OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-4

Exception Handling

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.

Exception

- 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.
- Note that an exceptional circumstance is not necessarily an error.

Exception Handling

- Exception handling is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program.
- Exception handling mechanism consists of following parts:
  - Find the problem (Hit the exception)
  - Inform about its occurrence (Throw the exception)
  - Receive error information (Catch the exception)
  - Take proper action (Handle the exception)

Exception Handling

C++ consists of 3 keywords for handling the exception:

- **try:** Try block consists of the code that may generate exception. Exception are thrown from inside the try block.
- **throw:** Throw keyword is used to throw an exception encountered inside try block. After the exception is thrown, the control is transferred to catch block.
- **catch:** Catch block catches the exception thrown by throw statement from try block. Then, exception are handled inside catch block.

Exception Handling

Syntax:

```
try
{
        statements;
        … … …
        throw exception;
}
catch (type argument)
{
        statements;
        … … …
}
```

Example:

```
int main ()
{
try
{
        throw 20;
}
catch (int e)
{
        cout << "An exception
        occurred with value  " << e;
}
        return 0;
}
```

Exception Handling

Example:

```
int main()
{
   int x = -1;
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }
    cout << "After catch (Will be executed) \n";
   return 0;
}
```

Output:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

Multiple Catch Exception

Multiple catch exception statements are used when a user wants to handle different exceptions differently. For this, a user must include catch statements with different declaration.

Syntax:

```
try {
        body of try block }
 catch (type1 argument1)
 {   statements;   … … … }
 catch (type2 argument2)
 {   statements;   … … … }
 catch (typeN argumentN)
 {   statements;   … … … }
```

Catch All Exceptions

Sometimes, it may not be possible to design a separate catch block for each kind of exception. In such cases, we can use a single catch statement with ellipsis (...) that catches all kinds of exceptions which are not caught by other handlers.

Syntax:

```
 catch (...)
{
        statements;
        ... ... ...
}
```

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Example

Program to divide two numbers using try catch block.

```
int main()
{
    int a,b;
    cout << "Enter 2 numbers: ";
    cin >> a >> b;
    try
    {
    if (b != 0)
    {
    float div = (float)a/b;
    if (div < 0) throw 'e';
//'e' is not implicitly converted to int
            cout << "a/b = " << div;
    }
    else
            throw b;
}
catch (int e)
{
cout << "Exception: Division by zero";
}
catch (char st)
{
cout << "Exception: Division is less than 1";
}
catch(...)
{
cout << "Exception: Unknown";
}
return 0;
}
```

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Nested try-catch blocks

```
int main()
{
    try {
        try  {
            throw 20;
        }
        catch (int n) {
             cout << "Handle Partially ";
             throw; //Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Output:

```
Handle Partially Handle remaining
```

Throwing exception from a function

Example:

```
void xTest(int test)
{
    cout<<"Inside xTest: "<<test;
    if(test)
            throw test;
}
main()
{
    cout<<"Main Start";
    try{
            cout<<"Inside try block";
            xTest(0);
            xTest(1);
            xTest(2);
    }
        catch (int i)
        {
                cout<<"Caught exception : ";
                cout<<i;
        }
        cout<<"Main End";
    }
```

Output:

```
Main Start
Inside try block
Inside xTest: 0
Inside xTest: 1
Caught Exception : 1
Main End
```

Exception Specification

To specify the set of exceptions that might be thrown as part of the function declaration e.g.

