IS 0020
Program Design and Software Tools

Exception Handling

Lecture 11

April 7, 2005

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

3

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

4

Exception-Handling Overview

- 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

Exception-Handling Overview

- 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

Exception-Handling Overview

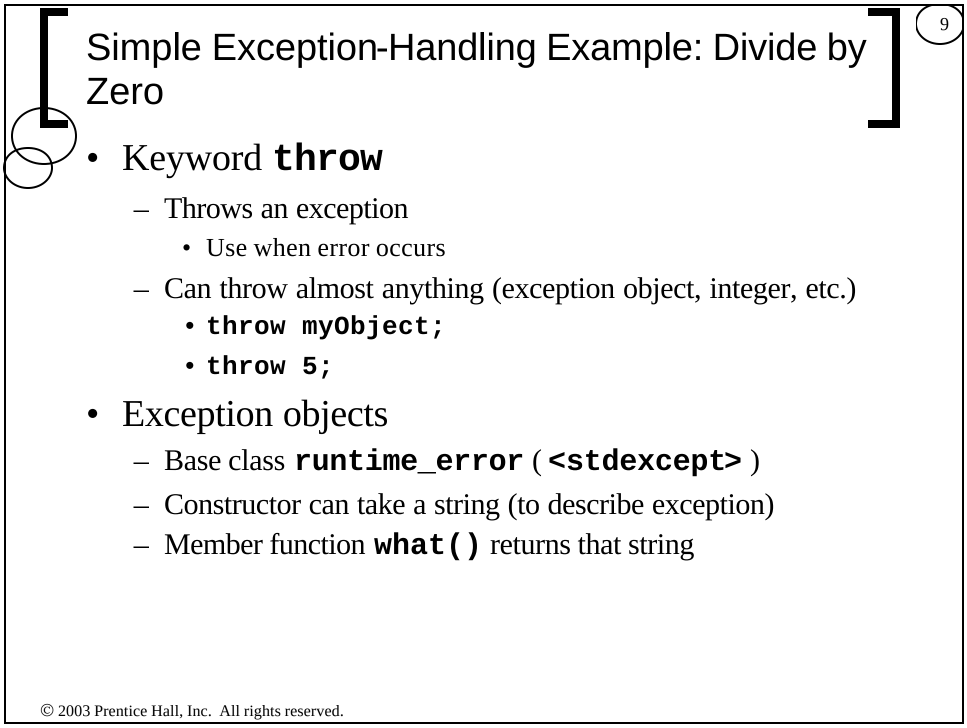
- Throw point
  - Location in **`try`** block where exception occurred
  - If exception handled
    - Program skips remainder of **`try`** block
    - Resumes after **`catch`** blocks
  - If not handled
    - Function terminates
    - Looks for enclosing **`catch`** block (stack unwinding)
- If no exception
  - Program skips **`catch`** blocks

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

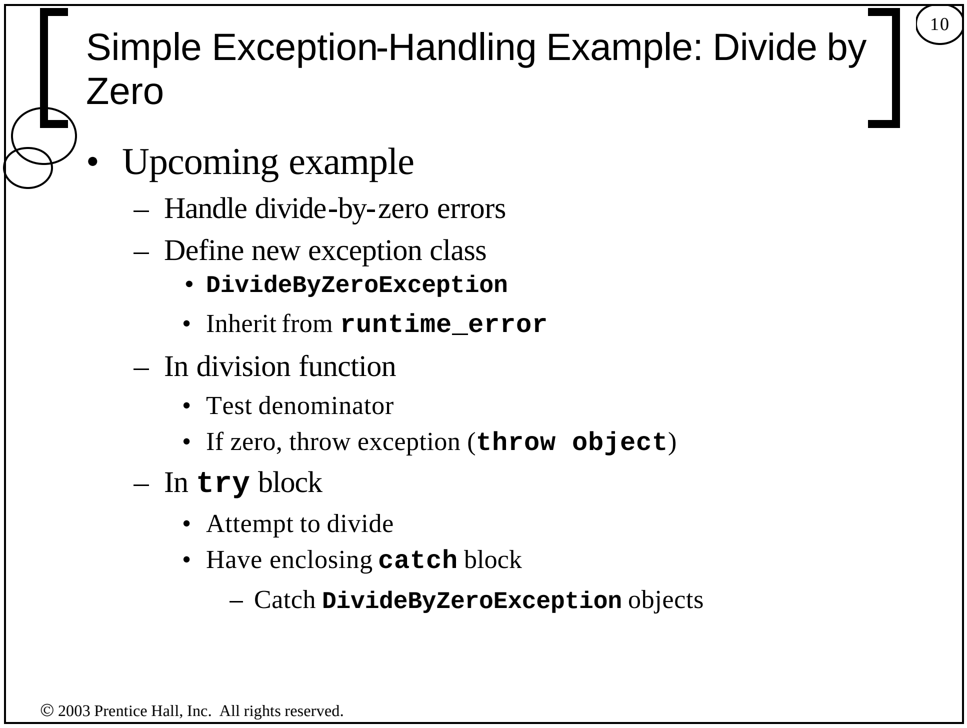
Other Error-Handling Techniques

- **`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



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 **`runtime_error`** (**`<stdexcept>`**)
  - Constructor can take a string (to describe exception)
  - Member function **`what()`** returns that string



