Pearson Problem Solving with C++ TENTH EDITION Savitch GLOBAL EDITION

Unit 10 (Ch 16)

Exception Handling

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat’l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *10.1 Exception-Handling Basics*
- 10.2 Programming Techniques for Exception Handling

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 take care of exceptional cases
  - This is called **exception handling**
  - Once an error is handled, it is no longer an error …
- C++ provides exception-handling facilities
  - Separate normal code from exception handling code
    - Better maintainability
  - Separate exception detection and exception handling
    - Different programs can handle an exception in different ways

Exception Handling Mechanism

- First, some library software or your code signals that something unusual has happened
  - This is called throwing an exception
- The code that deals with the exceptional case is placed at some other place in your program
  - This is called handling the exception
  - Can have different actions at different program
- Exception handling should be used sparingly
  - Not in the normal program flow
- Difficult to teach with large examples
  - Use simple toy example that would not normally use exception handling

A Toy Example

- Suppose people rarely run out of milk

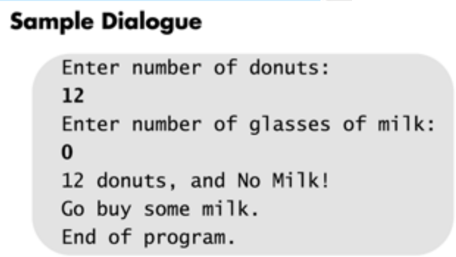
```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts /static_cast<double>(milk);
cout  << donuts << " donuts.\n"
        << milk << " glasses of milk.\n"
        << "You have " << dpg
        << " donuts per glass of milk.\n";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

- Our program still has to handle the situation of running out of milk
  - If there is no milk, this code results in a division by zero

Using if-else for No Milk Problem

- Program should accommodate unlikely situation of running out of milk
  - We could add a test case for this situation
- Traditionally, it is solved by using a simple if-else structure

```
if (milk <= 0) {
    cout << donuts << " donuts, and No Milk!\n";
    cout << "Go buy some milk.\n";
}
else
{  /* regular flow here */  }
```



Sample Dialogue

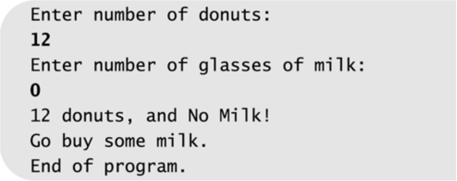
```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

Using Exception Handling

```
#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;   // jump to catch block

      // continue regular flow, no else is required
      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";
   }  // end try
   catch(int e)
   {
      cout << e << " donuts, and No Milk!\n"
           << "Go buy some milk.\n";
   }

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

donuts

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.
```

Try-Block and Catch-Block

- Try-block:
  - Enclose code that want to “try”, but it may cause a problem
  - Same code from ordinary version, except simple if statement

```
Try {
   ……
   if (milk <= 0)
       throw donuts;
   …… }
```

→ do something exceptional

- Catch-block:
  - Provide the “something exceptional” in this block → code for exception handling
- Provide separation of normal from exceptional
  - No big deal for this simple toy example, but very important for large complicated software system

Throw an Exception

- When something unusual happens, a throw-statement is used to throw a value
  - In this milk example:

```
try {
    /* normal code */
    if (exception happened)
        throw donuts;   // throw an integer value
    /* more code */
 }
```

- Keyword throw followed by an exception object
  - Called "throwing an exception"
  - You can throw an object or value of any type

The Catch-Block

- In C++, flow of control goes from try-block to catch-block after throwing an exception
  - The try-block stops executing and the catch-block begins execution
  - If no exception is thrown, the catch-block is ignored during program execution
- Executing the catch-block is called "catching the exception" or "handling the exception"
- The catch-block is called "exception handler"

```
catch (int e)
{
    cout << e << ......
    /* more code */
}
```

← The thrown exception object become its input. Its type identifies the kind of value can catch.

Try Blocks and if-else

- 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, execution skips the catch-blocks after the try-block is completed
- This mechanism looks similar to if-else statement
- A big difference between them:
  - The try-block is able to send a message, i.e. parameter, to one of its branches

Throwing a Class for Exception

- Since a throw-statement can throw a value of any type, why not throwing a object?
  - A class object can carry more information you want while being thrown to the catch-block
- Class objects are able to handle different types
  - Identify each possible kind of exceptional situation
  - A more important reason for a specialized exception class
- An exception class is a class that is used when an exception occurs

```
class NoMilk
{
 public:
    NoMilk();
    NoMilk(int howMany);
    int getDonuts();
 private:
    int count;
};
```

Example for Exception Class

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

class NoMilk
{ /* as defined in previous slide */ };

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 NoMilk(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(NoMilk e)
  {
    cout << e.getDonuts()
         << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
  }
  cout << "End of program.";
  return 0;
}