```
returnType functionName(Arg_list) throw(type_list)
{
        :
}
```

This specifies functionName may throw only exceptions of type_list and exceptions derived from these types but no others.

To throw character as well as integer from xTest() use:

```
void xTest(int test) throw (int, char)
```

To throw no type exception from xTest() use:

```
Void xTest(int test) throw()
```

Re- Throwing an exception

Example:

```
void f(int a)
{
try
{
if(a==0)
        throw a;
else
        cout<<“Good”;
}
catch(int)
{
cout<<“Rethrowing”;
throw;
}
}
```

```
main()
{
        try{
        f(0);
        }
        catch(int)
        {
        cout<<“Caught Exception”;
        }
}
```

Output:

```
    Rethrowing
    Caught Exception
```

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Exception & Classes

When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

```
class Test {
public:
   Test() { cout << "Constructor of Test \n"; }
  ~Test() { cout << "Destructor of Test \n"; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

Output:

```
Constructor of Test
Destructor of Test
Caught 10
```

Exception & Classes

- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class. If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints *"Caught Base Exception"*

```
class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try
   {
      throw d;
   }
```

```
catch(Base b)
{
        cout<<"Caught Base Exception";
}
catch(Derived d)
{
       //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
 }
   return 0;
}
```

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Exception & Classes

- In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modified program and it prints *“Caught Derived Exception”*

```
class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try {
       throw d;
   }
```

```
 catch(Derived d)
{
        cout<<"Caught Derived Exception";
  }
catch(Base b)
{
        cout<<"Caught Base Exception";
}
  return 0;
}
```

To be continued…

OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-4

Standard Library, Streams & File

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

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.

Standard Library

A standard library is something that every implementer must supply so that every programmer can rely on it.

The C++ standard library:

- Provides support for language features such as memory management and run time type information
- Supplies information about implementation-defined aspects of the language such as the largest float value
- Supplies functions that can not be implemented optimally in the language itself for every system such as sqrt()
- Supplies facilities that a programmer can rely on for portability such as lists, sort functions & I/o streams
- Provides common foundation for other libraries

Standard Library

The facilities of the standard library are defined in the std namespace and presented as a set of headers.

```
using namespace std;
```

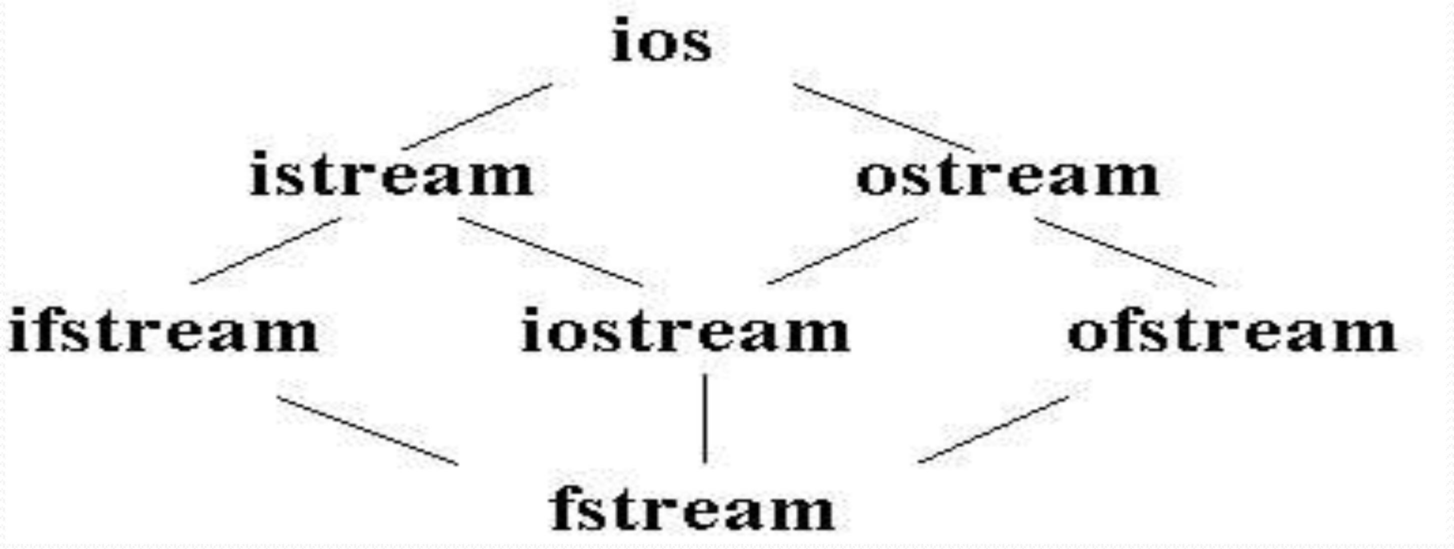
Standard library is a collection of classes and functions which are written in the core language. Header files in the c++ standard library do not end with '.h' extension e.g.

