Exception Handling

Agenda

- What exceptions are and when to use them
- Using **`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 understand the standard exception hierarchy

Fundamental Philosophy

- Mechanism for sending an exception signal up the call stack

- Regardless of intervening calls

- Note: there is a mechanism based on same philosophy in C
- **`setjmp(), longjmp()`**
- More available in man pages

Traditional Exception Handling

- Intermixing program and error-handling logic
  - Pseudocode outline

```
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
…
```

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

Note:– In most large systems, code to handle errors and exceptions represents >80% of the total code of the system

Fundamental Philosophy (continued)

- Remove 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
  - All exceptions of a group of related types

Fundamental Philosophy (continued)

- Programs can
  - Recover from exceptions
  - Hide exceptions
  - Pass exceptions up the “chain of command”
  - Ignore certain exceptions and let someone else handle them

Fundamental Philosophy (continued)

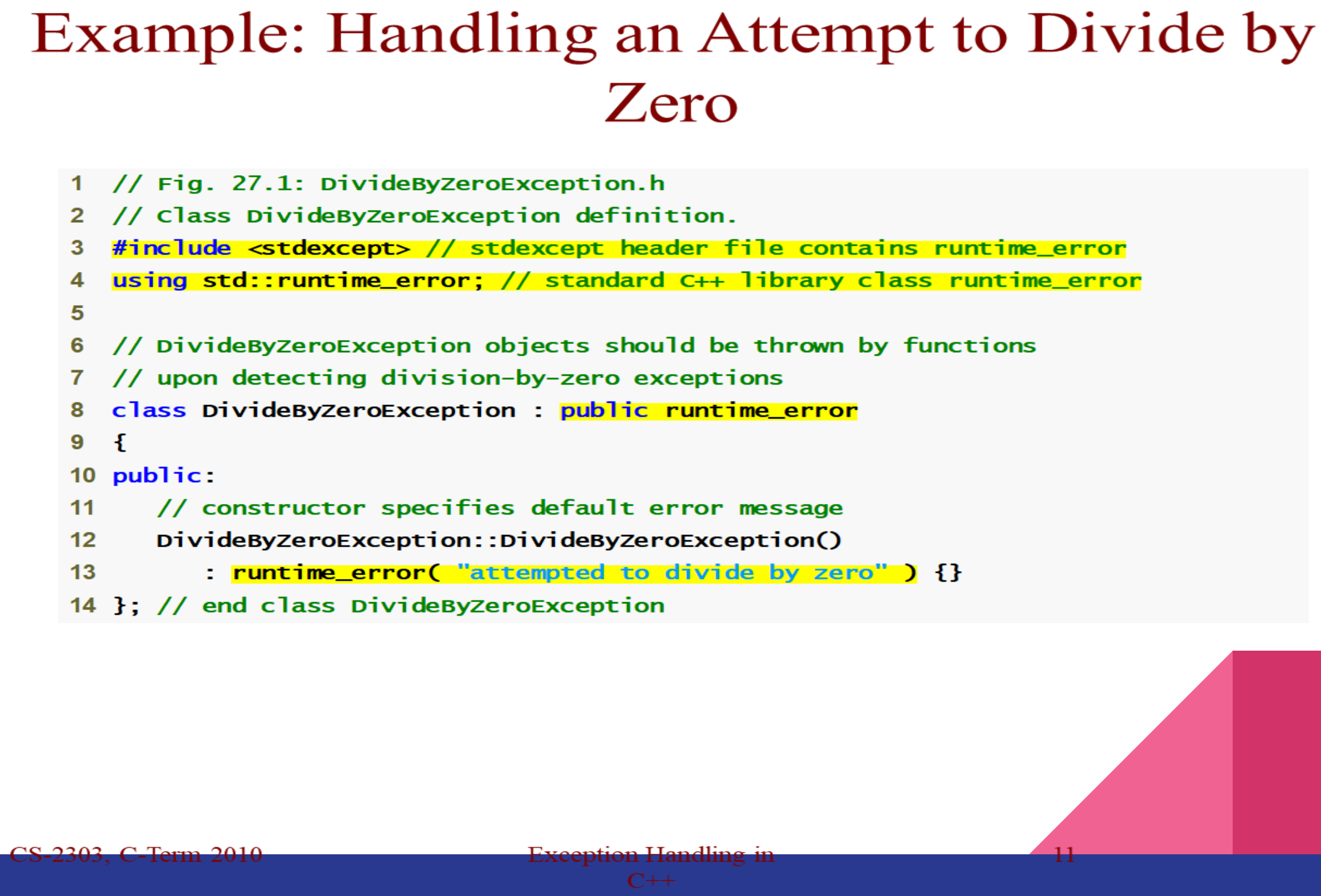
- An *exception* is a class
  - Usually derived from one of the system’s exception base classes
- If an exceptional or error situation occurs, program *throws* an object of that class
  - Object crawls up the call stack

