UNIT - 6

FILE HANDLING
EXCEPTION HANDLING
GENERIC PROGRAMMING

File Handling

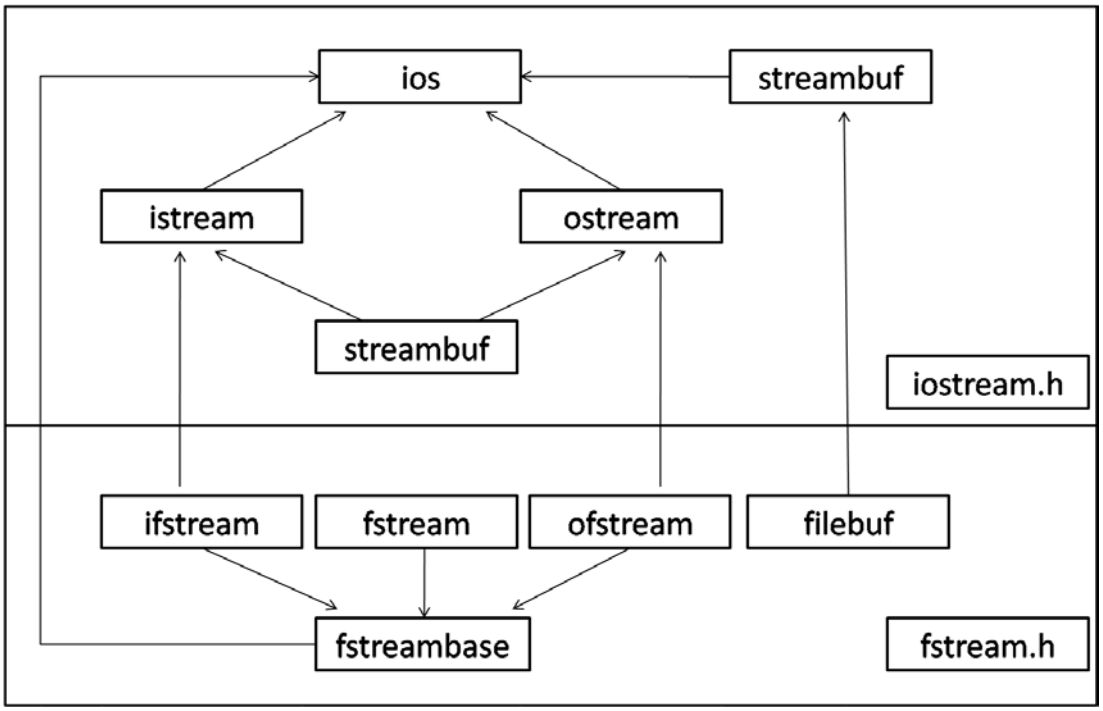
Introduction

A file is a collection of related data stored on a disk. It is advantageous to work with data stored in files rather than working with data given from keyboard due to following reasons:

- It is difficult to read huge amount of data from keyboard.
- Data entered from keyboard is volatile i.e., that data is cleared when power goes off.

File Handling Classes

Following figure illustrates the stream classes for working with files:



fstreambase: The *fstreambase* is the base class for *ifstream, ofstream,* and *fstream* classes. Functions such are *open()* and *close()* are defined in this class.

fstream: It allows simultaneous input and output operations on *filebuf*. This class inherits *istream* and *ostream* classes. Member functions of the base classes *istream* and *ostream* starts the input and output.

ifstream: This class inherits both *fstreambase* and *istream* classes. It can access member functions such as *get(), getline(), seekg(), tellg(),* and *read()*. It allows input operations and provides *open()* with the default input mode.

ofstream: This class inherits both *fstreambase* and *ostream* classes. It can access member functions such as *put(), seekp(), write(),* and *tellp()*. It allows output operations and provides the *open()* function with the default output mode.

filebuf: The filebuf is used for input and output operations on files. This class inherits *streambuf* class. It also arranges memory for keeping input data and for sending output. The I/O functions of *istream* and *ostream* invoke the filebuf functions to perform the insertion and extraction on the streams.

