# Exception Handling 16

*It's the exception that proves the rule.*

COMMON MAXIM *(possibly a corruption of something like: It's the exception that tests the rule.)*

## INTRODUCTION

One way to write a program is to first assume that nothing unusual or incorrect will happen. For example, if the program takes an entry off a list, you might assume that the list is not empty. Once you have the program working for the core situation where things always go as planned, you can then add code to take care of the exceptional cases. In C++, there is a way to reflect this approach in your code. Basically, you write your code as if nothing very unusual happens. After that, you use the C++ exception-handling facilities to add code for those unusual cases. Exception handling is commonly used to handle error situations, but perhaps a better way to view exceptions is as a way to handle "exceptional situations." After all, if your code correctly handles an "error," then it no longer is an error.

Perhaps the most important use of exceptions is to deal with functions that have some special case that is handled differently depending on how the function is used. Perhaps the function will be used in many programs, some of which will handle the special case in one way and some of which will handle it in some other way. For example, if there is a division by zero in the function, then it may turn out that for some invocations of the function, the program should end, but for other invocations of the function something else should happen. You will see that such a function can be defined to throw an exception if the special case occurs, and that exception will allow the special case to be handled outside of the function. That way, the special case can be handled differently for different invocations of the function.

In C++, exception handling proceeds as follows: Either some library software or your code provides a mechanism that signals when something unusual happens. This is called *throwing an exception*. At another place in your program, you place the code that deals with the exceptional case. This is called *handling the exception*. This method of programming makes for cleaner code. Of course, we still need to explain the details of how you do this in C++.

## PREREQUISITES

With the exception of one subsection that can be skipped, Section 16.1 uses material only from Chapters 2 to 6 and 10 to 11. The Pitfall subsection of Section 16.1 entitled "Exception Specification in Derived Classes" uses material from Chapter 15. This Pitfall subsection can be skipped without loss of continuity.

With the exception of one subsection that can be skipped, Section 16.2 uses material only from Chapters 2 to 8 and 10 to 12 and Section 15.1 of Chapter 15 in addition to Section 16.1. The subsection of Section 16.2 entitled "Testing for Available Memory" uses material from Chapter 15. This subsection can be skipped without loss of continuity.

## 16.1 EXCEPTION-HANDLING BASICS

*Well, the program works for most cases. I didn't know it had to work for that case.*

COMPUTER SCIENCE STUDENT, APPEALING A GRADE

Exception handling is meant to be used sparingly and in situations that are more involved than what is reasonable to include in a simple introductory example. So, we will teach you the exception-handling details of C++ by means of simple examples that would not normally use exception handling. This makes a lot of sense for learning about exception handling, but do not forget that these first examples are toy examples, and in practice, you would not use exception handling for anything that simple.

### A Toy Example of Exception Handling

For this example, suppose that milk is such an important food in our culture that people almost never run out of it, but still we would like our programs to accommodate the very unlikely situation of running out of milk. The basic code, which assumes we do not run out of milk, might be as follows:

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts/static_cast<double>(milk);
cout << donuts << " donuts.\n"
     << milk << " glasses of milk.\n"
     << "You have " << dpg
     << " donuts for each glass of milk.\n";
```

If there is no milk, then this code will include a division by zero, which is an error. To take care of the special situation in which we run out of milk, we can add a test for this unusual situation. The complete program with this added test for the special situation is shown in Display 16.1. The program in Display 16.1 does not use exception handling. Now, let's see how this program can be rewritten using the C++ exception-handling facilities.

## DISPLAY 16.1 Handling a Special Case Without Exception Handling

```
1   #include <iostream>
2   using namespace std;

3   int main()
4   {
5       int donuts, milk;
6       double dpg;
7       cout << "Enter number of donuts:\n";
8       cin >> donuts;
9       cout << "Enter number of glasses of milk:\n";
10      cin >> milk;

11      if (milk <= 0)
12      {
13          cout << donuts << " donuts, and No Milk!\n"
14               << "Go buy some milk.\n";
15      }
16      Else
17      {
18          dpg = donuts/static_cast<double>(milk);
19          cout << donuts << " donuts.\n"
20               << milk << " glasses of milk.\n"
21               << "You have " << dpg
22               << " donuts for each glass of milk.\n";
23      }

24      cout << "End of program.\n";
25      return 0;
26  }
```

### *Sample Dialogue*

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

In Display 16.2, we have rewritten the program from Display 16.1 using an exception. This is only a toy example, and you would probably not use an exception in this case. However, it does give us a simple example. Although the program as a whole is not simpler, at least the part between the words *try* and *catch* is cleaner, and this hints at the advantage of using exceptions. Look

**DISPLAY 16.2 Same Thing Using Exception Handling** *(part 1 of 2)*

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6       int donuts, milk;
7       double dpg;
8
9       try
10      {
11          cout << "Enter number of donuts:\n";
12          cin >> donuts;
13          cout << "Enter number of glasses of milk:\n";
14          cin >> milk;
15
16          if (milk <= 0)
17              throw donuts;
18
19          dpg = donuts/static_cast<double>(milk);
20          cout << donuts << " donuts.\n"
21               << milk << " glasses of milk.\n"
22               << "You have " << dpg
23               << " donuts for each glass of milk.\n";
24      }
25      catch(int e)
26      {
27          cout << e << " donuts, and No Milk!\n"
28               << "Go buy some milk.\n";
29      }
30
31      cout << "End of program.\n";
32      return 0;
33  }
```

