Concept of const, static, friend & inner class

Const Data Member in Class

- Data members of a class could be made constant.
- Constant data members must be initialized through constructors using constructor initialization list.
- Use of constant inside a class means, “It will be constant for the lifetime of the Object”
- Each object will have a different copy of the constant data member probably having different value.

Initializing Const Data Member of Class

```
class Data {
    int val;
    const int someConst;
public:
    Data() : someConst(0) { // constructor initializer syntax
        val = 0;
    }
    Data( int val, int c ) : someConst(c) {
        this->val  = val;
    }
};
```

Const Object & Const Member Function

- To create constant object use const keyword
  - const Pixel p1(2,3); // statement in main
- No data members of Constant object could be changed.
- const objects can invoke const member function, which guarantees that no data members of the object will be changed.
- Characteristic of const functions is it is 'Read Only' function.
- Constant functions can not write data members.
- To make a function constant place the const specifier after argument list in definition and declaration of function.
- Any function which is not going to modify data members should be made constant.
- Constant functions can be invoked by non constant objects also.
- Constructor and destructor can not be made const.

Example

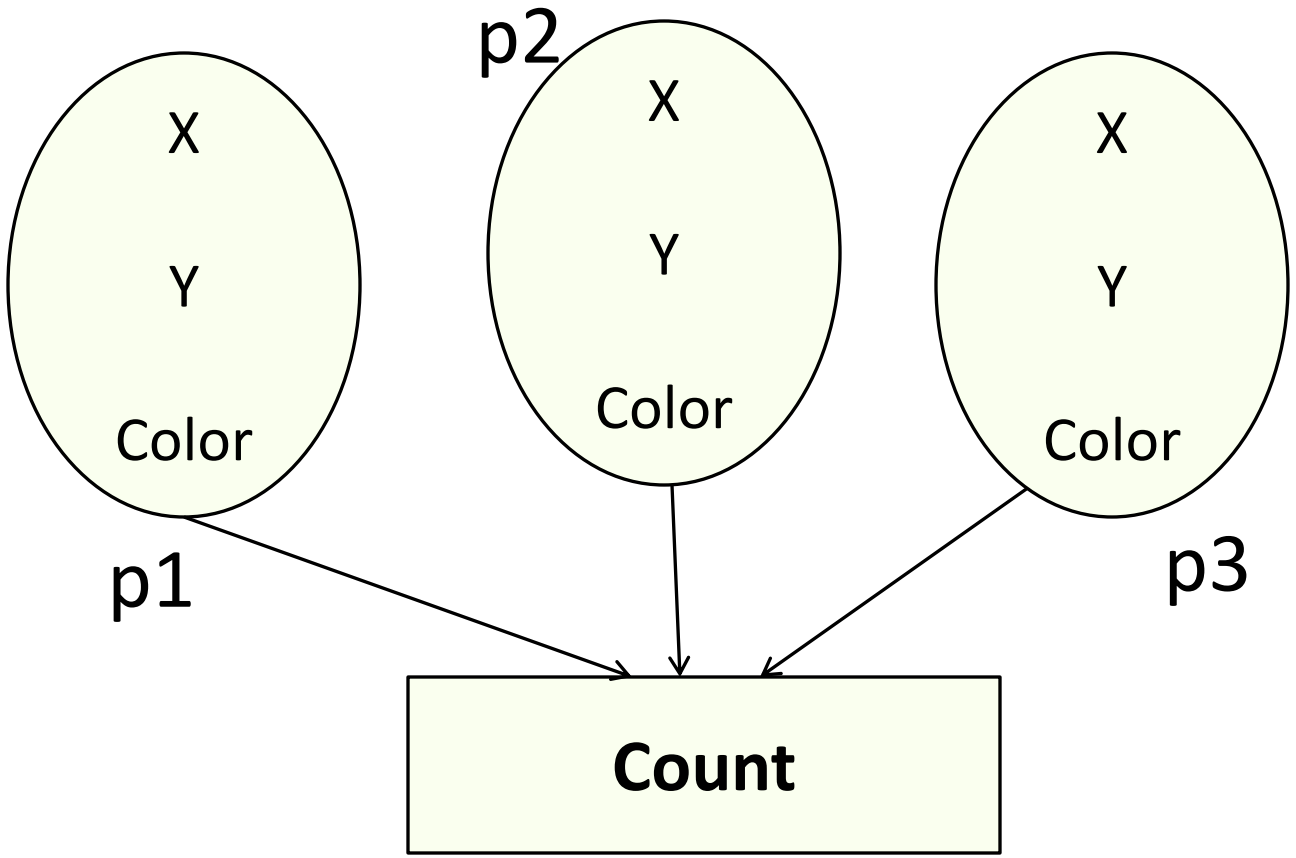
```
class Integer {
    int i;
public:
    Integer( int i ) {
        this->i = i;
    }
    void setI( int i ) {
        this->i = i;
    }
    int getInt() const; // const keyword makes the function constant
};

//getInt can not make changes to data member I as it is const function

int Integer::getInt() const { //const keyword must be used in def also
    return i;
}
int main() {
    const Integer i1( 10 );
    i1.getInt();
    //i1.setInt(); Error can not execute non const fun on const object
    return 0;
}
```

