Object-Oriented Programming
(in C++)

Exception Handling

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

Robustness

- An important design goal of software engineering
- Capable of handling the unexpected
- Capable of providing the correct response, even when the input is incorrect
- Advantages
  - Our programs are more fault-tolerant
  - They won’t crash when there’s a problem
  - They are safer

Exceptions

- Unexpected problems that occur when the program is running
  - Occur infrequently
  - Affect the operation of the program
- Examples
  - Trying to access an array outside of its bounds
  - Trying to delete an element from an empty list
  - Trying to divide by zero
  - Unable to allocate memory needed by the program

Straightforward Error Handling

```
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
```

- However, if the potential problems occur infrequently, intermixing program logic with error-handling logic can make the program difficult to read, modify, maintain, and debug
  - And can degrade a program’s performance

Exception Handling

- Provides a standard mechanism for processing errors
- Enables you to remove error-handling code from the “main line” of the program’s execution
- You can decide to handle any exceptions you choose

- With programming languages that do not support exception handling, programmers often delay writing error-processing code or sometimes forget to include it

Syntax of Exception Handling

```
try {
   Performing a task
   If the task did not execute correctly
        throw an_exception;
} catch (type_of_the_exception_to_catch) {
   Performing exception handling
} catch (another_type_of_the_exception_to_catch) {
   Performing exception handling
}
...
```

An expression

Throwing 5 does not imply anything. So associating each type of runtime error with an appropriately named exception object improves program clarity

- The operand of a throw can be of any type
  - E.g., throw 5
- If the operand of a throw is an object, we call it an exception object

C++ Exception Objects

- When there’s a potential error, an exception object is instantiated and used in the exception-handling procedures

- C++ exception objects
  - The program is written so that the code “throws” an exception in response to an unexpected event
  - Then the exception is “caught” and an appropriate action can occur

The catch Handler (Function)

- At least one catch handler (function) must immediately follow each try block
  - catch is the name of all exception handlers
  - Overloading is allowed
    - So the formal parameter of each catch function must be unique
- Each catch handler can have only a single formal parameter
  - The formal parameter need not have a variable
- The formal parameter can be an ellipsis (...), in which case it handles all exceptions not yet handled

Execution Flow of Exception Handling (1/2)

- No exceptions occur in a `try` block
  - No `catch` handlers will be called
- An exception occurs in a `try` block
  - The function that contains the statement terminates immediately
  - Only the first matching `catch` handler is executed
    - Handlers for specific exceptions are placed at the top of the list
  - When the `catch` handler finishes processing, program control resumes with the first statement after the last `catch` handler (i.e., after the `try-catch` clause)

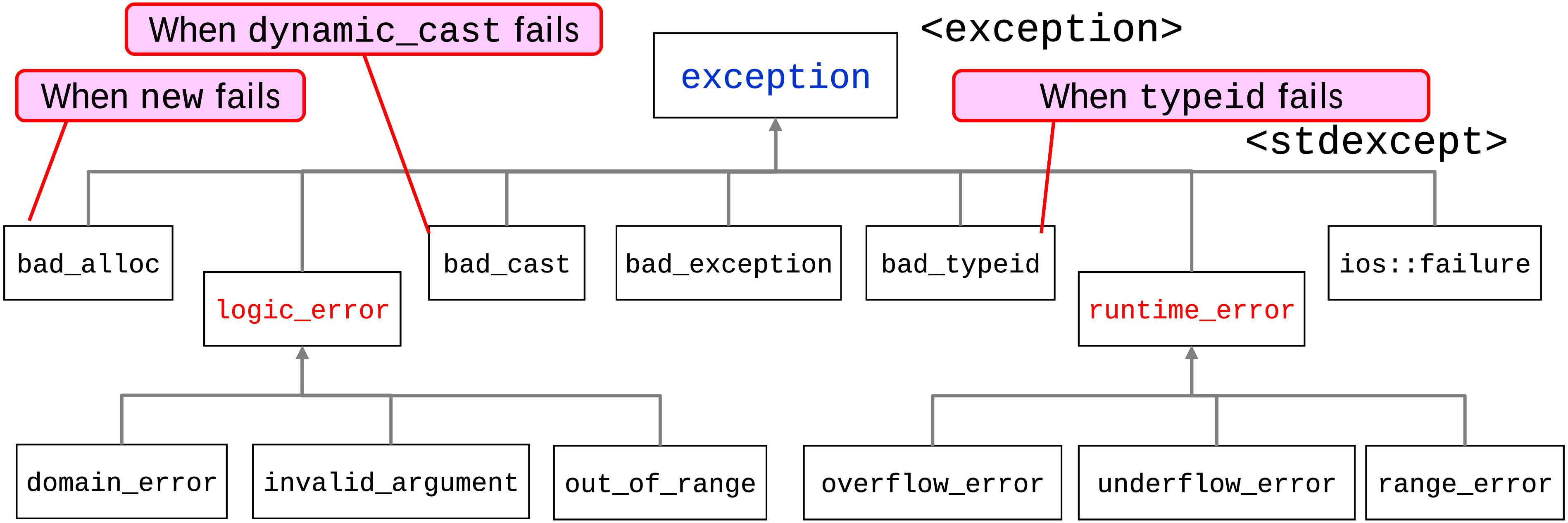
Execution Flow of Exception Handling (2/2)