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

Stream I/O Library Header Files

- A stream is a sequence of bytes.
- A stream acts as an interface between program & i/p, o/p device.
- A stream is nothing but a flow of data.
  - ***iostream*** -- contains basic information required for all stream I/O operations
  - ***iomanip*** -- contains information for performing formatted I/O with parameterized stream manipulators
  - ***fstream*** -- contains information for performing file I/O operations
  - ***strstream*** -- contains information for performing in-memory I/O operations (i.e., into or from strings in memory)

Stream Classes for I/O & File Operations



- ios is the base class.
- istream and ostream inherit from ios
- ifstream inherits from istream (and ios)
- ofstream inherits from ostream (and ios)
- iostream inherits from istream and ostream (& ios)
- fstream inherits from ifstream, iostream, and ofstream

C++ Stream I/O -- Stream Manipulators

- Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream.
- Stream manipulators are defined in <iomanip>
- These manipulators provide capabilities for
  - setting field widths,
  - setting precision,
  - setting and unsetting format flags,
  - flushing streams,
  - inserting a "newline" and flushing output stream,
  - skipping whitespace in input stream

C++ Stream I/O -- Stream Manipulators

setprecision (int precision)

- To set floating point precision i.e. number of significant digits to be printed e.g.
  cout << setprecision (2) << 1.2345;
  Output: 1.23 // two significant digits

setw (int width)

- To set the field width (can be used on input or output, but only applies to next insertion or extraction) e.g.
  cout << setw (10) <<12345;
  Output: _ _ _ _ _ 1 2 3 4 5 //5 blanks+5 digits

C++ Stream I/O -- Stream Manipulators

setfill (char fillChar)

- To set the fill character.
  e.g.
  cout << setfill('*') << setw(10)<<12345;
  Output: * * * * * 1 2 3 4 5 //5 stars + 5 digits

endl

- To insert new line and flush stream
  cout<<endl;

C++ Stream I/O - Stream Manipulators

- Various ios *format flags* specify the kinds of formatting to be performed during stream I/O.

- There are member functions (and/or stream manipulators) which control flag settings.

- There are various flags for trailing zeros and decimal points, justification, number base, floating point representation, and so on.

C++ Stream I/O -- Stream Manipulators

setiosflags(long f)

- Sets the format flag e.g.

```
        cout<<setiosflags(ios::scientific)<<1.0000;
Output:   1.0000e+000
```

resetiosflags(long f)

- Clears the flag specified by setiosflag e.g.

```
        cout<<resetiosflags(ios::scientific)<<1.0000;
Output:   1.0000
```

I/O Stream Member Functions

.setf ( )

- Allows the setting of an I/O stream format flag.

- e.g.

```
// To show the + sign in front of positive numbers
cout.setf (ios::showpos) ;

// To output the number in hexadecimal
cout.setf (ios::hex, ios::basefield) ;
```

I/O Stream Member Functions

.precision ( ) ;

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

```
cout.precision (2) ;    // two significant digits
```

.width ( ) ;

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

```
cout.width (4) ;      // field is four positions wide
```

I/O Stream Member Functions

.eof ( ) ;

- Tests for end-of-file condition.
- e.g.

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

.fail ( ) ;

- Tests if a stream operation has failed.
- e.g.

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

I/O Stream Member Functions

.clear ( ) ;

- Normally used to restore a stream's state to "good" so that I/O may proceed or resume on that stream.