NoMilk::NoMilk() { }
NoMilk::NoMilk(int howMany) : count(howMany) { }
int NoMilk::getDonuts()
{  return count;  }
```

What Happen in Throwing an Object?

- The program in previous slide 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
  - Only one exception can be thrown at a time
  - Each catch-block can catch only one exception
  - Multiple catch-blocks may be used
- Catch-blocks are tried in order. The first one matching the type of exception is executed
  - When catching multiple exceptions, write the catch-blocks for the most specific exceptions first
- It is suggested to add a default (and last) catch-block to catch any exception
  - Use "…" as the catch-block parameter
    → catch (…) { /* the catch block code */ }

Code for Multiple Throw (1/2)

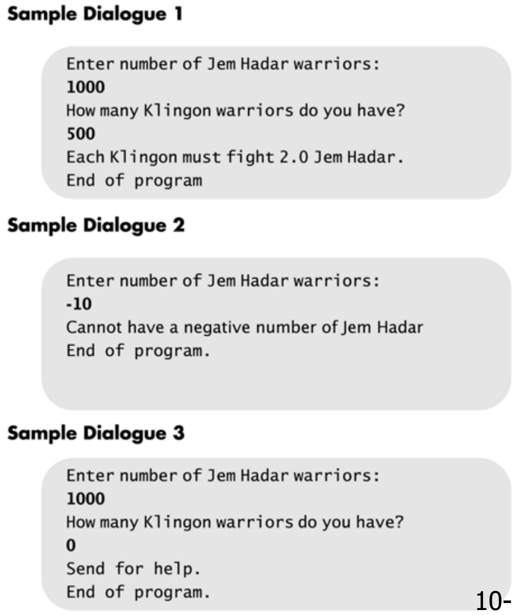
```
#include <iostream>
#include <string>
using namespace std;

class NegativeNumber
{
 public:
    NegativeNumber();
    NegativeNumber(string takeMe);
    string getMessage();
 private:
    string message;
};

class DivideByZero { /* nothing */ };

int main()
{
    int jemHadar, klingons;
    double portion;
```

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

   cout << "How many Klingon warriors do you have?\n";
   cin >> klingons;
   if (klingons < 0)
       throw NegativeNumber("Klingons");
   if (klingons != 0)
       portion = jemHadar/static_cast<double>(klingons);
   else
       throw DivideByZero();  // no parameter is passed

   cout << "Each Klingon must fight "
        << portion << " Jem Hadar.\n";
}
```

Code for Multiple Throw (2/2)

```
  catch(NegativeNumber e)
  {
    cout << "Cannot have a negative number of "
         << e.getMessage() << endl;
  }
  catch(DivideByZero)
  {
    cout << "Send for help.\n";
  }

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

NegativeNumber::NegativeNumber() { }

NegativeNumber::NegativeNumber(string takeMe)
                          : message(takeMe) { }
```

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



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.
```

Exception Class w/o Parameter

- In this example, exception class DivideByZero was defined as *class DivideByZero { };* =)
  - Has no member variables or member functions
  - Provide you a way for just throwing "nothing" when exception occurs
- DivideByZero is called a trivial exception class
  - Simply used to activate the appropriate catch-block
  - There is nothing to do with the catch-block parameter
  - Sometimes it can be omitted → think carefully

throw DivideByzero()
導向 parameter
是 DivideByzero
的 catch block

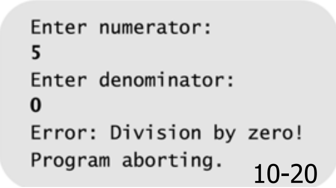
Handle Exceptions Elsewhere

- In some cases, an exception generated in a function is not handled in the same function
  - Try and catch can be located in different functions
- When exception occurs, some programs should end, while others might do something else
  - Might not know how to handle the exception at that time
  - Handle the exception in a following catch-block after the function call
- In the following example, we assume the bottom is not zero in function safeDivide()
  - If an exception is thrown, no catch is found in the function
  - Handle the exception in main() after the function call

Throw an Exception inside Function

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

class DivideByZero { };
double safeDivide(int top, int bottom)
                    throw (DivideByZero);

int main()
{
   int numerator, denominator;
   double quotient;
   cout << "Enter numerator:\n";
   cin >> numerator;
   cout << "Enter denominator:\n";
   cin >> denominator;
   try {
      quotient = safeDivide(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;
}

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

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



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

Exception Specification

- If a function does not catch an exception in it, explicitly list the exceptions that might be thrown out
  - An **exception specification**, also called a throw list, appears in the function declaration and definition

    ex: double safeDivide(int n, int d) throw (DivideByZero);
- Here are some examples:
  - void someFunction( ) throw (DivideByZero, OtherException);
    - Multiple exceptions are allowed to be thrown
  - void someFunction ( ) throw ( ); // cannot throw exceptions
    - Empty exception list. All exceptions in it terminate the program
  - void someFunction( ); // can throw any exceptions
    - All exceptions of all types treated "normally"
    - The same as if all possible exceptions are listed

Uncaught Exceptions

- If an exception is thrown but not caught
  - std::terminate() is called → abort the program
- If an exception is not listed in an exception specification and not caught by the function
  - std::unexpected() is called → abort the program
- Both situations ends the program abnormally
  - Should be avoided at all
- Exception specification is used in old C++ only (i.e., before C++11)
  - It has been deprecated in C++11 and even removed in C++17

Derived Classes and Exceptions

- If D is a derived class of B, but only B is in an exception specification
  - Although D is not in the throw list, a thrown object of class D will be treated normally
  - An object of a derived class is also an object of the base class → D object can be treated as B object
- Functions redefined or overloaded in derived classes should have the same throw list
  - The exception specification can be a subset of the exception specification in the base class
    - You cannot add exceptions, but you can delete some

Overview

- 10.1 Exception-Handling Basics
- *10.2 Programming Techniques for Exception Handling*

Throw and Catch in Separate Function

- A general guideline for exception handling:
  - 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

