Chapter 17: Exceptions

From the board game MONOPOLY, the rule to follow when your man lands on the "illegal" square:

> Go to jail. Go directly to jail, do not pass GO and do not collect $200.

17.1 Handling Errors in a Program

Function calls, loops, conditionals, switches, and breaks permit the programmer to control and direct the sequence of evaluation of a program and modify the default order of execution, which is sequential. These statements permit controlled, local perturbations in the order of execution. Almost all situations that arise in programming can be handled well using some combination of these statements.

The goto statement is also supported in C, but its use is discouraged because it makes uncontrolled, non-local changes in execution sequence. Use of a goto is even worse in C++; if it is used to jump around the normal block entry or exit code, the normal construction, initialization, and deletion of objects can be short-circuited, potentially leaving the memory in an inconsistent state. Therefore, this statement should simply not be used.

There are some situations in which the ordinary structured control statements do not work well. These involve unusual situations, often caused by hardware or input errors, which prevent the program from making further fruitful progress. In C++, a failure during execution of a constructor is one situation that calls for use of exceptions.

In the old days, these situations would be handled by a variety of strategies:

1. Do nothing. Don't check for errors; hope they don't happen. Let the program crash or produce garbage answers if an error happens.

2. The program could identify the error and call an error function. However, this defers the problem instead of solving it. The error function still must do something about the error.

3. Identify the error, print an error comment, and call exit(). (This is equivalent to using the fatal() function in the toolspp library.) However, aborting execution is not permissible in many real-life situations. For example, aborting execution of a program that handles bank accounts could leave those accounts in an inconsistent or incorrect state.

4. One could use assert() to check for errors. This is similar to option (1) but worse because it gives no opportunity to print out information about the error or its cause.

5. The function being executed could return with an error code. The function that called it would need to check for that error code and return to *its* caller with an error code, and so on, until control returned to the level at which the error could be handled.

   This method usually works but distorts the logic of the program and clutters it with a large amount of code that is irrelevant 99% of the time. Using this technique discourages use of functions and modular code.

6. A long-distance goto could be used to return to the top level. Using this strategy, any information about the error would have to be stored in global variables before executing the goto. This is like programming in BASIC. Use of this control pattern destroys the modularity that C programs can otherwise achieve. It is not recommended.

17.1.1 What an Exception Handler Can Do

An exception handler provides an additional control option that is designed for dealing with unusual situations (exceptions) in which the ordinary structured control statements do not work well. (Exceptions can be used in non-error-conditions, but should not be used that way.) An exception handler lets control go ...

- From the level at which an error is discovered, often deep within the code, at a stage when several functions have been called and have not yet returned ...
- Carrying as much information as necessary about the error ...
- To the level at which the error can be handled, often in the main function or a high-level function that handles the overall progress of the program or operator interaction...
- And back into normal execution, if that is appropriate.

An exception handler and the exceptions it can handle are defined at a high level. When an exception condition is identified at a lower level, an exception object is created and "thrown" upward. This causes immediate exit from the originating function, with proper clean-up of its stack frame. The function that called it then "has the ball" and can either catch the exception or ignore it (ignoring the exception causes an immediate return to *its* caller). Thus, control passes backward along the chain of function calls until some function in the chain-of-command "catches" the exception. All stack frames in this chain are deleted and the relevant destructors are run. The programmer should be careful, therefore, not to allow exceptions to be thrown when some objects are half-constructed.

Implementing a compiler and run-time-system that can do this efficiently is a hard problem. Exceptions can come in many types and each type of exception must have a matching handler. An exception is an object that may have as many fields as necessary to contain all the important facts about where and why the exception was thrown. This object must outlive the function that created it (it must be created in dynamic storage, not on the stack) and it must carry an identifying type-tag, so it can be matched to the appropriate exception handler. An uncaught exception will abort a process.

What next? After catching an exception, there are several options for handling it:

1. The catcher can clean up the data and execute return with some appropriate value.
2. The catcher can clean up the data and call the containing function recursively.
3. The catcher can clean up the data and continue from the line that follows the exception handler (not the line that follows the function call that caused the exception).
4. The catcher can comment, get more information from the operator, and carry on in either of the two preceding ways.
5. The catcher can abort the process or return to its caller, as in C.
6. The catcher can fix some data fields and rethrow the same exception or some other exception.

17.2 Defining Exceptions in C++

One major principle applies to the use of exceptions: they are not a substitute for proper use of `else` or `while`, so the function that throws an exception should not be the function that catches it. Exceptions are intended for global, not local, use. Their proper use is to enable control and information to pass, directly, across classes and through a chain of function calls.