- e.g.

```
cin.clear ( ) ;    // allow I/O to resume on a "bad"
                   // stream, in this case "cin",
                   // on which error had previously
                   // occurred
```

Using Manipulators & Member Functions

```
#include <iostream>          // No ".h" (standard header)
#include <iomanip>           // No ".h" (standard header)
using namespace std;         // To avoid "std::"
int main ( )
{
   int a, b, c = 8, d = 4 ;
   float k ;
   char name[30] ;
   cout  <<  "Enter  your name"  <<  endl ;
   cin.getline (name, 30) ;
   cout << "Enter two integers and a float " << endl ;
   cin  >>  a  >>  b  >>  k ;
```

Using Manipulators & Member Functions

```
// Now, let's output the values that were read in
  cout << "\nThank you, " << name <<  ", you entered"
    <<  endl  <<  a  <<   ", "  <<  b  <<  ", and " ;
   cout.width (4) ;
   cout.precision (2) ;
   cout << k << endl ;
//  Control the field and precision another way
   cout <<"\nThank you, " << name << ", you entered"
     << endl  <<  a << ", " <<  b << ", and " << setw (c)
     << setprecision (d) << k << endl ;
}
```

Example Program Output

```
Enter your name
Dr. Gaurav
Enter two integers and a float
12  24  67.85

Thank you, Dr. Gaurav, you entered
12, 24, and 68

Thank you, Dr. Gaurav, you entered
12, 24, and 67.85
```

Creating own manipulator

Syntax:

```
ostream & manipulator_name(ostream &out)
{
    :
    return out;
}
```

How to use it:

```
ostream &unit(ostream &out)
{
    out<<inches;
    return out;
}
```

Example:

```
        cout<<36<<unit;        ➔ 36 inches
```

Own manipulator with multiple operations

Example:

```
ostream &currency(ostream & output)
{
    output<<“Rs”;
    return output;
}
ostream &show(ostream &output)
{
    output.setf(ios::showpos);
    output.setf(ios::showpoint);
    output.fill(‘*’);
    output.precision(2);
    output.setiosflags(ios::fixed)<<setw(10);
    return output;
}
main()
{
cout<<currency<<show<<503.2;
}
```

Output:

```
Rs***+503.20
```

Files & Streams

- A file is a collection of related data stored in a particular area on the disk.
- C++ views each file as a sequence of bytes.
- Each file ends with an ***end-of-file*** marker.
- When a file is ***opened***, an object is created and a stream is associated with the object.
- To perform file processing in C++, the header files **<iostream.h>** and **<fstream.h>** must be included.
- <fstream.h> includes <ifstream> and <ofstream>

How to open a file?

A file can be opened in 2 ways:

- ***Using Constructor:*** Used when only one file is in the stream. *Syntax:*
  FileStreamClass objectName("fileName", open_Mode)

e.g. ofstream outFile("Try.dat",ios::out);

OR

- ***Using member function – open():*** Used to manage multiple files using one stream. *Syntax:*

  ```
  FileStreamClass objectName;
  objectName.open ("fileName", open_Mode)
  ```

  e.g.

  ```
  ofstream outFile;
  outFile.open("clients.dat", ios:out)
  ```

File Open Modes

- **ios:: app** - (append) write all output to the end of file
- **ios:: ate** - data can be written anywhere in the file
- **ios:: binary** - read/write data in binary format
- **ios:: in** - (input) open a file for input i.e. for read only
- **ios:: out** - (output) open a file for output i.e. for write only
- **ios:: trunc** -(truncate) discard the file's content if it exists
- **ios:: nocreate** - if the file does **NOT** exists, the open operation fails
- **ios:noreplace** - if the file exists, the open operation fails

How to close a file in C++?

