Topic 8
Lecture 8
Exception Handling (Chapter 17)

CSCI 140: C++ Language & Objects

Prof. Dominick Atanasio

Book: C++: How to Program 10 ed.

Agenda

- Introduction
- Exception-Handling Flow of Control; Defining an Exception Class
- Rethrowing an Exception
- Stack Unwinding
- When to Use Exception Handling
- noexcept : Declaring Functions That Do Not Throw Exceptions
- Constructors, Destructors and Exception Handling
- Processing new Failures

17.1 Introduction

- An exception is an indication of a problem that occurs during a program's execution.
- Exception handling enables you to create applications that can handle exceptions and perform appropriate cleanup when an exception that cannot or should not be handled occurs.
- In many cases, this allows a program to continue executing as if no problem had been encountered.
- The features presented in this chapter enable you to write robust and fault-tolerant programs that can deal with problems that may arise and continue executing or terminate gracefully.

17.2 Exception Handling Flow of Control; Defining an Exception Class

- Let's consider a simple example of exception handling (Figs. 17.1–17.2).
- We show how to deal with a common arithmetic problem—division by zero.
- Division by zero using integer arithmetic typically causes a program to terminate prematurely.
- In floating-point arithmetic, many C++ implementations allow division by zero, in which case positive or negative infinity is displayed as inf or -inf, respectively.
- Typically, a program would simply test for division by zero before attempting the calculation—we use exceptions here to present the flow of control when a program executes successfully and when an exception occurs.

17.2 Example: Handling an Attempt to Divide by Zero (cont.)

- In this example, we define a function named quotient that receives two integers entered by the user and divides its first int parameter by its second int parameter.
- Before performing the division, the function casts the first int parameter's value to type double.
- Then, the second int parameter's value is promoted to type double for the calculation.
- So, function quotient actually performs the division using two double values and returns a double result.

17.2 Example: Handling an Attempt to Divide by Zero (cont.)

- For the purpose of this example, we treat any attempt to divide by zero as an error.
- Thus, function quotient tests its second parameter to ensure that it isn't zero before allowing the division to proceed.
- If the second parameter is zero, the function uses an exception to indicate to the caller that a problem occurred.
- The caller (main in this example) can then process the exception and allow the user to type two new values before calling function quotient again.
- In this way, the program can continue to execute even after an improper value is entered, thus making the program more robust.

17.2 Example: Handling an Attempt to Divide by Zero (cont.)

- DivideByZeroException.h (Fig. 17.1) defines an exception class that represents the type of the problem that might occur in the example, and fig17_02.cpp (Fig. 17.2) defines the quotient function and the main function that calls it.
- Function main contains the code that demonstrates exception handling.

17.2.1 Defining an Exception Class to Represent the Type of Problem That Might Occur

- Figure 17.1 defines class DivideByZeroException as a derived class of Standard Library class runtime_error (from header file <stdexcept>).
- Class runtime_error—a derived class of exception (from header file <exception>)—is the C++ standard base class for representing runtime errors.
- Class exception is the standard C++ base class for exception in the C++ Standard Library.
  - Section 17.10 discusses class exception and its derived classes in detail.

17.2.1 Defining an Exception Class to Represent the Type of Problem That Might Occur

- A typical exception class that derives from the runtime_error class defines only a constructor (e.g., lines 10–11) that passes an error-message string to the base-class runtime_error constructor.
- Every exception class that derives directly or indirectly from exception contains the virtual function what, which returns an exception object's error message.
- You are not required to derive a custom exception class, such as DivideByZeroException, from the standard exception classes provided by C++.
  - Doing so allows you to use the virtual function what to obtain an appropriate error message and also allows you to polymorphically process the exceptions by catching a reference to the base-class type.
- We use an object of this DivideByZeroException class in Fig. 17.2 to indicate when an attempt is made to divide by zero.

```
1  // Fig. 17.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header contains runtime_error
4
5  // DivideByZeroException objects should be thrown by functions
6  // upon detecting division-by-zero exceptions
7  class DivideByZeroException : public std::runtime_error {
8  public:
9     // constructor specifies default error message
10    DivideByZeroException()
11       : std::runtime_error{"attempted to divide by zero"} {}
12 };
```

Fig. 17.1 | Class `DivideByZeroException` definition.

