EXCEPTIONS

Exceptions provide a systematic, object-oriented approach to handle **runtime** errors generated by C++ classes.

To qualify as an exception, such errors must occur as a result of some action taken within a program and they must be the ones the program itself can discover.

For example, a constructor in a user-written string class might generate an exception if the application tries to initialize an object with a string that's too long.

Similarly, a program can check if a file was opened or written successfully and generate an exception if it was not.

Let's look at how the process was handled in the past.

In C language programs, an error is often signaled by returning a particular value from the function in which it occurred.

For example, many math functions return a special value to indicate an error, and disk file functions often return NULL or 0 to signal an error.

Each time you call one of these functions, you check the return value.

Obsolete error handling:

```
if( somefunc() == ERROR_RETURN_VALUE )
        // handle the error or call error-handler function
else
        // proceed normally
if( anotherfunc() == NULL )
        // handle the error or call error-handler function
else
        // proceed normally
if( thirdfunc() == 0 )
        // handle the error or call error-handler function
else    // proceed normally
```

The problem with this approach is that every single call to such a function must be examined by the program.

Surrounding each function call with an if...else statement and inserting statements to handle the error (or to call an error-handler routine) makes the listing long and hard to read.

Also, it's not practical for some functions to return an error value.

For example, imagine a min() function that returns the minimum of two values.

All possible return values from this function represent valid outcomes.

There's no value left to use as an error return.

The problem becomes more complex when classes are used because errors may take place without a function being explicitly called.

For example, suppose an application defines objects of a class:

```
SomeClass obj1, obj2, obj3;
```

How will the application find out if an error occurred in the class constructor?

The constructor is called implicitly, so there's no return value to be checked.

Exception Syntax

If an error is detected in a member function, this member function informs the application that an error has occurred.

When exceptions are used, this is called *throwing an exception*.

In the application, a separate section of code is installed to handle the error.

This code is called an **exception handler** or ***catch block***: it catches the exceptions thrown by the member function.

Any code in the application that uses objects of the class is enclosed in a ***try block***.

The exception mechanism uses three new C++ keywords: **throw**, **catch**, and **try**.

Throwing an exception:
Syntax of a function f that throws an exception:

```
return_type f( parameters ) {
    if ( exception_condition ) throw exceptioncode;
        // normal operation
    return expression;
 }
```

Here *exceptioncode* can be any variable or constant of any built-in type (as char, int, char *) or it can also be an object that defines the exception.

Example; a fraction function: It takes the numerator and denominator as parameters, calculates the result of the fraction and returns it back.
If the denominator is zero an exception must be thrown.

```
float fraction(int num, int denom)
{
    if(denom==0) throw "Divide by zero";          // Exception condition
    return static_cast<float>(num) / denom;       // Normal operation
}
int main()
{
    int numerator, denominator;
    cout << endl << "Enter the numerator ";
    cin >> numerator;
    cout << endl << "Enter the denominator ";
    cin >> denominator;
    try{
        cout << fraction(numerator, denominator);
    }
    catch (const char * result){
         cout << endl << result;
   }
  cout << endl << "End of Program";
  return 0;
}
```

Try block.

The catch block must immediately follow the try block.

See Example: e10_1.cpp

In a catch block you may catch only the type of the exception-code, if the code itself is not necessary.

```
catch (const char *){
        cout << endl << "ERROR";            // The thrown data is unknown
}
```

A function may throw more then one exceptions. For example if we don't want negative denominators, we can write the fraction function as follows:

```
float fraction(int num, int denom)
{
    if(denom == 0) throw "Divide by zero";
    if(denom < 0) throw "Negative denominator";
    return static_cast<float>(num) / denom;
}
```

A function may also throw exceptions of different types.

```
float fraction(int num, int denom)
{
    if(denom == 0) throw "Divide by zero";             // throws char *
    if(denom < 0) throw "Negative denominator";        // throws char *
    if(denom > 1000) throw -1;                         // throws int
    return static_cast<float>(num) / denom;
}
```

If a function throws exceptions of different types, then a separate catch block must be written for each exception type.

```
try {
       cout << fraction(numerator , denominator);
}
catch (const char * result) {              // Catch block for exceptions of type char *
        cout << endl << result;
}
catch (int) {                  // Catch block for exceptions of type int   (value is not taken)
        cout << endl << "ERROR";
}
```

See Example: e10_2.cpp

Like built-in data types, objects can also be thrown and caught as exceptions. Examine the example e10_3.cpp. In this program we have a class: Stack. This class includes two functions push and pop. If an error occurs, these functions throw an object of class Error.

See Example: e10_3.cpp

Exceptions and Constructors

Exceptions are necessary to find out if an error occurred in the class constructor. Constructors are called implicitly and there’s no return value to be checked.

Example: The creator of the String class does not allow the contents of the String to be longer than 10 characters.

```
class String{
      enum { MAX_SIZE = 10 };          // MAX_SIZE is a constant
      int size;
      char *contents;
 public:
      String(const char *);            // Constructor
      void print() const;              // A member function
      ~String();                       // Destructor
};

String::String(const char *in_data)
{
      size = strlen(in_data);
      if (size > MAX_SIZE) throw "String too long";
      contents = new char[size +1];         // Normal operations
      strcpy(contents, in_data);
 }
```

```
int main()
{
      char input[20];                      // To take strings from keyboard
      String *str;                         // Pointer to objects
      bool again;                           // loop condition
      do{
            again = false;
            cout << " Enter a string: ";
            cin >> input;
            try{
               str= new String(input);         // calls the constructor
            }
            catch (const char *){
                 cout << "String is too long" << endl;
                 again = true;
            }
      }while(again);
      str->print();                      // The creation of the object is guaranteed
      delete str;
      return 0;
}
```

The only way to exit the do-while loop is giving strings shorter than 10 characters. Otherwise the object is not created.

See Example: e10_4.cpp