C++ Exception and Error Handling

Spring 2018

Introduction to Exceptions

- An exception is an indication of a problem that occurs during a program's execution.
- Exception handling enables you to create applications that can resolve (or handle) exceptions.
- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been "thrown."
- An *exception handler* is a section of code that gracefully responds to exceptions.
- The default exception handler prints an error message and terminates the program.

C++ Exception Header Files & Classes

- Header Files:
  - <stdexcept>
  - <exception>

- Classes:
  - runtime_error

- Example:

```
class DivideByZeroException :  public  std::runtime_error
{
    public:
       DivideByZeroException() : std::runtime_error("divide by zero") { }
} ;

...

if (denominator == 0)
  throw  DivideByZeroException() ;
```

How to Handle Exceptions

- To handle an exception, you use a *try* statement.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
```

- *try block*: First the keyword `try` indicates a block of code will be attempted (the curly braces are required).

- A catch clause begins with the key word `catch`:

  ```
  catch (ExceptionType ParameterName)
  ```

  - *`ExceptionType`* is the name of an exception class and
  - *`ParameterName`* is a variable name which will reference the exception object if the code in the try block throws an exception.

Software Engineering Observation 17.3

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.

An example of the Exception Handling

```
try
{
      double result = quotient( num1, num2 );
      cout << "Quotient: " << result << endl;
}
catch (DivideByZeroException &exceptionObj)
{
      cout << "Exception occurs. "  <<
  exceptionObj.what() << endl ;
}
```

Handling Exceptions

- The parameter must be of a type that is compatible with the thrown exception's type.

- A `catch` clause needs to be written for each type of exception that could potentially be thrown.
  - The `catch` clauses must be listed from most specific to most general.

The Stack Trace

- The *call stack* is an internal list of all the methods that are currently executing.
- A *stack trace* is a list of all the methods in the call stack.
- It indicates:
  - the method that was executing when an exception occurred and
  - all of the methods that were called in order to execute that method.

Common Programming Error 17.1

It's a syntax error to place code between a `try` block and its corresponding `catch` handlers or between its `catch` handlers.

Common Programming Error 17.2

Each `catch` handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.

```
try
  {
    Item item2(-88.88);
    cout << item2.toString() << endl ;
  }
  catch(InvalidPrice &err1, runtime_error &err2) // compiling errors
  {
   err.why() ;
  }
```

Common Programming Error 17.3

It's a compilation error to catch the same type in multiple `catch` handlers following a single `try` block.

```
try
 {
  cout << "Amount: " << getAmountWithErrorHandling() << endl ;
  cout << "Read the amount successfully" << endl;
 }
 catch (runtime_error & err1)
 {
  cout << err1.what() << endl;
 }
 catch (runtime_error & err2)  // catch same type twice
 {
  cout << err2.what() << endl;
 }
```

Common Programming Error 17.4

Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.

Error-Prevention Tip 17.2

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what's called mission-critical computing or business-critical computing.

Error-Prevention Tip 17.3

In general, you should throw only objects of exception class types.

Exercise

1. Write a C++ program that reads in an integer

2. Run the program and enter “ABC” instead of a number. What will happen?

3. Use try-catch statement to handle the error and ask the user to enter the number again

Rethrowing an Exception

- A function might use a resource—like a file—and might want to release the resource (i.e., close the file) if an exception occurs.

- An exception handler, upon receiving an exception, can release the resource then notify its caller than an exception occurred by rethrowing the exception via the statement
  - `throw;`

- Regardless of whether a handler can process an exception, the handler can *rethrow* the exception for further processing outside the handler.

- The next enclosing `try` block detects the rethrown exception, which a `catch` handler listed after that enclosing `try` block attempts to handle.

An example of the Exception Handling