***Sample Dialogue 1***

```
Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.
```

*(continued)*

**DISPLAY 16.2 Same Thing Using Exception Handling** *(part 2 of 2)*

***Sample Dialogue 2***

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

at the code between the words *try* and *catch*. That code is basically the same as the code in Display 16.1, but rather than the big *if-else* statement (shown in color in Display 16.1) this new program has the following smaller *if* statement (plus some simple nonbranching statements):

```
if (milk <= 0)
    throw donuts;
```

This *if* statement says that if there is no milk, then do something exceptional. That something exceptional is given after the word *catch*. The idea is that the normal situation is handled by the code following the word *try*, and that the code following the word *catch* is used only in exceptional circumstances. We have thus separated the normal case from the exceptional case. In this toy example, this separation does not really buy us too much, but in other situations it will prove to be very helpful. Let's look at the details.

The basic way of handling exceptions in C++ consists of the *try-throw-catch* threesome. A ***try* block** has the syntax

```
try
{
    Some_Code
}
```

This *try* block contains the code for the basic algorithm that tells the computer what to do when everything goes smoothly. It is called a *try* block because you are not 100 percent sure that all will go smoothly, but you want to "give it a try."

Now if something *does* go wrong, you want to throw an exception, which is a way of indicating that something went wrong. The basic outline, when we add a *throw*, is as follows:

```
try
{
```

```
    Code_To_Try
    Possibly_Throw_An_Exception
    More_Code
}
```

The following is an example of a *try* block with a *throw* statement included (copied from Display 16.2):

```
try
{
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;
    if (milk <= 0)
        throw donuts;
    dpg = donuts/static_cast<double>(milk);
    cout << donuts << " donuts.\n"
         << milk << " glasses of milk.\n"
         << "You have " << dpg
         << " donuts for each glass of milk.\n";
}
```

The following statement **throws** the *int* value donuts:

```
throw donuts;
```

The value thrown, in this case `donuts`, is sometimes called an **exception,** and the execution of a *throw* statement is called **throwing an exception.** You can throw a value of any type. In this case, an *int* value is thrown.

### *throw* Statement

**SYNTAX**

```
throw Expression_for_Value_to_Be_Thrown;
```

When the *throw* statement is executed, the execution of the enclosing *try* block is stopped. If the *try* block is followed by a suitable *catch* block, then flow of control is transferred to the *catch* block. A *throw* statement is almost always embedded in a branching statement, such as an *if* statement. The value thrown can be of any type.

**EXAMPLE**

```
if (milk <= 0)
    throw donuts;
```

As the name suggests, when something is "thrown," something goes from one place to another place. In C++, what goes from one place to another is the flow of control (as well as the value thrown). When an exception is thrown, the code in the surrounding *try* block stops executing and another portion of code, known as a ***catch* block,** begins execution. This executing of the *catch* block is called catching the exception or handling the exception. When an exception is thrown, it should ultimately be handled by (caught by) some *catch* block. In Display 16.2, the appropriate *catch* block immediately follows the *try* block. We repeat the *catch* block here:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

This *catch* block looks very much like a function definition that has a parameter of a type *int*. It is not a function definition, but in some ways, a *catch* block is like a function. It is a separate piece of code that is executed when your program encounters (and executes) the following (within the preceding *try* block):

```
throw Some_int;
```

So, this *throw* statement is similar to a function call, but instead of calling a function, it calls the *catch* block and says to execute the code in the *catch* block. A *catch* block is often referred to as an **exception handler,** which is a term that suggests that a *catch* block has a function-like nature.

What is that identifier `e` in the following line from a *catch* block?

```
catch(int e)
```

That identifier `e` looks like a parameter and acts very much like a parameter. So, we will call this `e` the ***catch*-block parameter.** (But remember, this does not mean that the *catch* block is a function.) The *catch*-block parameter does two things:

1. The *catch*-block parameter is preceded by a type name that specifies what kind of thrown value the *catch* block can catch.

2. The *catch*-block parameter gives you a name for the thrown value that is caught, so you can write code in the *catch* block that does things with the thrown value that is caught.

We will discuss these two functions of the *catch*-block parameter in reverse order. In this subsection, we will discuss using the *catch*-block parameter as a name for the value that was thrown and is caught. In the subsection entitled "Multiple Throws and Catches," later in this chapter, we will discuss which *catch* block (which exception handler) will process a value that is thrown. Our

current example has only one *catch* block. A common name for a *catch*-block parameter is e, but you can use any legal identifier in place of e.

Let's see how the *catch* block in Display 16.2 works. When a value is thrown, execution of the code in the *try* block ends and control passes to the *catch* block (or blocks) that are placed right after the *try* block. The *catch* block from Display 16.2 is reproduced here:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

When a value is thrown, the thrown value must be of type *int* in order for this particular *catch* block to apply. In Display 16.2, the value thrown is given by the variable donuts, and since donuts is of type *int*, this *catch* block can catch the value thrown.

