No rule is so general, which admits not some exception.

Robert Burton

Don't interrupt me while I'm interrupting.

Winston S. Churchill

8.1 Introduction

The author of a library can detect run-time errors but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors but cannot detect them or else they would have been handled in the user's code and not left for the library to find. The notion of an exception is provided to help deal with such problems. The fundamental idea is that a function that finds a problem that it cannot cope with throws an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle that kind of problem can indicate that it is willing to catch that exception.

All programs have bugs. The bigger the program, more bugs it will contain, and the more likely that many of these bugs will find their way into related software. The first priority of any serious programmer is to make robust and bug-free programs. The types of bugs that can afflict a program are:

1. Bugs that arise out of a mistake in syntax.
2. Logic errors, which arise because the programmer misunderstood the problem or how to solve it.
3. Exceptions, which arise out of unusual but predictable problems such as running out of resources (memory or disk space), accessing an array index that is out of bounds, divide by zero etc.

Programmers use powerful compilers and add to their existing code error handling code. To find logic errors, programmers use design reviews and exhaustive testing. Exceptions are different. It doesn't mean avoiding errors, but only how to handle it. When an exception occurs the user can choose one of the following:

1. **Terminate the program** is what is done by default when an exception isn't caught. For most errors we can and must do better. In particular, a library that doesn't know about the purpose and general strategy of the program in which it is embedded cannot simply exit() or abort(). A library that unconditionally terminates cannot be used in a program that cannot afford to crash. One way of viewing exceptions is as a way of giving control to caller when no meaningful action can be taken locally.

2. **Return a value representing "error"** isn't always feasible because there is often no acceptable "error" value. For example, if a function returns an int, every int might be a plausible result. Even where this approach is feasible, it is often inconvenient because every call must be checked for the error value. This can easily double the size of the program.

3. **Return a legal value and leave the program in an illegal state** has the problem that the calling function may not notice that the program has been put in an illegal state. For example many standard C library functions set the global variable errno to indicate error. However,

programs typically fail to test errno consistently enough to avoid consequential errors caused by values returned from failed calls. Furthermore, the use of global variables for recording error conditions doesn't work well in the presence of concurrency.

4. **Take corrective action and continue without disturbing the user**. C++ exception handling mechanism provides an alternative to the traditional techniques when they are insufficient, inelegant, and error-prone. It provides a way of explicitly separating error-handling code from "ordinary" code, thus making the program more readable and more amenable to tools. The exception handling mechanism provides a more regular style of error handling, thus simplifying cooperation between separately written program fragments.

The basic philosophy of Software should be "badly-formed code should not run". Though the ideal time to catch errors is compile time, not all errors can be detected at compile time. They must be handled at run-time through some formality. The word "exception" is meant in the sense of "I take exception to that". The exception handling mechanism of C++, is based on Ada .

Exception handling is about handling exceptional conditions. Not to confuse with ordinary error's, for example "int account_number = -100" (account numbers may not be negative, in your program) may be termed an ordinary error where as "int account_number = 100a" (int may not have alphabets in its value, in any program) is an exception, although ordinary error's can be dealt with exception handling.

First let us see how exception handling is done in C.

```
FILE * f1 = fopen ("file1");
if(f1 == NULL)
{
   // if the file cannot be opened processing cannot be done
   // handle the error and exit
}

FILE * f2 = fopen ("file2");
if(f2 == NULL)
{
   // close file1, handle the error and exit
}

FILE * f3 = fopen ("file3");
if(f3 == NULL)
{
   // close file1, close file2 and handle the error and exit
}
// if everything goes fine process the files here
```

Now let us see how the same code can be implemented in C++.

```
// think of FileClass as some class which supports file I/O
try
{
   f1 = new FileClass ("file1");
   f2 = new FileClass ("file2");
   f3 = new FileClass ("file3");
   // if every thing goes fine process the files here
```

```
}
catch(exception e)
{
   // in case exceptions occur close the files, handle the errors and exit
}
```