17.2.2 Demonstrating Exception Handling

- Fig. 17.2 uses exception handling to wrap code that might throw a DivideByZeroException and to handle that exception, should one occur.
- Function quotient divides its first parameter (numerator) by its second parameter (denominator).
- Assuming that the user does not specify 0 as the denominator for the division, quotient returns the division result.
- However, if the user inputs 0 for the denominator, function quotient throws an exception.

```
1  // Fig. 17.2: fig17_02.cpp
2  // Example that throws exceptions on
3  // attempts to divide by zero.
4  #include <iostream>
5  #include "DivideByZeroException.h" // DivideByZeroException class
6  using namespace std;
7
8  // perform division and throw DivideByZeroException object if
9  // divide-by-zero exception occurs
10 double quotient(int numerator, int denominator) {
11    // throw DivideByZeroException if trying to divide by zero
12    if (denominator == 0) {
13       throw DivideByZeroException{}; // terminate function
14    }
15
16    // return division result
17    return static_cast<double>(numerator) / denominator;
18 }
19
```

Fig. 17.2 | Example that throws exceptions on attempts to divide by zero. (Part 1 of 3.)

```
20  int main() {
21     int number1; // user-specified numerator
22     int number2; // user-specified denominator
23
24     cout << "Enter two integers (end-of-file to end): ";
25
26     // enable user to enter two integers to divide
27     while (cin >> number1 >> number2) {
28        // try block contains code that might throw exception
29        // and code that will not execute if an exception occurs
30        try {
31           double result{quotient(number1, number2)};
32           cout << "The quotient is: " << result << endl;
33        }
34        catch (const DivideByZeroException& divideByZeroException) {
35           cout << "Exception occurred: "
36              << divideByZeroException.what() << endl;
37        }
38
39        cout << "\nEnter two integers (end-of-file to end): ";
40     }
41
42     cout << endl;
43  }
```

Fig. 17.2 | Example that throws exceptions on attempts to divide by zero. (Part 2 of 3.)

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

Fig. 17.2 | Example that throws exceptions on attempts to divide by zero. (Part 3 of 3.)

17.2.3 Enclosing Code in a try Block

- Exception handling is geared to situations in which the function that detects an error is unable to handle it.
- try blocks enable exception handling.
- The try block encloses statements that might cause exceptions and statements that should be skipped if an exception occurs.
- In this example, because the invocation of function quotient (line 31) can throw an exception, we enclose this function invocation in a try block.
- Enclosing the output statement (line 32) in the try block ensures that the output will occur only if function quotient returns a result.

17.2.4 Defining a catch Handler to Process a DivideByZeroException

- Exceptions are processed by catch handlers.
- At least one catch handler (lines 34–37) must immediately follow each try block.
- An exception parameter should always be declared as a reference to the type of exception the catch handler can process (DivideByZeroException in this case)—this prevents copying the exception object when it's caught and allows a catch handler to properly catch derived-class exceptions as well.
- When an exception occurs in a try block, the catch handler that executes is the first one whose type matches the type of the exception that occurred (i.e., the type in the catch block matches the thrown exception type exactly or is a direct or indirect base class of it).

17.2.4 Defining a catch Handler to Process a DivideByZeroException

- If an exception parameter includes an optional parameter name, the catch handler can use that parameter name to interact with the caught exception in the body of the catch handler, which is delimited by braces ({ and }).
- A catch handler typically reports the error to the user, logs it to a file, terminates the program gracefully or tries an alternate strategy to accomplish the failed task.
- In this example, the catch handler simply reports that the user attempted to divide by zero. Then the program prompts the user to enter two new integer values.

17.2.5 Termination Model of Exception Handling

- If an exception occurs as the result of a statement in a try block, the try block expires (i.e., terminates immediately).
- Next, the program searches for the first catch handler that can process the type of exception that occurred.
- The program locates the matching catch by comparing the thrown exception's type to each catch's exception-parameter type until the program finds a match.
- A match occurs if the types are identical or if the thrown exception's type is a derived class of the exception-parameter type.
- When a match occurs, the code in the matching catch handler executes.