Opening and Closing Files

For reading or writing data into files, the general operations are: open a connection to the file, read or write data to file, and close the connection to the file.

We can open a connection to the file in two ways: one way is by using the *open()* function and second way is by using the appropriate constructor.

Syntax for opening a connection to a file using *open()* function in input mode is as follows:

```
ifstream object-name;
object-name("filename");
```

In the above syntax, *filename* is the name of file from which you want to read the data. Syntax for opening a connection to a file using the constructor in input mode is as follows:

```
ifstream object-name("filename");
```

Syntax for opening a connection to a file using *open()* function in output mode is as follows:

```
ofstream object-name;
object-name("filename");
```

In the above syntax, *filename* is the name of file into which you want to write the data. Syntax for opening a connection to a file using the constructor in output mode is as follows:

```
ofstream object-name("filename");
```

After reading or writing data into file we can close the connection by writing the following:

```
object-name.close();
```

Reading and Writing into Files

After opening a connection to the file, reading and writing data to the file is an easy task. We can write data using the following syntax:

opfile<<data;

In the above syntax, *opfile* is the object of *ofstream* and *data* is any variable holding data. We can read data from a file using the following syntax:

ipfile>>data;

In the above syntax, *ipfile* is the object of *ifstream* and *data* is any variable into which the data is stored to process in the program.

Note: When a file is opened for writing data into it, if the specified file is not there, a new file with the same name is created and opened for output. If the file is already available, the old content in the file is cleared.

Following program demonstrates writing data into the file:

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
	string name;
	int salary;
	cout<<"Enter name and salary: ";
	cin>>name>>salary;
	ofstream opfile("data.txt");	//Opening connection to a file using constructor
	opfile<<name<<"\t"<<salary;//Writing data into file
	opfile.close();		//Closing connection to the file
	cout<<"Enter name and salary: ";
	cin>>name>>salary;
	ofstream newopfile;
	newopfile.open("data.txt");	//Opening connection to a file using open() function
	newopfile<<name<<"\t"<<salary;	//Writing data into file
	newopfile.close();		//Closing connection to the file
	return 0;
}
```

Input for the above program is as follows:
Enter name and salary: suresh 7000
Enter name and salary: ramesh 8000

Output in the file "data.txt" is as follows:
ramesh 8000

Note: In the above program when the file is opened for the second time, the old data will be cleared.

Following program demonstrates reading data from a file:

```
#include<iostream>
```

```
#include<fstream>
using namespace std;
int main()
{
	string name;
	int salary;
	ifstream ipfile("data.txt");	//Opening connection to a file using constructor
	ipfile>>name>>salary;		//Reading data from file
	cout<<"Name is: "<<name<<endl;
	cout<<"Salary is: "<<salary;
	ipfile.close();			//Closing connection to the file
	return 0;
}
```

Input from the file "data.txt" for above program is as follows:
ramesh 8000

Output for the above program is as follows:
Name is: ramesh
Salary is: 8000

Detecting End-of-File

While reading data from a file, if the file contains multiple rows, it is necessary to detect the end of file. This can be done using the *eof()* function of *ios* class. It returns 0 when there is data to be read and a non-zero value if there is no data. Following program demonstrates writing and reading multiple rows of data into the file:

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
	string name;
	int salary;
	ofstream opfile("data.txt");
	for(int i=1; i<=3; i++)
	{
		cout<<"Enter name and salary of employee "<<i<<":";
		cin>>name>>salary;
		opfile<<name<<"\t"<<salary;
	}
	opfile.close();
	cout<<"Data stored into file."<<endl;
	cout<<"Reading data from file..."<<endl;
	ifstream ipfile("data.txt");
	while(!ipfile.eof())	//Checking for end of file
	{
```

```
            ipfile>>name>>salary;
            cout<<"Name: "<<name<<" Salary: "<<salary<<endl;
        }
        ipfile.close();
        return 0;
}
```