- A calling program can choose to *catch* exceptions of certain classes
  - Take action based on the exception object

Class Exception

- The standard C++ base class for all exceptions
- Provides derived classes with virtual function **`what`**
  - Returns the exception’s stored error message

Example:Handling an Attempt to Divide by Zero

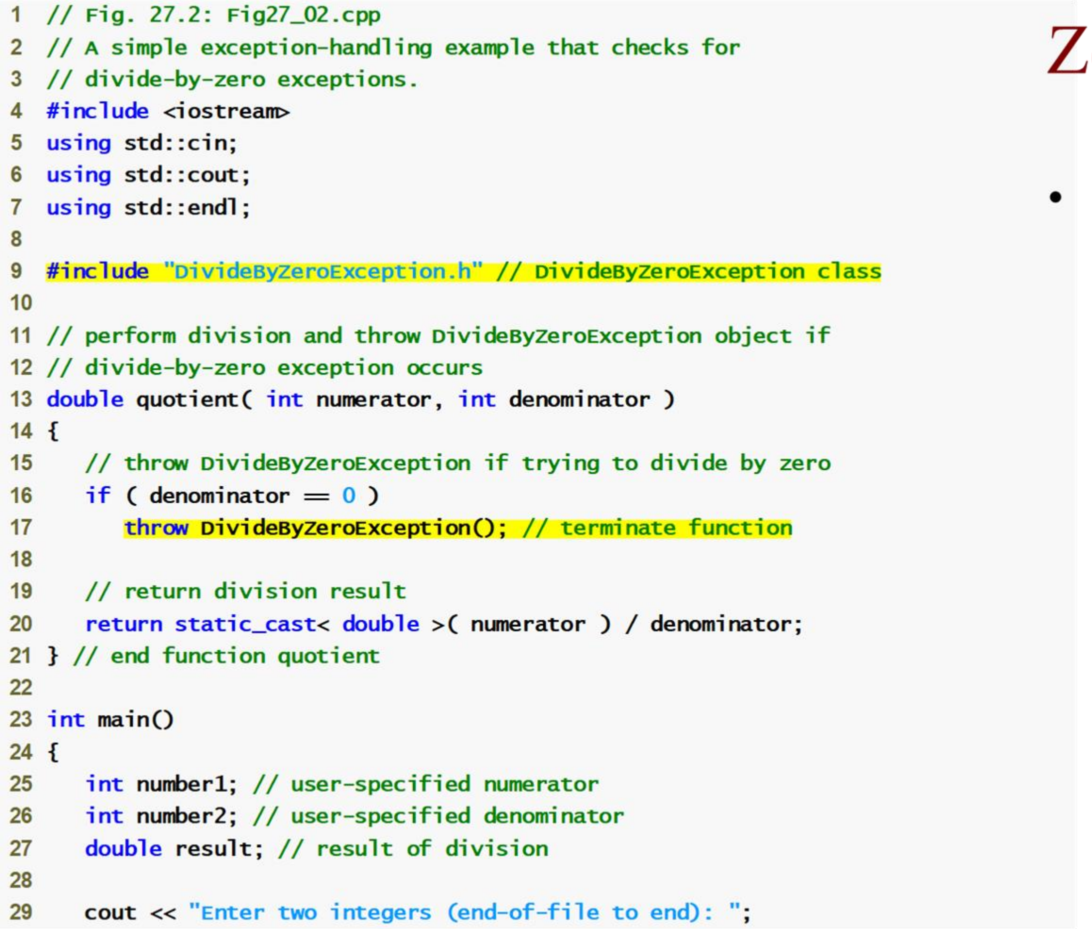