17.2.5 Termination Model of Exception Handling

- When a catch handler finishes processing by reaching its closing right brace (}), the exception is considered handled, and the local variables defined within the catch handler (including the catch parameter) go out of scope.
- Program control does not return to the point at which the exception occurred (known as the throw point), because the try block has expired.
- Rather, control resumes with the first statement after the last catch handler following the try block.
- This is known as the termination model of exception handling.
- As with any other block of code, when a try block terminates, local variables defined in the block go out of scope.

17.2.5 Termination Model of Exception Handling

- If the try block completes its execution successfully (i.e., no exceptions occur in the try block), then the program ignores the catch handlers and program control continues with the first statement after the last catch following that try block.
- If an exception that occurs in a try block has no matching catch handler, or if an exception occurs in a statement that is not in a try block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing try block in the calling function.
- This process is called stack unwinding and is discussed in Section 17.4.

17.2.7 Flow of Program Control When the User Enters a Denominator of Zero)

- As part of throwing an exception, the throw operand is created and used to initialize the parameter in the catch handler, which we discuss momentarily.
- Central characteristic of exception handling: If your program explicitly throws an exception, it should do so before the error has an opportunity to occur.
- In general, when an exception is thrown within a try block, the exception is caught by a catch handler that specifies the type matching the thrown exception.

Software Engineering Observation 17.4

A central characteristic of exception handling is: If your program explicitly throws an exception, it should do so before the error has an opportunity to occur.

17.2.7 Flow of Program Control When the User Enters a Denominator of Zero)

- In this program, the catch handler specifies that it catches DivideByZeroException objects—this type matches the object type thrown in function quotient.
- Actually, the catch handler catches a reference to the DivideByZeroException object created by function quotient's throw statement.
- The exception object is maintained by the exception-handling mechanism.

Good Programming Practice 17.1

Associating each type of runtime error with an appropriately named exception type improves program clarity.

17.3 Rethrowing an Exception

- A function might use a resource—like a file—and might want to release the resource (i.e., close the file) if an exception occurs.
- An exception handler, upon receiving an exception, can release the resource then notify its caller that an exception occurred by rethrowing the exception via the statement
  - throw;
- Regardless of whether a handler can process an exception, the handler can rethrow the exception for further processing outside the handler.
- The next enclosing try block detects the rethrown exception, which a catch handler listed after that enclosing try block attempts to handle.

Common Programming Error 17.5

Executing an empty `throw` *statement outside a* `catch` *handler terminates the program immediately.*

17.3 Rethrowing an Exception (cont.)

- Fig. 17.3 demonstrates rethrowing an exception.
- Since we do not use the exception parameters in the catch handlers of this example, we omit the exception parameter names and specify only the type of exception to catch (lines 14 and 30).

```
1  // Fig. 17.3: fig17_03.cpp
2  // Rethrowing an exception.
3  #include <iostream>
4  #include <exception>
5  using namespace std;
6
7  // throw, catch and rethrow exception
8  void throwException() {
9     // throw exception and catch it immediately
10    try {
11       cout << "  Function throwException throws an exception\n";
12       throw exception{}; // generate exception
13    }
14    catch (const exception&) { // handle exception
15       cout << "  Exception handled in function throwException"
16          << "\n  Function throwException rethrows exception";
17       throw; // rethrow exception for further processing
18    }
19
20    cout << "This should not print\n";
21 }
```

Fig. 17.3 | Rethrowing an exception. (Part 1 of 2.)

```
22
23  int main() {
24     // throw exception
25     try {
26        cout << "\nmain invokes function throwException\n";
27        throwException();
28        cout << "This should not print\n";
29     }
30     catch (const exception&) { // handle exception
31        cout << "\n\nException handled in main\n";
32     }
33
34     cout << "Program control continues after catch in main\n";
35  }
```

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

Fig. 17.3 | Rethrowing an exception. (Part 2 of 2.)

