Computer Programming
Exception Handling & Design Patterns 22nd Lecture

엄현상 (Eom, Hyeonsang)
컴퓨터공학부
서울대학교

©Copyrights 2011 Eom, Hyeonsang All Rights Reserved
순서

- Exception Handling
- Design Patterns
  - Visitor
- Possible Topics for Final (생략)
- Q&A
Introduction to Exceptions

Exceptions

- Indicate problems that occur during a program’s execution
- Occur infrequently

Exception handling

- Can resolve exceptions
  - Allow a program to continue executing or
  - Notify the user of the problem and
  - Terminate the program in a controlled manner
- Makes programs robust and fault-tolerant
Exception handling provides a standard mechanism for processing errors. This is especially important when working on a project with a large team of programmers.
Exception-Handling Overview

Intermixing program and error-handling logic

- Pseudocode example

  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

Exception-Handling Overview Cont’d

- Exception handling
  - Removes 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 or
    - All exceptions of a group of related types
Performance Tip 1

If the potential problems occur infrequently, intermixing program logic and error-handling logic can degrade a program’s performance, because the program must (potentially frequently) perform tests to determine whether the task executed correctly and the next task can be performed.
Example: Handling an Attempt to Divide by Zero

- **Class runtime_error**
  - Is a standard C++ base class for creating new exception types
  - Provides its derived classes with virtual function **what**
    - Returns the exception’s stored error message
Example: Handling an Attempt to Divide by Zero Cont’d

- **try Blocks**
  - Keyword `try` followed by braces (`{}`).
  - Should enclose
    - Statements that might cause exceptions and
    - Statements that should be skipped in case of an exception.
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.
Example: Handling an Attempt to Divide by Zero Cont’d

- **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
Example: Handling an Attempt to Divide by Zero Cont’d

- catch handlers Cont’d
  - Executes if exception parameter type matches the exception thrown in the try block
    - Could be a base class of the thrown exception’s class
Common Programming Error 1

Each `catch` handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.
Example: Handling an Attempt to Divide by Zero Cont’d

Termination model of exception handling

- **try** block expires when an exception occurs
  - Local variables in **try** block go out of scope
- The 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
Example: Handling an Attempt to Divide by Zero Cont’d

- **Stack unwinding**
  - Occurs if no matching catch handler is found
  - Program attempts to locate another enclosing try block in the calling function
Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.
Example: Handling an Attempt to Divide by Zero Cont’d

- 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
// Fig. 16.1: 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
// Fig. 16.2: Fig16_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): ";
// 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;
cout << endl;
return 0; // terminate normally
} // end main
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
Good Programming Practice 1

Associating each type of runtime error with an appropriately named exception object improves program clarity.
Performance Tip 2

When no exceptions occur, exception-handling code incurs little or no performance penalties. Thus, programs that implement exception handling operate more efficiently than do programs that intermix error-handling code with program logic.
Exception Specifications

Exception specifications (a.k.a. throw lists)

- Keyword `throw`
- Comma-separated list of exception classes in parentheses

Example

```cpp
int someFunction( double value )
    throw ( ExceptionA, ExceptionB, ExceptionC )
```

Indicates `someFunction` can throw exceptions of types `ExceptionA`, `ExceptionB` and `ExceptionC`
Exception Specifications Cont’d

Exception specifications Cont’d

- A function can throw only exceptions of types in its specification or types derived from those types
  - If a function throws a non-specification exception, function unexpected is called
  - This normally terminates the program
- No exception specification indicates the function can throw any exception
- An empty exception specification, throw(), indicates the function can not throw any exceptions
Common Programming Error 3

Throwing an exception that has not been declared in a function’s exception specification causes a call to function unexpected.
Error-Prevention Tip
The compiler will not generate a compilation error if a function contains a throw expression for an exception not listed in the function’s exception specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, carefully check your code to ensure that functions do not throw exceptions not listed in their exception specifications.

Design Patterns

Definition
- Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

Essential Elements
- Pattern name
- Problem
- Solution
- Consequences
  - Results and trade-off of applying the pattern

Design Patterns, E. Gamma, R. Helm, R. Johnson & J. Vlissides, Addison Wesley, '95
Visitor: A Design Pattern

- The operation that gets executed depends on both the type of Visitor and the type of Element it visits.

