1 Exception Handling

1.1 Introduction

- Exceptions
  - Indicates problem occurred in program
  - Not common; An *exception* to a program that usually works
- Exception Handling
  - Resolve exceptions
  - Program may be able to continue; Controlled termination
  - Write fault-tolerant programs; As an example, we will handle a divide-by-zero error

1.2 Exception-Handling Overview

- Consider pseudocode

```
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
```

- Mixing logic and error handling
  - Can make program difficult to read/debug
  - Exception handling removes error correction from *main line* of program
- Exception handling
  - For synchronous errors (divide by zero, null pointer)
    - Cannot handle asynchronous errors (independent of program)
    - Disk I/O, mouse, keyboard, network messages
  - Easy to handle errors
- Terminology

  - Function that has error *throws an exception*
  - *Exception handler* (if it exists) can deal with problem; *Catches* and *handles* exception
  - If no exception handler, *uncaught* exception; Could terminate program

- C++ code

```
try{
   code that may raise exception
   }
catch(exceptionType){
                    code to handle exception
                    }
```

- **try** block encloses code that may raise exception
- One or more **catch** blocks follow
  - Catch and handle exception, if appropriate
  - Take parameter; if named, can access exception object
- Throw point
  - Location in **try** block where exception occurred
  - If exception handled
    * Program skips remainder of **try** block
    * Resumes after **catch** blocks
  - If not handled
    * Function terminates
    * Looks for enclosing **catch** block (stack unwinding, 13.8)
- If no exception
  - Program skips **catch** blocks

1.3 Other Error-Handling Techniques

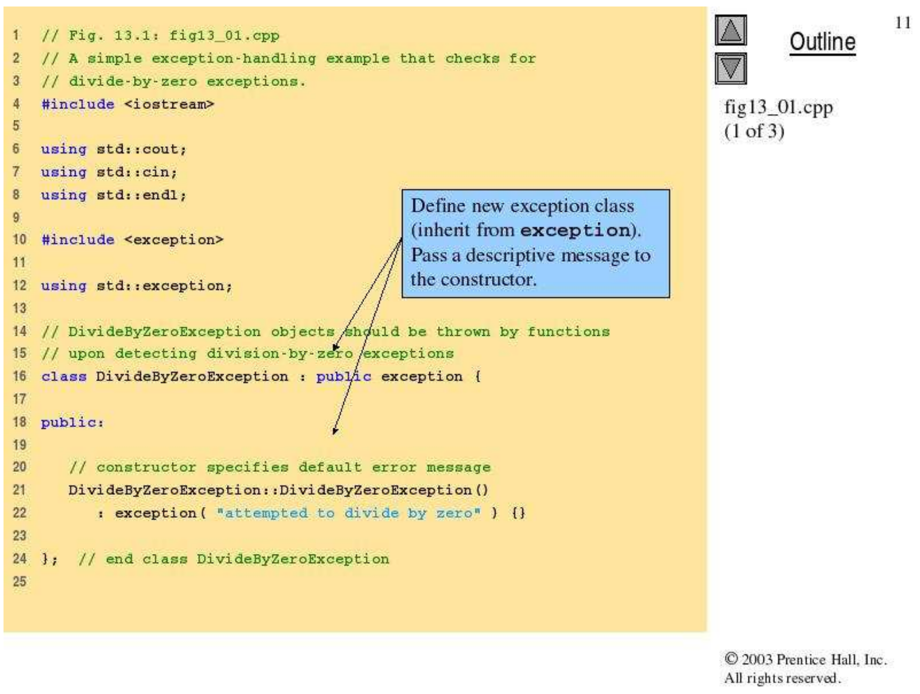
- Ignore exception
  - Typical for personal (not commercial) software
  - Program may fail
- Abort program
  - Usually appropriate
  - Not appropriate for mission-critical software
- Set error indicators
  - Unfortunately, may not test for these when necessary
- Test for error condition
  - Call exit (<**cstdlib**>) and pass error code
- **setjump** and **longjump**
  - <**csetjmp**>
  - Jump from deeply nested function to call error handler
  - Can be dangerous
- Dedicated error handling
  - **new** can have a special handler
  - Discussed 13.11