17.2.1 Defining an Exception Type

An exception is an object and its type is defined like any class. It can (but usually does not) have public, protected, and private parts. The exception class definition can be global or it can be contained within the definition of another class. Related exceptions can be created by derivation. For example, the Bad, BadSuit, and

BadSpot classes, below, define three exception types that might be used in a program that plays an interactive card game and reads input from the keyboard. The class Bad is a base class for the others and defines the functionality common to all three classes. The UML diagram in Figure17.1 shows the derivation hierarchy. The purple circle with a V marks a virtual function.

```
1  //============================================================================
2  // Exception Classes for Playing Card Errors.                    file: bad.hpp
3  // Exception demonstration program:  March 2009
4  // ---------------------------------------------- Error reading input stream.
5  #pragma once;
6  #include "tools.hpp"
7
8  class Bad {
9  public:
10     char spot;
11     char suit;
12     //------------------------------------------- Suit and Spot both wrong.
13     Bad (char n, char s) : spot(n),  suit(s) {};
14     virtual void print(){
15         cerr <<"  Both spot value and suit are wrong\n"
16         <<"  Legal spot values are 2..9, T, J, Q, K, A\n"
17         <<"  Legal suits are H D C S\n";
18         pr();
19     }
20     void pr(){
21         cerr <<"  You entered "<<spot <<" of " <<suit
22         <<". Please reenter. \n";
23     }
24 };
25
26 // ------------------------------------------------- Only the suit is wrong.
27 class  BadSuit : public Bad {
28 public:
29     BadSuit (char n, char s) : Bad(n, s) {}
30     virtual void print(){
31         cerr <<"  Legal suits are H D C S\n";
32         pr();
33     }
34 };
35
36 //--------------------------------------------- Only the spot value is wrong.
37 class  BadSpot : public Bad {
38 public:
39     BadSpot (char n, char s) : Bad(n, s) {}
40     virtual void print(){
41         cerr <<"  Legal spot values are 2..9, T, J, Q, K, A\n";
42         pr();
43     }
44 };
```

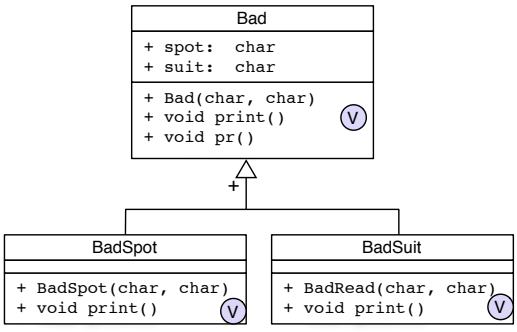

Figure 17.1: UML diagram for the exception classes.

17.2.2 Exception Specifications

In Java, every class must declare the exceptions that might happen within it, unless they are also caught within it. This requirement is confusing to beginners, but it helps make Java programs more predictable and it leads to better-informed error handling. In C++, a function *may* declare a list of potential exceptions, or declare that it does not throw exceptions. However, such a declaration is not required. Since an uncaught exception will terminate a program, this can be an important kind of documentation. Note: this is a new feature of C++ and may not be supported by all compilers.

The exception specification follows the parameter list in a prototype and precedes the semicolon. For example, suppose a fuction named `divide(int, int)` could throw two kinds of exceptions. The declaration would be:

```
int divide (int, int) throw (Zerodivide, MyException);
```

We can also declare that a fuction does not throw exceptions at all. The syntax is:

```
int safeFunction (int, int) throw ();
```

Technically, this declaration is legal even if some function called by safeFunction throws exceptions. However, since we can't know (let alone control) which library functions throw exceptions, this kind of declaration has questionable value.

17.2.3 Playing card demo.

Next is a simple class that models a playing card. We will use it to demonstrate throwing and catching exceptions. The constructor for this class takes input from the keyboard, a notoriously error-prone data source. Each card is represented by two chars representing the face value (spot) and the suit of the card. Spaces and capitalization are not important and can be used freely. After reading the two chars, the Card constructor validates them and (possibly) throws an exception. The nature of the input error is encoded in the name of the exception, enabling the program to use virtual functions to display a specific and appropriate error comment each time.

```
45  //============================================================================
46  //  A playing card class and related exception classes.
47  //  Alice E. Fischer, March 2009                                  cards.hpp
48  //============================================================================
49  #pragma once;
50  #include "tools.hpp"
51  #include "bad.hpp"
52
53  enum SuitType{ spades, hearts, diamonds, clubs, bad };
54
55  //============================================================================
56  // This is the main data class; it represents one playing card.
57  //
58  class  Card {
59      int         spot_;
60      SuitType    suit_;
61  public:
62      Card (istream& sin) throw (Bad, BadSpot, BadSuit);
63      int         spot(){ return spot_; }
64      SuitType    suit(){ return suit_; }
65      ostream&    print(ostream&);
66      static void instructions(ostream&, int n);
67  };
```