Simple Exception-Handling Example: Divide by Zero

- Upcoming example
  - Handle divide-by-zero errors
  - Define new exception class
    - **`DivideByZeroException`**
    - Inherit from **`runtime_error`**
  - 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)

```
// 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 runtime_error {

public:

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

};  // end class DivideByZeroException

```

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

Outline

fig13_01.cpp
(2 of 3)

```
// 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.

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 integ

   }  // end while

   cout << endl;

   return 0;  // terminate norma

}  // 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.

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
```

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)

```
// 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)

```

   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.

Outline

fig13_02.cpp
output (1 of 1)

```
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
```

Exception Specifications

- List of exceptions function can throw
  - Also called throw list

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

  - Can only throw **`ExceptionA`**, **`ExceptionB`**, and **`ExceptionC`** (and derived classes)
    - If throws other type, function **`unexpected`** called
    - By default, terminates program (more 13.7)
  - If no throw list, can throw any exception
  - If empty throw list, cannot throw any exceptions

Processing Unexpected Exceptions

- Function **`unexpected`**
  - Calls function registered with **`set_unexpected`**
    - **`<exception>`**
    - Calls **`terminate`** by default
  - **`set_terminate`**
    - Sets what function **`terminate`** calls
    - By default, calls **`abort`**
      - If redefined, still calls **`abort`** after new function finishes
- Arguments for set functions
  - Pass pointer to function
    - Function must take no arguments
    - Returns **`void`**

Stack Unwinding

- If exception thrown but not caught
  - Goes to enclosing **`try`** block
  - Terminates current function
    - Unwinds function call stack
  - Looks for **`try`**/**`catch`** that can handle exception
    - If none found, unwinds again
- If exception never caught
  - Calls **`terminate`**

Outline

fig13_03.cpp
(1 of 2)

```
// Fig. 13.3: fig13_03.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 )
{
   throw runtime_error( "runtime_error in function3" ); // fourth
}

// function2 invokes function3
void function2() throw ( runtime_error )
{
   function3(); // third
}

```

Note the use of the throw list. Throws a runtime error exception, defined in **`<stdexcept>`**.

Outline

fig13_03.cpp
(2 of 2)

```
// function1 invokes function2
void function1() throw ( runtime_error )
{
   function2(); // second
}

// demonstrate stack unwinding
int main()
{
   // invoke function1
   try {
      function1(); // first

   } // end try

   // handle run-time error
   catch ( runtime_error &error ) // fifth
   {
      cout << "Exception occurred: " << error.what() << endl;

   } // end catch

   return 0;

}  // end main
```

`function1` calls **`function2`** which calls **`function3`**. The exception occurs, and unwinds until an appropriate **`try`**/**`catch`** block can be found.

```
Exception occurred: runtime_error in function3
```

Constructors, Destructors and Exception Handling

- Error in constructor
  - **`new`** fails; cannot allocate memory
  - Cannot return a value - how to inform user?
    - Hope user examines object, notices errors
    - Set some global variable
  - Good alternative: throw an exception
    - Destructors automatically called for member objects
    - Called for automatic variables in **`try`** block
- Can catch exceptions in destructor

Exceptions and Inheritance

- Exception classes
  - Can be derived from base classes
    - I.e., **`runtime_error;`** **`exception`**
  - If **`catch`** can handle base class, can handle derived classes
    - Polymorphic programming

Processing new Failures

- When **`new`** fails to get memory
  - Should **`throw`** **`bad_alloc`** exception
    - Defined in **`<new>`**
  - Some compilers have **`new`** return 0
  - Result depends on compiler

Outline

fig13_04.cpp
(1 of 2)

```
// Fig. 13.4: fig13_04.cpp
// Demonstrating pre-standard new returning 0 when memory
// is not allocated.
#include <iostream>

using std::cout;

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

   // allocate memory for ptr
   for ( int i = 0; i < 50; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      // new returns 0 on failure to alloc
      if ( ptr[ i ] == 0 ) {
         cout << "Memory allocation failed for ptr[ "
              << i << " ]\n";

         break;

      } // end if

```

Demonstrating `new` that returns `0` on allocation failure.