Input and output for the above program is as follows:

```
Enter name and salary of employee 1:suresh 7000
Enter name and salary of employee 2:ramesh 8000
Enter name and salary of employee 3:mahesh 9000
Data stored into file.
Reading data from file...
Name: suresh Salary: 7000
Name: ramesh Salary: 8000
Name: mahesh Salary: 9000
```

File Modes

Until now when using the constructor or *open()* function for reading or writing data to a file, we are passing only one argument i.e., the file name. We can also pass a second argument which specifies the file mode. The mode parameter specifies the mode in which the file has to be opened. We can specify one of the following modes available in *ios* file:

File Mode	Meaning
ios::app	When file is opened in append mode, the data in the file remains and the new data is appended to the end of file.
ios::ate	A file is opened with the file pointer set to the end of file.
ios::binary	A file opened in binary mode.
ios::nocreate	When a file is opened in out mode and the file is not there, no new file created.
ios::noreplace	If the file is already available, it cannot be opened. If the file is not there, a new file is created.
ios::trunc	When a file is opened, the data in the file gets deleted.
ios::in	Opens the file for input or to read from file. This is default for ifstream.
ios::out	Opens the file for output or to write to the file. This is default for ofstream.

A programmer can open a file in append mode by writing as follows:

fstream object-name("filename", ios::app);

A programmer can combine multiple modes using the | symbol as follows:

fstream object-name("filename", ios::out | ios::nocreate);

File Pointers and Manipulation

Every file will contain two pointers: a read pointer or also known as a get pointer and a write pointer also known as a put pointer. The read pointer or a get pointer is used to read data and the write pointer or put pointer is used to write data to a file. These pointers can be manipulated using the functions from stream classes. Those functions are as follows:

Function	Class Name	Description	Example	Effect
seekg()	ifstream	Moves the get pointer to specified position	seekg(20)	Moves the get pointer to byte 20
tellg()	ifstream	Gives the position of get pointer	tellg()	Returns 20 if the get pointer is at byte 20
seekp()	ofstream	Moves the put pointer to specified position	seekp(20)	Moves the put pointer to byte 20
tellp()	ofstream	Moves the put pointer to specified position	tellp()	Returns 20 if the put pointer is at byte 20

The seekg() and seekp() functions specified above can be used to move the pointers in the file for random access. The syntax of these functions is as follows:

seekg(int offset, reference_position)
seekp(int offset, reference_position)

The offset is an integer parameter which specifies the position in bytes and *reference_position* can be any one of the following:

ios::beg (Moves the pointer from beginning of file)
ios::cur (Moves the pointer from current position of pointer)
ios::end (Moves the pointer from ending of file)

For example if we give *seekg(20, ios::cur)* moves the get pointer by 20 bytes from the current position. If we give *seekg(-20, ios::end)* moves the get pointer by 20 bytes back from the end of file.

Types of Files

C++ supports two types of files. They are text files and binary files.

ASCII Text Files

A text file is collection of characters that can be processed sequentially by a computer. As text files only process characters, they can read or write data one character at a time. In a text file, each line contains zero or more characters and ends with one or more characters that specify the end of a line.

In a text file data of all types are stored as a sequence of characters. Each file ends with a special character called as end of file marker.

Binary Files

A binary file is a file which contains any type of data, converted to binary format for computer storage and processing. A binary file is non-human readable. A binary file does not require any special processing of the data and each byte of data is transferred from the disk unprocessed. While text files can be processed sequentially, binary files can be processed sequentially or in a random fashion.

Binary files store data in internal representation format i.e., if we store an integer 51, it only occupies only one byte. Whereas in text file it occupies two bytes. One byte to store character 5 and another byte to store character 1.