- What about an unhandled exception?
  - There is no matching `catch` handler in the `try` clause

  or

  - The exception occurs outside a `try` block
- If this is the case, the exception is propagated to an enclosing `try` block
  - If no handler is found in the outer `try` block, the exception is propagated to the caller of the function in which it is raised
    - The process is called stack unwinding
    - This propagation continues to the `main` function
  - If no handler is found (all the way to main), the default handler (function `terminate`), which terminates the program by default, is called

Exception Hierarchy in C++ Standard Library



- Class `logic_error`
  - Defines the type of objects presumably detectable before the program executes
- Class `runtime_error`
  - Defines the type of objects presumably detectable only when the program executes

Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

Example: Divide-by-Zero Problem

- Division by zero typically causes a program to terminate prematurely

```
int Quotient (int numerator, int denominator}
{
    if (denominator!= 0)
        return (numerator/denominator);
    else
        // What to do?
}
```

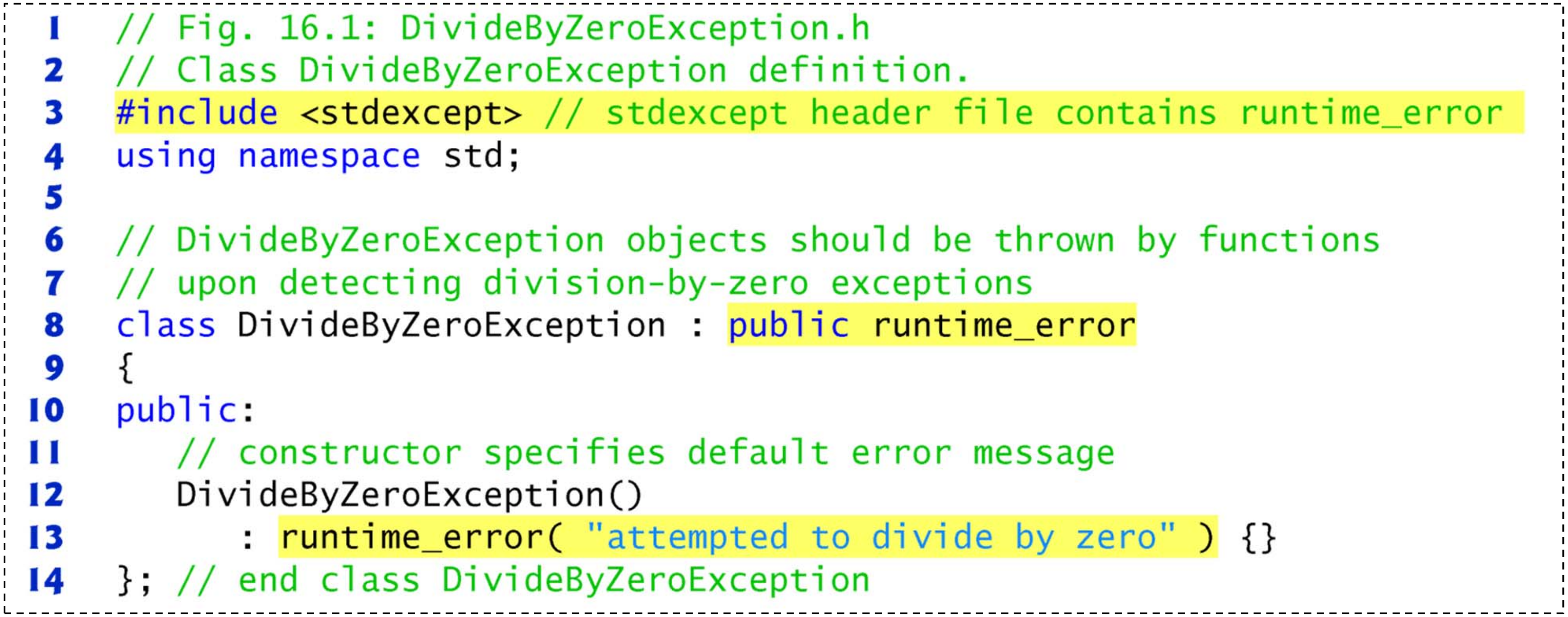
How to Handle This Problem?