Suppose the value of donuts is 12 and the value of milk is 0, as in the second sample dialogue in Display 16.2. Since the value of milk is not positive, the *throw* statement within the *if* statement is executed. In that case, the value of the variable donuts is thrown. When the *catch* block in Display 16.2 catches the value of donuts, the value of donuts is plugged in for the *catch*-block parameter e and the code in the *catch* block is executed, producing the following output:

```
12 donuts, and No Milk!
Go buy some milk.
```

If the value of donuts is positive, the *throw* statement is not executed. In this case, the entire *try* block is executed. After the last statement in the *try* block is executed, the statement after the *catch* block is executed. Note that if no exception is thrown, then the *catch* block is ignored.

This makes it sound like a *try-throw-catch* setup is equivalent to an *if-else* statement. It almost is equivalent, except for the value thrown. A *try-throw-catch* setup is similar to an *if-else* statement *with the added ability to send a message to one of the branches*. This does not sound much different from an *if-else* statement, but it turns out to be a big difference in practice.

To summarize in a more formal tone, a *try* block contains some code that we are assuming includes a *throw* statement. The *throw* statement is normally executed only in exceptional circumstances, but when it is executed, it throws a value of some type. When an exception (a value like donuts in Display 16.2) is thrown, that is the end of the *try* block. All the rest of the code in the *try* block is ignored and control passes to a suitable *catch* block. A *catch* block applies only to an immediately preceding *try* block. If the exception is thrown, then that exception object is plugged in for the *catch*-block parameter, and the statements in the *catch* block are executed. For example, if you look at the dialogues in Display 16.2, you will see that as soon as the user

### *catch*-Block Parameter

The *catch*-block parameter is an identifier in the heading of a *catch* block that serves as a placeholder for an exception (a value) that might be thrown. When a (suitable) value is thrown in the preceding *try* block, that value is plugged in for the *catch*-block parameter. You can use any legal (nonreserved word) identifier for a *catch*-block parameter.

**EXAMPLE**

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

e is the *catch*-block parameter.

enters a nonpositive number, the *try* block stops and the *catch* block is executed. For now, we will assume that every *try* block is followed by an appropriate *catch* block. We will later discuss what happens when there is no appropriate *catch* block.

Next, we summarize what happens when no exception is thrown in a *try* block. If no exception (no value) is thrown in the *try* block, then after the *try* block is completed, program execution continues with the code after the *catch* block. In other words, if no exception is thrown, then the *catch* block is ignored. Most of the time when the program is executed, the *throw* statement will not be executed, and so in most cases, the code in the *try* block will run to completion and the code in the *catch* block will be ignored completely.

### *try-throw-catch*

This is the basic mechanism for throwing and catching exceptions. The *throw* **statement** throws the exception (a value). The *catch* **block** catches the exception (the value). When an exception is thrown, the *try* block ends and then the code in the *catch* block is executed. After the *catch* block is completed, the code after the *catch* block(s) is executed (provided the *catch* block has not ended the program or performed some other special action).

If no exception is thrown in the *try* block, then after the *try* block is completed, program execution continues with the code after the *catch* block(s). (In other words, if no exception is thrown, then the *catch* block(s) are ignored.)

**SYNTAX**

```
try
{
    Some_Statements
        < Either some code with a throw statement or a
              function invocation that might throw an
              exception>
    Some_More_Statements
}
catch(Type_Name e)
{
        < Code to be performed if a value of the
          catch-block parameter type is thrown in the
          try block>
}
```

**EXAMPLE**

See Display 16.2.

## SELF-TEST EXERCISES

1. What output is produced by the following code?

```
int wait_time = 46;
try
{
    cout << "Try block entered.\n";
    if (wait_time > 30)
        throw wait_time;
    cout << "Leaving try block.\n";
}
catch(int thrown_value)
{
    cout << "Exception thrown with\n"
         << "wait_time equal to " << thrown_value << endl;
}
cout << "After catch block." << endl;
```

2. What would be the output produced by the code in Self-Test Exercise 1 if we make the following change? Change the line

```
int wait_time = 46;
```

to

```
int wait_time = 12;
```

3. In the code given in Self-Test Exercise 1, what is the *throw* statement?

4. What happens when a *throw* statement is executed? This is a general question. Tell what happens in general, not simply what happens in the code in Self-Test Question 1 or some other sample code.

5. In the code given in Self-Test Exercise 1, what is the *try* block?

6. In the code given in Self-Test Exercise 1, what is the *catch* block?

7. In the code given in Self-Test Exercise 1, what is the *catch*-block parameter?

## Defining Your Own Exception Classes

A *throw* statement can throw a value of any type. A common thing to do is to define a class whose objects can carry the precise kind of information you want thrown to the *catch* block. An even more important reason for defining a specialized exception class is so that you can have a different type to identify each possible kind of exceptional situation.

An exception class is just a class. What makes it an exception class is how it's used. Still, it pays to take some care in choosing an exception class's name and other details. Display 16.3 contains an example of a program with a programmer-defined exception class. This is just a toy program to illustrate some C++ details about exception handling. It uses much too much machinery for such a simple task, but it is an otherwise uncluttered example of some C++ details.

Notice the *throw* statement, reproduced in what follows:

```
throw NoMilk(donuts);
```

The part `NoMilk(donuts)` is an invocation of a constructor for the class `NoMilk`. The constructor takes one *int* argument (in this case `donuts`) and creates an object of the class `NoMilk`. That object is then "thrown."