17.4 Stack Unwinding

- When an exception is thrown but not caught in a particular scope, the function call stack is “unwound,” and an attempt is made to catch the exception in the next outer try…catch block.
- Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables that have completed intitialization in that function are destroyed and control returns to the statement that originally invoked that function.
- If a try block encloses that statement, an attempt is made to catch the exception.
- If a try block does not enclose that statement, stack unwinding occurs again.
- If no catch handler ever catches this exception, the program terminates.
- The program of Fig. 17.4 demonstrates stack unwinding.

```
1  // Fig. 17.4: fig17_04.cpp
2  // Demonstrating stack unwinding.
3  #include <iostream>
4  #include <stdexcept>
5  using namespace std;
6
7  // function3 throws runtime error
8  void function3() {
9     cout << "In function 3" << endl;
10
11    // no try block, stack unwinding occurs, return control to function2
12    throw runtime_error{"runtime_error in function3"}; // no print
13 }
14
15 // function2 invokes function3
16 void function2() {
17    cout << "function3 is called inside function2" << endl;
18    function3(); // stack unwinding occurs, return control to function1
19 }
```

Fig. 17.4 | Demonstrating stack unwinding. (Part 1 of 3.)

```
20
21  // function1 invokes function2
22  void function1() {
23     cout << "function2 is called inside function1" << endl;
24     function2(); // stack unwinding occurs, return control to main
25  }
26
27  // demonstrate stack unwinding
28  int main() {
29     // invoke function1
30     try {
31        cout << "function1 is called inside main" << endl;
32        function1(); // call function1 which throws runtime_error
33     }
34     catch (const runtime_error& error) { // handle runtime error
35        cout << "Exception occurred: " << error.what() << endl;
36        cout << "Exception handled in main" << endl;
37     }
38  }
```

Fig. 17.4 | Demonstrating stack unwinding. (Part 2 of 3.)

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

Fig. 17.4 | Demonstrating stack unwinding. (Part 3 of 3.)

17.4 Stack Unwinding (cont.)

- Line 12 of function3 throws a runtime_error object.
- However, because no try block encloses the throw statement in line 12, stack unwinding occurs—function3 terminates at line 12, then returns control to the statement in function2 that invoked function3 (i.e., line 18).
- Because no try block encloses line 18, stack unwinding occurs again—function2 terminates at line 18 and returns control to the statement in function1 that invoked function2 (i.e., line 24).
- Because no try block encloses line 24, stack unwinding occurs one more time—function1 terminates at line 24 and returns control to the statement in main that invoked function1 (i.e., line 32).
- The try block of lines 30–33 encloses this statement, so the first matching catch handler located after this try block (line 34–37) catches and processes the exception.

17.5 When to Use Exception Handling

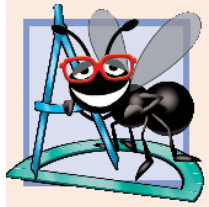
- Exception handling is designed to process synchronous errors, which occur when a statement executes, such as out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid function parameters and unsuccessful memory allocation (due to lack of memory).
- Exception handling is not designed to process errors associated with asynchronous events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

17.5 When to Use Exception Handling (cont.)

- Exception-handling also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes.
- Software elements often use exceptions to notify programs when problems occur.
- This enables you to implement customized error handling for each application.

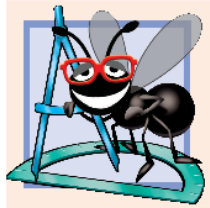
17.5 When to Use Exception Handling (cont.)

- Complex applications normally consist of predefined software components and application-specific components that use the predefined components.
- When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the predefined component cannot know in advance how each application processes a problem that occurs.



Software Engineering Observation 17.5

Exception handling provides a single, uniform technique for processing problems. This helps programmers on large projects understand each other's error-processing code. It also enables predefined software components (such as Standard Library classes) to communicate problems to application-specific components, which can then process the problems in an application-specific manner



Software Engineering Observation 17.6

Functions with common error conditions should return `nullptr`, `0` *or other appropriate values, such as* `bool`s, *rather than throw exceptions. A program calling such a function can check the return value to determine success or failure of the function call.*

17.6 noexcept: Declaring Functions That Do Not Throw Exceptions

- As of C++11, if a function does not throw any exceptions and does not call any functions that throw exceptions, you can explicitly state that a function does not throw exceptions.
- This indicates to client-code programmers that there's no need to call the function in a try block.
- Add noexcept to the right of the function's parameter list in both the prototype and the definition.
- For a const member function, place noexcept after const.
- If a function that's declared noexcept calls another function that throws an exception or executes a throw statement, the program terminates.

