Illustrations List *(Main Page)*

```
// Fig. 13.1: fig13_01.cpp
// A simple exception handling example.
// Checking for a divide-by-zero exception.
#include <iostream.h>

// Class DivideByZeroException to be used in exception
// handling for throwing an exception on a division by zero.
class DivideByZeroException {
public:
   DivideByZeroException()
      : message( "attempted to divide by zero" ) { }
   const char *what() const { return message; }
private:
   const char *message;
};

// Definition of function quotient. Demonstrates throwing
// an exception when a divide-by-zero exception is encountered.
double quotient( int numerator, int denominator )
{
   if ( denominator == 0 )
      throw DivideByZeroException();

   return static_cast< double > ( numerator ) / denominator;
}

// Driver program
int main()
{
   int number1, number2;
   double result;

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

   while ( cin >> number1 >> number2 ) {

      // the try block wraps the code that may throw an
      // exception and the code that should not execute
      // if an exception occurs
      try {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      }
```

Fig. 13.1 A simple exception-handling example with divide by zero (part 1 of 2).

```
      catch ( DivideByZeroException ex ) { // exception handler
         cout << "Exception occurred: " << ex.what() << '\n';
      }

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

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

```
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): 33 9
The quotient is: 3.66667

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

Fig. 13.1 A simple exception-handling example with divide by zero (part 2 of 2).

```cpp
// Fig. 13.2: fig13_02.cpp
// Demonstration of rethrowing an exception.
#include <iostream>
#include <exception>

using namespace std;

void throwException() throw ( exception )
{
   // Throw an exception and immediately catch it.
   try {
      cout << "Function throwException\n";
      throw exception();  // generate exception
   }
   catch( exception e )
   {
      cout << "Exception handled in function throwException\n";
      throw;  // rethrow exception for further processing
   }
```

Fig. 13.2 Rethrowing an exception (part 1 of 2).

```cpp

   cout << "This also should not print\n";
}

int main()
{
   try {
      throwException();
      cout << "This should not print\n";
   }
   catch ( exception e )
   {
      cout << "Exception handled in main\n";
   }

   cout << "Program control continues after catch in main"
        << endl;
   return 0;
}
```

```
Function throwException
Exception handled in function throwException
Exception handled in main
Program control continues after catch in main
```

Fig. 13.2 Rethrowing an exception (part 2 of 2).

```
// Fig. 13.3: fig13_03.cpp
// Demonstrating stack unwinding.
#include <iostream>
#include <stdexcept>

using namespace std;

void function3() throw ( runtime_error )
{
   throw runtime_error( "runtime_error in function3" );
}

void function2() throw ( runtime_error )
{
   function3();
}

void function1() throw ( runtime_error )
{
   function2();
}

int main()
{
   try {
      function1();
   }
   catch ( runtime_error e )
   {
      cout << "Exception occurred: " << e.what() << endl;
   }

   return 0;
}
```

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

Fig. 13.3 Demonstration of stack unwinding.

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

int main()
{
   double *ptr[ 10 ];

   for ( int i = 0; i < 10; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      if ( ptr[ i ] == 0 ) { // new failed to allocate memory
         cout << "Memory allocation failed for ptr[ "
              << i << " ]\n";
         break;
      }
      else
         cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";
   }

   return 0;
}
```

Fig. 13.4 Demonstrating `new` returning 0 on failure (part 1 of 2).

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

Fig. 13.4 Demonstrating `new` returning 0 on failure (part 2 of 2).

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

int main()
{
   double *ptr[ 10 ];

   try {
      for ( int i = 0; i < 10; i++ ) {
         ptr[ i ] = new double[ 5000000 ];
         cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";
      }
   }
   catch ( bad_alloc exception ) {
      cout << "Exception occurred: "
           << exception.what() << endl;
   }

   return 0;
}
```

Fig. 13.5 Demonstrating `new` throwing `bad_alloc` on failure (part 1 of 2).

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

Fig. 13.5 Demonstrating `new` throwing `bad_alloc` on failure (part 2 of 2).

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

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

int main()
{
   double *ptr[ 10 ];
   set_new_handler( customNewHandler );

   for ( int i = 0; i < 10; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

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

   return 0;
}
```

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

Fig. 13.6 Demonstrating `set_new_handler`.

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

using namespace std;

class Integer {
public:
   Integer( int i = 0 ) : value( i )
      { cout << "Constructor for Integer " << value << endl; }
   ~Integer()
      { cout << "Destructor for Integer " << value << endl; }
   void setInteger( int i ) { value = i; }
   int getInteger() const { return value; }
private:
   int value;
};

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

   auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );

   cout << "Using the auto_ptr to manipulate the Integer\n";
   ptrToInteger->setInteger( 99 );
   cout << "Integer after setInteger: "
        << ( *ptrToInteger ).getInteger()
        << "\nTerminating program" << endl;

   return 0;
}
```

```
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
```

Fig. 13.7 Demonstrating `auto_ptr`.