Static Data Member

- Useful when all objects of the same class must share a common item of information.
- Data to be shared by all objects is stored as static data members.
- There is single piece of storage for static data members.
- It's a class variable. Static data members could be made private, public or protected.
- Static data members are class members, they belong to class and not to any object.
- The static data member should be created and initialized before the main() program begins.



- A single copy of a static member is created per class. All objects share the same copy of static variable.
- Compiler will not allocate memory to static for each object.
- If static data members are declared and not defined linker will report an error.
- Static data members must be defined outside the class. Only one definition for static data members is allowed.

Static Member Function

Static member functions can access static data members only.

Static member function is invoked using class name

- *class name : : function name()*

Pointer ***this*** is never passed to a static member function

Can be invoked before creation of any object.

```
class ABC
{
   private:
       static int ref;
       static ABC* self;
       ABC()
       {
        cout<<"\n constructor..";
       }

   public:
       static ABC* getInstance();
       static int getref();
};

int ABC::getref()
{
       return ref;
}
```

```
ABC* ABC::getInstance()
{
   if (self == 0)
   {
     //  ref=1;
                self = new ABC();
   }
   ref=ref+1;
   return self;
}


ABC* ABC::self;
int ABC::ref=0;
```

```
int main()
{
   //ABC obj;
   ABC* ptr = ABC::getInstance();
   ABC* sptr = ABC::getInstance();
   ABC* pptr = ABC::getInstance();

   std::cout << ptr << std::endl;
   std::cout << sptr << std::endl;
   std::cout << ABC::getref();
}
```

Friend class and functions

A class grants access privileges to its friends.
Friend class can access private data members of class.

```
class A{
  int data;
  public:
     void setdata() {
            cout<<"Enter a number";
            cin>>data;
      }
      void display() {
            cout<<"Value of A class Data from class A function
"<<data<<endl;
      }
      friend class B;
};
```

```
class B{
      public:
            void display(A obj)
            {
                  cout<<"Value of A Class
data from class B function
"<<obj.data<<endl;
            }
};
```

```
int main() {
      A obj;
      B obj1;
      obj.setdata();
      obj.display();
      obj1.display(obj);
}
```

```
class B;
class A {
  int Adata;
  public:
    void getdata()
    {
      cout<<"Enter value";
      cin>>Adata;
    }
    void display()
    {
      cout<<"Value of Adata "<<Adata<<endl;
    }
    void friend swap(A &, B &);
};
```

```
class B{
        int Bdata;
        public:
                void getdata()
                {
                        cout<<"Enter value of B class";
                        cin>>Bdata;
                }
                void display()
                {
                        cout<<"Value of Bdata "<<Bdata<<endl;
                }
                void friend swap(A &obj1, B &obj2);
};
```

```
void swap(A &obj1, B &obj2)
{   int t;
    t=obj1.Adata;   obj1.Adata=obj2.Bdata;    obj2.Bdata=t;
    obj1.display();
 }
```

Friendship rules

The privileges of friendship aren't inherited.

> Derived classes of a friend aren't necessarily friends. If class **myClass** declares that class **Base** is a friend, classes derived from **Base** don't have any automatic special access rights to **myClass** objects.

The privileges of friendship aren't transitive.

> A friend of a friend isn't necessarily a friend. If class **Tom** declares class **Jerry** as a friend, and class **Jerry** declares class **Tuffy** as a friend, class **Tuffy** doesn't necessarily have any special access rights to **Tom** objects.

The privileges of friendship aren't reciprocal.

> If class **Tom** declares that class **Jerry** is a friend, **Jerry** objects have special access to **Tom** objects but **Tom** objects do not automatically have special access **Jerry** objects.