The file is closed implicitly when a destructor for the corresponding object is called

OR

by using member function *close:*

```
ofstream outFile;
outFile.close();
```

Sequential file (Using Constructor)

```
#include <iostream.h>
#include <fstream.h>
int main()
{
   // ofstream constructor opens file
    ofstream outFile( “File”);
    char name[30]=“Gaurav”;
    int year=2018;
    outFile<<name<<“\n”;
    outFile<<year<<“\n”;
    outFile.close();
```

```
    ifstream inFile(“File”);
    inFile>>name;
    inFile>>year;
    cout<<name;
    cout<<year;
    }
```

Sequential file (Using open())

```
#include <iostream.h>
#include <fstream.h>
int main()
{
   char str[100];
    ofstream outFile;
    outFile.open("Country");
    outFile<<"India\n";
    outFile<<"Nepal\n";
    outFile.close();
    outFile.open("States");
    outFile<<"Delhi";
    outFile<<"Kathmandu";
    outFile.close();
    ifstream inFile;
    inFile.open("Country");
while(inFile)  //(inFile.eof()!=0)
{
    inFile.getline(str,100);
    cout<<str;
}
inFile.close();
inFile.open("States");
while(inFile)  //eof()=0 on
    reaching end of file condition
{
    inFile.getline(str,100);
    cout<<str;
}
inFile.close();
}
```

Reading and printing a sequential file

```
// Reading and printing a sequential file
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
void outputLine( int, const char *, double );
int main()
{
   // ifstream constructor opens the file
   ifstream inFile( "clients.dat", ios::in );
   if ( !inFile ) {
      cerr << "File could not be opened\n";
      exit( 1 );
   }
```

```
    int account;
   char name[ 30 ];
   double balance;
   cout << setiosflags( ios::left ) << setw( 10 ) << "Account“ << setw( 13 ) <<
"Name" << "Balance\n";

   while ( inFile >> account >> name >> balance )
      outputLine( account, name, balance );

   return 0;  // ifstream destructor closes the file
}

void outputLine( int acct, const char *name, double bal )
{
   cout << setiosflags( ios::left ) << setw( 10 ) << acct << setw( 13 ) << name <<
setw( 7 ) << setprecision( 2 ) << resetiosflags( ios::left )
 << setiosflags( ios::fixed | ios::showpoint )
        << bal << '\n';
}
```

File position pointer

- <istream> and <ostream> classes provide member functions for repositioning the *file pointer* (the byte number of the next byte in the file to be read or to be written.)
- These member functions are:

  seekg (seek get) for istream class – Moves get (i/p) pointer to a specified location

  Syntax: **seekg(offset, refPosition);**

Where ***offset*** is no. of bytes, file pointer is to be moved from refPosition.

***refPosition*:**

- ios::beg →start of file
- ios::curr→Current position of pointer
- ios::end → End of file

seekp (seek put) for ostream class – Moves put pointer to a specified location.

seekp(offset, refPosition);

Examples of moving a file pointer

- **inFile.seekg(0)** - repositions the file get pointer to the beginning of the file
- **inFile.seekg(n, ios::beg)** - repositions the file get pointer to the n-th byte of the file
- **inFile.seekg(m, ios::end)** -repositions the file get pointer to the m-th byte from the end of file
- **inFile.seekg(0, ios::end)** - repositions the file get pointer to the end of the file
- The same operations can be performed with <ostream> member function - ***seekp.***

Member functions tellg() and tellp()

- Member functions **tellg** and **tellp** are provided to return the current locations of the get and put pointers, respectively.

  long location = inFile.tellg();
- To move the pointer relative to the current location use:

  ios::cur
- inFile.seekg(n, ios::cur) - moves the file get pointer n bytes forward from current position of pointer.

Sequential file-I/O operations (put() & get())

```
int main()
{
    char str[100];
    cin>>str;
    int len=strlen(str);
    fstream file;
    file.open("Try",ios::in|ios::out);
    for(int i=0;i<len;i++)
            file.put(str[i]);
    file.seekg(0);   //Go to start
    char ch;
    while(file)
    {
         file.get(ch);
         cout<<ch;
      }
}
```

Sequential file - write() & read()

Used to handle data in binary form. Syntax:

- inFile.read((char*)&var,sizeof(var));
- outFile.write((char*)&var,sizeof(var));

```
main()
{
float arr[4]={1.2,2.3,3.4,4.5,5.6};
ofstream outfile;
outfile.open(“Try”);
outfile.write((char*)&arr,sizeof(arr));
outfile.close();
arr={0,0,0,0};
ifstream infile;
infile.open(“Try”);
infile.read((char*)&arr,sizeof(arr));
for(i=0;i<4;i++)
cout<<arr[i];
infile.close();
}
```

Sequential file – Reading & Writing Class Object

Data members are written to disk file not the member function

```
class ABC
{
char name[50];
int roll;
public:
    void GetData()   {
    cin>>name>>roll;  }
    void Display()    {
    cout<<name<<roll;
}};

main()
{
    ABC obj[5];
    fstream file;
file.open("Try.DAT", ios::in|ios::out);
cout<<"Enter Data: ";
for(int i=0;i<5;i++)       {
obj[i].GetData();
file.write((char*)&obj[i],sizeof(obj[i]));
}
file.seekg(0);
cout<<"Output : ";
for(int i=0;i<5;i++)       {
file.read((char*)&obj[i],sizeof(obj[i]));
obj[i].Display();
}
file.close();
}
```

Updating a sequential file

- Data that is formatted and written to a sequential file **cannot be modified easily** without the risk of destroying other data in the file.
- If we want to modify a record of data, the new data may be longer than the old one and it could overwrite parts of the record following it.

Problems with sequential files

- Sequential files are inappropriate for so-called “instant access” applications in which a particular record of information must be located immediately.
- These applications include banking systems, point-of-sale systems, airline reservation systems, (or any data-base system.)

Random access files

- Instant access is possible with random access files.
- Individual records of a **random access file** can be accessed directly (and quickly) without searching many other records.

<ostream> function *write*

- The <ostream> member function *write* outputs a fixed number of bytes beginning at a specific location in memory to the specific stream.
- When the stream is associated with a file, the data is written beginning at the location in the file specified by the “put” file pointer.
- The *write* function expects a first argument of type *const char* *, the address of the *object is converted* to a *const char* *. The second argument of *write* is an integer of type size_t specifying the number of bytes to written. *Thus* the *sizeof(object)*.

<istream> function *read*

The <istream> function inputs a specified (by sizeof(object)) number of bytes from the current position of the specified stream into an object.

Operations on random access file

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
classs ABC
{
    char name[20];
    int roll;
    public:
            void GetData();
            void Display();
};
main()
{
ABC obj;
fstream inoutfile;
inoutfile.open("Try.DAT", ios::ate|
ios::in|ios::out|ios::binary);
```

```
inoutfile.seekg(0,ios::beg);
while(inoutfile.read((char*)&obj,
            sizeof(obj)))      {
    obj.Display();
}
inoutfile.clear();

//Adding Data
obj.GetData();
inoutfile.write((char*)&obj,sizeof(obj));

//Display the appended file
inoutfile.seekg(0);
while(inoutfile.read((char*)&obj,sizeof(
    obj)) {
    obj.Display();
}
```

Operations on random access file

```
//Find no. of objects in the file
int last=inoutfile.tellg(); //current
    position of get pointer which
    will be last position
int n=last/sizeof(obj); //no. of
    objects

//Modify details
cout<<"Enter object no. to be
    updated : ";
int objNo;
int location=(objNo-1) *
    sizeof(obj);
if(inoutfile.eof())
    inoutfile.clear();  //To turnoff
    eof flag
inoutfile.seekp(location);
cout<<"Enter value for object : ";
obj.GetData();

