Chapter 16

Exception Handling

Adapted from Savitch, “Problem Solving with C++”

Overview

16.1 Exception-Handling Basics

16.2 Programming Techniques for Exception Handling

16.1

Exception-Handling Basics

Exception Handling Basics

- It is often easier to write a program by first assuming that nothing incorrect will happen
- Once it works correctly for the expected cases, add code to handle the exceptional cases
- Exception handling is commonly used to handle error situations
  - Once an error is handled, it is no longer an error

Functions and Exception Handling

- A common use of exception handling:
  - Functions with a special case that is handled in different ways depending on how the function is used
  - If the function is used in different programs, each program may require a different action when the special case occurs

Exception Handling Mechanism

- In C++, exception handling proceeds by:
  - Some library software or your code signals that something unusual has happened
    - This is called throwing an exception
  - At some other place in your program you place the code that deals with the exceptional case
    - This is called handling the exception

A Toy Example

- Exception handling is meant to be used sparingly in situations that are generally not reasonable introductory examples
- For this example:
  - Suppose milk is so important that we almost never run out
  - We still would like our program to handle the situation of running out of milk

The Milk Example (cont.)

- Code to handle the normal situations involving milk, might be:

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts /static_cast<double>(milk);
cout << donuts << " donuts.\n"
     << milk << " glasses of milk.\n"
     << "You have " << dpg
     << " donuts per glass of milk.\n";
```

The No Milk Problem

- If there is no milk, the code on the previous slide results in a division by zero
  - We could add a test case for this situation
  - **Display 16.1** shows the program with the test case

    Display 16.2 (1-2) shows the program rewritten using an exception

Display 16.1

Back Next

Handling a Special Case without Exception Handling

```
#include <iostream>
using namespace std;

int main()
{
    int donuts, milk;
    double dpg;
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;

    if (milk <= 0)
    {
        cout << donuts << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }
    else
    {
        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Sample Dialogue

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

Same Thing Using Exception Handling (*part 1 of 2*)

```
#include <iostream>
using namespace std;

int main()
{
    int donuts, milk;
    double dpg;

    try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;

        if (milk <= 0)
               throw donuts;

        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }
    catch(int e)
    {
        cout << e << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Display 16.2 (2/2)

Same Thing Using Exception Handling (*part 2 of 2*)

Sample Dialogue 1

```
Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.
```

Sample Dialogue 2

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

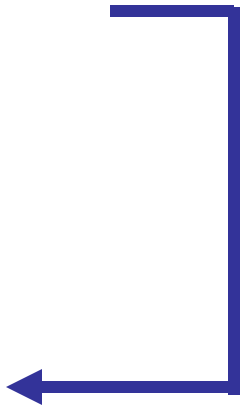
The try Block

- The program of Display 16.2 replaces the test case in the if-else statement with

```
if(milk <= 0)
        throw donuts;
```

- This code is found in the try block

```
try
{
    Some_Code
}
```



which encloses the code to handle the normal situations

The Try Block Outline

- The try block encloses code that you want to "try" but that could cause a problem
- The basic outline of a try block is:

```
try
{
        Code_To_Try
        Possibly_Throw_An_Exception
        More_Code
}
```

The Exception

- To throw an exception, a throw-statement is used to throw a value
  - In the milk example:

    throw donuts;

    throws an integer value.
  - The value thrown is sometimes called an exception
  - You can throw a value of any type

The catch-block

- Something that is thrown goes from one place to another
- In C++ throw causes the flow of control to go to another place
  - When an exception is thrown, the try block stops executing and the catch-block begins execution
  - This is catching or handling the exception

The Milk catch-block

- The catch-block from the milk example looks like, but is not, a function definition with a parameter:

```
catch(int e)
 {
    cout << e << donuts, and no milk!\n"
         << "Go buy some milk.\n";
 }
```

  - If no exception is thrown, the catch-block is ignored during program execution
  - The identifier *e* is called the *catch-block parameter*.

The catch-block Parameter

- The catch-block parameter, (recall that the catch-block is not a function) does two things:
  - The type of the catch-block parameter identifies the kind of value the catch-block can catch
  - The catch-block parameter provides a name for the value caught so you can write code using the value that is caught

try-blocks and if-else

- try-blocks are very similar to if-else statements
  - If everything is normal, the entire try-block is executed
  - else, if an exception is thrown, the catch-block is executed
- A big difference between try-blocks and if-else statements is the try-block's ability to send a message to one of its branches

try-throw-catch Review

- This is the basic mechanism for throwing and catching exceptions
  - The try-block includes a throw-statement
  - If an exception is thrown, the try-block ends and the catch-block is executed
  - If no exception is thrown, then after the try-block is completed, execution continues with the code following the catch-block(s)
  - A catch-block applies only to an immediately preceding try block.

Defining an Exception Class

- Because a throw-statement can throw a value of any type, it is common to define a class whose objects can carry the kind of information you want thrown to the catch-block
- A more important reason for a specialized exception class is so you can have a different type to identify each possible kind of exceptional situation

The Exception Class

- An exception class is just a class that happens to be used as an exception class
- An example of a program with a programmer defined exception class is in **Display 16.3 (1-2)**

DISPLAY 16.3 Defining Your Own Exception Class *(part 1 of 2)*

This is just a toy example to learn C++ syntax. Do not take it as an example of good typical use of exception handling.

```
1  #include <iostream>
2  using namespace std;
3  class NoMilk
4  {
5  public:
6      NoMilk();
7      NoMilk(int how_many);
8      int get_donuts();
9  private:
10     int count;
11 };
12 int main()
13 {
14     int donuts, milk;
15     double dpg;
16     try
17     {
18         cout << "Enter number of donuts:\n";
19         cin >> donuts;
20         cout << "Enter number of glasses of milk:\n";
21         cin >> milk;
22         if (milk <= 0)
23                 throw NoMilk(donuts);
24         dpg = donuts/static_cast<double>(milk);
25         cout << donuts << " donuts.\n"
26              << milk << " glasses of milk.\n"
27              << "You have " << dpg
28              << " donuts for each glass of milk.\n";
29     }
30     catch(NoMilk e)
31     {
32         cout << e.get_donuts() << " donuts, and No Milk!\n"
33              << "Go buy some milk.\n";
34     }
35     cout << "End of program.";
36     return 0;
37 }
38
39 NoMilk::NoMilk()
40 {}
```

(continued)

Display 16.3 (2/2)

Back Next

DISPLAY 16.3 Defining Your Own Exception Class *(part 2 of 2)*

```
41   NoMilk::NoMilk(int how_many) : count(how_many)
42   {}
43
44   int NoMilk::get_donuts()
45   {
46        return count;
47   }
```

The sample dialogues are the same as in Display 16.2.

Throwing a Class Type

- The program in Display 16.3 uses the throw-statement

  throw NoMilk(donuts);

  - This invokes a constructor for the class NoMilk
  - The constructor takes a single argument of type int
  - The NoMilk object is what is thrown
  - The catch-block then uses the statement

    e.get_donuts( )

    to retrieve the number of donuts

Multiple Throws and Catches

- A try-block can throw any number of exceptions of different types
  - In any one execution, only one exception can be thrown
  - Each catch-block can catch only one exception
  - Multiple catch-blocks may be used
    - A parameter is not required in a catch-block
      - List the type with no parameter, e.g.,

        catch (DivideByZero) { }
- A sample program with two catch-blocks is found in

Display 16.4 (1-3)

Display 16.4 (1/3)

Back Next

Catching Multiple Exceptions (*part 1 of 3*)

```
#include <iostream>
#include <string>
using namespace std;

class NegativeNumber
{
public:
    NegativeNumber();
    NegativeNumber(string take_me_to_your_catch_block);
    string get_message();
private:
    string message;
};

class DivideByZero
{};

int main()
{
    int jem_hadar, klingons;
    double portion;

    try
    {
        cout << "Enter number of Jem Hadar warriors:\n";
        cin >> jem_hadar;
        if (jem_hadar < 0)
            throw NegativeNumber("Jem Hadar");

        cout << "How many Klingon warriors do you have?\n";
        cin >> klingons;
        if (klingons < 0)
            throw NegativeNumber("Klingons");
```

Although not done here, exception classes can have their own interface and implementation files and can be put in a namespace.

This is another toy example.

Display 16.4 (2/3)

Catching Multiple Exceptions (*part 2 of 3*)