Example: Handling an Attempt to Divide by Zero

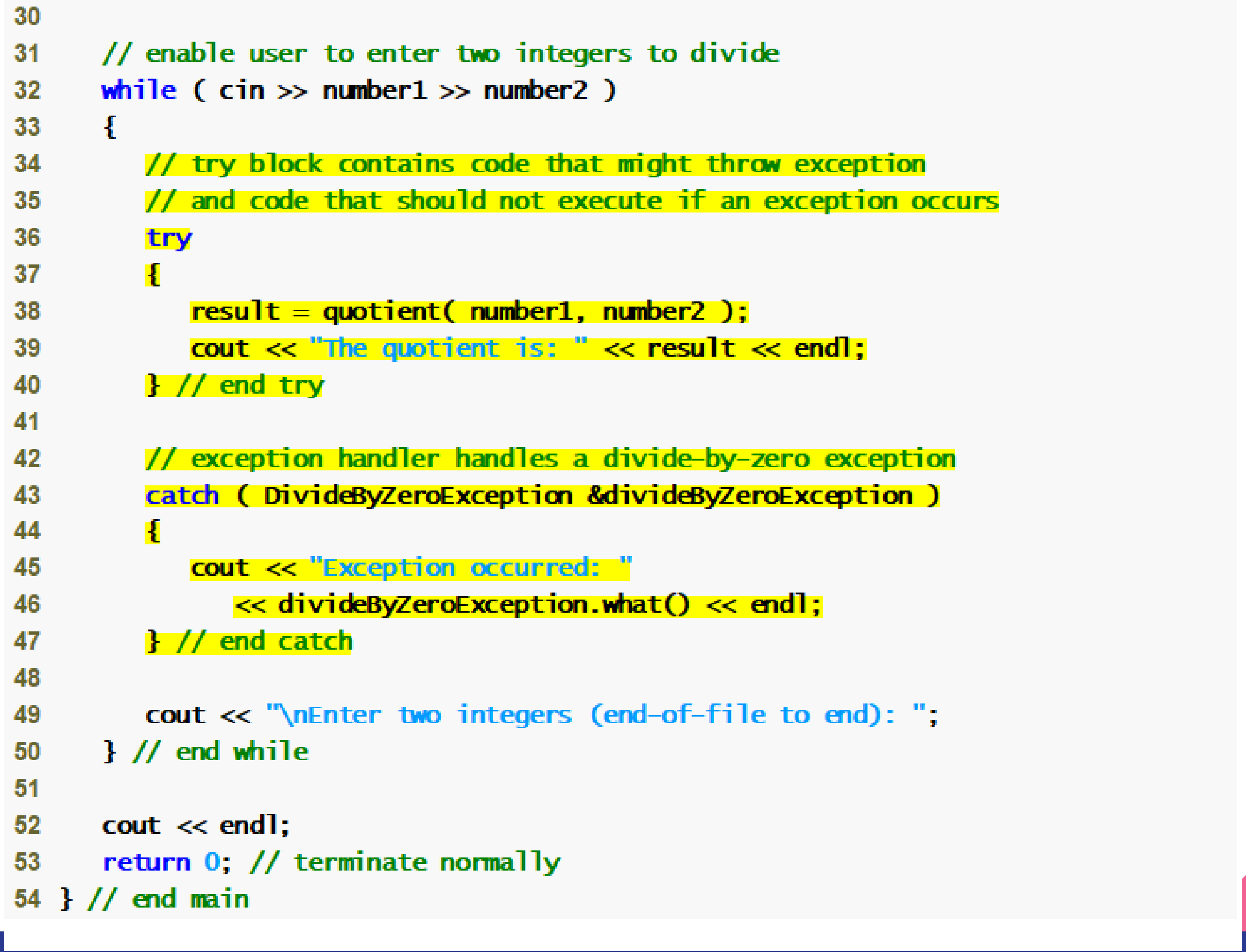
```
// Fig. 27.1: DivideByZeroException.h
// Class DivideByZeroException definition.
#include <stdexcept> // stdexcept header file contains runtime_error
using std::runtime_error; // standard C++ library class runtime_error

// 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
```