In earlier languages like C, exception handling was established by convention and is not a part of the language itself. First, the programmer was at liberty not to implement the error handling mechanism in the application. The three if blocks in the above fragment need not be implemented and the program would still compile. This unwarranted freedom provided by the languages encouraged sloppy programming culture. Secondly, even if you are a die-hard conservative programmer and wish to provide for all exceptions, you still can't provide for without making the program into an unreadable nightmare. Imagine having to provide a series of error checks for every printf() and scanf(). Thirdly you cannot separate the code for exceptions from code for normal functioning.

We typically put the code for normal functioning of the application inside the try block, but are forced to provide code for errors in a corresponding catch block. A try block must always be associated with at least one catch block, or as many catch blocks as necessary.

```
try
{
   // code for normal functioning
}
catch (IOexception e)
{
   // code upon exceptions
}
catch (Fileexception e)
{
   // code upon exceptions
}
```

8.2 Exceptions

The C++ language provides built-in support for handling anomalous situations, known as “exceptions,” which may occur during the execution of your program. The **try**, **throw**, and **catch** statements have been added to the C++ language to implement exception handling. With C++ exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control. An exception in C++ is an object that is thrown from the area of code where a problem occurs to the part of the code that will handle the problem. The type of the exception determines which area of code will deal with the problem, and the contents of the object thrown, if any, will be used to provide feedback to the user.

The steps in using exceptions include the following:

1. Identify those areas of the program that begin an operation that might raise an exception, and put them in a try block.
2. Create catch blocks to catch the exceptions if they are thrown, and to clean up allocated memory and inform the user appropriately.

When an exception is thrown, control is transferred to the catch block immediately following the current try block.

8.1 Code Listing

// Illustrates how to throw and catch a simple exception - Divide by Zero

```
# include <iostream.h>

double divide(long numerator, long denominator)
{
   if(denominator == 0) throw "Division by Zero";
      return (double) numerator / denominator;
}

int main()
{
   long numerator, denominator;

   cout << "Enter two numbers";
   cin >> numerator >> denominator;
   // Press ctrl z for eof

   while(!cin.eof())
   {
      try
      {
         cout << divide(numerator, denominator);
      }
      catch(const char* message)
      {
         cerr << message << endl;
      }

      cout << "Enter another numerator and denominator ";
      cin >> numerator >> denominator;
   }
   return 0;
}
```

8.2 Code Listing

/* The following example shows how class functions can use exception handling to detect run time errors. In the following example there are two such errors

1. the constructor function detects that no heap space is available
2. the overloaded operator [] () function detects an invalid index

In both cases an exception is thrown. If you wish to emulate the first error, then just uncomment the statement in the constructor function that sets the member ptr to 0 */

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

```
class array
{
   private:
      int *ptr;
      int size;
   public:
      array(int = 1);
      ~array();
      int& operator[] (int);
      friend ostream& operator << (ostream&, const array&);
};

array :: array(int n)
{
   size = n;

   if(size < 0)
      throw ("Negative size for memory allocation not allowed");

   ptr = new int[size];
   if (ptr == 0)
      throw("Out of memory");

   for(int i = 0; i < size; i++)
      ptr[i] = 0;
}

array :: ~array()
{
   delete [] ptr;
}

int& array :: operator[] (int x)
{
   if((x < 0) || (x >= size))
      throw x;

   return ptr[x];
}

ostream & operator << (ostream& os, const array& a)
{
   cout << "The array:\n";
   for(int i = 0; i < a.size; i++)
      os  << '[' << i << ']' << " " << a.ptr[i] << endl;

   return os;
}

int main()
{
   cout << "Input the array size ";
   int length;
   cin >> length;            // Press ctrl z to encounter eof

   while(!cin.eof())
```

```
   {
      try
      {
         array a(length);
         cout << "Enter an index and a value: ";
         int index, value;
         cin >> index >> value;

         while(!cin.eof())
         {
            a[index] = value;
            cout << a << endl;
            cout << "Enter next index and value: ";
            cin >> index >> value;
         }
      }
      catch(const char* message)
      {
         cerr << message << endl;
      }
      catch(int m)
      {
         cerr << "Invalid index : " << m << endl;
      }
      cout << "Enter next array length: ";
      cin >> length;
   }
   return 0;
}
```