Options:

- Print an error message and halt the program
- Rewrite the function with a third parameter (`bool`) indicating success or failure
- Allow the function to have a precondition: Test for `denominator==0` before function is called
- Use C++ exception-handling mechanism

Example: Defining a Divide-by-Zero Exception

- We’d like to use exception handling to prevent the common arithmetic problem

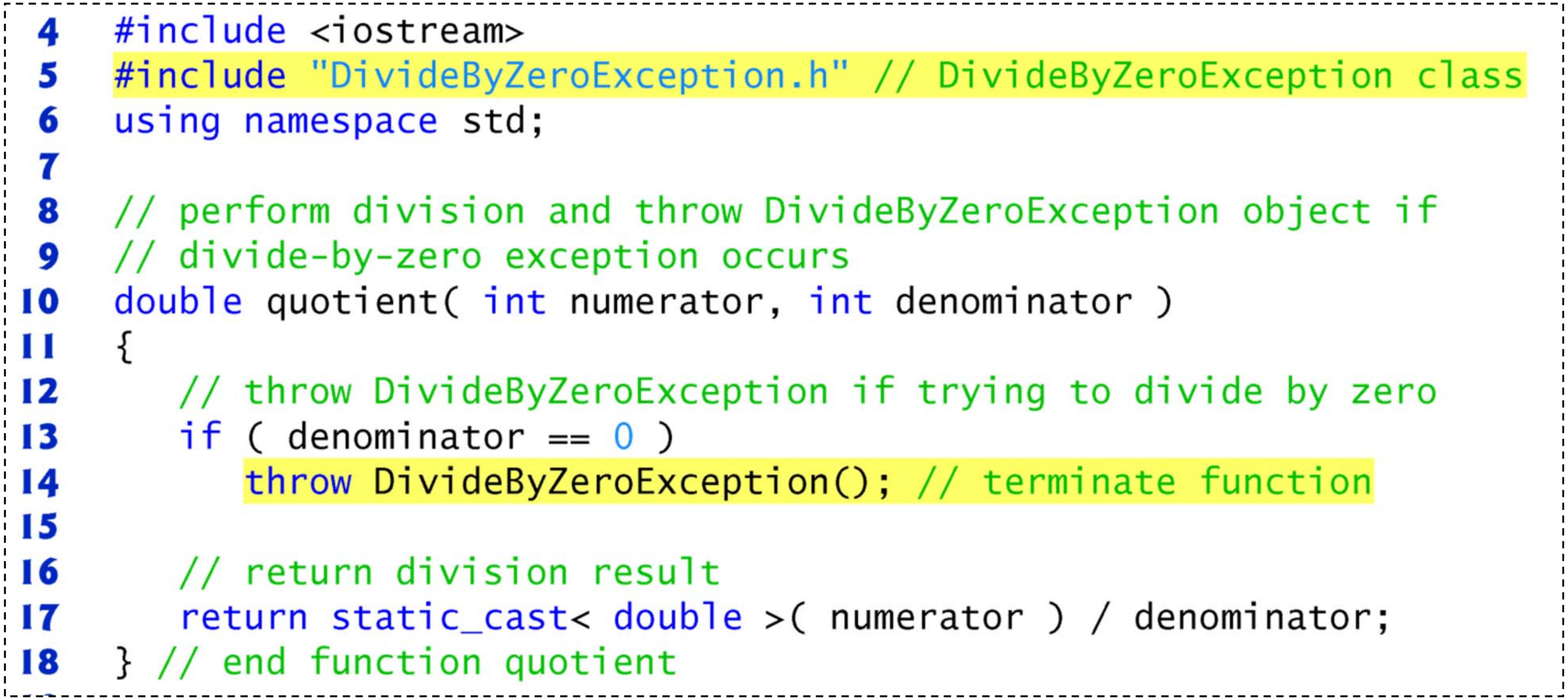


```
// Fig. 16.1: DivideByZeroException.h
// Class DivideByZeroException definition.
#include <stdexcept> // stdexcept header file contains runtime_error
using namespace std;

// 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()
      : runtime_error( "attempted to divide by zero" ) {}
}; // end class DivideByZeroException
```

Every exception class deriving from `exception` contains the `virtual` function `what`, which returns an exception object’s error message