17.2.4 Throwing an Exception

The `throw` statement is used to construct an exception and propagate it. One type of exception is commonly used without definition; a function may throw a literal string (type `string` or `char*`). Throwing an exception causes control to pass backwards through the chain of function calls to the nearest previous catch clause that

handles that particular type of exception. The destructors will be run for all objects in the stack frames between the throw and the catch.

```
68  //============================================================================
69  //  Functions and constants for the Card class.
70  //  Alice E. Fischer, March 2009                                  cards.cpp
71  //============================================================================
72  #include "cards.hpp"
73  const char* suitlabels[5] = {"spades", "hearts", "diamonds", "clubs", "bad"};
74
75  const char* spotlabels[16] = {
76      "bad","Ace","2","3","4","5","6","7","8","9",
77      "10","Jack","Queen","King","Ace"
78  };
79
80  // ---------------------------------------------------------------------------
81  void Card::instructions( ostream& out, int n) {
82      out << "Please enter " << n << " cards.\n"
83          << "Spot codes are 2..9, T, J, Q, K, A \n"
84          << "Suit codes are S H D C \n";
85  }
86
87  // ---------------------------------------------------------------------------
88  Card::Card (istream& sin) throw (Bad, BadSpot, BadSuit) {
89      char inspot, insuit;
90      sin >> inspot >> insuit;
91      if (!sin.good()) throw "Low level read error\n";
92      if (inspot >='2' && inspot<='9') spot_ = inspot - '0';
93      else switch( toupper(inspot) ){
94          case 'T': spot_ = 10; break;
95          case 'J': spot_ = 11; break;
96          case 'Q': spot_ = 12; break;
97          case 'K': spot_ = 13; break;
98          case 'A': spot_ = 1; break;
99          default : spot_ = 0;
100     };
101     switch( toupper(insuit) ){
102         case 'S': suit_ = spades;   break;
103         case 'H': suit_ = hearts;   break;
104         case 'D': suit_ = diamonds; break;
105         case 'C': suit_ = clubs;    break;
106         default : suit_ = bad;
107     };
108     if (spot_ == 0 && suit_ == bad) throw Bad(inspot, insuit);
109     if (spot_ == 0)   throw BadSpot(inspot, insuit);
110     if (suit_ == bad) throw BadSuit(inspot, insuit);
111 }
112
113 // ---------------------------------------------------------------------------
114 ostream&
115 Card::print(ostream& sout) {
116     return sout <<spotlabels[spot_] <<" of " <<suitlabels[suit_] <<endl;
117 }
```

In this program, line 91 throws a string exception. This will be caught by the general exception handler on line 153. Lines 108–10 throw exceptions from the Bad hierarchy. These will be caught on line 146. The names of these exceptions are announced on line 88. (This is a worthy but optional form of program documentation. It is not clear how much checking, if any, compiler does with these declarations.)

17.2.5 Catching an Exception

```
118  //=============================================================================
119  //  Exception handling demonstration.
120  //  Alice E. Fischer, August 2009                                    except.cpp
121  //=============================================================================
122  #include "tools.hpp"
123  #include "cards.hpp"
124  #include "bad.hpp"
125  #define NCARDS 3
126
127  int main( void )
128  {
129      Card* hand[NCARDS];
130      int k;
131      bool success;
132      Card::instructions( cout, NCARDS );
133      //---------------------------------------- Main loop that reads all cards.
134      for (k=0; k<NCARDS; ){
135          success = false;         // Will not be changed if an exception happens.
136          //------------------------------ Here is the single line of active code.
137          try {
138              cout << "\nEnter a card (spot code, suit code): " ;
139              hand[k] = new Card(cin);    //---------------------- Input one card.
140              success = true;             //-- No exception - we have a good card.
141              cout << "  Card successfully entered into hand: ";
142              hand[k]->print(cout);
143              ++k;
144          }
145          //----------------- Check for the three application-specific exceptions.
146          catch (Bad& bs) { bs.print(); }          // Will catch all 3 Bad errors.
147
148          //------------------- Now check for general exceptions thrown by system.
149          catch (bad_alloc bs) {          //-------------- Catch a malloc failure.
150              cerr << "  Allocation error for card #" <<k <<".\n";
151              return 1;
152          }
153          catch (...) {                   //--------------- Catch everything else.
154              cerr << "  Last-ditch effort to catch exceptions.\n";
155          }
156          // ----------------- Control comes here after the try/catch is finished.
157          if(!success) delete hand[k];       // ----- Delete the half-made object.
158      }
159      cout << "\nHand is complete:" << endl;
160      for (k = 0; k < NCARDS; ++k) { hand[k]->print( cout ); }
161  }
```

Exceptions are caught and processed by exception handlers. A handler is defined like an ordinary function; the type of its parameter is the type of exception that it will catch.