```
try
{
      cout << "Hello" << endl;

      throw exception() ;
}
catch (exception &e)
{
      throw ;
}
```

Stack Unwinding

- When an exception is thrown but not caught in a particular scope, the function call stack is "unwound," and an attempt is made to `catch` the exception in the next outer `try`…`catch` block.
- Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables that have completed intitialization in that function are destroyed and control returns to the statement that originally invoked that function.
- If a `try` block encloses that statement, an attempt is made to `catch` the exception.
- If a `try` block does not enclose that statement, stack unwinding occurs again.
- If no `catch` handler ever catches this exception, the program terminates.

When to Use Exception Handling

- Exception handling is designed to process synchronous errors, which occur when a statement executes, such as *out-of-range array subscripts*, *arithmetic overflow* (i.e., a value outside the representable range of values), *division by zero*, *invalid function parameters* and uns*uccessful memory allocation* (due to lack of memory).

- Exception handling is not designed to process errors associated with asynchronous events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

Polymorphic References To Exceptions

- When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause.

- Most exceptions are derived from the `exception` class.

- A `catch` clause that uses a parameter variable of the `exception` type is capable of catching any exception that is derived from the `exception` class.

Constructors, Destructors and Exception Handling

- What happens when an error is detected in a *constructor*?
- For example, how should an object's constructor respond when it receives invalid data?
- Because the constructor *cannot return a value* to indicate an error, we must choose an alternative means of indicating that the object has not been constructed properly.
- One scheme is to return the improperly constructed object and hope that anyone using it would make appropriate tests to determine that it's in an inconsistent state.
- Another scheme is to set some variable outside the constructor.

Constructors, Destructors and Exception Handling (cont.)

- The preferred alternative is to require the constructor to `throw` an exception that contains the error information, thus offering an opportunity for the program to handle the failure.
- Before an exception is thrown by a constructor, destructors are called for any member objects whose constructors have run to completion as part of the object being constructed.
- Destructors are called for every automatic object constructed in a `try` block before the exception is caught.
- Stack unwinding is guaranteed to have been completed at the point that an exception handler begins executing.
- If a destructor invoked as a result of stack unwinding throws an exception, the program terminates.
- This has been linked to various security attacks.

Error-Prevention Tip 17.7

A constructor should throw an exception if a problem occurs while initializing an object. Before doing so, the constructor should release any memory that it dynamically allocated.

Error-Prevention Tip 17.8

Using inheritance with exceptions enables an exception handler to `catch` related errors with concise notation. One approach is to `catch` each type of reference to a derived-class exception object individually, but a more concise approach is to `catch` pointers or references to base-class exception objects instead. Also, catching pointers or references to derived-class exception objects individually is error prone, especially if you forget to test explicitly for one or more of the derived-class reference types.

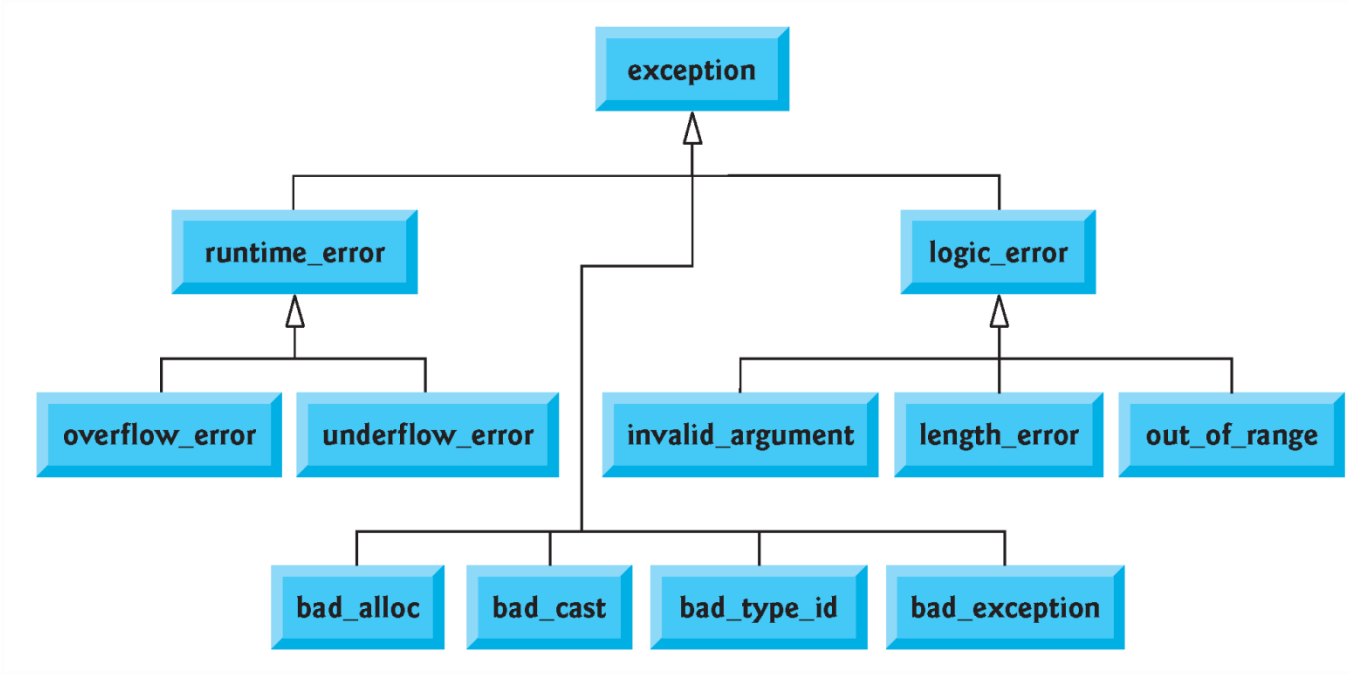

Fig. 17.10 | Some of the Standard Library exception classes.

Common Programming Error 17.6

Placing a `catch` handler that catches a base-class object before a `catch` that catches an object of a class derived from that base class is a logic error. The base-class `catch` catches all objects of classes derived from that base class, so the derived-class `catch` will never execute.

