18

Exception Handling

OBJECTIVES

In this chapter you will learn:

- What exceptions are and when to use them.
- To use `try`, `catch` and `throw` to detect, handle and indicate exceptions, respectively.
- To process uncaught and unexpected exceptions.
- To declare new exception classes.
- How stack unwinding enables exceptions not caught in one scope to be caught in another scope.
- To handle `new` failures.
- To use `auto_ptr` to prevent memory leaks.
- To understand the standard exception hierarchy.

Outline

18.1 Introduction

18.2 Exception-Handling Overview

18.3 Example: Handling an Attempt to Divide by Zero

18.4 When to Use Exception Handling

18.5 Rethrowing an Exception

18.6 Exception Specifications

18.7 Processing Unexpected Exceptions

18.8 Stack Unwinding

18.9 Constructors, Destructors and Exception Handling

18.10 Exceptions and Inheritance

18.11 Processing `new` Failures

18.12 Class `auto_ptr` and Dynamic Memory Allocation

18.13 Standard Library Exception Hierarchy

18.14 Other Error-Handling Techniques

18.15 Wrap-Up

18.1 Introduction

- Exceptions
  - Indicate problems that occur during a program’s execution
  - Occur infrequently
- Exception handling
  - Can resolve exceptions
    - Allow a program to continue executing or
    - Notify the user of the problem and
    - Terminate the program in a controlled manner
  - Makes programs robust and fault-tolerant

◀ ▶

Error-Prevention Tip 18.1

Exception handling helps improve a program’s fault tolerance.

Software Engineering Observation 18.1

Exception handling provides a standard mechanism for processing errors. This is especially important when working on a project with a large team of programmers.

18.2 Exception-Handling Overview

- **Intermixing program and error-handling logic**
  - **Pseudocode example**

```
Perform a task
If the preceding task did not execute correctly
    Perform error processing
Perform next task
If the preceding task did not execute correctly
    Perform error processing
…
```

Open file

check if file is open

allocate memory

check if the allocate is succ

open a network conn

Connected?

  - **Makes the program difficult to read, modify, maintain and debug**

```
try{
    open file
    alloc mem
    com. net
}
catch(exception ){
    file error
    mem err
    net err.
```

18.2 Exception-Handling Overview (Cont.)

- Exception handling
  - Removes error-handling code from the program execution’s “main line”
  - Programmers can handle any exceptions they choose
    - All exceptions,
    - All exceptions of a certain type or
    - All exceptions of a group of related types

18.3 Example: Handling an Attempt to Divide by Zero

- Class `exception`
  - Is the standard C++ base class for all exceptions
  - Provides its derived classes with `virtual` function `what`
    - Returns the exception's stored error message

Outline

```cpp
// Fig. 16.1: DivideByZeroException.h
// Class DivideByZeroException definition.
#include <stdexcept> // stdexcept header file contains runtime_error
using std::runtime_error; // standard C++ library class runtime_error

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public runtime_error
{
public:
   // constructor specifies default error message
   DivideByZeroException::DivideByZeroException()
      : runtime_error( "attempted to divide by zero" ) {}
}; // end class DivideByZeroException
```

DivideByZeroException.h

(1 of 1)

```
// Fig. 16.2: Fig16_02.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include "DivideByZeroException.h" // DivideByZeroException class

// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs
double quotient( int numerator, int denominator )
{
   // throw DivideByZeroException if trying to divide by zero
   if ( denominator == 0 )
      throw DivideByZeroException(); // terminate function

   // return division result
   return static_cast< double >( numerator ) / denominator;
} // end function quotient

int main()
{
   int number1; // user-specified numerator
   int number2; // user-specified denominator
   double result; // result of division

   cout << "Enter two integers (end-of-file to end): ";
```

throw 10;

throw Mang(10)

Fig18_02.cpp

(1 of 3)