17.7 Constructors, Destructors and Exception Handling

- What happens when an error is detected in a constructor?
- For example, how should an object's constructor respond when it receives invalid data?
- Because the constructor cannot return a value to indicate an error, we must choose an alternative means of indicating that the object has not been constructed properly.
- One scheme is to return the improperly constructed object and hope that anyone using it would make appropriate tests to determine that it's in an inconsistent state.
- Another scheme is to set some variable outside the constructor.
- The preferred alternative is to require the constructor to throw an exception that contains the error information, thus offering an opportunity for the program to handle the failure.

17.7.1 Destructors Called Due to Exceptions

- If an exception occurs during object construction, destructors may be called:
  - If an exception is thrown before an object is fully constructed, destructors will be called for any member objects that have been constructed so far.
  - If an array of objects has been partially constructed when an exception occurs, only the destructors for the array's constructed objects will be called.
- Destructors are called for every automatic object constructed in a try block before an exception that occurred in that block is caught.
- Stack unwinding is guaranteed to have been completed at the point that an exception handler begins executing.
- If a destructor invoked as a result of stack unwinding throws an exception, the program terminates.
- This has been linked to various security attacks.

17.9 Class unique_ptr and Dynamic Memory Allocation

- A common programming practice is to allocate dynamic memory, assign the address of that memory to a pointer, use the pointer to manipulate the memory and deallocate the memory with delete when the memory is no longer needed.
- If an exception occurs after successful memory allocation but before the delete statement executes, a memory leak could occur.
- C++ 11 provides class template unique_ptr in header file <memory> to deal with this situation.

17.9 Class unique_ptr and Dynamic Memory Allocation (cont.)

- A unique_ptr maintains a pointer to dynamically allocated memory.
- When a unique_ptr object's destructor is called (for example, when an unique_ptr object goes out of scope), it performs a delete operation on its pointer data member.
- Class template unique_ptr provides overloaded operators * and -> so that an unique_ptr object can be used just as a regular pointer variable is.
- Figure 17.9 demonstrates an unique_ptr object that points to a dynamically allocated object of class Integer (Figs. 17.7–17.8).

```
1  // Fig. 17.7: Integer.h
2  // Integer class definition.
3
4  class Integer {
5  public:
6     Integer(int i = 0); // Integer default constructor
7     ~Integer(); // Integer destructor
8     void setInteger(int i); // set Integer value
9     int getInteger() const; // return Integer value
10 private:
11    int value;
12 };
```

Fig. 17.7 | `Integer` class definition.

```
1  // Fig. 17.8: Integer.cpp
2  // Integer member function definitions.
3  #include <iostream>
4  #include "Integer.h"
5  using namespace std;
6
7  // Integer default constructor
8  Integer::Integer(int i)
9     : value{i} {
10    cout << "Constructor for Integer " << value << endl;
11 }
12
13 // Integer destructor
14 Integer::~Integer() {
15    cout << "Destructor for Integer " << value << endl;
16 }
17
```

Fig. 17.8 | `Integer` member function definitions. (Part 1 of 2.)

```
18   // set Integer value
19   void Integer::setInteger(int i) {
20      value = i;
21   }
22
23   // return Integer value
24   int Integer::getInteger() const {
25      return value;
26   }
```

Fig. 17.8 | `Integer` member function definitions. (Part 2 of 2.)

```
1  // Fig. 17.9: fig17_09.cpp
2  // Demonstrating unique_ptr.
3  #include <iostream>
4  #include <memory>
5  using namespace std;
6
7  #include "Integer.h"
8
9  // use unique_ptr to manipulate Integer object
10 int main() {
11    cout << "Creating a unique_ptr object that points to an Integer\n";
12
13    // "aim" unique_ptr at Integer object
14    unique_ptr<Integer> ptrToInteger{make_unique<Integer>(7)};
15
16    cout << "\nUsing the unique_ptr to set the Integer\n";
17    ptrToInteger->setInteger(99); // use unique_ptr to set Integer value
18
19    // use unique_ptr to get Integer value
20    cout << "Integer after setInteger: " << (*ptrToInteger).getInteger()
21       << "\n\nTerminating program" << endl;
22 }
```