## Multiple Throws and Catches

A *try* block can potentially throw any number of exception values, and they can be of differing types. In any one execution of the *try* block, only one exception will be thrown (since a thrown exception ends the execution of the *try* block), but different types of exception values can be thrown on different occasions when the *try* block is executed. Each *catch* block can only catch values of one type, but you can catch exception values of differing types by placing more than one *catch* block after a *try* block. For example, the program in Display 16.4 has two *catch* blocks after its *try* block.

Note that there is no parameter in the *catch* block for `DivideByZero`. If you do not need a parameter, you can simply list the type with no parameter.

## DISPLAY 16.3 Defining Your Own Exception Class

```
#include <iostream>
using namespace std;

class NoMilk
{
public:
    NoMilk();
    NoMilk(int how_many);
    int get_donuts();
private:
    int count;
};

int main()
{
    int donuts, milk;
    double dpg;
    try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;
        if (milk <= 0)
                throw NoMilk(donuts);
        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }
    catch(NoMilk e)
    {
        cout << e.get_donuts() << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }
    cout << "End of program.";
    return 0;
}

NoMilk::NoMilk()
{}
NoMilk::NoMilk(int how_many) : count(how_many)
{}

int NoMilk::get_donuts()
{
    return count;
}
```

*This is just a toy example to learn C++ syntax. Do not take it as an example of good typical use of exception handling.*

*The sample dialogues are the same as in Display 16.2.*

**DISPLAY 16.4 Catching Multiple Exceptions** *(part 1 of 2)*

*Although not done here, exception classes can have their own interface and implementation files and can be put in a namespace. This is another toy example.*

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class NegativeNumber
6   {
7   public:
8       NegativeNumber();
9       NegativeNumber(string take_me_to_your_catch_block);
10      string get_message();
11  private:
12      string message;
13  };
14
15  class DivideByZero
16  {};
17
18  int main()
19  {
20      int jem_hadar, klingons;
21      double portion;
22
23  try
24  {
25      cout << "Enter number of JemHadar warriors:\n";
26      cin >> jem_hadar;
27      if (jem_hadar< 0)
28          throw NegativeNumber("JemHadar");
29
30      cout << "How many Klingon warriors do you have?\n";
31      cin >> klingons;
32      if (klingons< 0)
33          throw NegativeNumber("Klingons");
34      if (klingons != 0)
35          portion = jem_hadar/static_cast<double>(klingons);
36      else
37          throw DivideByZero();
38      cout << "Each Klingon must fight "
39           << portion << " JemHadar.\n";
40  }
41  catch(NegativeNumber e)
42  {
43      cout << "Cannot have a negative number of "
44           << e.get_message() << endl;
45  }
```

*(continued)*

**DISPLAY 16.4 Catching Multiple Exceptions** *(part 2 of 2)*

```
46          catch (DivideByZero)
47          {
48              cout << "Send for help.\n";
49          }
50
51          cout << "End of program.\n";
52          return 0;
53      }
54
55
56      NegativeNumber::NegativeNumber()
57      {}
58
59      NegativeNumber::NegativeNumber(string take_me_to_your_catch_block)
60           : message(take_me_to_your_catch_block)
61      {}
62
63      string NegativeNumber::get_message()
64      {
65          return message;
66      }
```

***Sample Dialogue 1***

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
500
Each Klingon must fight 2.0 JemHadar.
End of program
```

***Sample Dialogue 2***

```
Enter number of JemHadar warriors:
-10
Cannot have a negative number of JemHadar
End of program.
```

***Sample Dialogue 3***

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
0
Send for help.
End of program.
```

This case is discussed a bit more in the Programming Tip section entitled "Exception Classes Can Be Trivial."

## PITFALL Catch the More Specific Exception First

When catching multiple exceptions, the order of the *catch* blocks can be important. When an exception value is thrown in a *try* block, the following *catch* blocks are tried in order, and the first one that matches the type of the exception thrown is the one that is executed.

For example, the following is a special kind of *catch* block that will catch a thrown value of any type:

```
catch(...)
{
    <Place whatever you want in here>
}
```

The three dots do not stand for something omitted. You actually type in those three dots in your program. This makes a good default *catch* block to place after all other *catch* blocks. For example, we could add it to the *catch* blocks in Display 16.4 as follows:

```
catch(NegativeNumber e)
{
    cout << "Cannot have a negative number of "
         << e.get_message() <<endl;
}
catch(DivideByZero)
{
    cout<< "Send for help.\n";
}
catch(...)
{
    cout << "Unexplained exception.\n";
}
```

However, it only makes sense to place this default *catch* block at the end of a list of *catch* blocks. For example, suppose we instead used:

```
catch(NegativeNumber e)
{
    cout << "Cannot have a negative number of "
         << e.get_message() <<endl;
}
catch(...)
{
    cout << "Unexplained exception.\n";
}
catch(DivideByZero)
```

```
{
    cout << "Send for help.\n";
}
```

With this second ordering, an exception (a thrown value) of type `NegativeNumber` will be caught by the `NegativeNumber` *catch* block, as it should be. However, if a value of type `DivideByZero` were thrown, it would be caught by the block that starts *catch*`(...)`. So, the `DivideByZero` *catch* block could never be reached. Fortunately, most compilers tell you if you make this sort of mistake. ■

## ■ PROGRAMMING TIP Exception Classes Can Be Trivial

Here we reproduce the definition of the exception class `DivideByZero` from Display 16.4:

```
class DivideByZero
{};
```

This exception class has no member variables and no member functions (other than the default constructor). It has nothing but its name, but that is useful enough. Throwing an object of the class `DivideByZero` can activate the appropriate *catch* block, as it does in Display 16.4.

When using a trivial exception class, you normally do not have anything you can do with the exception (the thrown value) once it gets to the *catch* block. The exception is just being used to get you to the *catch* block. Thus, you can omit the *catch*-block parameter. (You can omit the *catch*-block parameter anytime you do not need it, whether the exception type is trivial or not.) ■

### Throwing an Exception in a Function

Sometimes it makes sense to delay handling an exception. For example, you might have a function with code that throws an exception if there is an attempt to divide by zero, but you may not want to catch the exception in that function. Perhaps some programs that use that function should simply end if the exception is thrown, and other programs that use the function should do something else. So you would not know what to do with the exception if you caught it inside the function. In these cases, it makes sense to not catch the exception in the function definition, but instead to have any program (or other code) that uses the function place the function invocation in a *try* block and catch the exception in a *catch* block that follows that *try* block.

Look at the program in Display 16.5. It has a *try* block, but there is no *throw* statement visible in the *try* block. The statement that does the throwing in that program is

```
if (bottom == 0)
    throw DivideByZero();
```

**DISPLAY 16.5 Throwing an Exception Inside a Function** *(part 1 of 2)*

```
1   #include <iostream>
2   #include <cstdlib>
3   using namespace std;
4
5   class DivideByZero
6   {};
7
8   double safe_divide(int top, int bottom) throw (DivideByZero);
9
10  int main()
11  {
12      int numerator;
13      int denominator;
14      double quotient;
15      cout << "Enter numerator:\n";
16      cin >> numerator;
17      cout << "Enter denominator:\n";
18      cin >> denominator;
19
20      try
21      {
22          quotient = safe_divide(numerator, denominator);
23      }
24      catch(DivideByZero)
25      {
26          cout << "Error: Division by zero!\n"
27               << "Program aborting.\n";
28          exit(0);
29      }
30
31      cout << numerator << "/" << denominator
32           << " = " << quotient <<endl;
33
34      cout << "End of program.\n";
35      return 0;
36  }
37
38
39  double safe_divide(int top, int bottom) throw (DivideByZero)
40  {
41      if (bottom == 0)
42          throw DivideByZero();
43
44      return top/static_cast<double>(bottom);
45  }
```

*(continued)*

**DISPLAY 16.5 Throwing an Exception Inside a Function** *(part 2 of 2)*

***Sample Dialogue 1***

```
Enter numerator:
5
Enter denominator:
10
5/10 = 0.5
End of Program.
```

***Sample Dialogue 2***

```
Enter numerator:
5
Enter denominator:
0
Error: Division by zero!
Program aborting.
```

This statement is not visible in the *try* block. However, it is in the *try* block in terms of program execution, because it is in the definition of the function `safe_divide` and there is an invocation of `safe_divide` in the *try* block.

## Exception Specification

If a function does not catch an exception, it should at least warn programmers that any invocation of the function might possibly throw an exception. If there are exceptions that might be thrown, but not caught, in the function definition, then those exception types should be listed in an **exception specification**, which is illustrated by the following function declaration from Display 16.5:

```
double safe_divide(int top, int bottom) throw (DivideByZero);
```

As illustrated in Display 16.5, the exception specification should appear in both the function declaration and the function definition. If a function has more than one function declaration, then all the function declarations must have identical exception specifications. The exception specification for a function is also sometimes called the **throw list**.

If there is more than one possible exception that can be thrown in the function definition, then the exception types are separated by commas, as illustrated here:

```
void some_function( ) throw (DivideByZero, OtherException);
```

All exception types listed in the exception specification are treated normally. When we say the exception is treated normally, we mean it is treated as we have described before this subsection. In particular, you can place the function invocation in a *try* block followed by a *catch* block to catch that type of exception, and if the function throws the exception (and does not catch it inside the function), then the *catch* block following the *try* block will catch the exception. If there is no exception specification (no throw list) at all (not even an empty one), then it is the same as if all possible exception types were listed in the exception specification; that is, any exception that is thrown is treated normally.

What happens when an exception is thrown in a function but is not listed in the exception specification (and not caught inside the function)? In that case, the program ends. In particular, notice that if an exception is thrown in a function but is not listed in the exception specification (and not caught inside the function), then it will not be caught by any *catch* block, but instead your program will end. Remember, if there is no specification list at all, not even an empty one, then it is the same as if all exceptions were listed in the specification list, and so throwing an exception will not end the program in the way described in this paragraph.

Keep in mind that the exception specification is for exceptions that "get outside" the function. If they do not get outside the function, they do not belong in the exception specification. If they get outside the function, they belong in the exception specification no matter where they originate. If an exception is thrown in a *try* block that is inside a function definition and is caught in a *catch* block inside the function definition, then its type need not be listed in the exception specification. If a function definition includes an invocation of another function and that other function can throw an exception that is not caught, then the type of the exception should be placed in the exception specification.

To say that a function should not throw any exceptions that are not caught inside the function, you use an empty exception specification like so:

```
void some_function( ) throw ( );
```

By way of summary:

```
void some_function( ) throw (DivideByZero, OtherException);
//Exceptions of type DivideByZero or OtherException are
//treated normally. All other exceptions end the program
//if not caught in the function body.