Inner class / Nested class

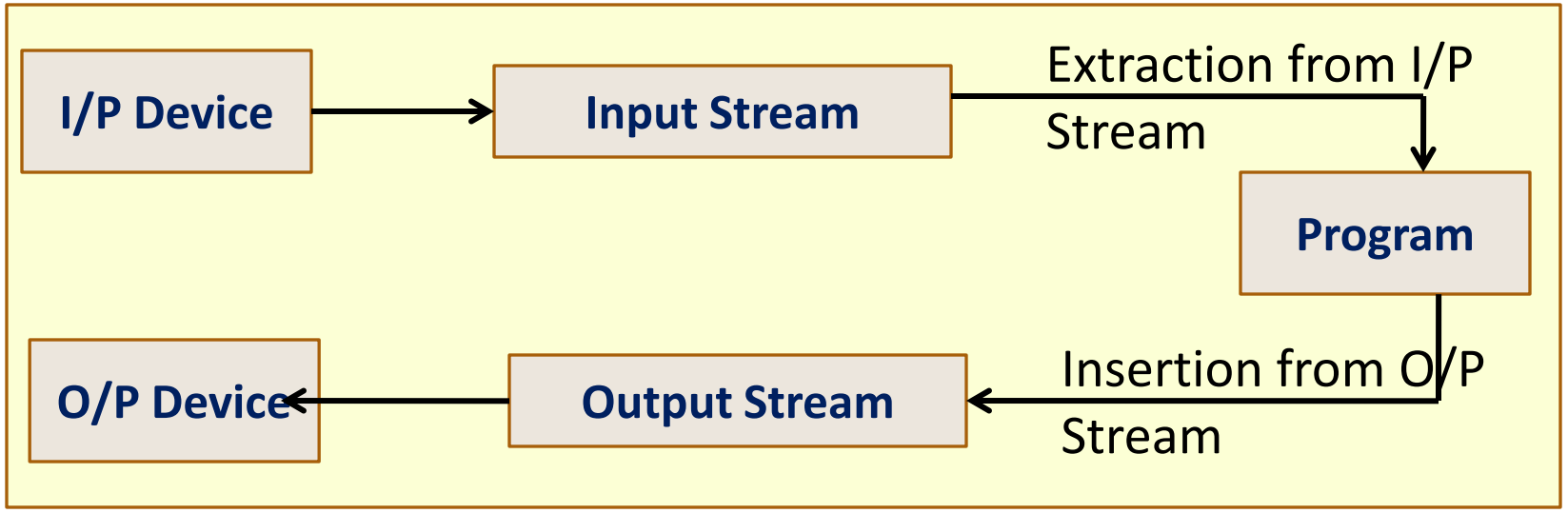
An inner class is a class which is declared in another outer class.
The inner class is a member and as such has the same access rights as any other member.
The members of an outer class have no special access to members of inner class, the usual access rules shall be obeyed.

```
class outer {     int outer_i;
  public:
     outer() { outer_i=20;}
     void disp_outer() { cout<<"\nouter_i = "<<outer_i;}

     class inner {   int inner_i;
       public :
       inner() { inner_i = 30;}
       void disp_inner(outer o) {
          cout<<"\n outer_i from inner class = "<<o.outer_i;
          cout<<"\n inner_i = "<<inner_i; }
     };
};
```

```
int main()
{
    outer o;
    outer::inner obj;
    obj.disp_inner(o);
    return 0;
}
```

I/O and File Handling



Data Stream- Console IO

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

File

A file is a collection on information, usually stored on a computer's disk. Information can be saved to files and then later reused.

File Name-

- All files are assigned a name that is used for identification purposes by the operating system and the user.

File Name and Extension	File Contents
MYPROG.BAS	BASIC program
MENU.BAT	DOS Batch File
INSTALL.DOC	Documentation File
CRUNCH.EXE	Executable File
BOB.HTML	HTML (Hypertext Markup Language) File
3DMODEL.JAVA	Java program or applet
INVENT.OBJ	Object File
PROG1.PRJ	Borland C++ Project File
ANSI.SYS	System Device Driver
README.TXT	Text File

File Handling Steps

Using a file in a program is a simple three-step process

- The file must be opened. If the file does not yet exits, opening it means creating it.
- Information is then saved to the file, read from the file, or both.
- When the program is finished using the file, the file must be closed.