Outline

fig13_04.cpp
(2 of 2)

fig13_04.cpp
output (1 of 1)

```
      // successful memory allocation
      else
         cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";

   } // end for

   return 0;

}  // end main
```

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Memory allocation failed for ptr[ 4 ]
```

Outline

fig13_05.cpp
(1 of 2)

```
// Fig. 13.5: fig13_05.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.
#include <iostream>

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

#include <new> // standard operator new

using std::bad_alloc;

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

   // attempt to allocate memory
   try {

      // allocate memory for ptr[ i ]; new throws bad_alloc
      // on failure
      for ( int i = 0; i < 50; i++ ) {
         ptr[ i ] = new double[ 5000000 ];
         cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";
      }

   } // end try
```

Demonstrating **new** that throws an exception.

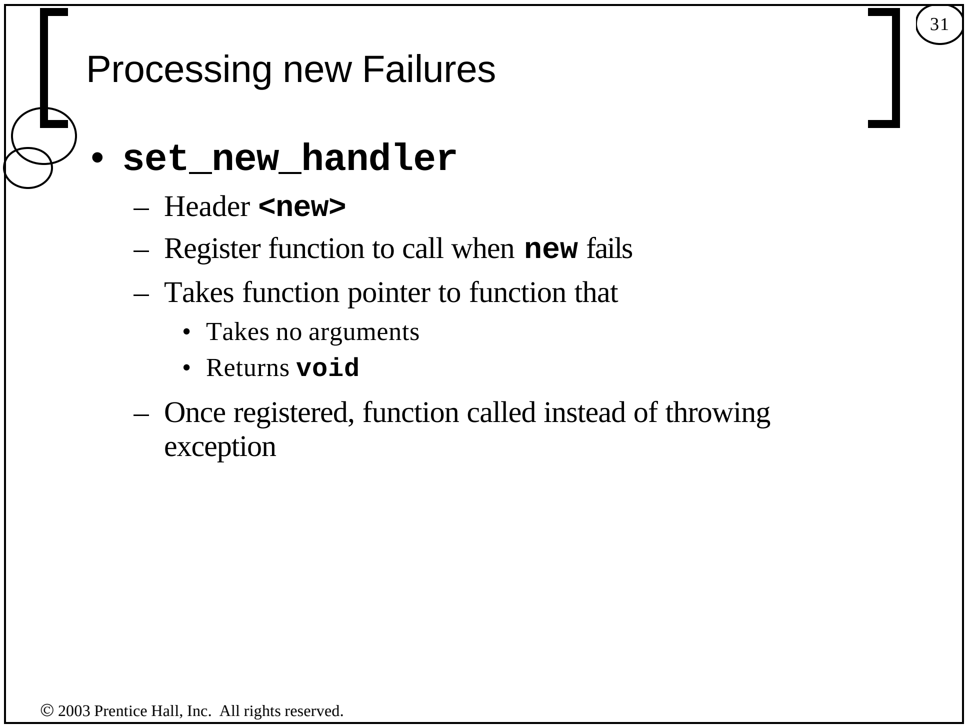
Outline

fig13_05.cpp
(2 of 2)

fig13_05.cpp
output (1 of 1)

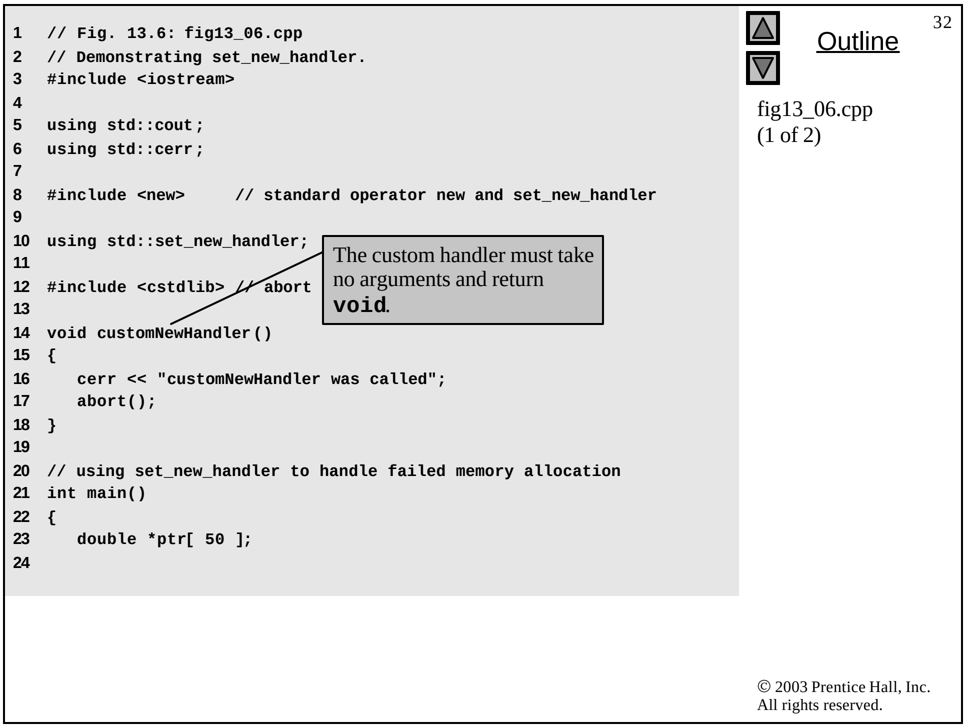
```