Fig. 17.9 | `unique_ptr` object manages dynamically allocated memory. (Part 1 of 2.)

```
Creating a unique_ptr object that points to an Integer
Constructor for Integer 7

Using the unique_ptr to set the Integer
Integer after setInteger: 99

Terminating program
Destructor for Integer 99
```

Fig. 17.9 | `unique_ptr` object manages dynamically allocated memory. (Part 2 of 2.)

17.9 Class unique_ptr and Dynamic Memory Allocation (cont.)

- Because ptrToInteger is a local automatic variable in main, ptrToInteger is destroyed when main terminates.
- The unique_ptr destructor forces a delete of the Integer object pointed to by ptrToInteger, which in turn calls the Integer class destructor.
- The memory that Integer occupies is released, regardless of how control leaves the block (e.g., by a return statement or by an exception).
- Most importantly, using this technique can prevent memory leaks.

17.9.1 unique_ptr Ownership

- The class is called unique_ptr because only one unique_ptr at a time can own a dynamically allocated object.
- When you assign one unique_ptr to another, the unique_ptr on the assignment's right transfers ownership of the dynamic memory it manages to the unique_ptr on the assignment's left.
- The same is true when one unique_ptr is passed as an argument to another unique_ptr's constructor.
- The last unique_ptr object that maintains the pointer to the dynamic memory will delete the memory.
- This makes unique_ptr an ideal mechanism for returning dynamically allocated memory to client code.
- When the unique_ptr goes out of scope in the client code, the unique_ptr's destructor deletes the dynamically allocated object—if the object has a destructor, it is called before the memory is returned to the system.

17.9.2 unique_ptr to a Built-In Array

- You can also use a unique_ptr to manage a dynamically allocated built-in array.
- For example, consider the statement
  - unique_ptr<string[]> ptr{make_unique<string[]>(10)};
- Because make_unique's type is specified as string[], the function obtains a dynamically allocated built-in array of the number of elements specified by its argument (10).
- By default, the elements of arrays allocated with make_unique are initialized to 0 for fundamental types, to false for bools or via the default constructor for objects of a class—so in this case, the array would contain 10 string objects initialized with the empty string.

17.9.2 unique_ptr to a Built-In Array

- A unique_ptr that manages an array provides an overloaded [] operator for accessing the array's elements.
- For example, the statement
  - ptr[2] = "hello";
- assigns "hello" to the string at ptr[2] and the statement
  - cout << ptr[2] << endl;
- displays that string.

17.10 Standard Library Exception Hierarchy

- Experience has shown that exceptions fall nicely into a number of categories.
- The C++ Standard Library includes a hierarchy of exception classes, some of which are shown in Fig. 17.10.
- As we first discussed in Section 17.2, this hierarchy is headed by base-class exception (defined in header file <exception>), which contains virtual function what that derived classes can override to issue appropriate error messages.
- If a catch handler catches a reference to an exception of a base-class type, it also can catch a reference to all objects of classes derived publicly from that base class—this allows for polymorphic processing of related errors

17.10 Standard Library Exception Hierarchy (cont.)

- Immediate derived classes of base-class exception include runtime_error and logic_error (both defined in header <stdexcept>), each of which has several derived classes.
- Also derived from exception are the exceptions thrown by C++ operators—for example, bad_alloc is thrown by new (Section 17.8), bad_cast is thrown by dynamic_cast (Chapter 12) and bad_typeid is thrown by typeid (Chapter 12).

17.10 Standard Library Exception Hierarchy (cont.)

- Class logic_error is the base class of several standard exception classes that indicate errors in program logic.
  - For example, class invalid_argument indicates that a function received an invalid argument.
  - Proper coding can, of course, prevent invalid arguments from reaching a function.
- Class length_error indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object.
- Class out_of_range indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

17.10 Standard Library Exception Hierarchy (cont.)

- Class runtime_error, which we used briefly in Section 17.4, is the base class of several other standard exception classes that indicate execution-time errors.
  - For example, class overflow_error describes an arithmetic overflow error (i.e., the result of an arithmetic operation is larger than the largest number that can be stored in the computer) and class underflow_error describes an arithmetic underflow error (i.e., the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer).