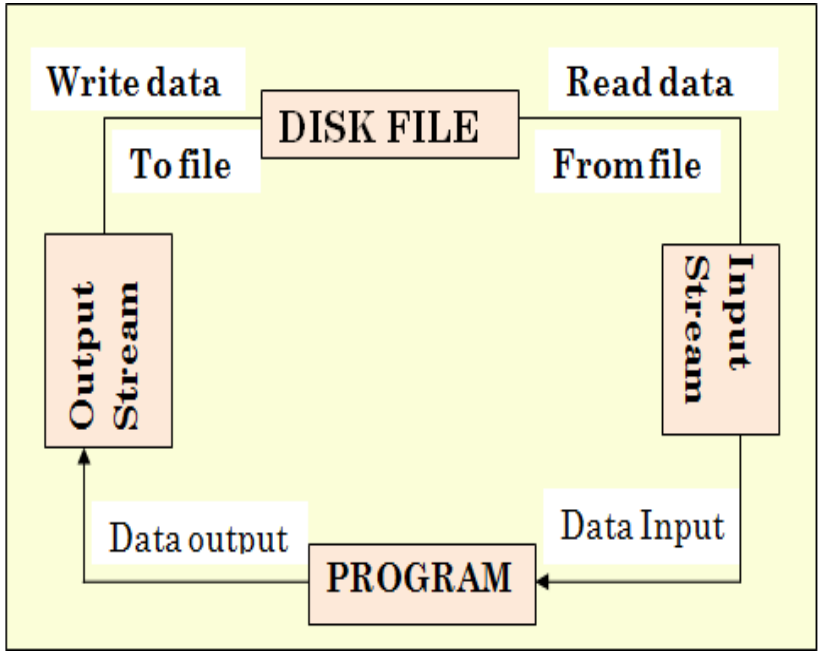
Using Input / Output Stream

Streams act as an interface between files and programs. In C++ . A stream is used to refer to the flow of data from a particular device to the program's variables.

Every stream is associated with a class having member functions and operations for a particular kind of data flow.

- File -> Program (Input stream) - reads
- Program -> File (Output stream) – write

All designed into fstream.h and hence needs to be included in all file handling programs.


Write data
DISK FILE
Read data
To file
From file
Output Stream
Input Stream
Data output
Data Input
PROGRAM

Using Input / Output File

stream - a sequence of characters

- interactive (iostream)
  - **istream** - input stream associated with **keyboard.**
  - **ostream** - output stream associated with **display.**
- file (fstream)
  - **ifstream** - defines new input stream (normally associated with a file).
  - **ofstream** - defines new output stream (normally associated with a file).

- Stream of bytes to do input and output to different devices.
- Stream is the basic concepts which can be attached to files, strings, console and other devices.
- User can also create their own stream to cater specific device or user defined class.

Opening and Closing File

Before data can be written to or read from a file, the file must be opened.

- ifstream inputFile;

inputFile.open("customer.dat");//opens file in read mode

- ofstream outputFile;

outputFile.open("customer.dat");//opens file in write mode

- fstream iofile;

iofile.open("customer.dat", filemode);// opens file in specified mode

Closing File- A file should be closed when a program is finished using it

- inputFile.close();
- outputFile.close();
- Iofile.close();

File Modes

ios::app	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
ios::ate	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
ios::binary	Binary mode. When a file is opened in binary mode, information is written to or read from it in pure binary format. (The default mode is text.)
ios::in	Input mode. Information will be read from the file. If the file does not exist, it will not be created and the open function will fail.
ios::nocreate	If the file does not already exist, this flag will cause the open function to fail. (The file will not be created.)
ios::noreplace	If the file already exists, this flag will cause the open function to fail. (The existing file will not be opened.)
ios::out	Output mode. Information will be written to the file. By default, the file's contents will be deleted if it already exists.
ios::trunc	If the file already exists, its contents will be deleted (truncated). This is the default mode used by ios::out.

Writing/Reading Data From/To File

Writing data to file-

- The stream insertion operator (<<) may be used to write information to a file.

  outputFile << “I love C++ programming !”

- File output may be formatted the same way as screen output.

Reading Data From File-

- The stream extraction operator (>>) may be used to read information from a file.

End of File Detetcion-

- The eof() member function reports when the end of a file has been encountered.