1.4 Simple Exception-Handling Example: Divide by Zero

- Keyword throw
  - Throws an exception; Use when error occurs
  - Can throw almost anything (exception object, integer, etc.); **throw myObject;**, **throw 5;**
- Exception objects
  - Base class **exception** (<**exception**>)

  - Constructor can take a string (to describe exception)
  - Member function **what()** returns that string

- Upcoming example
  - Handle divide-by-zero errors
  - Define new exception class
    - **DivideByZeroException**
    - Inherit from **exception**
  - In division function
    - Test denominator
    - If zero, throw exception (**throw object**)
  - In **try** block
    - Attempt to divide
    - Have enclosing **catch** block; Catch **DivideByZeroException** objects

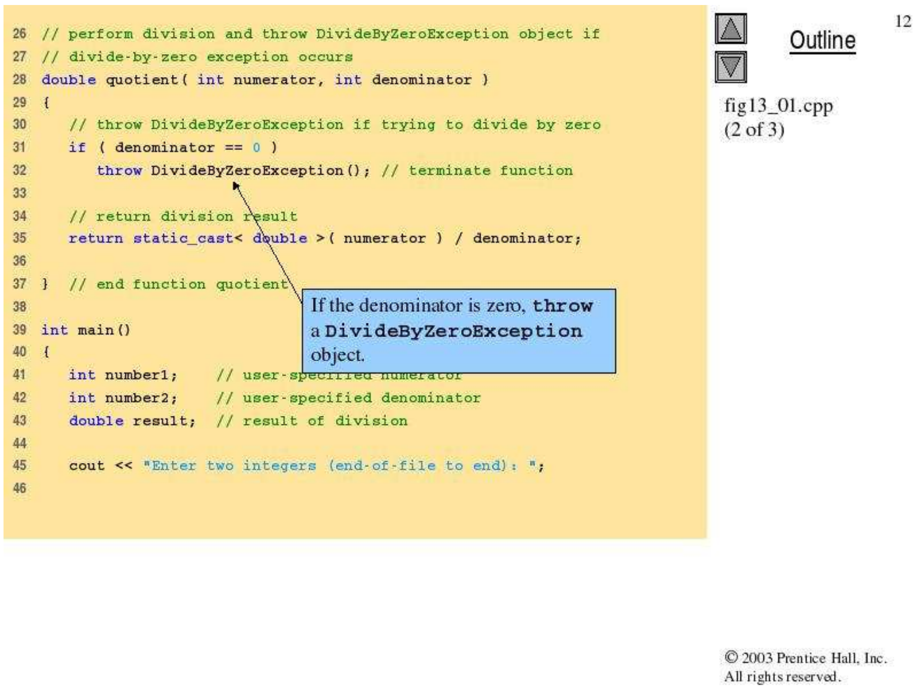


Outline

fig13_01.cpp
(1 of 3)

```cpp
// Fig. 13.1: fig13_01.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <exception>

using std::exception;

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public exception {

public:

   // constructor specifies default error message
   DivideByZeroException::DivideByZeroException()
      : exception( "attempted to divide by zero" ) {}

};  // end class DivideByZeroException

```

Define new exception class (inherit from **`exception`**). Pass a descriptive message to the constructor.