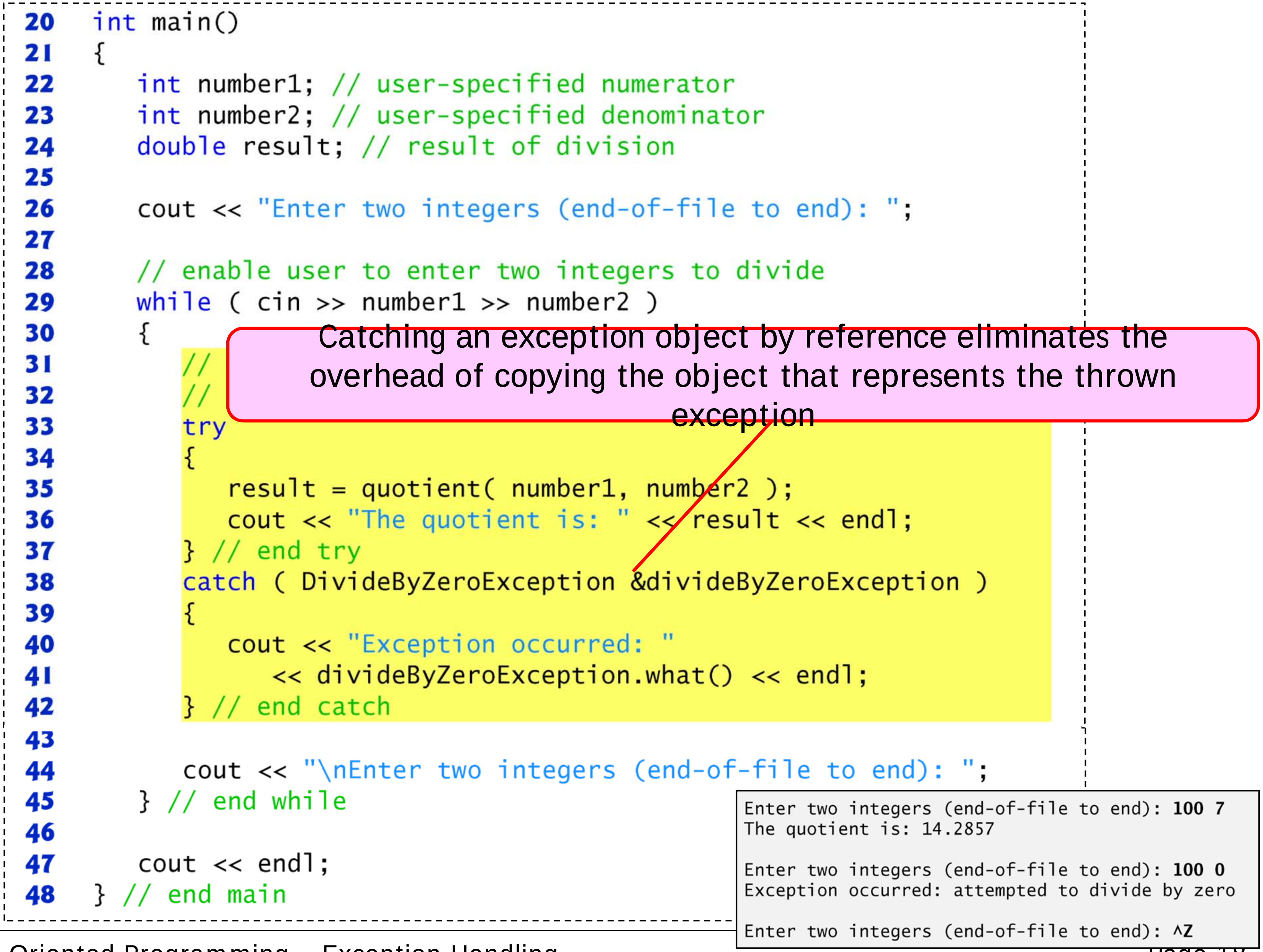
Exception class need not be derived from class `exception`

Throwing a Divide-by-Zero Exception



```cpp
#include <iostream>
#include "DivideByZeroException.h" // DivideByZeroException class
using namespace std;

// 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
```

Catching a Divide-by-Zero Exception



```
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): ";

   // enable user to enter two integers to divide
   while ( cin >> number1 >> number2 )
   {
      //
      //
      try
      {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      } // end try
      catch ( DivideByZeroException &divideByZeroException )
      {
         cout << "Exception occurred: "
            << divideByZeroException.what() << endl;
      } // end catch

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

   cout << endl;
} // end main
```

Catching an exception object by reference eliminates the overhead of copying the object that represents the thrown exception

```
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
```

Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

When to Use Exception Handling

- To process synchronous errors, which occur when a statement executes. For example,
  - Out-of-range array subscripts
  - Arithmetic overflow
  - Division by zero
  - Invalid function parameters
  - Unsuccessful memory allocation
  - …
- Not for processor errors associated with asynchronous events. For example,
  - Disk I/O completions
  - Mouse clicks
  - ...

Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

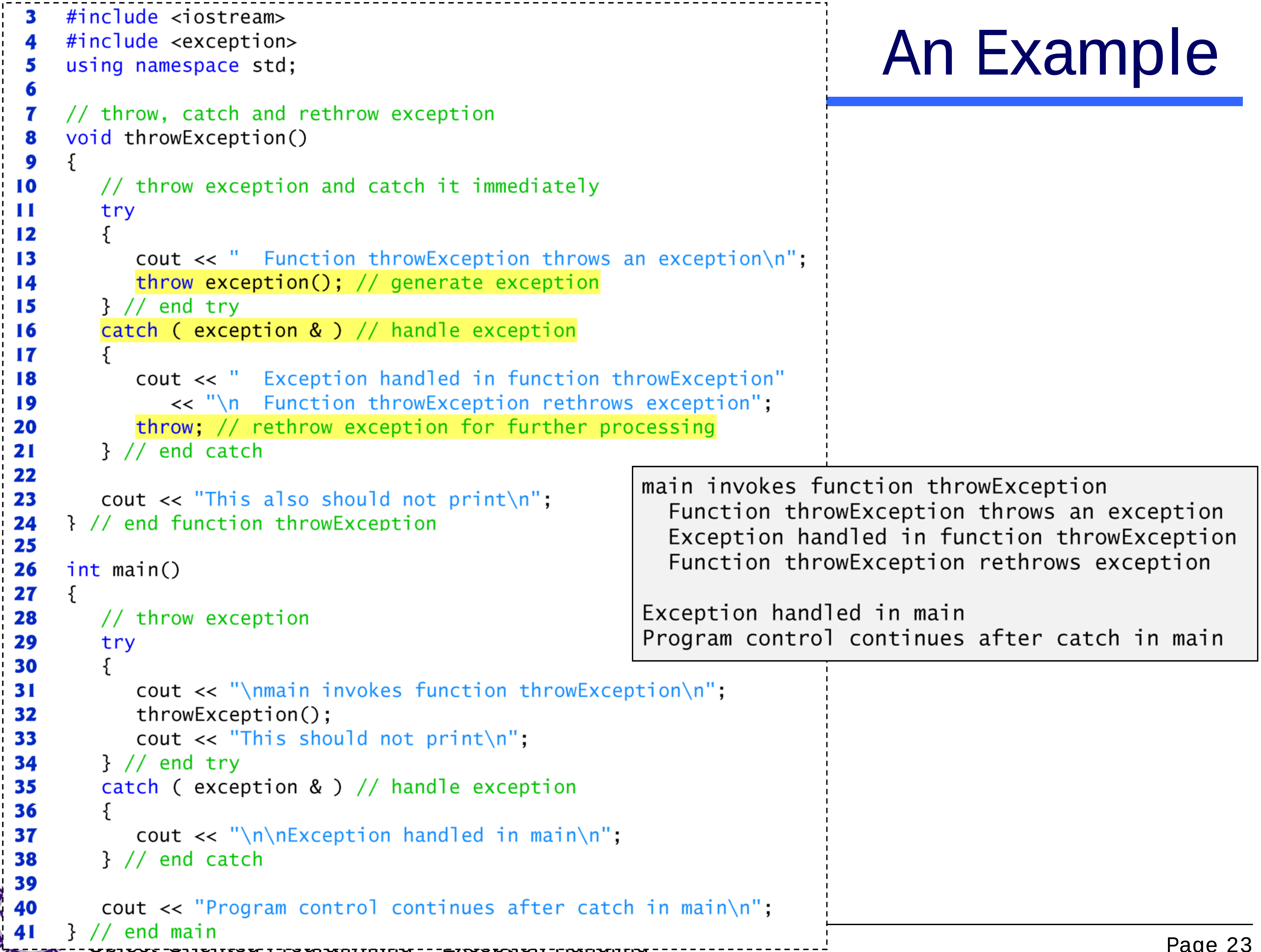
Rethrowing an Exception

- It is possible that an exception handler might decide either that
  - it cannot process that exception or that
  - it can process the exception only partially
- In either case, the exception handler can rethrow exception via the statement

```
throw;
```

  - Executing a throw statement without an operand outside a `catch` handler calls the default exception handler