Zero Divide Example

- **Fig27-2**
  - (1 of 2)



```cpp
// Fig. 27.2: Fig27_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): ";
```



```cpp

   // 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
```

Try Blocks

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

Software Engineering Observations

- 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.

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

Catch Handlers (continued)

```
try {
 // code to try
}
catch (exceptionClass1 &name1) {
 // handle exceptions of exceptionClass1
}
catch (exceptionClass2 &name2) {
 // handle exceptions of exceptionClass2
}
catch (exceptionClass3 &name3) {
 // handle exceptions of exceptionClass3
}
...
/* code to execute if
  no exception or
   catch handler handled exception*/
```

Common Programming Errors

Syntax error to place code between a **`try`** block and its corresponding **`catch`** handlers

-

Each **`catch`** handler can have only a single parameter

•Specifying a comma-separated list of exception parameters is a syntax error

•Logic error to catch the same type in two different **`catch`** handlers following a single **`try`** block

Fundamental Philosophy (continued)

- Termination model of exception handling
  - `try` block *expires* when an exception occurs
    - Local variables in try block go out of scope
  - Code within the matching catch handler executes
  - Control resumes with the first statement after the last catch handler following the try block
-

Control *does not* return to throw point

- Stack unwinding
  - Occurs if no matching `catch` handler is found
  - Program attempts to locate another enclosing `try` block in the calling function

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
    - Invokes destructors
  - Control returns to point where function was invoked
- Attempts are made to catch the exception in outer **`try`**…**`catch`** blocks
- If the exception is never caught, the function **`terminate`** is called

Observations

- With exception handling, a program can continue executing (rather than terminating) after dealing with a problem.
- This helps to support robust applications that contribute to *mission-critical* computing or *business-critical* computing
- When no exceptions occur, there is no performance penalty

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

Observations

- Catching an exception object by reference eliminates the overhead of copying the object that represents the **`thrown`** exception
-
- Associating each type of runtime error with an appropriately named exception object improves program clarity.

When to use Exception Handling

Don't use for routine stuff such as end-of-file or null string checking

- To process synchronous errors
  - Occur when a statement executes
- Not to process asynchronous errors
  - Occur in parallel with, and independent of, program execution
- To process problems arising in predefined software elements
  - Such as predefined functions and classes
  - Error handling can be performed by the program code to be customized based on the application's needs

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

Executing an empty **`throw`** statement outside a **`catch`** handler causes a function call to terminate

- Abandons exception processing and terminates the program immediately

Constructors and Destructors

- Exceptions and constructors
  - Exceptions enable constructors to report errors
    - Unable to return values
  - 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

Exceptions and Inheritance

- 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
  - Enables **`catching`** related errors with a concise notation