Outline

```

   // enable user to enter two integers to divide
   while ( cin >> number1 >> number2 )
   {
      // try block contains code that might throw exception
      // and code that should not execute if an exception occurs
      try
      {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      } // end try

      // exception handler handles a divide-by-zero exception
      catch ( DivideByZeroException &divideByZeroException )
      {
         cout << "Exception occurred: "
            << divideByZeroException.what() << endl;
      } // end catch

      cout << "\nEnter two integers (end-of-file to end): ";
   } // end while

   cout << endl;
   return 0; // terminate normally
} // end main
```

throws on exception

Fig18_02.cpp

(2 of 3)

Outline

```
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
```

Fig18_02.cpp

(3 of 3)

18.3 Example: Handling an Attempt to Divide by Zero (Cont.)

- `try` Blocks
  - Keyword `try` followed by braces (`{}`)
  - Should enclose
    - Statements that might cause exceptions and
    - Statements that should be skipped in case of an exception

11:37

```
try{
  1
  2
  3
  4
  5
}
catch(Param){

}
```

Software Engineering Observation 18.2

Exceptions may surface through explicitly mentioned code in a `try` block, through calls to other functions and through deeply nested function calls initiated by code in a `try` block.

18.3 Example: Handling an Attempt to Divide by Zero (Cont.)

- `catch` handlers
  - Immediately follow a `try` block
    - One or more `catch` handlers for each `try` block
  - Keyword `catch`
  - Exception parameter enclosed in parentheses
    - Represents the type of exception to process
    - Can provide an optional parameter name to interact with the caught exception object
  - Executes if exception parameter type matches the exception thrown in the `try` block
    - Could be a base class of the thrown exception's class

```
while(true){
  try{
    getvals();
    use vals();
  }
  catch(Exception&){
    ~~cout << "Enter valid";~~
    cerr << "err";
    exit(-1);
  }
}
```

```
if
 :
else
 :
```

Common Programming Error 18.1

It is a syntax error to place code between a `try` block and its corresponding `catch` handlers.

Common Programming Error 18.2

Each `catch` handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.

catch(exception& e, ~~int b~~) {

Common Programming Error 18.3

It is a logic error to catch the same type in two different `catch` handlers following a single `try` block.

Common Programming Error 18.4

Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.

Error-Prevention Tip 18.2

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what is called mission-critical computing or business-critical computing.

18.3 Example: Handling an Attempt to Divide by Zero (Cont.)

- Throwing an exception
  - Use keyword `throw` followed by an operand representing the type of exception
    - The `throw` operand can be of any type
      - If the `throw` operand is an object, it is called an exception object
  - The `throw` operand initializes the exception parameter in the matching `catch` handler, if one is found

Good Programming Practice 18.1

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

Software Engineering Observation 18.3

Incorporate your exception-handling strategy into your system from the design process's inception. Including effective exception handling after a system has been implemented can be difficult.

Software Engineering Observation 18.4

Exception handling provides a single, uniform technique for processing problems. This helps programmers working on large projects understand each other's error-processing code.

Software Engineering Observation 18.5

Avoid using exception handling as an alternate form of flow of control. These “additional” exceptions can “get in the way” of genuine error-type exceptions.

Software Engineering Observation 18.6

Exception handling simplifies combining software components and enables them to work together effectively by enabling predefined components to communicate problems to application-specific components, which can then process the problems in an application-specific manner.

Performance Tip 18.3

When no exceptions occur, exception-handling code incurs little or no performance penalties. Thus, programs that implement exception handling operate more efficiently than do programs that intermix error-handling code with program logic.