Sequential and Random I/O

C++ allows data to be read or written from a file in sequential or random fashion. Reading data character by character or record by record is called sequential access. Reading data in any order is known as random access.

The *fstream* class provides functions like *get(), read()* for reading data and *put(), write()* for writing data to a file. The functions *get()* and *put()* are character-oriented functions. Syntax of these functions is as follows:

get(char)
put(char)

Following program demonstrates *get()* and *put()* functions:

```
#include<iostream>
#include<fstream>
#include<string.h>
using namespace std;
int main()
{
	char data[50], ch;
	cout<<"Enter a line of text: ";
	cin.getline(data, 50);
	fstream file("data.txt", ios::in | ios::out);
	int len = strlen(data);
	for(int i=0; i<len; i++)
	{
		file.put(data[i]);
	}
	file.seekg(0);
	for(int i=0; i<len; i++)
	{
		file.get(ch);
		cout<<ch;
	}
	file.close();
	return 0;
```

```
}
```

Input and output for the above program is as follows:

```
Enter a line of text: Welcome to file handling
Welcome to file handling
```

Until now we working with text files. But, as mentioned previously, reading and writing data in binary format is more efficient. We can store and retrieve objects in to binary files using *read()* and *write()* methods whose syntax is as follows:

```
read((char *)&var, sizeof(var));
write((char *)&var, sizeof(var));
```

Both functions accept to parameters. First parameter is the address of the variable or object and the second parameter is the size of variable or object. Following program demonstrates storing and retrieving an object from file:

```
#include<iostream>
#include<fstream>
using namespace std;
class Student
{
	private:
		int regdno;
		int marks;
	public:
		Student(){}
		Student(int no, int m)
		{
			regdno = no;
			marks = m;
		}
		void show()
		{
			cout<<"Regd.No. is: "<<regdno<<endl;
			cout<<"Marks are: "<<marks<<endl;
		}
};
int main()
{
	Student s1(501, 100);
	ofstream opfile("data.txt", ios::binary);
	opfile.write((char *)&s1, sizeof(s1));
	opfile.close();
	Student s2;
	ifstream ipfile("data.txt", ios::binary);
	ipfile.read((char *)&s2, sizeof(s2));
	ipfile.close();
	s2.show();
```

```
        return 0;
}
```

Output (in data.txt) for above program is as follows:

```
Regd.No. is: 501
Marks are: 100
```

Command Line Arguments

The data or parameters provided while invoking the program are known as command-line parameters or arguments. Command-line arguments can be accessed using the variables declared in the signature of *main()* function as shown below:

*int main(int argc, char *argv[])*

In the above syntax, first parameter *argc* holds the count of command-line arguments and the second parameter, an array of character pointers holds the actual command-line arguments.

Note: *Remember that the first element in the array, i.e., argv[0] holds the filename. First command-line parameter will be available in argv[1], second parameter in argv[2] and so on.*

Following program demonstrates working with command-line arguments:

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main(int argc, char* argv[])
{
        cout<<"Arg count = "<<argc<<endl;
        int x = atoi(argv[1]);
        cout<<"First parameter = "<<x;
}
```

Input for the above program is: 10 20

Output for the above program is as follows:

```
Arg count = 3
First parameter = 10
```

Exception Handling

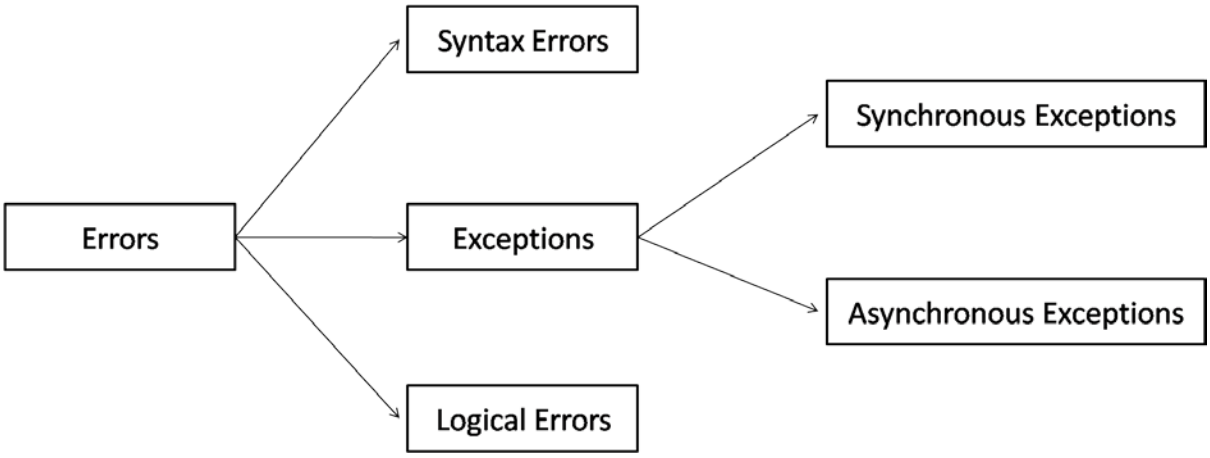
Introduction

In programming, it is common for programmers to make mistakes which leads to abnormal conditions called as errors. In general, these errors are of three types: 1) Syntax errors, 2) Logical errors, and 3) Run-time errors.

Syntax errors are most frequent type of errors. For example, if we forget to place a semi-colon, it is a syntax error. Logical errors arise when programmer performs a mistake in the logic of a program. For example, in addition of two numbers program, if the programmers places a minus instead of plus, it is a logical error. These are very hard to detect.

The last category of errors are the errors which occur while executing the program. These errors are known as run-time errors or exceptions. Exceptions are of two types: 1) Synchronous, and 2) Asynchronous.

Synchronous exceptions are the run-time exceptions which occur due to the code written by the programmer and they can be handled by the programmer. Asynchronous exceptions are run-time exceptions which occur due to code outside the program. For example, if no memory is available in the RAM, it will lead to out of memory error which is an asynchronous exception. Such asynchronous exceptions cannot be handled. Errors in programs can be illustrated as shown below:



.

Exception Handling

Handling synchronous exceptions is known as exception handling. Exception handling deals with detecting run-time exceptions and reporting it to the user for taking appropriate action. C++ provides *try*, *catch*, and *throw* elements for handling exceptions.

A *try* block contains code which might raise exceptions. Syntax of try block is as follows:

```
try
{
        //Code
        ...
}
```

A *catch* block contains code for handling exceptions that may raise in the try block. Syntax of catch block is as follows:

```
catch(Exception-type)
{
        //Code to handle exception
        ...
}
```

The *throw* clause is used to throw an exception which will be caught by the *catch* block. Syntax of throw clause is as follows:

```
throw exception;
```

Following program handles divide by zero exception:

```
#include<iostream>
using namespace std;
int main()
{
        int a, b;
        cout<<"Enter a and b values: ";
        cin>>a>>b;
        try
        {
                if(b == 0)
                        throw b;
                else
                        cout<<"Result of a/b = "<<(a/b);
        }
        catch(int b)
        {
                cout<<"b cannot be zero";
        }
        return 0;
}
```

Input and output for the above program is as follows:

```
Enter a and b values: 10 0
b cannot be zero
```

Note: If an exception arises and there is no catch block to handle that exceptions, *terminate()* function will execute which in turn calls *abort() function and the execution of program stops abruptly*.

Multiple Catch Statements

A try block can be associated with more than one catch block. Multiple catch statements can follow a try block to handle multiple exceptions. The first catch block that matches the exception type will execute and the control shifts to next statement after all the available catch blocks.

A try block should be followed by at least one catch block. If non catch block matches with the exception, then *terminate()* function will execute. Following program demonstrates handling multiple exceptions using multiple catch statements:

```
#include<iostream>
using namespace std;
int main()
{
	int a, b;
	cout<<"Enter a and b values: ";
	cin>>a>>b;
	try
	{
		if(b == 0)
			throw b;
		else if(b < 0)
			throw "b cannot be negative";
		else
			cout<<"Result of a/b = "<<(a/b);
	}
	catch(int b)
	{
		cout<<"b cannot be zero";
	}
	catch(const char* msg)
	{
		cout<<msg;
	}
	return 0;
}
```

Input and output for the above program is as follows:

First Run:
Enter a and b values: 10 0
b cannot be zero

Second Run:

```
Enter a and b values: 10 -5
b cannot be negative
```

Catch All Exceptions

While writing programs if the programmer doesn't know what kind of exception the program might raise, the catch all block can be used. It is used to catch any kind of exception. Syntax of catch all block is as follows:

```
catch(...)
{
        //Code to handle exception
        ...
}
```

While writing multiple catch statements care should be taken such that a catch all block should be written as a last block in the catch block sequence. If it is written first in the sequence, other catch blocks will never be executed. Following program demonstrates catch all block:

```
#include<iostream>
using namespace std;
int main()
{
        int a, b;
        cout<<"Enter a and b values: ";
        cin>>a>>b;
        try
        {
                if(b == 0)
                        throw b;
                else if(b < 0)
                        throw "b cannot be negative";
                else
                        cout<<"Result of a/b = "<<(a/b);
        }
        catch(int b)
        {
                cout<<"b cannot be zero";
        }
        catch(...)
        {
                cout<<"Unkown exception in program";
        }
        return 0;
}
```

Input and output for the above program is as follows:

```
Enter a and b values: 10 -5
Unkown exception in program
```

Rethrowing an Exception

In C++ if a function or a nested try-block does not want to handle an exception, it can rethrow that exception to the function or the outer try-block to handle that exception. Syntax for rethrowing and exception is as follows:

throw;

Following program demonstrates rethrowing and exception to outer try-catch block:

```
#include<iostream>
using namespace std;
int main()
{
	int a, b;
	cout<<"Enter a and b values: ";
	cin>>a>>b;
	try
	{
		try
		{
			if(b == 0)
				throw b;
			else if(b < 0)
				throw "b cannot be negative";
			else
				cout<<"Result of a/b = "<<(a/b);
		}
		catch(int b)
		{
			cout<<"b cannot be zero";
		}
		catch(...)
		{
			throw;
		}
	}
	catch(const char* msg)
	{
		cout<<msg;
	}
	return 0;
}
```

Input and output for the above program is as follows:

```
Enter a and b values: 10 -2
b cannot be negative
```

In the above program we can see that the exception is raised in the inner try block. The catch all block catches the exception and is rethrowing it to the outer try-catch block where it got handled.

Throwing Exceptions in Function Definition

A function can declare what type of exceptions it might throw. Syntax for declaring the exceptions that a function throws is as follows:

```
return-type function-name(params-list) throw(type1, type2, ...)
{
        //Function body
        ...
}
```

Following program demonstrates throwing exceptions in a function definition:

```
#include<iostream>
using namespace std;
void sum() throw(int)
{
        int a, b;
        cout<<"Enter a and b values: ";
        cin>>a>>b;
        if(a==0 || b==0)
                throw 1;
        else
                cout<<"Sum is: "<<(a+b);
}
int main()
{
        try
        {
                sum();
        }
        catch(int)
        {
                cout<<"a or b cannot be zero";
        }
        return 0;
}
```

Input and output for the above program is as follows:

```
Enter a and b values: 5 0
a or b cannot be zero
```

In the above program the *sum()* function can throw an exception of type *int*. So, the calling function must provide a catch block for exception of type *int*.

Throwing Exceptions of Class Type

Instead of throwing exceptions of pre-defined types like *int, float, char, etc.*, we can create classes and throw those class types as exceptions. Empty classes are particularly useful in exception handling. Following program demonstrates throwing class types as exceptions:

```
#include<iostream>
using namespace std;
class ZeroError {};
void sum()
{
	int a, b;
	cout<<"Enter a and b values: ";
	cin>>a>>b;
	if(a==0 || b==0)
		throw ZeroError();
	else
		cout<<"Sum is: "<<(a+b);
}
int main()
{
	try
	{
		sum();
	}
	catch(ZeroError e)
	{
		cout<<"a or b cannot be zero";
	}
	return 0;
}
```

Input and output for the above program is as follows:

```
Enter a and b values: 0 8
a or b cannot be zero
```

In the above program *ZeroError* is an empty class created for handling exception.

Exception Handling and Inheritance

In inheritance, while throwing exceptions of derived classes, care should be taken that catch blocks with base type should be written after the catch block with derived type. Otherwise, the catch block with base type catches the exceptions of derived class types too. Consider the following example:

```
#include<iostream>
using namespace std;
class Base {};
class Derived : public Base {};
int main()
{
	try
	{
		throw Derived();
	}
	catch(Base b)
	{
		cout<<"Base object caught";
	}
	catch(Derived d)
	{
		cout<<"Derived object caught";
	}
	return 0;
}
```

Output for the above program is as follows:

Base object caught

You can see that in the above program even though the exception thrown is of the type *Derived* it is caught by the catch block of the type *Base*. To avoid that we have to write the catch block of *Base* type at last in the sequence as follows:

```
#include<iostream>
using namespace std;
class Base {};
class Derived : public Base {};
int main()
{
	try
	{
		throw Derived();
	}
	catch(Derived d)
	{
		cout<<"Derived object caught";
	}
```

```
	catch(Base b)
	{
		cout<<"Base object caught";
	}

	return 0;
}
```

Output of the above program is as follows:

Derived object caught

Exceptions in Constructors and Destructors

It is possible that exceptions might raise in a constructor or destructors. If an exception is raised in a constructor, memory might be allocated to some data members and might not be allocated for others. This might lead to memory leakage problem as the program stops and the memory for data members stays alive in the RAM.

Similarly, when an exception is raised in a destructor, memory might not be deallocated which may again lead to memory leakage problem. So, it is better to provide exception handling within the constructor and destructor to avoid such problems. Following program demonstrates handling exceptions in a constructor and destructor:

```
#include<iostream>
using namespace std;
class Divide
{
	private:
		int *x;
		int *y;
	public:
		Divide()
		{
			x = new int();
			y = new int();
			cout<<"Enter two numbers: ";
			cin>>*x>>*y;
			try
			{
				if(*y == 0)
				{
					throw *x;
				}
			}
			catch(int)
			{
				delete x;
```

```
                        delete y;
                        cout<<"Second number cannot be zero!"<<endl;
                        throw;
                }
        }
        ~Divide()
        {
                try
                {
                        delete x;
                        delete y;
                }
                catch(...)
                {
                        cout<<"Error while deallocating memory"<<endl;
                }
        }
        float division()
        {
                return (float)*x / *y;
        }
};
int main()
{
        try
        {
                Divide d;
                float res = d.division();
                cout<<"Result of division is: "<<res;

        }
        catch(...)
        {
                cout<<"Unkown exception!"<<endl;
        }
        return 0;
}
```

Input and output for the above program is as follows:

```
Enter two numbers: 5 0
Second number cannot be zero!
Unkown exception!
```

Advantages of Exception Handling

Following are the advantages of exception handling:

- Exception handling helps programmers to create reliable systems.
- Exception handling separates the exception handling code from the main logic of program.
- Exceptions can be handled outside of the regular code by throwing the exceptions from a function definition or by re-throwing an exception.
- Functions can handle only the exceptions they choose i.e., a function can throw many exceptions, but may choose handle only some of them.
- A program with exception handling will not stop abruptly. It terminates gracefully by giving appropriate message.

Generic Programming