   // handle bad_alloc exception
   catch ( bad_alloc &memoryAllocationException ) {
      cout << "Exception occurred: "
           << memoryAllocationException.what() << endl;

   } // end catch

   return 0;

}  // end main
```

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Exception occurred: Allocation Failure
```



Processing new Failures

- **`set_new_handler`**
  - Header **`<new>`**
  - Register function to call when **`new`** fails
  - Takes function pointer to function that
    - Takes no arguments
    - Returns **`void`**
  - Once registered, function called instead of throwing exception



```
// Fig. 13.6: fig13_06.cpp
// Demonstrating set_new_handler.
#include <iostream>

using std::cout;
using std::cerr;

#include <new>     // standard operator new and set_new_handler

using std::set_new_handler;

#include <cstdlib> // abort

void customNewHandler()
{
   cerr << "customNewHandler was called";
   abort();
}

// using set_new_handler to handle failed memory allocation
int main()
{
   double *ptr[ 50 ];

```

The custom handler must take no arguments and return **`void`**.

fig13_06.cpp
(1 of 2)

Outline

fig13_06.cpp
(2 of 2)

fig13_06.cpp
output (1 of 1)

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

   // allocate memory for ptr[ i ]; customNewHandler will be
   // called on failed memory allocation
   for ( int i = 0; i < 50; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      cout << "Allocated 5000000 doubles in ptr[ "
           << i << " ]\n";

   } // end for

   return 0;

}  // end main
```

Note call to **set_new_handler**.

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
customNewHandler was called
```

Class auto_ptr and Dynamic Memory Allocation

- Declare pointer, allocate memory with **new**
  - What if exception occurs before you can **delete** it?
  - Memory leak
- Template class **auto_ptr**
  - Header **<memory>**
  - Like regular pointers (has ***** and **->**)
  - When pointer goes out of scope, calls **delete**
  - Prevents memory leaks
  - Usage

    auto_ptr< MyClass > newPointer(new MyClass());

    - **newPointer** points to dynamically allocated object

Outline

fig13_07.cpp
(1 of 3)

```
// Fig. 13.7: fig13_07.cpp
// Demonstrating auto_ptr.
#include <iostream>

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

#include <memory>

using std::auto_ptr; // auto_ptr class definition

class Integer {

public:

   // Integer constructor
   Integer( int i = 0 )
      : value( i )
   {
      cout << "Constructor for Integer " << value << endl;

   } // end Integer constructor

```

Outline

fig13_07.cpp
(2 of 3)

```
   // Integer destructor
   ~Integer()
   {
      cout << "Destructor for Integer "  << value << endl;

   } // end Integer destructor

   // function to set Integer
   void setInteger( int i )
   {
      value = i;

   } // end function setInteger

   // function to return Integer
   int getInteger() const
   {
      return value;

   } // end function getInteger

private:
   int value;

};  // end class Integer

```

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

   // use auto_ptr to set Integer value
   ptrToInteger->setInteger( 99 );

   // use auto_ptr to get Integer value
   cout << "Integer after setInteger: "
        << ( *ptrToInteger ).getInteger()
        << "\n\nTerminating program" << endl;

   return 0;

}  // end main
```

Create an **auto_ptr**. It can be manipulated like a regular pointer.

delete not explicitly called, but the **auto_ptr** will be destroyed once it leaves scope. Thus, the destructor for class **Integer** will be called.

fig13_07.cpp
(3 of 3)

```
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

Terminating program
Destructor for Integer 99
```

fig13_07.cpp
output (1 of 1)

Standard Library Exception Hierarchy

- Exception hierarchy
  - Base class **`exception`** (**`<exception>`**)
    - Virtual function **`what`**, overridden to provide error messages
  - Sample derived classes
    - **`runtime_error`**, **`logic_error`**
    - **`bad_alloc`**, **`bad_cast`**, **`bad_typeid`**
      - Thrown by **`new`**, **`dynamic_cast`** and **`typeid`**
- To catch all exceptions
  - **`catch(...)`**
  - **`catch( exception AnyException)`**
    - Will not catch user-defined exceptions