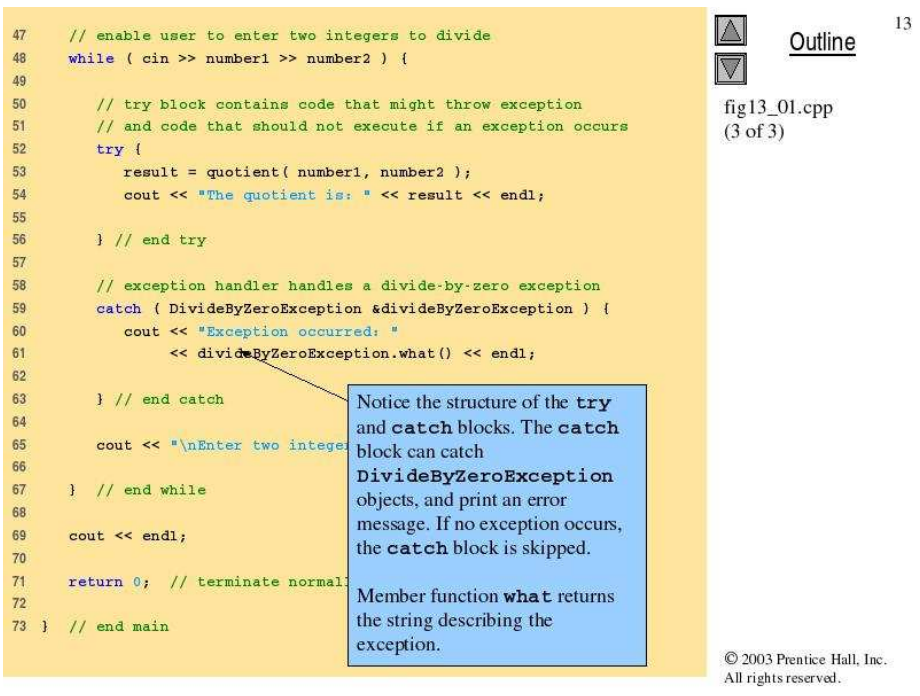
Outline

fig13_01.cpp
(2 of 3)

```cpp
// 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): ";

```

If the denominator is zero, **`throw`** a **`DivideByZeroException`** object.

Figure 1: Exception-handling example that throws exceptions on attempts to divide by zero. (Part 1 of 2)



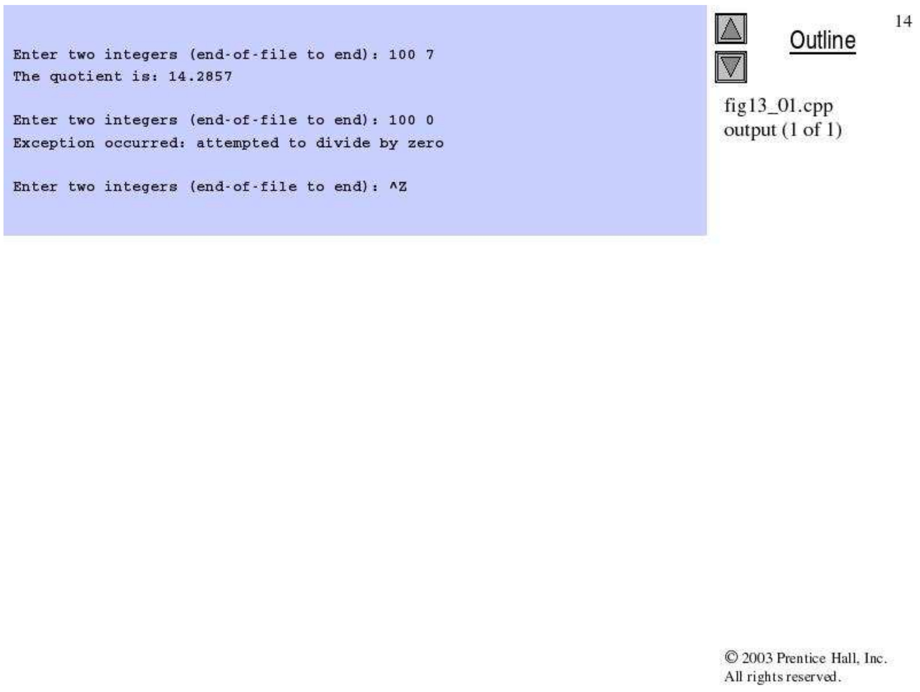
Outline

fig13_01.cpp
(3 of 3)

```
   // 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 intege

   }  // end while

   cout << endl;

   return 0;   // terminate normal

}  // end main
```

Notice the structure of the **`try`** and **`catch`** blocks. The **`catch`** block can catch **`DivideByZeroException`** objects, and print an error message. If no exception occurs, the **`catch`** block is skipped.

