Exceptions

Writing code to detect errors that may occur in the execution of class methods also requires a programmer to specify how to handle the errors. A common approach is to simply issue an error message and/or abort execution by means of the exit() function or assert() function. In many cases, however, it would be better to only signal the error and let the user of the class take appropriate action, and exceptions are designed to make this possible.

When a function detects an error, it can throw an exception, which is usually an error-message string or a class object that conveys information to the exception handler, which will catch the exception and take appropriate action. If there is no handler for that type of exception, execution terminates. To illustrate, suppose that a class Time contains a Set operation that accepts values for parameters hours, minutes, and am_pm to be used to set data members myHours, myMinutes, and myAMorPM in the class, provided that they are valid values for a Time object. If they are not, the function might throw an exception as shown in the following code:

```cpp
if (hours >= 1 && hours <= 12 &&
    minutes >= 0 && minutes <= 59 &&
    (am_pm == 'A' || am_pm == 'P'))
{
    . . .
} else
{
    char illegal_Time_Error[] =
        "*** Can’t set time with these values ***\n"
    throw illegal_Time_Error;
}
```

A program or function that calls this function encloses the function call and associated code in a try block of the form

```cpp
try
{
    ... statements that may cause error
}
```

This is followed by one of more catch blocks, each of which specifies an exception type and contains code for handling that exception. They have the form

```cpp
catch(exception_type optional_parameter_name)
{
    ... the exception handler
}
```

For example, the following code attempts to use the Set operation in a Time object mealTime and catches the exception thrown in Set():

```cpp
try
{
    mealTime.Set(13, 30, 'P');
    cout << "This is a valid time\n";
}
catch (char badTime[])  
{  
cout << "ERROR: " << badTime << endl;  
exit(-1);  
}  
cout << "Proceeding. . .\n";

When the code in the try block is executed and no exceptions are thrown, all of the catch blocks are skipped and execution continues with the statement after the last one. If an exception is thrown, execution leaves the try block and the attached catch blocks are searched for one whose parameter type matches the type of exception. If one is found, its exception handler is executed; otherwise, the catch blocks of any enclosing try blocks are searched. If none are found, execution terminates.

The types of exceptions that a function can throw can be declared by attaching an exception specification of the form throw(exception_list):

    ReturnType Name(parameterlist) throw(exc1, exc2, ...);

This function can throw only the exceptions listed and exceptions derived from them. If it attempts to do otherwise, the function std::unexpected() is called, which will terminate execution (unless unexpected() is redefined by calling set_unexpected()) or which will throw bad_exception if std::bad_exception is included in the list of exceptions.

There are several standard exceptions provided in C++. They are listed in the following table. They are all derived from the class exception() provided in <stdexcept>, which in addition to member function throw() also has a virtual member function what().

**Standard Exceptions**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Thrown by</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad_alloc</td>
<td>new()</td>
</tr>
<tr>
<td>bad_cast</td>
<td>dynamic_cast()</td>
</tr>
<tr>
<td>bad_typeid</td>
<td>typeid()</td>
</tr>
<tr>
<td>bad_exception</td>
<td>exception specification</td>
</tr>
<tr>
<td>out_of_range</td>
<td>at() and [] in bitset</td>
</tr>
<tr>
<td>invalid_argument</td>
<td>bitset constructor</td>
</tr>
<tr>
<td>overflow_error</td>
<td>to_ulong() in bitset</td>
</tr>
<tr>
<td>ios_base::failure</td>
<td>ios_base::clear()</td>
</tr>
</tbody>
</table>

For example, by the at member function of vector throws an out-of-range exception if the index gets out of range. The following code segment illustrates:

```cpp
vector<int> v(4, 99);

try  
{  
for (int i = 0; i < 5; i++)  
cout << v.at(i) << endl;  
}  
...  
catch(out_of_range exception)  
{  
cout << "Exception occurred: "  
    << exception.what() << endl;  
}  
```
The member function `what()` used in the output statement returns a string describing the exception. In one version of C++, the output produced was

```
99
99
99
99
```

`Exception occurred: vector::at out of range`

The statements

```cpp
double * ptr;
try {
    ptr = new double[1000000];
}
catch(bad_alloc exception) {
    cout << "Exception occurred: 
         << exception.what() << endl;
}
```

produced

```
Exception occurred: Allocation Failure
```