inoutfile.write((char*)&obj,sizeof(obj));
//Display updated file
inoutfile.seekg(0);
while(inoutfile.read((char*)&obj,sizeof(
    obj))  {
    obj.Display();
}
inoutfile.close();
}
```

To be continued…

OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-4

Standard Template Library

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

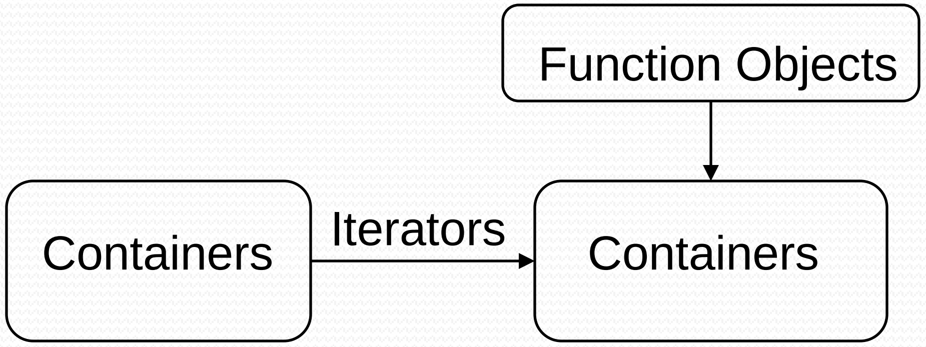
Unit-IV

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.**

Standard Template Library(STL)

STL is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

- STL provides a collection of classes to make C++ programming more efficient
  - Algorithms, Containers, and Iterators
  - Function objects, Allocators, and Adapters



Components of STL

- **Containers** - Generic class templates for storing collection of data. Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, etc.
- **Algorithms** - Generic function templates for operating on containers. Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
- **Iterators** - Generalized 'smart' pointers that facilitate use of containers. They provide an interface that is needed for STL algorithms to operate on STL containers.

Containers

Three types of containers

- *Sequence containers:*
  - linear data structures such as vectors and linked lists
  - vector (dynamic table)
  - dequeue (double ende queue, using dynamic table)
  - list (double linked one)
  - (arrays and strings not STL containers, but operable by STL algorithms)
- *Associative containers:*
  - non-linear containers such as hash tables
  - set (using binary tree)
  - multiset
  - map, multimap
- *Container adapters:*
  - constrained sequence containers such as stacks and queues

Iterators

- Iterators are pointers to elements of first-class containers
  - Type ***const_iterator*** defines an iterator to a container element that *cannot* be modified
  - Type ***iterator*** defines an iterator to a container element that *can* be modified
- Used as arguments to algorithms
- Containers provide several iterators
  - Begin, End, Reverse Begin, Reverse End
- 5 categories – each has specific operations
  - Input, Output, Forward, Bidirectional, Random

Algorithms

- Operate on containers using iterator interface
  - generic, but not as fast as containers' methods
  - slow on some combinations of algorithm/container
  - may operate on various containers at the same time
- 4 categories
  - Non-modifying

    For each, Find, Count, Equal, Search, etc.
  - Mutating

    Copy, Generate, Reverse, Remove, Swap, etc.
  - Sorting Related

    Sort, Stable sort, Binary search, Merge, etc.
  - Numeric

    Accumulate, Inner product, Partial sum, Adjacent difference

Function Object as Algorithm Argument

- Used by some algorithms
  - Applied to each element in input by For each
  - Used to create new elements by Generate
- Predicates - Function Objects that return bool
  - Unary and Binary predicates used to test equality
  - Find If uses unary, Sort uses binary
- function objects are
  - easily optimizable by compiler
  - may have class member variables,internal state" passed by the constructor argument
  - we may have many function objects of the same class

Vector container

Provides an alternative to the built in array. A vector is self grown. Use It instead of the built in array!

- Vector – similar to an array
  - Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector
- Vector vs array
  - Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence

Syntax:

vector<of type>

Example :

vector<int> - vector of integers.

vector<string> - vector of strings.

vector<Shape> - vector of Shape objects where Shape is a user defined class.

Vector container

Example :

```
#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;
} // use iterator to access the values
vector<int>::iterator v = vec.begin();
while( v != vec.end())
{
cout << "value of v = " << *v << endl;
v++;
}
return 0; }
```

Strings

- In C, char * was used to represent a string.
- The C++ standard library provides a common implementation of a string class abstraction named string.
- To use the string type simply include its header file.