Code that may generate exceptions (at any nesting level) and wishes to catch those exceptions must be enclosed in a `try` block (lines 137..144). The try block can contain multiple statements of any and all sorts. The exception handlers are written in `catch` blocks that immediately follow the `try` block (lines 146..155). Several things should be noted here:

1. The fields inside an exception object may be used according to the normal rules of public and/or private access. The data can be public because there is no need to protect it.

2. If the handler does not need to access the information in the exception, the parameter name may be omitted.

3. The order in which the handlers are written is important; the general case must come after all related specific cases.

4. A base-class exception handler will catch exceptions of all derived types. When an exception of a derived type is caught, the fields that belong to the derived type are not "visible" to the handler. If the exception is rethrown, all fields (including the fields of the derived type) are part of the rethrown object.

5. An exception handler whose parameter is ... will catch all exceptions.

In this example, line 139 calls the Card constructor, which can throw exceptions from the Bad class and its derived classes. The handler for those exceptions is on line 146. This line will catch all three kinds of exceptions and process them by calling the print function in the exception class.

Processing the exception. When an exception is caught by line 146, we call the `bs.print()` to process it. Note that the print function is virtual in the Bad class hierarchy, and has three defining methods. When `bs.print()` is called, the system will inspect the run-time type-tag attached to `bs` to find out which class or subclass constructed this particular exception. Then the `print()` function of the matching class will be called. This is how we get three different kinds of output from one catch clause. (Note the first, second, and third blocks of error comments in the output transcript, below.) If `Bad::print()` were not virtual, the base-class `print()` function would always be used.

Output The output that follows shows two sample runs of this program with different exception handlers active. The first output is from the program as shown:

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter a card (spot code, suit code): mn
  Both spot value and suit are wrong
  Legal spot values are 2..9, T, J, Q, K, A
  Legal suits are H D C S
  You entered m of n. Please reenter.

Enter a card (spot code, suit code): 2m
  Legal suits are H D C S
  You entered 2 of m. Please reenter.

Enter a card (spot code, suit code): 1s
  Legal spot values are 2..9, T, J, Q, K, A
  You entered 1 of s. Please reenter.

Enter a card (spot code, suit code): 2s
  Card successfully entered into hand: 2 of spades

Enter a card (spot code, suit code): kh
  Card successfully entered into hand: King of hearts

Enter a card (spot code, suit code): JC
  Card successfully entered into hand: Jack of clubs

Hand is complete:
2 of spades
King of hearts
Jack of clubs
```

The following output was produced after commenting out the first catch clause (line 146).

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter a card (spot code, suit code): mn
  Last-ditch effort to catch exceptions.

Enter a card (spot code, suit code): 3s
   Card successfully entered into hand: ... and so on.
```

17.2.6 Built-in Exceptions

Simple objects like iintegers and strings can also be thrown and caught. In addition, C++ has about a dozen built-in exceptions (see Schildt, pages 922-924). One of these, `bad_alloc` changes the way we write programs.

In C, it was necessary to check the result of every call on `malloc()`, to find out whether the system was able to fulfill the request. In C++, such a check is not necessary or helpful. If there is an allocation failure, the run-time system will throw a `bad_alloc` exception and control will not return to the failed call. If your program does not have a handler for such exceptions, and one occurs, the program will be terminated.

The `bad_cast` exception is used when a program executes an invalid downward dynamic cast, and `bad_exception` is thrown when a function violates its own exception specification.

One group of standard exceptions called `runtime_errors` are beyond the programmer's control and are used for mistakes in library functions or the run-time system. These include `overflow_error`, `range_error`, and `underflow_error`.

A final group of exceptions called `logic_errors` are defined by the standard but, so far as I know, not used by the system. They seem to be intended for use by any program when a run-time error is discovered. These include `domain_error`, `invalid_argument`, `length_error`, and `out_of_range`.

17.2.7 Summary

In the hands of an expert, exception handlers can greatly simplify error handling in a large application. However, they are not easy to use and using them without understanding is likely to lead to errors that are difficult to diagnose and cure. They are also difficult for a compiler to handle and force the compiler to interact with the host system in awkward ways. Virtual print functions are a powerful and simple tool to produce good error comments in situations where multiple faults can occur. Be aware, though, that processing a call on a virtual function takes more time than processing a non-virtual function in the same class.