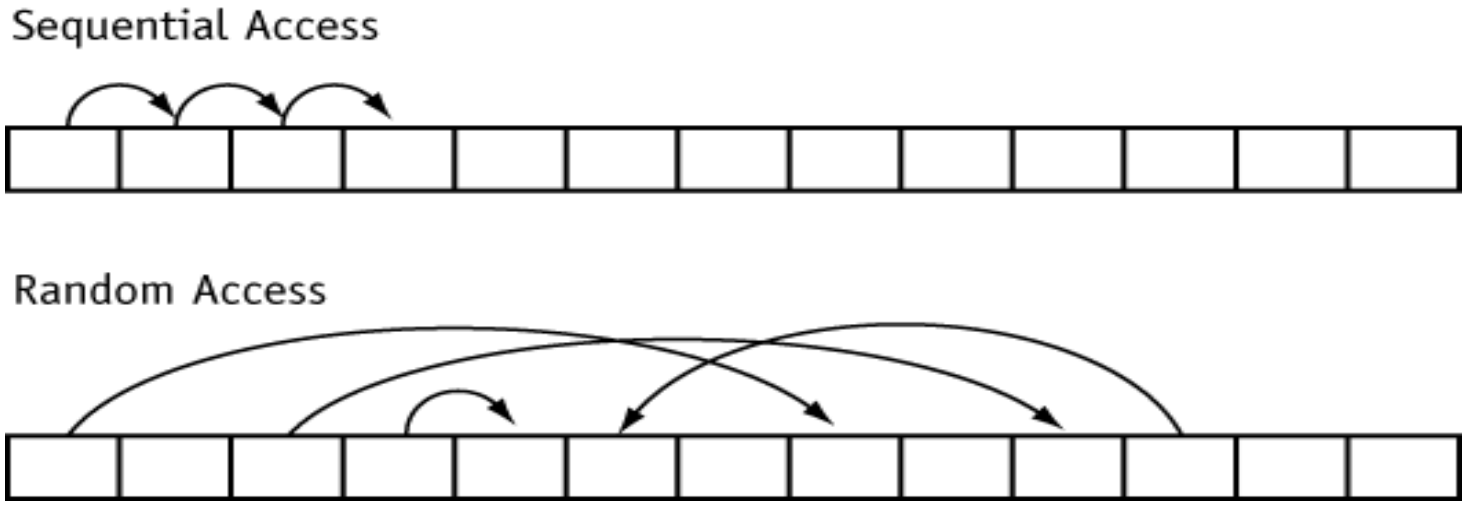
```
if (inFile.eof())
  inFile.close();
```

Random Access File

Random Access means non-sequentially accessing information in a file.

Different modes to operate Random AccessFile-

- ios::beg : The offset is calculated from the beginning of the file.
- ios::end : The offset is calculated from the end of the file.
- ios::cur : The offset is calculated from the current position.



seekp() and seekg()

Statement	How it affects the position
File.seekp(32L, ios::beg);	Sets the write position to the 33rd byte (byte 32) from the beginning of the file.
file.seekp(-10L, ios::end);	Sets the write position to the 11th byte (byte 10) from the end of the file.
file.seekp(120L, ios::cur);	Sets the write position to the 121st byte (byte 120) from the current position.
file.seekg(2L, ios::beg);	Sets the read position to the 3rd byte (byte 2) from the beginning of the file.
file.seekg(-100L, ios::end);	Sets the read position to the 101st byte (byte 100) from the end of the file.
file.seekg(40L, ios::cur);	Sets the read position to the 41st byte (byte 40) from the current position.
file.seekg(0L, ios::end);	Sets the read position to the end of the file.

tellp() and tellg()

tellp returns a long integer that is the current byte number of the file's write position.

tellg returns a long integer that is the current byte number of the file's read position.

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{
        char ch;
        ofstream f1("first.txt");

        f1<<"Hi";
        f1<<" Hello";
        f1<<" Good Afternoon";
        f1.put('C');

        f1.close();
```

```
        ifstream f2("first.txt");
        while((!f2.eof()))
        {
                ch=f2.get();
                cout<<ch;
        }
        f2.close();
}
```

Program to write to file

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{
	char arr[]="Infoway Technologies Pvt. Ltd., Pune";
	ofstream f1("text1.txt");
   f1<<arr;
/*	or
   for(int i=0;arr[i]!='\0';i++)
	{
		f1.put(arr[i]);
	}
*/	f1.seekp(5,ios::beg);
	f1.put('P');
	f1.put('p');
}
```

Program to read from file

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{
	char ch;
	ifstream f1("text1.txt");
	f1.seekg(2,ios::beg);
	while(f1)
	{
		f1.get(ch);
		cout<<ch;
	/*	if(ch=='E')
		  f1.seekg(1,ios::cur);
	f1.get(ch);
		cout<<ch;
	 */}
}
```

Binary / Record File