18.5 Rethrowing an Exception

- **Rethrowing an exception**
  - **Empty `throw`; statement**
  - **Use when a `catch` handler cannot or can only partially process an exception**
  - **Next enclosing `try` block attempts to match the exception with one of its `catch` handlers**

Common Programming Error 18.6

Executing an empty `throw` statement that is situated outside a `catch` handler causes a call to function terminate, which abandons exception processing and terminates the program immediately.

```cpp
// Fig. 16.3: Fig16_03.cpp
// Demonstrating exception rethrowing.
#include <iostream>
using std::cout;
using std::endl;

#include <exception>
using std::exception;

// throw, catch and rethrow exception
void throwException()
{
   // throw exception and catch it immediately
   try
   {
      cout << "  Function throwException throws an exception\n";
      throw exception(); // generate exception
   } // end try
   catch ( exception & ) // handle exception
   {
      cout << "  Exception handled in function throwException"
         << "\n  Function throwException rethrows exception";
      throw; // rethrow exception for further processing
   } // end catch

   cout << "This also should not print\n";
} // end function throwException
```

Rethrow the exception

Fig18_03.cpp

(1 of 2)

Outline

Fig18_03.cpp

(2 of 2)

```cpp

int main()
{
   // throw exception
   try
   {
      cout << "\nmain invokes function throwException\n";
      throwException();
      cout << "This should not print\n";
   } // end try
   catch ( exception & ) // handle exception
   {
      cout << "\n\nException handled in main\n";
   } // end catch

   cout << "Program control continues after catch in main\n";
   return 0;
} // end main
```

Catch rethrown exception

```
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
```

18.6 Exception Specifications

- Exception specifications (a.k.a. `throw` lists)
  - Keyword `throw`
  - Comma-separated list of exception classes in parentheses
  - Example

```
• int someFunction( double value )
     throw ( ExceptionA, ExceptionB,
             ExceptionC )
  {
     …
  }
```

C++98

deprecated with C++11

  - Indicates `someFunction` can throw exceptions of types `ExceptionA`, `ExceptionB` and `ExceptionC`

Common Programming Error 18.7

Throwing an exception that has not been declared in a function’s exception specification causes a call to function `unexpected`.

unhandled exception

```
int main(){
  quotient(1,0);
  return 0;
}
```

Error-Prevention Tip 18.3

The compiler will not generate a compilation error if a function contains a `throw` expression for an exception not listed in the function's exception specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, carefully check your code to ensure that functions do not throw exceptions not listed in their exception specifications.

18.7 Processing Unexpected Exceptions

- Function `unexpected`
  - Called when a function `throw`s an exception not in its exception specification
  - Calls the function registered with function `set_unexpected`
  - Function `terminate` is called by default
- Function `set_unexpected` of `<exception>`
  - Takes as argument a pointer to a function with no arguments and a `void` return type
  - Returns a pointer to the last function called by `unexpected`
    - Returns `0` the first time

18.7 Processing Unexpected Exceptions (Cont.)

- Function `terminate`
  - Called when
    - No matching `catch` is found for a thrown exception
    - A destructor attempts to `throw` an exception during stack unwinding
    - Attempting to rethrow an exception when no exception is being handled
    - Calling function `unexpected` before registering a function with function `set_unexpected`
  - Calls the function registered with function `set_terminate`
  - Function `abort` is called by default

18.7 Processing Unexpected Exceptions (Cont.)

- Function `set_terminate`
  - Takes as argument a pointer to a function with no arguments and a `void` return type
  - Returns a pointer to the last function called by `terminate`
    - Returns `0` the first time
- Function `abort`
  - Terminates the program without calling destructors for automatic or static storage class objects
    - Could lead to resource leaks

18.8 Stack Unwinding

- Stack unwinding
  - Occurs when a thrown exception is not caught in a particular scope
  - Unwinding a function terminates that function
    - All local variables of the function are destroyed
    - Control returns to the statement that invoked the function
  - Attempts are made to `catch` the exception in outer `try`…`catch` blocks
  - If the exception is never caught, function `terminate` is called

Outline

Fig18_04.cpp

(1 of 3)

```cpp
// Fig. 16.4: Fig16_04.cpp
// Demonstrating stack unwinding.
#include <iostream>
using std::cout;
using std::endl;

#include <stdexcept>
using std::runtime_error;

// function3 throws run-time error
void function3() throw ( runtime_error )
{      Money   m1(10);
   cout << "In function 3" << endl;

   // no try block, stack unwinding occur, return control to function2
   throw runtime_error( "runtime_error in function3" );
} // end function3

// function2 invokes function3
void function2() throw ( runtime_error )
{      PAccngLists   a(10);
   cout << "function3 is called inside function2" << endl;
   function3(); // stack unwinding occur, return control to function1
} // end function2
```

int = 7;