8.3 Code Listing

/* This program illustrates how to store data as part of exception class and to derive from the existing exception classes. The following two classes: withdrawException and duplicateAccNoException are derived from Exception. If the account number already exists, an object of duplicateAccNoException is thrown. After the account is created, if we perform a withdraw operation, and the new balance is less than the minimum balance, an object of withdrawException is thrown */

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

class Exception
{
   char message[100];
   public:
      Exception(){}

      Exception(char * m)
      {
         strcpy(message, m);
      }
      char * getMessage()
      {
         return message;
```

```
      }
      ~Exception(){}
};

class withdrawException : public Exception
{
   public:
   withdrawException():Exception() {}
   withdrawException(char * err):Exception(err) {}
};

class duplicateAccNoException : public Exception
{
   int actNo;

   public:
   duplicateAccNoException():Exception() {}
   duplicateAccNoException(int act_no, char * message) :
Exception(message)
   {
      this->actNo = act_no;
   }
   int getActNo()
   {
      return actNo;
   }
};

const int minBal = 1000;
const int acc_no[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

class Account
{
   int accno;
   char name[30];
   double balance;
   void checkDupAccNo(int);

   public:
   void getData(void);
   void withDraw(double);
   void Deposit(double);
   void Show();
};

void Account :: checkDupAccNo(int num)
{
   for(int i = 0; i < 10; i++)
   {
      if(acc_no[i] == num)
         throw duplicateAccNoException(num, "Existing account");
   }
}

void Account :: getData()
{
```

```
   while(1)
   {
      cout << "Enter Account number: ";
      cin >> accno;

      try
      {
         checkDupAccNo(accno);
         break;
      }
      catch(duplicateAccNoException d)
      {
         throw d;
      }
   }

   cin.ignore();
   cout << "Enter Name: ";
   cin >> name;
   cout << "Enter Initial Balance: ";
   cin >> balance;
}

void Account :: withDraw(double amt)
{
   double b = balance - amt;

   if (b < minBal)
      throw withdrawException("Cannot withdraw more than minimum
Balance");
   else
      balance = b;
}

void Account :: Deposit(double amt)
{
   balance += amt;
}

void Account::Show()
{
   cout << "Account No: " << accno << endl;
   cout << "Name: "<< name << endl;
   cout << "Balance: "<< balance << endl;
}

void main()
{
   Account A;
   try
   {
      A.getData();
      A.Deposit(500);
      A.withDraw(5000);
      A.Show();
   }
```

```
   catch(duplicateAccNoException m)
   {
      cout << m.getMessage() << ": " << m.getActNo() << endl;
   }
   catch(withdrawException d)
   {
      cout << d.getMessage() << endl;
   }
   catch(...)   // Catch all
   {
      cout << "\n something went wrong";
   }
}
```

8.3 Exercise

Theory questions

1. What is the purpose of C++ exceptions handling mechanism?
2. What are the traditional error handling mechanisms?
3. What is an error? What are the different kinds of errors?
4. When should an exception be raised?
5. How can throw rather than error codes make the system easier to understand?
6. Why is it helpful to separate the normal logic from the exception handling logic?
7. What should be placed in a try block?
8. What should be placed in a catch block?

Problems

1. Modify problem 1 to add data into the exception, along with an accessor function, and use it in the catch block.

2. Add exception handlers to the int_vector problem discussed in chapter 6.

3. Add exception handlers to the menu driven file application discussed in chapter 7.