```
#include<fstream>
#include<iostream>
using namespace std;
class student{
   int rno;
   char name[15];
   public:
     void set_data()
    {
        cout<<"Enter Rno";
        cin>>rno;
        cout<<"\nEnter Name";
        cin>>name;
    }
    void display()
   {
      cout<<"Rno: "<<rno;
      cout<<"Name: "<<name<<endl;
    }
};
```

```
int main()
{
   ofstream f1;
   ifstream f2;
   f1.open("mydata.dat");
   student obj1,obj2;
   obj1.set_data();
   f1.write((char *)&obj1,sizeof(obj1));
   obj1.display();
   f1.close();

   f2.open("mydata.dat");
   f2.read((char *)&obj2,sizeof(obj2));
   obj2.display();
   f2.close();
}
```

Exception Handling C++

What are exceptions

- Exceptions are runtime anomalies that a program may detect
- e.g.
  - Division by 0
  - Access to an array outside it's bounds
  - Exhaustion of the free memory on heap

Exception Handling

- C++ provides built in features to raise and handle exceptions.
- These language features activates a runtime mechanism to communicate exceptions between two unrelated portions of C++ program.
- C++ exception handling is built upon three keywords: **try, catch,** and
- **throw.**
- try block is a block surrounded by braces in which exception may be thrown
- A catch block is the block immediately following a try block, in which exceptions are handled

Syntax -

```
try{
    // something unusual but still predictable
}
catch (out of memory) {
    // take some action
}
catch (File not found) {
    // take other action
}
```

While using Exception Handling …

- **Note**
  - When an exception is raised, program flow continues after catch block.
  - Control never comes back to the point from where exception is thrown
  - Memory leakage in context with exception when object is created on heap.
- Duplicate cleanup code that is common to both normal and exceptional path of control

```
// First Demo
int main() {
    cout << "Start\n"; try {
    // start a try block
    cout << "Inside try block\n";
    throw 100; // throw an error
    cout << "This will not execute";
    }
    catch (int i) { // catch an error
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }
    cout << "End";
    return 0;
}
}
```

Throwing an Exception from function

- If a function is throwing an exception which is not handled locally, it will be escalated to the calling function. If the calling function is not handling the exception, it will be escalated to the next outer scope, till the time it is not handled.

- If no handler is found then program is terminated, by invoking terminate() function.

- We can define our own terminate() function, and set it using the set_terminate() void myTerminate() {
  // do whatever required
  }
  void main () { set_terminate(myTerminate);
  // regular execution
  }

set_terminate is defined in <exception>

Throwing an Exception from function

```
void fun( int num )
{
   if( num <= 0 )
      throw num;
   else
      throw ‘p’;
   cout<<"num = "<<num<<endl;
}
```