An Example



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

// 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

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";
} // end main
```

```
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
```

Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

Exception Specifications (1/2)

- Used to enumerates a list of exceptions that a function can throw
  - Optional
  - Also called throw list
  - If the function throws an exception that does not belong to a specified type, function `unexpected` (calling function `terminate` by default) is called
- Begins with keyword `throw` immediately following the closing parenthesis of the function's parameter list
- E.g.,

```
int someFunction( double value )
   throw ( ExceptionA, ExceptionB, ExceptionC )
{
   // function body
}
```

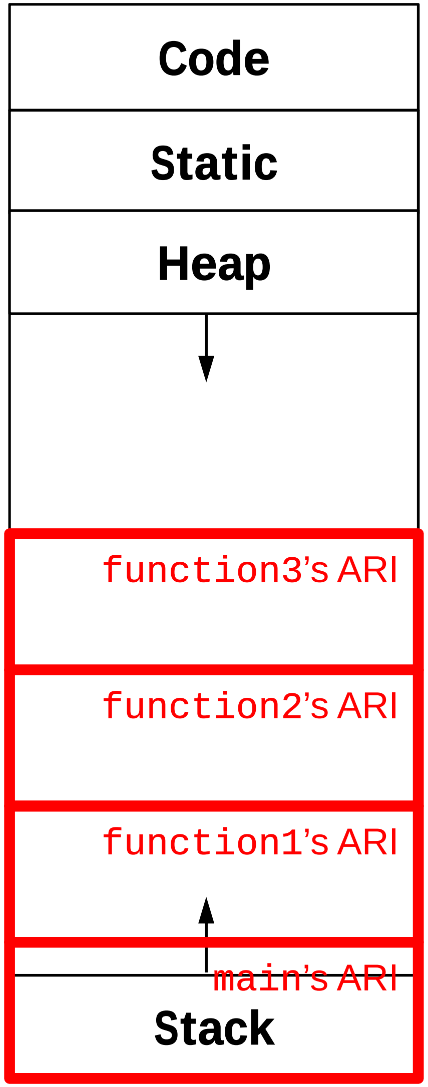
Exception Specifications (2/2)

- A function without an exception specification can throw any exception
- A function with an empty exception specification (`throw()`) means that the function does not throw exceptions
  - If the function attempts to throw an exception, function `unexpected` is called

Example: Exception Specification & Stack Unwinding

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

// function3 throws runtime error
void function3() throw ( runtime_error )
{
   cout << "In function 3" << endl;

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

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

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

// demonstrate stack unwindi
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 runtime error
   {
      cout << "Exception occurred: " << error.what() << endl;
      cout << "Exception handled in main" << endl;
   } // end catch
} // end main
```

```
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
```



Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
    - Constructors, Destructors, and Exception Handling
    - Exceptions and Inheritance
    - Function `terminate`
    - Processing new Failures
    - Class `auto_ptr`

Errors Occurring in a Constructor (1/2)

- How should an object’s constructor respond when new fails? (the constructor cannot return a value to indicate an error)
- Options
  - To return the improperly constructed object and hope that anyone who use it would make appropriate tests to determine that it’s improperly constructed
  - To set some variable outside the constructor to indicate the error
  - (Preferred) To require the constructor to throw an exception that contains the error information

Errors Occurring in a Constructor (2/2)

- Before an exception is thrown by a constructor, destructors are called for
  - every member object built as part of the object
  - every automatic object constructed in a `try` block
- If a destructor invoked during stack unwinding throws an exception, function `terminate` is called

Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

Exceptions and Inheritance

- If a `catch` handler catches a pointer or reference on an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes publicly derived from that base class
- Using inheritance with exceptions enables an exception handler to `catch` related errors with concise notation

Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

Function terminate

- Cases in which function terminate is called include:
  - No matching catch is found for a thrown exception
  - A destructor attempts to throw an exception during stack unwinding
  - Rethrowing an exception when there is no exception currently being handled
  - A call to function unexpected defaults to calling function terminate
- set_terminate can specify the function to invoke when terminate is called
  - terminate calls function abort by default

Outline

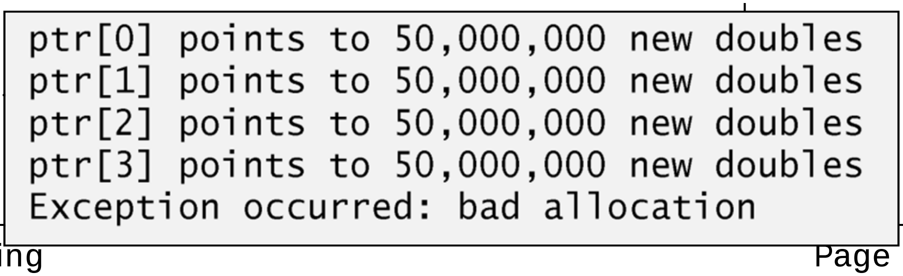
- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