```
        #include <string>
        string str = “some text”;
                        or
        string str(“some text”);
```

other ways:

```
        string s1(7,‘a’);
        string s2 = s1;
```

String Operations

String Length: The length of string is returned by its size() operation.

```
#include <string>
string str = "something";
cout << "The size of "<< str << "is " << str.size() <<
"characters." << endl;
```

String concatenation: Concatenating one string to another is done by the '+' operator.

```
string str1 = "Here ";
string str2 = "comes the sun";
string concat_str = str1 + str2;
```

Valarray slice

A *valarray slice* is defined by a *starting index*, a *size*, and a *stride*.

- The *starting index* (start) is the index of the first element in the selection.
- The *size* (size) is the number of elements in the selection.
- The *stride* (stride) is the span that separates the elements selected.
  Therefore, a slice with a stride higher than 1 does not select contiguous elements in the valarray;
- For example, slice(3,4,5) selects the elements 3, 8, 13 and 18.

Valarray slice

Example:

```
#include <iostream> // std::cout
#include <cstddef> // std::size_t
#include <valarray> // std::valarray, std::slice
using namespace std;
int main () {
    valarray<int> foo (12);
    for (int i=0; i<12; ++i)
        foo[i]=i;
    valarray<int> bar = foo[slice(2,3,4)];
    cout << "slice(2,3,4):";
    for (size_t n=0; n<bar.size(); n++)
      cout << ' ' << bar[n]<<'\n';
    return 0;
}
```

Output:

```
slice(2,3,4): 2 6 10
```

Generalized Numeric Algorithm

Some of the Numeric algorithms in STL are:

- ***iota Method:*** assigns all the successive elements in range [first, last] as an incremented value of the element itself. Syntax is:
  - iota(iterator first, iterator last, int value).
- ***accumulate Method:*** performs operation **op** on all the element in the range [first, last] and stores the result into the container result. Syntax is:
  - accumulate(iterator first, iterator last, object_type result, binaryoperator op)
- ***partial_sum Method***: assigns every element in the range starting from iterator **result** of the operation **op** on the successive range in [first, last]. Syntax is:
  - partial_sum(iterator first, iterator last, iterator result, binary_operation op)

iota method

```
#include<iostream>
#include<numeric>
#include<vector>
using namespace std;
int main()
{
vector<int> v(10); /* now vector v is : 0,0,0,0,0,0,0,0,0,0 */
iota(v.begin(), v.end(), 10 ); /* now the vector v is
   10,11,12,13,14,15,16,17,18,19 */
}
```

Accumulate method

```
#include<iostream>
#include<numeric>
#include<vector>
using namespace std;
int myoperator(int a, int b )
{
return a*b;
}
int main() {
    vector<int> v;
    for(int i = 0 ; i < 10; i++) {
            v.push_back(i);
    }
/* now vector v is :
    0,1,2,3,4,5,6,7,8,9 */
    int result;

accumulate(v.begin(), v.end(), result) ;
/* as no operator is specified,
accumulate add all the elements
between v.begin() and v.end() and store
the sum in result */
/* now result = 45 */

accumulate(v.begin(), v.end(), result,
myoperator) ;
/* applies myoperator on all the
elements in the range v.begin() and
v.end() and store them in result */
/* now result = 9! */
}
```

partial_sum method

```
#include<iostream>
#include<numeric>
#include<vector>
using namespace std;
int myoperator(int a, int b) {
    return a*b;
}
int main() {
    int a[] = {1,2,3,4,5};
    vector<int> v (a,a+5);
    vector<int> v2; /* vector v is 1,2,3,4,5 */
    v2.resize(v.size());
    partial_sum(v.begin(), v.end(), v2.begin());
    /* now v2 is : 1,3,6,10,15 */
    /* sum of the successive range in v.begin() and v.end() */
    partial_sum(v.begin(), v.end(), v2.begin(), myoperator);
    /* now v2 is : 1,2,6,24,120 */
}
```

End of Unit-IV…

Syllabus Finished…