Outline

```

// function1 invokes function2
void function1() throw ( runtime_error )
{
   cout << "function2 is called inside function1" << endl;
   function2(); // stack unwinding occur, return control to main
} // end function1

// demonstrate stack unwinding
int main()
{
   // invoke function1
   try
   {
      cout << "function1 is called inside main" << endl;
      function1(); // call function1 which throws runtime_error
   } // end try
   catch ( runtime_error &error ) // handle run-time error
   {
      cout << "Exception occurred: " << error.what() << endl;
      cout << "Exception handled in main" << endl;
   } // end catch

   return 0;
} // end main
```

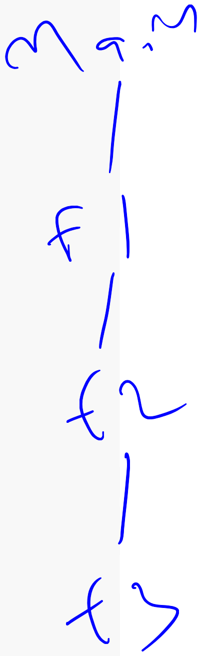


Fig18_04.cpp

(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
```

Fig18_04.cpp

(3 of 3)

18.9 Constructors, Destructors and Exception Handling

bitset<5> b("127ab");

- **Exceptions and constructors**
  - **Exceptions enable constructors, which cannot return values, to report errors to the program**
  - **Exceptions thrown by constructors cause any already-constructed component objects to call their destructors**
    - **Only those objects that have already been constructed will be destructed**
- **Exceptions and destructors**
  - **Destructors are called for all automatic objects in the terminated `try` block when an exception is thrown**
    - **Acquired resources can be placed in local objects to automatically release the resources when an exception occurs**
  - **If a destructor invoked by stack unwinding throws an exception, function `terminate` is called**

◀ ▶

Error-Prevention Tip 18.4

When an exception is thrown from the constructor for an object that is created in a `new` expression, the dynamically allocated memory for that object is released.

18.10 Exceptions and Inheritance

- Inheritance with exception classes
  - New exception classes can be defined to inherit from existing exception classes
  - A catch handler for a particular exception class can also catch exceptions of classes derived from that class

18.11 Processing new Failures

- `new` failures
  - Some compilers `throw` a `bad_alloc` exception
    - Compliant to the C++ standard specification
  - Some compilers return `0`
    - C++ standard-compliant compilers also have a version of `new` that returns `0`
      - Use expression `new( nothrow )`, where `nothrow` is of type `nothrow_t`
  - Some compilers `throw` `bad_alloc` if `<new>` is included

```
// Fig. 16.5: Fig16_05.cpp
// Demonstrating pre-standard new returning 0 when memory
// is not allocated.
#include <iostream>
using std::cerr;
using std::cout;

int main()
{
   double *ptr[ 50 ];

   // allocate memory for ptr
   for ( int i = 0; i < 50; i++ )
   {
      ptr[ i ] = new double[ 50000000 ];

      if ( ptr[ i ] == 0 ) // did new fail to allocate memory
      {
         cerr << "Memory allocation failed for ptr[ " << i << " ]\n";
         break;
      } // end if
      else // successful memory allocation
         cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
   } // end for

   return 0;
} // end main
```

Fig18_05.cpp

(1 of 2)

Allocate **50000000 double** values

new will have returned **0** if the memory allocation operation failed

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Memory allocation failed for ptr[ 3 ]
```

Outline

Fig18_03.cpp

(2 of 2)