Processing new Failures

- When operator new fails, it throws a `bad_alloc` exception (defined in <new>)
- Two ways to handle the exception
  - Write a `try-catch` clause to catch the exception
  - Use function `set_new_handler` (whose prototype is defined in <new>) to handle new failures
    - Once a new-handler is registered, operator new does not throw `bad_alloc` on failure

Write a try-catch Clause

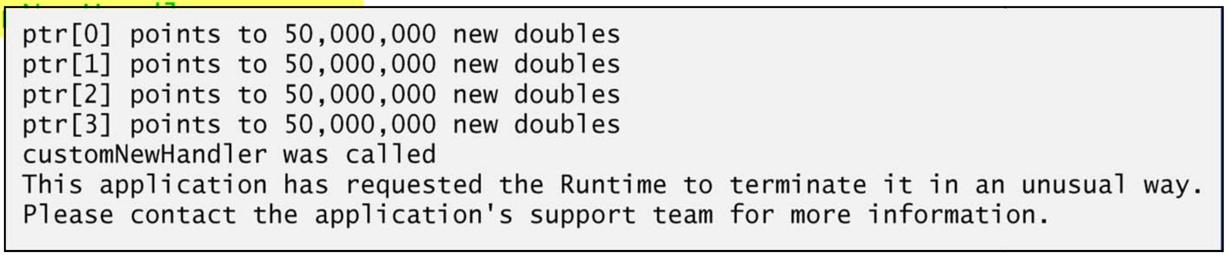
```
#include <iostream>
#include <new> // bad_alloc class is defined here
using namespace std;

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

   // aim each ptr[i] at a big block of memory
   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 << "ptr[" << i << "] points to 50,000,000 new doubles\n";
      } // end for
   } // end try
   catch ( bad_alloc &memoryAllocationException )
   {
      cerr << "Exception occurred: "
         << memoryAllocationException.what() << endl;
   } // end catch
} // end main
```



```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
Exception occurred: bad allocation
```