```
try
 {
  cout << "Amount: " << getAmountWithErrorHandling() << endl ;
  cout << "Read the amount successfully" << endl;
 }
 catch (exception & err)  // Logic error
 {
  cout << "A general error has occurred. " << endl;
 }
 catch (runtime_error & err1)
 {
  cout << err1.what() << endl;
 }
```

Standard Library Exception Hierarchy (cont.)

- Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic.
  - For example, class `invalid_argument` indicates that a function received an invalid argument.
  - Proper coding can, of course, prevent invalid arguments from reaching a function.

- Class `length_error` indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object.

- Class `out_of_range` indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Common Programming Error 17.7

Exception classes need not be derived from class `exception`, so catching type `exception` is not guaranteed to `catch` all exceptions a program could encounter.

Error-Prevention Tip 17.9

To `catch` all exceptions potentially thrown in a `try` block, use `catch(...)`. One weakness with catching exceptions in this way is that the type of the caught exception is unknown. Another weakness is that, without a named parameter, there's no way to refer to the exception object inside the exception handler.

Software Engineering Observation 17.8

The standard `exception` hierarchy is a good starting point for creating exceptions. You can build programs that can `throw` standard exceptions, `throw` exceptions derived from the standard exceptions or `throw` your own exceptions not derived from the standard exceptions.

Exercise

1. Define your own error class named “InvalidBalanceException”

2. Use it in the “BankAccount” class to prevent the creation of a “negative balance” BankAccount object

3. Write a simple “MyBank” program that reads an account number and the account’s balance. Create a BankAccount object and save it into a list (e.g. vector)

4. Enter “negative” balance

5. Modify the “MyBank” to be able to keep asking the user for a new balance as long as the user enters the negative balance