- Adds an operation to a class without modifying the class.
  - Every class has a virtual method `Accept(Visitor& v)`
  - For every concrete class S that has Accept, the Visitor has a method `VisitS(S* s)`
  - An object of class Visitor is passed to the `Accept` method
  - `Accept` immediately calls `VisitS`, passing the `this` pointer as an argument.
Visitor and ConcreteVisitor

Visitor

- Declares a Visit operation for each class of ConcreteElement in the object structure

ConcreteVisitor

- Implements each operation declared by Visitor
- Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure
- ConcreteVisitor provides the context for the algorithm and stores its local state

Design Patterns, E. Gamma, R. Helm, R. Johnson & J. Vlissides, Addison Wesley, ‘95
Element and ConcreteElement

- Element
  - Defines an Accept operation that takes a visitor as an argument

- ConcreteElement
  - Implements an Accept operation that takes a visitor as an argument

- ObjectStructure
  - Can enumerate its elements
  - May provide a high-level interface to allow the visitor to visit its elements
  - May either be a composite or a collection such as a list or a set
Visitor Class

class Visitor
{
    public:
        virtual void VisitElementA(ElementA*);
        virtual void VisitElementB(ElementB*);
        virtual void VisitCompositeElement(CompositeElement*);
    protected:
        Visitor();
};
ConcreteVisitor Class

class ConcreteVisitor : public Visitor
{
   public:
      ConcreteVisitor();
      virtual void VisitElementA(ElementA*);
      virtual void VisitElementB(ElementB*);
      virtual void VisitCompositeElement(CompositeElement*);
};
Element Class

class Element
{
    public:
        virtual ~Element();
        virtual void Accept(Visitor&) = 0;
    protected:
        Element();
};
Element Class

class ElementA : public Element
{
    public:
        ElementA();
        virtual void Accept(Visitor& v) {
            v.VisitElementA(this);
        }
};

class ElementB : public Element
{
    public:
        ElementB();
        virtual void Accept(Visitor& v) {
            v.VisitElementB(this);
        }
};

Design Patterns, E. Gamma, R. Helm, R. Johnson & J. Vlissides, Addison Wesley,'95
CompositeElement Class

class CompositeElement : public Element
{
    public:
        virtual void Accept(Visitor&);
    private:
        List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v)
{
    ListIterator<Element*> i(_children);
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}

Design Patterns, E. Gamma, R. Helm, R. Johnson & J. Vlissides, Addison Wesley, ‘95
How to Use?

**CompositeElement** e;

Visitor v;

... 

e->Accept(v);

Or

**ConcreteVisitor** cv;

...

e->Accept(cv);
**Consequences**

- **Visitor makes adding new OPs easy**
- **A Visitor gathers related operations and separates unrelated ones**
  - Related behavior is localized in a visitor while unrelated sets are partitioned in subclasses
- **Adding new ConcreteElement classes is hard**
- **Visiting across class hierarchies**
- **Accumulating state**
- **Breaking encapsulation**

*Design Patterns, E. Gamma, R. Helm, R. Johnson & J. Vlissides, Addison Wesley, '95*
Reference for Design Patterns

“Design Patterns: Elements of Reusable Object-Oriented Software,” Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995
Possible Topics for Final

- Unix/Linux
  - C Program Memory Layout
  - File System
- C Compiler and Linker
  - Static vs Shared Library
- Gdb & Make
- Scoping
- Memory Management in C
  - Possible Errors in Dynamic Memory Allocation
- Libraries
  - Standard I/O Library
Possible Topics for Final Cont’d

- Namespaces
- Imperative Language (vs. Declarative Lang.)
- Object-Oriented Programming
  - Design Principles
- Separating Interface from Implementation
- Pointers
- C vs C++
  - Static
Possible Topics for Final Cont’d

- Function Definitions
- Function Prototypes
- C++ Data Types
- Storage Classes and Scope Rules
- C++ Function Call Stack and Activation Records
- References
- Default Arguments
- Function Overloading and Templates
- Recursion
Possible Topics for Final Cont’d

- Constructors and Destructors
  - Copy Constructors
- const Members
- Member Initializer
- friend Functions and Classes
- Static Members
- Inheritance
- Polymorphism
  - Virtual
- Exception Handling
- Design Patterns