Member function **`what`** returns the string describing the exception.



Outline

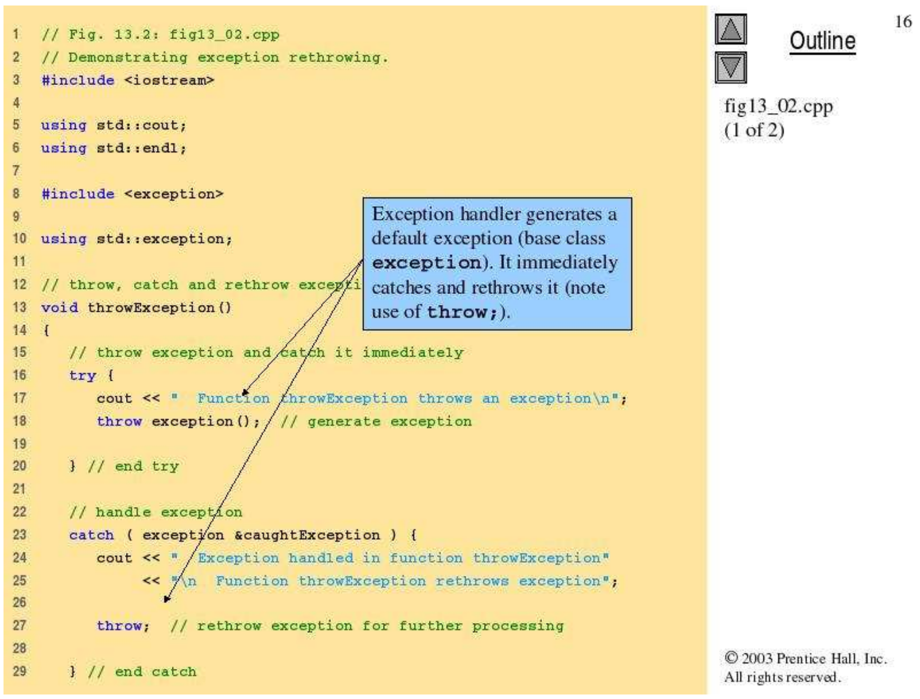
fig13_01.cpp
output (1 of 1)

```
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
```

Figure 2: Exception-handling example that throws exceptions on attempts to divide by zero. (Part 2 of 2)

1.5 Rethrowing an Exception

- Rethrowing exceptions
  - Use when exception handler cannot process exception; Can still rethrow if handler did some processing
  - Can rethrow exception to another handler
    - Goes to next enclosing **try** block
    - Corresponding **catch** blocks try to handle
- To rethrow
  - Use statement ***throw;***
    - No arguments
    - Terminates function