Fig18_06.cpp

(1 of 2)

```
// Fig. 16.6: Fig16_06.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;

#include <new> // standard operator new
using std::bad_alloc;

int main()
{
   double *ptr[ 50 ];

   // allocate memory for ptr
   try
   {
      // allocate memory for ptr[ i ]; new throws bad_alloc on failure
      for ( int i = 0; i < 50; i++ )
      {
         ptr[ i ] = new double[ 50000000 ]; // may throw exception
         cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
      } // end for
   } // end try
```

Allocate `50000000` `double` values

Outline

```
   // handle bad_alloc exception
   catch ( bad_alloc &memoryAllocationException )
   {
      cerr << "Exception occurred: "
         << memoryAllocationException.what() << endl;
   } // end catch

   return 0;
} // end main
```

new throws a **bad_alloc** exception if the memory allocation operation failed

Fig18_06.cpp

(2 of 2)

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Exception occurred: bad allocation
```

18.13 Standard Library Exception Hierarchy

Late binding

ptr -> f()

ref . f()

- **Exception hierarchy classes**
  - **Base-class `exception`**
    - **Contains `virtual` function `what` for storing error messages**
    - **Exception classes derived from `exception`**

      Pet & pr = Dog()

      Dog& dr = dynamic_cast<Dog&> pr;

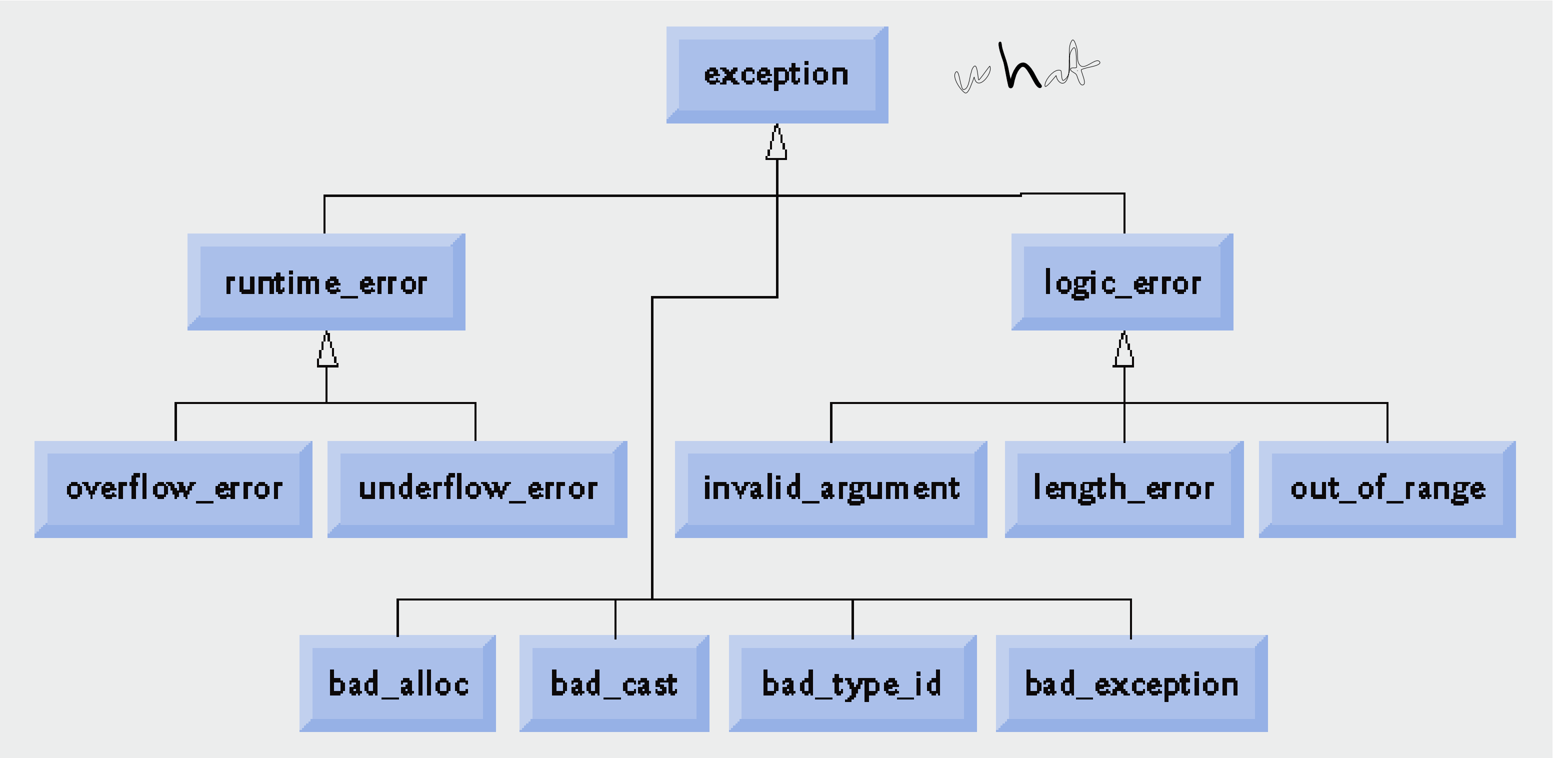
      - **`bad_alloc` – thrown by `new`**
      - **`bad_cast` – thrown by `dynamic_cast`**
      - **`bad_typeid` – thrown by `typeid`**
      - **`bad_exception` – thrown by `unexpected`**
        - **Instead of terminating the program or calling the function specified by `set_unexpected`**
        - **Used only if `bad_exception` is in the function’s `throw` list**

Common Programming Error 18.8

Placing a `catch` handler that catches a base-class object before a `catch` that catches an object of a class derived from that base class is a logic error. The base-class `catch` catches all objects of classes derived from that base class, so the derived-class `catch` will never execute.