```
        if (klingons != 0)
            portion = jem_hadar/static_cast<double>(klingons);
        else
            throw DivideByZero();
        cout << "Each Klingon must fight "
             << portion << " Jem Hadar.\n";
    }
    catch(NegativeNumber e)
    {
       cout << "Cannot have a negative number of "
            << e.get_message() << endl;
    }
    catch(DivideByZero)
    {
       cout << "Send for help.\n";
    }

    cout << "End of program.\n";
    return 0;
}

NegativeNumber::NegativeNumber()
{}

NegativeNumber::NegativeNumber(string take_me_to_your_catch_block)
    : message(take_me_to_your_catch_block)
 {}

string NegativeNumber::get_message()
{
    return message;
}
```

Display 16.4 (3/3)

Catching Multiple Exceptions (*part 3 of 3*)

Sample Dialogue 1

```
Enter number of Jem Hadar warriors:
1000
How many Klingon warriors do you have?
500
Each Klingon must fight 2.0 Jem Hadar.
End of program
```

Sample Dialogue 2

```
Enter number of Jem Hadar warriors:
-10
Cannot have a negative number of Jem Hadar
End of program.
```

Sample Dialogue 3

```
Enter number of Jem Hadar warriors:
1000
How many Klingon warriors do you have?
0
Send for help.
End of program.
```

A Default catch-block

- When catching multiple exceptions, write the catch-blocks for the most specific exceptions first
  - Catch-blocks are tried in order and the first one matching the type of exception is executed
- A default (and last) catch-block to catch any exception can be made using "…" as the catch-block parameter

```
catch(…)
  {   <the catch block code> }
```

Exception Class DivideByZero

- In Display 16.4, exception class DivideByZero was defined as

```
class DivideByZero
  { } ;
```

  - This class has no member variables or member functions
  - This is a trivial exception class
  - DivideByZero is used simply to activate the appropriate catch-block
  - There is nothing to do with the catch-block parameter so it can be omitted as shown in Display 16.4

Exceptions In Functions

- In some cases, an exception generated in a function is not handled in the function
  - It might be that some programs should end, while others might do something else, so within the function you might not know how to handle the exception
- In this case, the program places the function invocation in a try block and catches the exception in a following catch-block

Function safe_divide

- The program of Display 16.5 includes a function that throws, but does not catch an exception
  - In function safe_divide, the denominator is checked to be sure it is not zero. If it is zero, an exception is thrown:

```
if (bottom == 0)
    throw DivideByZero( );
```

  - The call to function safe_divide is found in the try-block of the program

Exception Specification

- If a function does not catch an exception it should warn programmers that an exception might be thrown by the function
  - An exception specification, also called a throw list, appears in the function declaration and definition:
    double safe_divide(int n, int d) throw (DivideByZero);
  - if multiple exceptions are thrown and not caught by the function:
    double safe_divide(int n, int d)
    throw (DivideByZero, OtherException);

Display 16.5 (1-2)

Throwing an Exception inside a Function (*part 1 of 2*)

```
#include <iostream>
#include <cstdlib>
using namespace std;

class DivideByZero
{};

double safe_divide(int top, int bottom) throw (DivideByZero);

int main()
{
    int numerator;
    int denominator;
    double quotient;
    cout << "Enter numerator:\n";
    cin >> numerator;
    cout << "Enter denominator:\n";
    cin >> denominator;

    try
    {
       quotient = safe_divide(numerator, denominator);
    }
    catch(DivideByZero)
    {
         cout << "Error: Division by zero!\n"
              << "Program aborting.\n";
         exit(0);
    }

    cout << numerator << "/" << denominator
         << " = " << quotient << endl;

    cout << "End of program.\n";
    return 0;
}
```

Display 16.5 (1/2)

Back Next

Display 16.5 (2/2)

Throwing an Exception inside a Function (*part 2 of 2*)

```
double safe_divide(int top, int bottom) throw (DivideByZero)
{
    if (bottom == 0)
        throw DivideByZero();

    return top/static_cast<double>(bottom);
}
```

Sample Dialogue 1

```
Enter numerator:
5
Enter denominator:
10
5/10 = 0.5
End of Program.
```

Sample Dialogue 2