```
int main ()
{
   try {
      fun( 0 );
   }
   catch(int excNum ) {
      cout<<"exception generated: "<<excNum<<endl;
   }
   catch(char ch) {
      cout<<"char exception generated: "<<ch;
   }
   return 0;
}
```

Multiple catch blocks

- Execution is similar to switch-case
- Once a matched catch block signature is found, other catch blocks are not executed
- Order of catch blocks is important. Specific first and general at last.
- Most general is Catch everything indicated by catch(…)

If Exceptions Are Ignored…

- If exceptions are ignored or not handled properly, the program is terminated.
- This will happen in cases where:
  - An exception is thrown out of a **try** block.
  - No appropriate catch block has been defined to handle an exception.
- In such a case, the standard termination function - **terminate()** is executed.
  - The function executes the **abort()** function.

Throwing Exceptions of Class Type

You can create your own classes to represent exception.

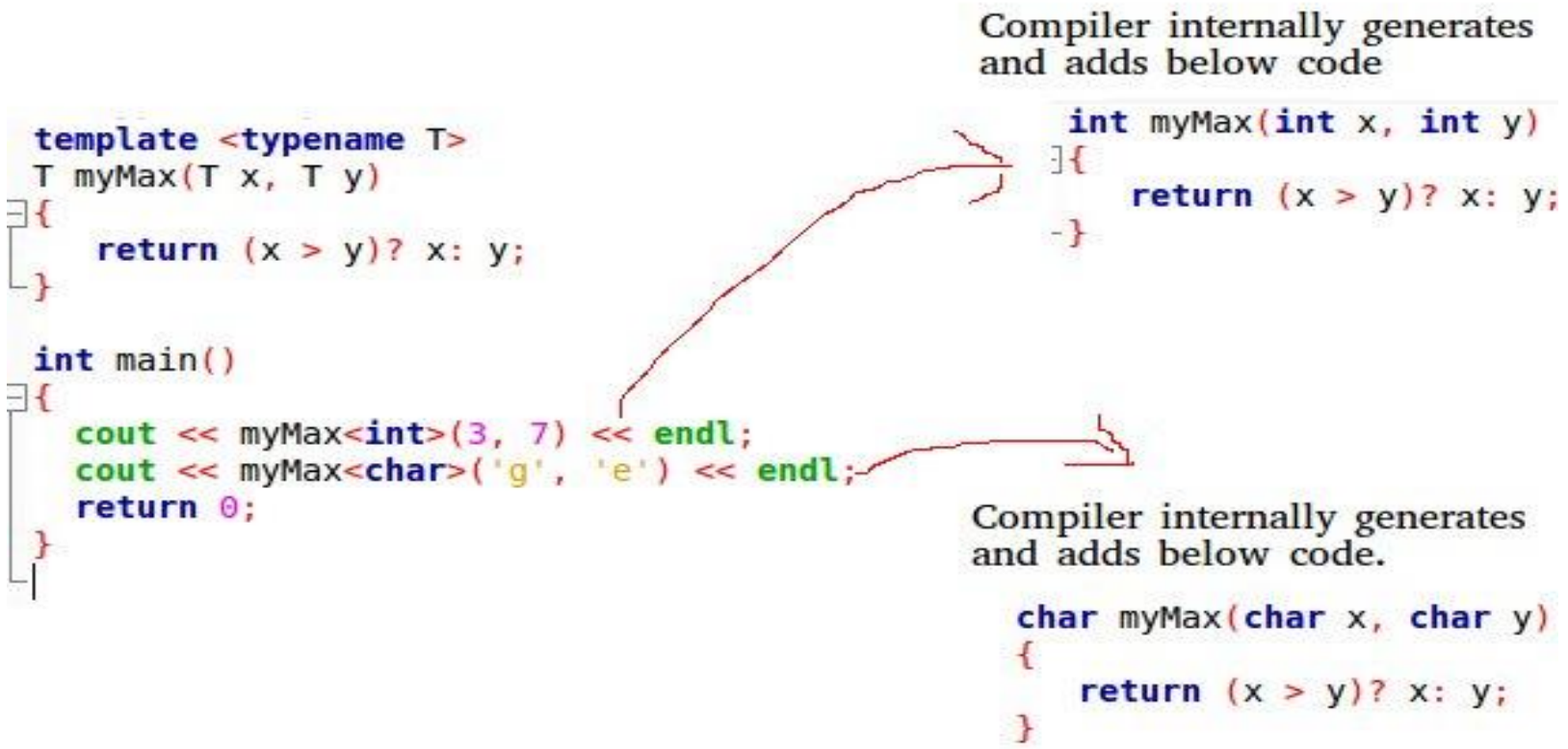
```
class MyException {
    char str_what[80];
    int what;
    public:
        int get_what() { return what;}
        char *get_str_what() { return str_what ;}
        MyException() {
            *str_what = 0;
            what = -999;
        }
        MyException(char *s, int e) {
            strcpy(str_what, s);
            what = e;
        }
    void print() { cout<<str_what<<"Error code: "<<what<<endl;
    }
};
```

```
int main() {
    int i;
    try {
        cout << "Enter a positive number: "; cin >> i;
        if(i<0)
            throw MyException("Not Positive", i);
        }
        catch (MyException e) {
        cout << e.get_what() << "\n";
        cout << e.get_str_what() << "\n";

        }
    return 0;
}
```

Templates, STL and RTTI Namespaces

Templates


template <typename T>
T myMax(T x, T y)
{
return (x > y)? x: y;
}
int main()
{
cout << myMax<int>(3, 7) << endl;
cout << myMax<char>('g', 'e') << endl;
return 0;
}
Compiler internally generates and adds below code
int myMax(int x, int y)
{
return (x > y)? x: y;
}
Compiler internally generates and adds below code.
char myMax(char x, char y)
{
return (x > y)? x: y;
}

Templates

- With function overloading same code needs to be repeated for different data types which leads towards waste of time & space.
- Templates enable us to define generic functions or classes which avoids above repetition of code for different data types .
- Generally templates are used if same algorithm works well for various data types eg sorting algorithms.
- There can be function templates or class templates.
- Function templates can be overloaded.

template < class type > ret type FnName (parameter list) template is a keyword used to create generic functions. class type is a placeholder

// swap function using function template