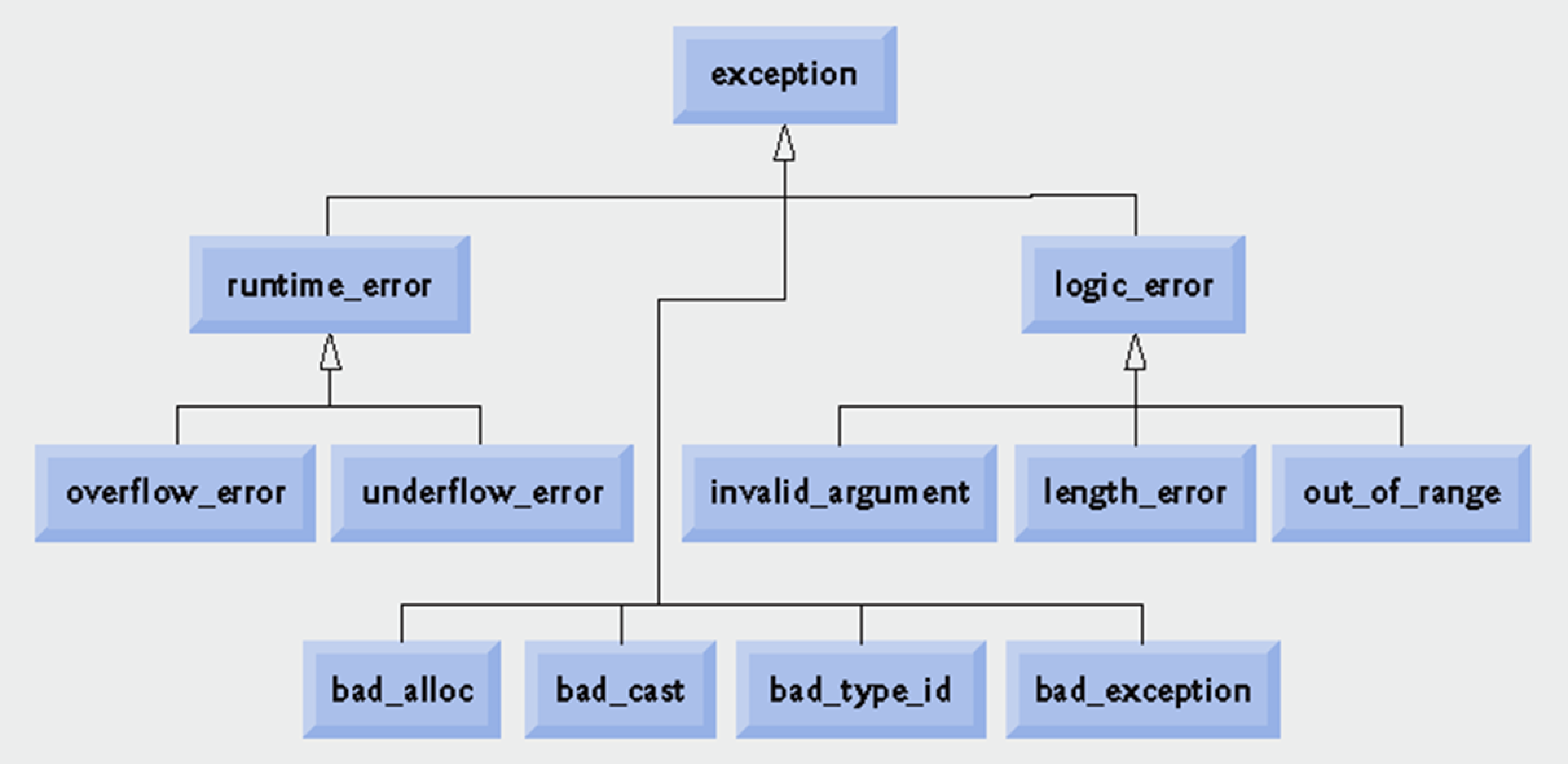
Failures of call to new()

- 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

Standard Library Exception Hierarchy

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

Standard Library Exception



Programming Exercises

1. (Throwing the Result of a Conditional Expression) Throw the result of a conditional expression that returns either a double or an int. Provide an int catch handler and a double catch handler. Show that only the double catch handler executes, regardless of whether the int or the double is returned.

1. (Local Variable Destructors) Write a program illustrating that all destructors for objects constructed in a block are called before an exception is thrown from that block

1. (Member Object Destructors) Write a program illustrating that member object destructors are called for only those member objects that were constructed before an exception occurred.