than vectors also.

Member functions in valarray class

① apply() :- This function applies manipulation given in its arguments to all valarray elements at once and returns new valarray with manipulated values.

② sum() :- This function returns the summation of all the elements of valarray at once.

③ min() : this function returns the smallest element of valarray.

④ max() :- This function returns the largest element of valarray.

⑤ shift() :- This function returns the new valarray after shifting elements by the number mentioned in its argument. If the number is positive, left-shift is applied. If the number is negative, right shift is applied.

⑥ cshift() :- This function returns new valarray after circularly shifting (rotating) elements by the number mentioned in its argument. If no. is +ve, left-circular shift is applied. If no is neg., right circular shift is applied.

Scanned by CamScanner

Object Oriented ...mming with C++ Balagurusamy Mc Graw Hill Educat

(7) swap() :- This function swaps one valarray with other.

Eg
```
#include<iostream>
#include<valarray>
int main()
{
valarray<int> varr = {10,2,20,1,30};
// valarray<int> varr1;
cout << "The sum:" <<;
cout << varr.sum() << endl;
} return 0;
```

O/P :

The sum : 63

SLICE

(defined in header <valarray>)

Class slice;

* slice is the selector class that identifies subset of valarray
* slice holds three values :-
  1. starting index :- index of first element
  2. stride :- span that separates the elements separated
  3. total no. of values in subset

Object of type slice can be used as indexes with valarray's operator [].

Scanned by CamScanner

bstract ... ber functions

(constructor)	slice constructor
start	Return start of slice
size	Return size of slice
stride	Return stride of slice

Generalised Numeric Algorithm

- It is used to generalized numeric operations and this header describes set of algorithms to perform certain operations on sequence of numeric values.
- It includes common mathematical functions and types, as well as optimized numeric arrays and support for random number generation

Functions

① accumulate :- used accumulate values in range
② adjacent_difference :- used to compute adjacent difference of range
③ inner_product : used to compute cumulative inner product range
④ partial_sum : used to compute partial sums of range
⑤ iota : used to store increasing sequence

Scanned by CamScanner

Page No.
Date

Own Manipulator

```
#include<iomanip>
using namespace std;

ostream &rightArrow(ostream &output)
{
  output<<" name --->";
  return output;
}

istream &getName(istream &input)
{  cout<<"Enter your name"<<endl;
    return input;
}

int main()
{ string name;
  cin>>getName>>name;
  cout<<rightArrow<<name;
  return 0;
}
```

OR

```
using namespace std;
ostream &userManip(ostream &out)
{  out<<"Rs";
    return out;
}
int main()
{ cout<<userManip<<50.00;
}
```

o/p

Rs. 50.0

Scanned by CamScanner