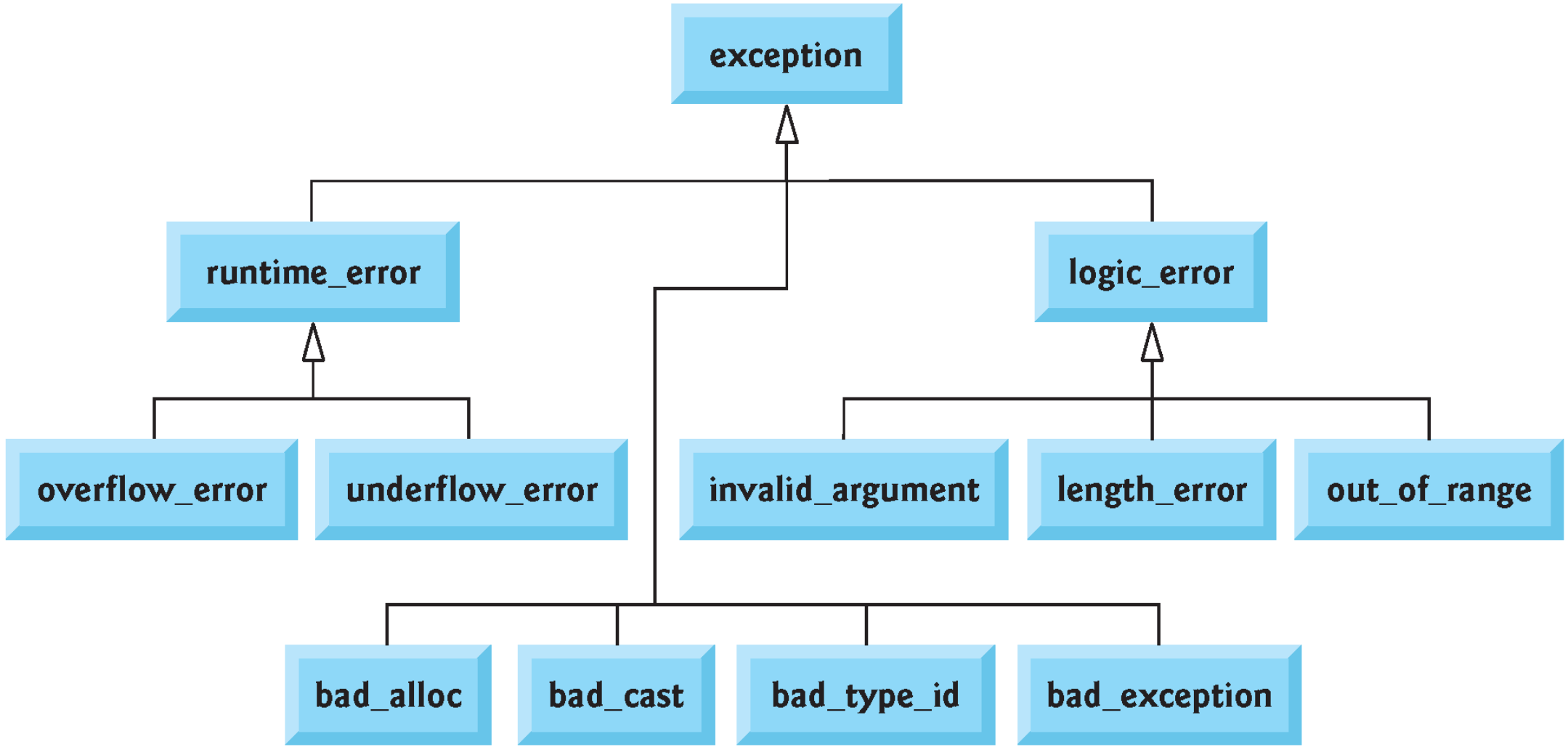


Fig. 17.10 | Some of the Standard Library exception classes.