```
void f1( ){
    throws bad_cast();
}
```

```
try{
    f1();
}
catch(exception& bc){
    ...
}
```



what

Fig. 18.11 | Standard Library exception classes.

18.13 Standard Library Exception Hierarchy (Cont.)

- Exception hierarchy classes (Cont.)
  - Class `logic_error`, derived from `exception`
    - Indicates errors in program logic
    - Exception classes derived from `logic_error`
      - `invalid_argument`
        - Indicates an invalid argument to a function
      - `length_error`
        - Indicates a length larger than the maximum size for some object was used
      - `out_of_range`
        - Indicates a value, such as an array subscript, exceeded its allowed range

18.13 Standard Library Exception Hierarchy (Cont.)

- Exception hierarchy classes (Cont.)
  - Class `runtime_error`, derived from `exception`
    - Indicates execution-time errors
    - Exception classes derived from `runtime_error`
      - `overflow_error`
        - Indicates an arithmetic overflow error – an arithmetic result is larger than the largest storable number
      - `underflow_error`
        - Indicates an arithmetic underflow error – an arithmetic result is smaller than the smallest storable number

Common Programming Error 18.9

Programmer-defined exception classes need not be derived from class `exception`. Thus, writing `catch( exception anyException )` is not guaranteed to `catch` all exceptions a program could encounter.

Error-Prevention Tip 18.6

To `catch` all exceptions potentially thrown in a `try` block, use `catch(...)`. One weakness with catching exceptions in this way is that the type of the caught exception is unknown at compile time. Another weakness is that, without a named parameter, there is no way to refer to the exception object inside the exception handler.

```
try{
  :
}catch(mem-err){
  ...
}catch(bad_type){
  :
}catch(...){
  cerr<<"send email to developer";
}
```

Software Engineering Observation 18.10

The standard `exception` hierarchy is a good starting point for creating exceptions. Programmers can build programs that can `throw` standard exceptions, `throw` exceptions derived from the standard exceptions or `throw` their own exceptions not derived from the standard exceptions.

Software Engineering Observation 18.11

Use `catch(...)` to perform recovery that does not depend on the exception type (e.g., releasing common resources). The exception can be rethrown to alert more specific enclosing `catch` handlers.

```
void f();
void f() throw();
```