```
template <class T>
void swap (T & a, T & b)
{
T temp; temp       = a; a = b;
b = temp;
}
```

```
int main() {
    int i = 10, j = 20;
    swap( i, j );
   char ch1 = 'a', ch2 = 'b'
   swap( ch1, ch2 );
   }
```

You can define more than one generic data type in the **template** statement by using a comma-separated list.

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

template <class type1, class type2>
void myfunc(type1 x, type2 y) {
        cout << x << ' ' << y << '\n';
}

int main() {

myfunc(10, "I like C++");
myfunc(98.6, 19L);

return 0;
}
```

STL (Standard Template Library)

- Provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues, and stacks.
- Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.
- At the core of the standard template library are four items: *containers, algorithms, functors* and *iterators*.

Containers

Containers are objects that hold other objects.

Sequence Containers

E.g. vector, deque

Associative Containers

E.g. map

Each container class defines a set of functions that may be applied to the container.

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 or extract the contents of containers.

Their capabilities include initialization, sorting, searching, and transforming the contents of containers.

Common algorithms - binary_search, count, find, merge, max, min, sort

Iterators

Iterators are objects that are, similar to pointers.

Iterators gives you the ability to cycle through the contents of a container.

Types of Iterators:

- Random Access
- Bidirectional
- Forward
- Input
- Output

You can increment and decrement Iterators.
You can apply the * operator to them. Iterators are declared using the **iterator** type defined by the various containers.

Functors

The STL includes classes that overload the **function call operator()**. Instances of such classes are called function objects or functors.

Functors are objects that can be treated as though they are a function or function pointer.

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

class Add {

  public:
    // overload function call operator
    // accept two integer arguments
    // return their sum
    int operator() (int a, int b) {
      return a + b;
    }
};
```

```
int main() {

  // create an object of Add class
  Add add;

  // call the add  object
  int sum = add(100, 78);

  cout << "100 + 78 = " << sum;

  return 0;
}
```

Vectors

The **vector** class supports a dynamic array.

you can use the standard array subscript notation to access its elements.

Any object that will be stored in a **vector** must define a default constructor.

Example:

```
vector<int> iv; // create zero-length int vector
vector<char> cv(5); // create 5-element char vector
vector<char> cv(5, 'x'); // initialize a 5-element char vector
vector<int> iv2(iv); // create int vector from an int vector
```

Vector Functions

size() – returns the current size of the vector

begin() – returns an iterator pointing to the beginning of vector

end() – returns an iterator pointing to the end of the vector

push_back() – inserts an element at the end of the vector

Insert() – used to insert an element in middle

erase() – used to remove an element form a vector

Clear() – removes all elements from the vector

Other similar containers are – List, Map, etc.

RTTI

RTTI stands for Run Time Type Identification

RTTI enables us to identify type of an object during execution of program

RTTI operators are runtime events for polymorphic classes and compile time events for all other types.

typeid

- typeid is an operator which returns reference to object of type_info class
- type_info class describes type of object

Casting Operators

Explicit conversion is referred as cast

- const cast
- static cast
- dynamic cast
- reinterpret cast

cast operators are sometimes necessary

Explicit cast, allows programmer to momentarily suspend type checking

Syntax

```
cast_name<type>(expression);
```

casting Operators : const_cast

casts away the constness of its expression

You can not use the const_cast operator to directly override a constant variable's constant status.

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

void print (char * str) {
    cout << str << endl;
}
```

```
int main () {
const char * c = "sample text";
print ( const_cast<char *> (c) );
return 0;
}
```

casting Operators : static_cast

Any conversions which compiler performs implicitly can be made explicit using static_cast

Warning messages for loss of precision will be turned off.

reader, programmer and compiler all are made aware of fact of loss of precision.

e.g. static_cast double dval; int ival;

casting Operators : reinterpret_cast

Complex <double> *pcom;

char *pc = reinterpret_cast < char*>(pcom)

Reinterpret cast performs low level interpretation of bit pattern

Is used to convert any data type to any other data type

The operator provides a conversion between pointers to other pointer types and numbers to pointers and vice versa.

Most dangerous

casting Operators : dynamic_cast

dynamic_cast can be used only with pointers and references to objects.

Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

dynamic_cast operators are used to obtain pointer to the derived class. dynamic_cast is always successful when we cast a class to one of its base classes.