void some_function( ) throw ( );
//Empty exception list; all exceptions end the
//program if thrown but not caught in the function body.

void some_function( );
//All exceptions of all types treated normally.
```

Keep in mind that an object of a derived class[1] is also an object of its base class. So, if D is a derived class of class B and B is in the exception specification, then a thrown object of class D will be treated normally, since it is an object of class B and B is in the exception specification. However, no automatic type conversions are done. If `double` is in the exception specification, that does not account for throwing an `int` value. You would need to include both `int` and `double` in the exception specification.

Warning!

One final warning: Not all compilers treat the exception specification as they are supposed to. Some compilers essentially treat the exception specification as a comment, and so with those compilers, the exception specification has no effect on your code. This is another reason to place all exceptions that might be thrown by your functions in the exception specification. This way all compilers will treat your exceptions the same way. Of course, you could get the same compiler consistency by not having any exception specification at all, but then your program would not be as well documented and you would not get the extra error checking provided by compilers that do use the exception specification. With a compiler that does process the exception specification, your program will terminate as soon as it throws an exception that you did not anticipate. (Note that this is a run-time behavior, but which run-time behavior you get depends on your compiler.)

### PITFALL Exception Specification in Derived Classes

When you redefine or override a function definition in a derived class, it should have the same exception specification as it had in the base class, or it should have an exception specification whose exceptions are a subset of those in the base class exception specification. Put another way, when you redefine or override a function definition, you cannot add any exceptions to the exception specification (but you can delete some exceptions if you want). This makes sense, since an object of the derived class can be used anyplace an object of the base class can be used, and so a redefined or overwritten function must fit any code written for an object of the base class. ■

## SELF-TEST EXERCISES

8. What is the output produced by the following program?

```
#include <iostream>
using namespace std;
void sample_function(double test) throw (int);
```

[1] If you have not yet learned about derived classes, you can safely ignore the remarks about them.

```
int main()
{
    try
    {
        cout << "Trying.\n";
        sample_function(98.6);
        cout << "Trying after call.\n";
    }
    catch(int)
    {
        cout << "Catching.\n";
    }
    cout << "End of program.\n";
    return 0;
}
void sample_function(double test) throw (int)
{
    cout << "Starting sample_function.\n";
    if (test < 100)
        throw 42;
}
```

9. What is the output produced by the program in Self-Test Exercise 8 if the following change were made to the program? Change

   `sample_function(98.6);`

   in the *try* block to

   `sample_function(212);`

## 16.2 PROGRAMMING TECHNIQUES FOR EXCEPTION HANDLING

*Only use this in exceptional circumstances.*

WARREN PEACE, *The Lieutenant's Tools*

So far, we have shown you lots of code that explains how exception handling works in C++, but we have not yet shown even one example of a program that makes good and realistic use of exception handling. However, now that you know the mechanics of exception handling, this section can go on to explain exception-handling techniques.

### When to Throw an Exception

We have given some very simple code in order to illustrate the basic concepts of exception handling. However, our examples were unrealistically simple. A more complicated but better guideline is to separate throwing an exception

and catching the exception into separate functions. In most cases, you should include any *throw* statement within a function definition, list the exception in the exception specification for that function, and place the *catch* clause in *a different function*. Thus, the preferred use of the *try-throw-catch* triad is as illustrated here:

```
void  functionA() throw (MyException)
{
        .
        .
        .
    throw MyException(<Maybe an argument>);
        .
        .
        .
}
```

Then, in *some other function* (perhaps even some other function in some other file), you have

```
void functionB()
{
        .
        .
        .
    try
    {
        .
        .
        .
      functionA();
        .
        .
        .
    }
    catch(MyException e)
    {
        <Handle exception.>
    }
        .
        .
        .
}
```

Moreover, even this kind of use of a *throw* statement should be reserved for cases in which it is unavoidable. If you can easily handle a problem in some other way, do not throw an exception. Reserve *throw* statements for situations in which the way the exceptional condition is handled depends on how and where the function is used. If the way that the exceptional condition is handled depends on how and where the function is invoked, then the

**When to Throw an Exception**

For the most part, *throw* statements should be used within functions and listed in an exception specification for the function. Moreover, they should be reserved for situations in which the way the exceptional condition is handled depends on how and where the function is used. If the way that the exceptional condition is handled depends on how and where the function is invoked, then the best thing to do is to let the programmer who invokes the function handle the exception. In all other situations, it is almost always preferable to avoid throwing an exception.

best thing to do is to let the programmer who invokes the function handle the exception. In all other situations, it is almost always preferable to avoid throwing exceptions.

## PITFALL Uncaught Exceptions

Every exception that is thrown by your code should be caught someplace in your code. If an exception is thrown but not caught anywhere, your program will end. ■

## PITFALL Nested *try-catch* Blocks

You can place a *try* block and following *catch* blocks inside a larger *try* block or inside a larger *catch* block. In rare cases, this may be useful, but if you are tempted to do this, you should suspect that there is a nicer way to organize your program. It is almost always better to place the inner *try-catch* blocks inside a function definition and place an invocation of the function in the outer *try* or *catch* block (or maybe just eliminate one or more *try* blocks completely).

If you place a *try* block and following *catch* blocks inside a larger *try* block, and an exception is thrown in the inner *try* block but not caught in the inner *try-catch* blocks, then the exception is thrown to the outer *try* block for processing and might be caught there. ■

## PITFALL Overuse of Exceptions

Exceptions allow you to write programs whose flow of control is so involved that it is almost impossible to understand the program. Moreover, this is not hard to do. Throwing an exception allows you to transfer flow of control

from anyplace in your program to almost anyplace else in your program. In the early days of programming, this sort of unrestricted flow of control was allowed via a construct known as a *goto*. Programming experts now agree that such unrestricted flow of control is very poor programming style. Exceptions allow you to revert to these bad old days of unrestricted flow of control. Exceptions should be used sparingly and only in certain ways. A good rule is the following: If you are tempted to include a *throw* statement, then think about how you might write your program or class definition without this *throw* statement. If you think of an alternative that produces reasonable code, then you probably do not want to include the *throw* statement. ■

## Exception Class Hierarchies


VideoNote
The STL Exception Class

It can be very useful to define a hierarchy of exception classes. For example, you might have an `ArithmeticError` exception class and then define an exception class `DivideByZeroError` that is a derived class of `ArithmeticError`. Since a `DivideByZeroError` is an `ArithmeticError`, every *catch* block for an `ArithmeticError` will catch a `DivideByZeroError`. If you list `ArithmeticError` in an exception specification, then you have, in effect, also added `DivideByZeroError` to the exception specification, whether or not you list `DivideByZeroError` by name in the exception specification.

## Testing for Available Memory

In Chapter 13, we created new dynamic variables with code such as the following:

```
struct Node
{
    int data;
    Node *link;
};
typedef Node* NodePtr;
    . . .
NodePtr pointer = new Node;
```

This works fine as long as there is sufficient memory available to create the new node. But, what happens if there is not sufficient memory? If there is not sufficient memory to create the node, then a `bad_alloc` exception is thrown. The type `bad_alloc` is part of the C++ language. You do not need to define it.

Since *new* will throw a `bad_alloc` exception when there is not enough memory to create the node, you can check for running out of memory as follows:

```
try
{
    NodePtr pointer = new Node;
}
```

```
catch (bad_alloc)
{
    cout << "Ran out of memory!";
}
```

Of course, you can do other things besides simply giving a warning message, but the details of what you do will depend on your particular programming task.

### Rethrowing an Exception

It is legal to throw an exception within a *catch* block. In rare cases, you may want to catch an exception and then, depending on the details, decide to throw the same or a different exception for handling farther up the chain of exception-handling blocks.

## SELF-TEST EXERCISES

10. What happens when an exception is never caught?

11. Can you nest a *try* block inside another *try* block?

## CHAPTER SUMMARY

- Exception handling allows you to design and code the normal case for your program separately from the code that handles exceptional situations.
- An exception can be thrown in a *try* block. Alternatively, an exception can be thrown in a function definition that does not include a *try* block (or does not include a *catch* block to catch that type of exception). In this case, an invocation of the function can be placed in a *try* block.
- An exception is caught in a *catch* block.
- A *try* block may be followed by more than one *catch* block. In this case, always list the *catch* block for a more specific exception class before the *catch* block for a more general exception class.
- Do not overuse exceptions.

### Answers to Self-Test Exercises

1.
```
Try block entered.
Exception thrown with
wait_time equal to 46
After catch block.
```

2. 
```
Try block entered.
Leaving try block.
After catch block.
```

3. 
```
throw wait_time;
```

   Note that the following is an *if* statement, not a *throw* statement, even though it contains a *throw* statement:

```
if (wait_time> 30)
    throw wait_time;
```

4. When a *throw* statement is executed, that is the end of the enclosing *try* block. No other statements in the *try* block are executed, and control passes to the following *catch* block(s). When we say control passes to the following *catch* block, we mean that the value thrown is plugged in for the *catch*-block parameter (if any), and the code in the *catch* block is executed.

5. 
```
try
{
    cout << "Try block entered.";
    if (wait_time > 30)
        throw (wait_time);
    cout << "Leaving try block.";
}
```

6. 
```
catch(int thrown_value)
{
    cout << "Exception thrown with\n"
         << "wait_time equal to" << thrown_value << endl;
}
```

7. `thrown_value` is the *catch*-block parameter.

8. 
```
Trying.
Starting sample_function.
Catching.
End of program.
```

9. 
```
Trying.
Starting sample_function.
Trying after call.
End of program.
```

10. If an exception is not caught anywhere, then your program ends.

11. Yes, you can have a *try* block and corresponding *catch* blocks inside another larger *try* block. However, it would probably be better to place the inner *try* and *catch* blocks in a function definition and place an invocation of the function in the larger *try* block.

## PRACTICE PROGRAMS

*Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.*

VideoNote
Solution to Practice Program 16.1

1. A function that returns a special error code is often better implemented by throwing an exception instead. This way, the error code cannot be ignored or mistaken for valid data. The following class maintains an account balance.