```
Enter numerator:
5
Enter denominator:
0
Error: Division by zero!
Program aborting.
```

Exceptions Not Listed

- If an exception is not listed in an exception specification and not caught by the function:
  - The program ends
- If there is no exception specification at all, it is the same as if all possible exceptions are listed
  - These exceptions will be treated "normally"
- An empty exception specification list means that no exceptions should be thrown and not caught

Sample Exception Specifications

- void some_function( ) throw(DivideByZero,
  OtherException);
  //Exceptions DivideByZero and OtherException
  //treated normally. All others end the program

- void some_function ( ) throw ( );
  //empty exception list; so all exceptions not
  // caught by the function end the program

- void some_function( );
  // All exceptions of all types treated normally
  // If not caught by a catch-block, the program ends

Derived Classes and Exceptions

- Remember that an object of a derived class is also an object of the base class
  - If D is a derived class of B and B is in an exception specification
    - A thrown object of class D will be treated normally since it is an object of class B

Type Conversion

- No automatic type conversions are done with exceptions
  - if double is in the exception specification, an int cannot be thrown unless int is also in the exception specification

Function Redefinitions in Derived Classes

- Functions redefined or overloaded in derived classes should have the same exception specification as in the base class
  - The exception specification can be a subset of the exception specification in the base class
    - You cannot add exceptions

Section 16.1 Conclusion

- Can you
  - List the three components of exception handling?
  - Write code to catch an exception of type char?
  - Create an exception specification for a function?
  - Create an exception class?

16.2

Programming Techniques for Exception-Handling

Programming Techniques for Exception Handling

- A guideline for exception handling is to separate throwing an exception and catching an exception into separate functions
  - Place the throw-statement in one function and list the exception in the exception specification
  - Place the function invocation and catch-clause in a try-block of a different function

try and throw…Again

- Here is a general example of in using throw:

```
void functionA( ) throw (MyException)
 {
    …
    throw MyException(<an argument?>);
 }
```

catch…again

- Using FunctionA from the previous slide, here is how to catch MyException:

```
 void functionB( )
{
        …
    try
    {
        …
        functionA( );
        …
    }
    catch(MyException e)
    {
        < handle the exception>
    }
}
```

When to Throw An Exception

- Throwing exceptions is generally reserved for those cases when handling the exceptional case depends on how and where the function was invoked
  - In these cases it is usually best to let the programmer calling the function handle the exception
  - An uncaught exception ends your program
- If you can easily write code to handle the problem do not throw an exception

Overuse of Exceptions

- Throwing an exception allows you to transfer flow of control to almost any place in your program
- Such un-restricted flow of control is generally considered poor programming style as it makes programs difficult to understand
- Exceptions should be used sparingly and only when you cannot come up with an alternative that produces reasonable code

Exception Class Hierarchies

- It can be useful to define a hierarchy of exception classes.
  - You might have an ArithmeticError exception class
    with DivideByZeroError as a derived class
  - Since a DivideByZeroError object is also an ArithmeticError object, every catch-block for an ArithmeticError will also catch a DivideByZeroError

Checking For Available Memory

- The new operator allocates memory from the freestore: NodePtr pointer = new Node;
  - What if there is no memory available?
  - bad_alloc is a predefined exception and can be used in this way since new throws a bad_alloc exception:

```
try
{
    NodePtr pointer = new Node;
}

catch(bad_alloc)
{
    cout << "Ran out of memory!";
}
```

Rethrowing an Exception

- The code within a catch-block can throw an exception
  - This feature can be used to pass the same or a different exception up the chain of exception handling blocks

Nested try-catch Blocks

- Although a try-block followed by its catch-block can be nested inside another try-block
  - It is almost always better to place the nested try-block and its catch-block inside a function definition, then invoke the function in the outer try-block
- An error thrown but not caught in the inner try-catch-blocks is thrown to the outer try-block where it might be caught

Section 16.2 Conclusion

- Can you
  - Describe what happens if an exception is never caught?
  - Write code that nests a try-block inside another try-block?
  - Describe the type of situations in which exceptions should not be thrown?

Chapter 16 -- End