Use Function set_new_handler

```
#include <iostream>
#include <new> // set_new_handler function prototype
#include <cstdlib> // abort function prototype
using namespace std;

// handle memory allocation failure
void customNewHandler()
{
   cerr << "customNewHandler was called";
   abort();
} // end function custo

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

   // specify that customNewHandler should be called on
   // memory allocation failure
   set_new_handler( customNewHandler );

   // aim each ptr[i] at a big block of memory; customNewHandler will be
   // called on failed memory allocation
   for ( int i = 0; i < 50; i++ )
   {
      ptr[ i ] = new double[ 50000000 ]; // may throw exception
      cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
   } // end for
}  // end main
```



```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
customNewHandler was called
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

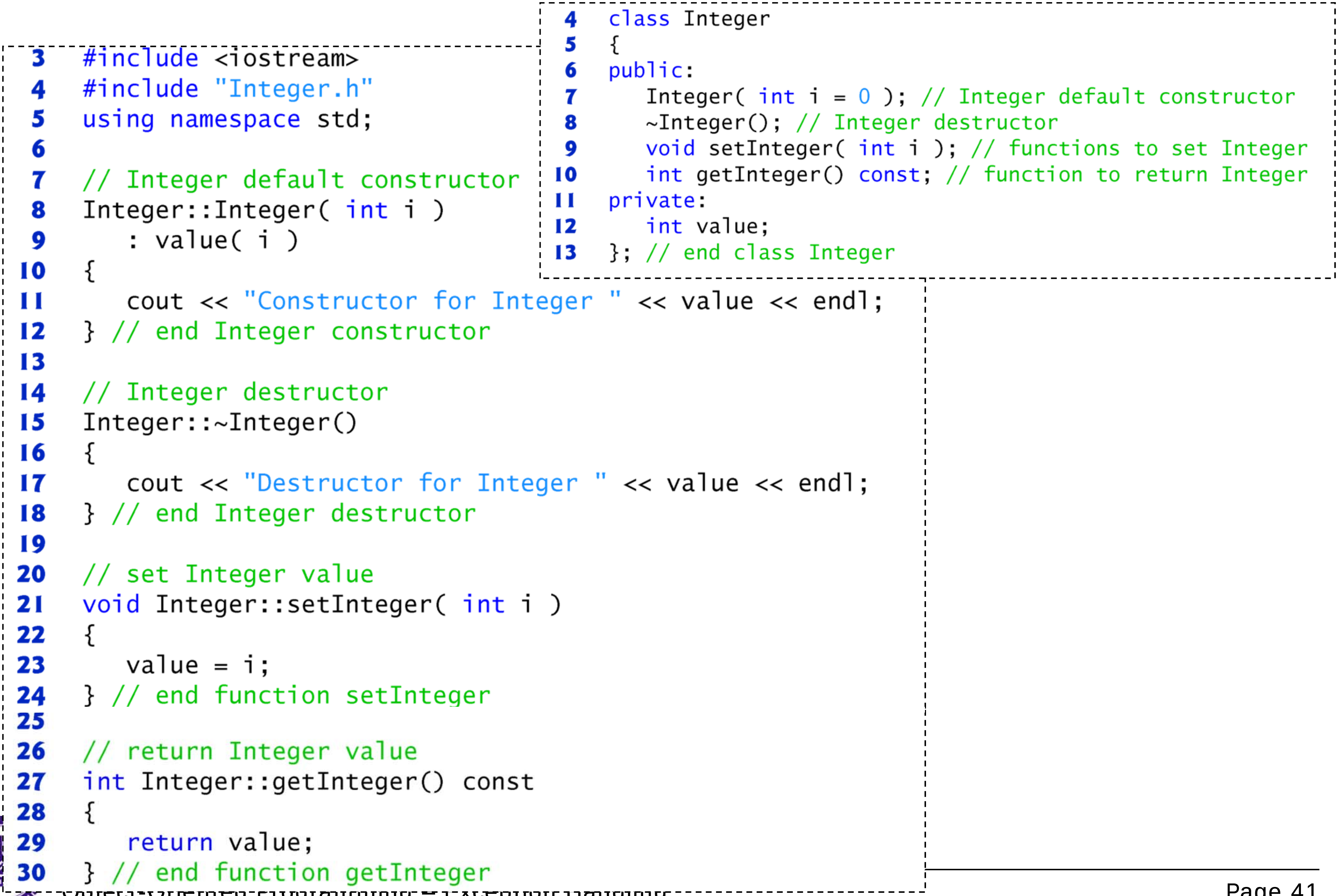
Outline

- Introduction to Exception
- Example: Division by Zero
- When to Use Exception Handling
- Rethrowing an Exception
- Exception Specifications
- Other Issues
  - Constructors, Destructors, and Exception Handling
  - Exceptions and Inheritance
  - Function `terminate`
  - Processing new Failures
  - Class `auto_ptr`

Class auto_ptr

- If an exception occurs after successful dynamic memory allocation but before the delete statement executes, a memory leak could occur
  - Class template auto_ptr (<memory>) is provided to deal with this situation
- An auto_ptr object maintains a pointer to dynamically allocated memory
  - When an auto_ptr object destructor is called (e.g., when an auto_ptr object goes out of scope), it performs a delete option on its pointer data member
- auto_ptr provides overloaded operators * and -> so that an auto_ptr object can be used as a regular pointer variable

Integer Class Definition



```cpp
class Integer
{
public:
   Integer( int i = 0 ); // Integer default constructor
   ~Integer(); // Integer destructor
   void setInteger( int i ); // functions to set Integer
   int getInteger() const; // function to return Integer
private:
   int value;
}; // end class Integer
```

```cpp
#include <iostream>
#include "Integer.h"
using namespace std;

// Integer default constructor
Integer::Integer( int i )
   : value( i )
{
   cout << "Constructor for Integer " << value << endl;
} // end Integer constructor

// Integer destructor
Integer::~Integer()
{
   cout << "Destructor for Integer " << value << endl;
} // end Integer destructor

// set Integer value
void Integer::setInteger( int i )
{
   value = i;
} // end function setInteger

// return Integer value
int Integer::getInteger() const
{
   return value;
} // end function getInteger
```

Using auto_ptr

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

#include "Integer.h"

// use auto_ptr to manipulate Integer object
int main()
{
   cout << "Creating an auto_ptr object that points to an Integer\n";

   // "aim" auto_ptr at Integer object
   auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );

   cout << "\nUsing the auto_ptr to manipulate the Integer\n";
   ptrToInteger->setInteger( 99 ); // use auto_ptr to set Integer value

   // use auto_ptr to get Integer value
   cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
} // end main
```

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99

Destructor for Integer 99
```

Notes on Using `auto_ptr`

- Because `auto_ptr` objects transfer ownership of memory when they are copied, they cannot be used with STL containers like `vector`
  - Container classes often make copies of objects
  - This causes ownership of a container element to be transferred to another object, which might then be accidentally deleted when the copy goes out of scope
- The `Boost.Smart_ptr` library provides memory management features similar to `auto_ptr` that can be used with containers