```
class Account
{
private:
    double balance;
public:
    Account()
    {
        balance = 0;
    }
    Account(double initialDeposit)
    {
        balance = initialDeposit;
    }
    double getBalance()
    {
        return balance;
    }
    // returns new balance or -1 if error
    double deposit(double amount)
    {
        if (amount > 0)
            balance += amount;
        else
            return -1; // Code indicating error
        return balance;
    }
    // returns new balance or -1 if invalid amount
    double withdraw(double amount)
    {
    if ((amount > balance) || (amount < 0))
        return -1;
    else
        balance -= amount;
    return balance;
    }
};
```

   Rewrite the class so that it throws appropriate exceptions instead of returning –1 as an error code. Write test code that attempts to withdraw and deposit invalid amounts and catches the exceptions that are thrown.

2. The Standard Template Library includes a class named `exception` that is the parent class for any exception thrown by an STL function. Therefore, any exception can be caught by this class. The following code sets up a *`try-catch`* block for STL exceptions:

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;

int main()
{
    string s = "hello";
    try
    {
       cout << "No exception thrown." << endl;
    }
    catch (exception& e)
    {
       cout << "Exception caught: " <<
            e.what() << endl;
    }
    return 0;
}
```

Modify the code so that an exception is thrown in the `try` block. You could try accessing an invalid index in a string using the `at` member function.

## PROGRAMMING PROJECTS

*Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.*

1. Write a program that converts 24-hour time to 12-hour time. The following is a sample dialogue:

```
Enter time in 24-hour notation:
13:07
That is the same as
1:07 PM
Again?(y/n)
y
Enter time in 24-hour notation:
10:15
That is the same as
10:15 AM
Again?(y/n)
y
```

```
Enter time in 24-hour notation:
10:65
There is no such time as 10:65
Try again:
Enter time in 24-hour notation:
16:05
That is the same as
4:05 PM
Again?(y/n)
n
End of program
```

You will define an exception class called `TimeFormatMistake`. If the user enters an illegal time, like `10:65` or even gibberish like `8&*68`, then your program will throw and catch a `TimeFormatMistake`.

2. Write a program that converts dates from numerical month/day format to alphabetic month/day (for example, 1/31 or 01/31 corresponds to January 31). The dialogue should be similar to that in Programming Project 1. You will define two exception classes, one called `MonthError` and another called `DayError`. If the user enters anything other than a legal month number (integers from 1 to 12), then your program will throw and catch a `MonthError`. Similarly, if the user enters anything other than a valid day number (integers from 1 to either 29, 30, or 31, depending on the month), then your program will throw and catch a `DayError`. To keep things simple, always allow 29 days for February.

VideoNote
Solution to Programming Project 16.3

3. Write a program that inputs numeric values from 1 through 10 and outputs a textual histogram of the values using *'s to count the number of occurrences of each value. The program should first ask the user how many numbers to enter. If the user enters a value that does not consist of all digits or a number outside the range 1 to 10, then an exception should be caught. (*Hint*: Input each number as a string, and then scan through the string to see if it contains all digits. If not, throw an exception. To convert a string `str` to an integer, use the following code:

```
atoi(str.c_str());
```

The `atoi` function is described in Chapter 8.) Here is a sample dialogue:

```
How many numbers to enter?
5
Enter number 1L
one
Please enter your number using digits only.  Try again.
Enter number 1:
9
Enter number 2:
```

```
3
Enter number 3:
3
Enter number 4:
33
The number must be between 1-10.  Try again.
Enter number 4:
3
Enter number 5:
7

Here is the histogram of values:
1 :
2 :
3 : ***
4 :
5 :
6 :
7 : *
8 :
9 : *
10:
```

4. Define a class named `CheckedArray`. The objects of this class are like regular arrays but have range checking. If `a` is an object of the class `CheckedArray` and `i` is an illegal index, then use of `a[i]` will cause your program to throw an exception (an object) of the class `ArrayOutOfRangeError`. Defining the class `ArrayOutOfRangeError` is part of this project. Note that, among other things, your `CheckedArray` class must have a suitable overloading of the `[]` operators, as discussed in Appendix 6.

5. Stacks were introduced in Chapters 13 and 14. Define a stack class for storing a stack of elements of type *`char`*. A stack object should be of fixed size; the size is a parameter to the constructor that creates the stack object. When used in a program, an object of the stack class will throw exceptions in the following situations:

   - Throw a `StackOverflowException` if the application program tries to push data onto a stack that is already full
   - Throw a `StackEmptyException` if the application program tries to pop data off an empty stack

   Defining the classes `StackOverflowException` and `StackEmptyException` is part of this project. Write a suitable test program.

6. (Based on a problem in Stroustrup, *The C++ Programming Language*, 3rd edition) Write a program consisting of functions calling one another to a calling depth of 10. Give each function an argument that specifies the level

at which it is to throw an exception. The `main` function prompts for and receives input that specifies the calling depth (level) at which an exception will be thrown. The `main` function then calls the first function. The `main` function catches the exception and displays the level at which the exception was thrown. Don't forget the case where the depth is 0, where `main` must both throw and catch the exception.

(*Hints:* You could use 10 different functions or 10 copies of the same function that call one another, but don't. Rather, for compact code, use a `main` function that calls another function that calls itself recursively. Suppose you do this; is the restriction on the calling depth necessary? This can be done without giving the function any additional arguments, but if you cannot do it that way, try adding an additional argument to the function.)

7. Programming Project 7 in Chapter 9 described a technique to emulate a two-dimensional array with wrapper functions around a one-dimensional array. If the indices of a desired entry in the two-dimensional array were invalid (for example, out of range), you were asked to print an error message and exit the program. Modify this program (or do it for the first time) to instead throw an `ArrayOutOfRangeError` exception if either the row or column indices are invalid. Your program should define the `ArrayOutOfRangeError` exception class.