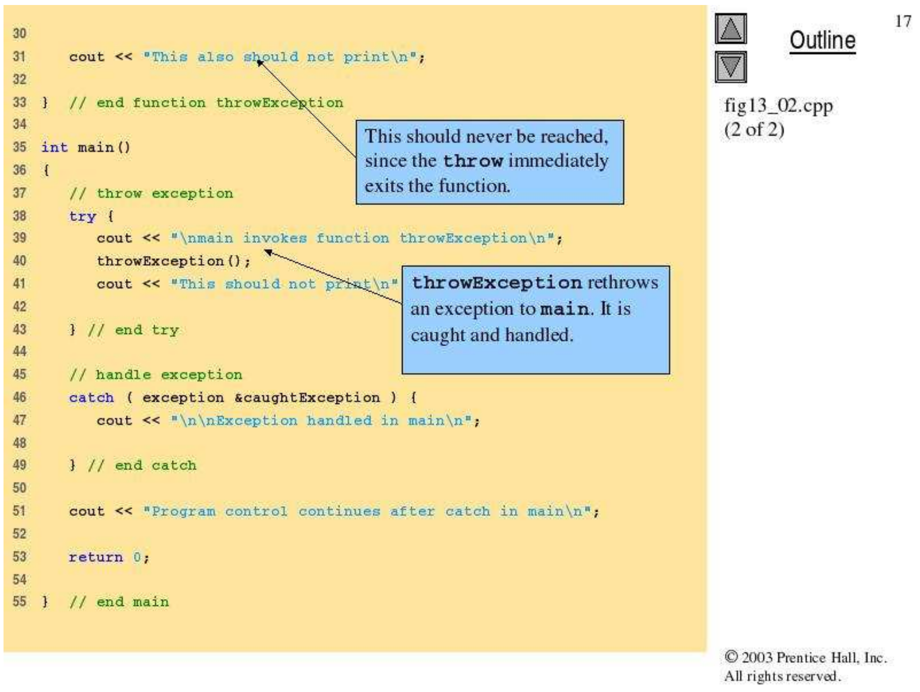


Outline

fig13_02.cpp
(1 of 2)

```cpp
// Fig. 13.2: fig13_02.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

   // handle exception
   catch ( exception &caughtException ) {
      cout << "  Exception handled in function throwException"
           << "\n  Function throwException rethrows exception";

      throw;  // rethrow exception for further processing

   } // end catch
```

Exception handler generates a default exception (base class **exception**). It immediately catches and rethrows it (note use of **throw;**).



Outline

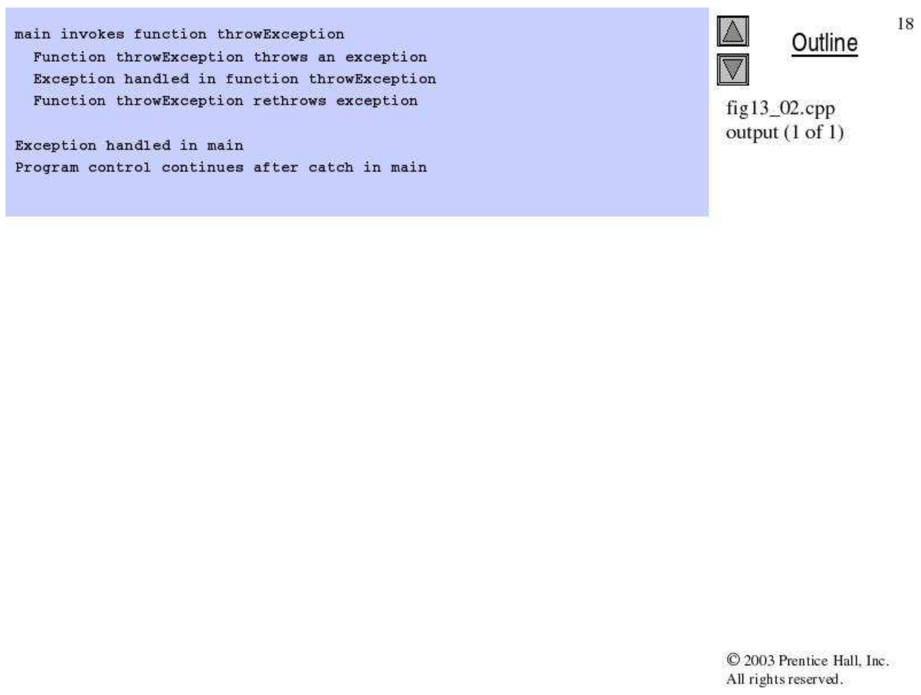
fig13_02.cpp
(2 of 2)

```cpp

   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

   // handle exception
   catch ( exception &caughtException ) {
      cout << "\n\nException handled in main\n";

   } // end catch

   cout << "Program control continues after catch in main\n";

   return 0;

}  // end main
```

This should never be reached, since the **throw** immediately exits the function.

throwException rethrows an exception to **main**. It is caught and handled.

Figure 3: Rethrowing an exception. (Part 1 of 2)



```
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

Outline

fig13_02.cpp
output (1 of 1)

Figure 4: Rethrowing an exception. (Part 2 of 2)