```
void functionA( ) throw (MyException)
{
    …
    throw MyException(<an argument?>);
    // no catch in this function
        …
}
```

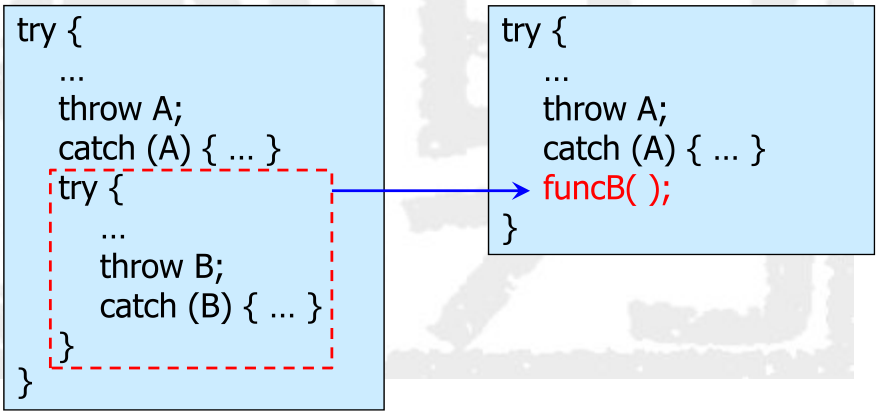
```
void functionB( ) {
    …
    try {
        …
        functionA( );
        …
    }
    catch(MyException e) {
        < handle the exception>
    }
}
```

When to Throw An Exception

- Exceptions should be used sparingly
  - Only when you cannot come up with an alternative way
- Such unrestricted flow of control is often considered as poor programming style
  - Allow you to jump to almost any place in your program
  - It makes programs difficult to understand
- Used for those cases when handling the exceptional case depends on where the function was invoked
  - Let programmer call correct function to handle the exception
  - An uncaught exception ends your program
- If you can easily write code to handle the problem, i.e. if-else, do not throw an exception

Nested try-catch Blocks

- A try-block followed by its catch-block can be nested inside another try-block
  - Better to place the inner try-catch-blocks inside a function definition, then invoke it in the outer try-block
- An error that is not caught in the inner try-catch-blocks might be caught in the outer try-block



```
try {
    …
    throw A;
    catch (A) { … }
    try {
        …
        throw B;
        catch (B) { … }
    }
}
```

```
try {
    …
    throw A;
    catch (A) { … }
    funcB( );
}
```

Exception Class Hierarchies

- It's useful to define a hierarchy of exception classes
  - Ex: MathError is base class. Overflow and ZeroDivide are derived classes
  - Every catch-block for a MathError will also catch a ZeroDivide exception
    - A ZeroDivide object is also a MathError object
  - If the exception has been caught in a catch-block, it cannot be caught again
    - Overflow cannot be caught by MathError in this example

```
class MathError { };
class Overflow : public MathError { };
class ZeroDivide : public MathError { };

void f() {
    try {
        /* throw exceptions */
    }
    catch (Overflow) {
        // handle Overflow
    }
    catch (MathError) {
        // handle any MathError that is
        // NOT Overflow, e.g., ZeroDivide
    }
}
```

Rethrowing an Exception

- If an exception handler cannot completely handle the error, you can throw it again !!
  - Pass the same or a different exception up the chain of exception handling blocks
  - Just do what can be done locally at each catch

```
class ExceptionB { };
class ExceptionD : public ExceptionB { };

void h() {
    try { throw ExceptionD(); }
    catch (ExceptionB) { cerr << "h's catch\n"; throw; }  // rethrow
}

void g() {
    try { h(); }
    catch (ExceptionD) { cerr << "g's catch\n"; }  // still caught here
}
```

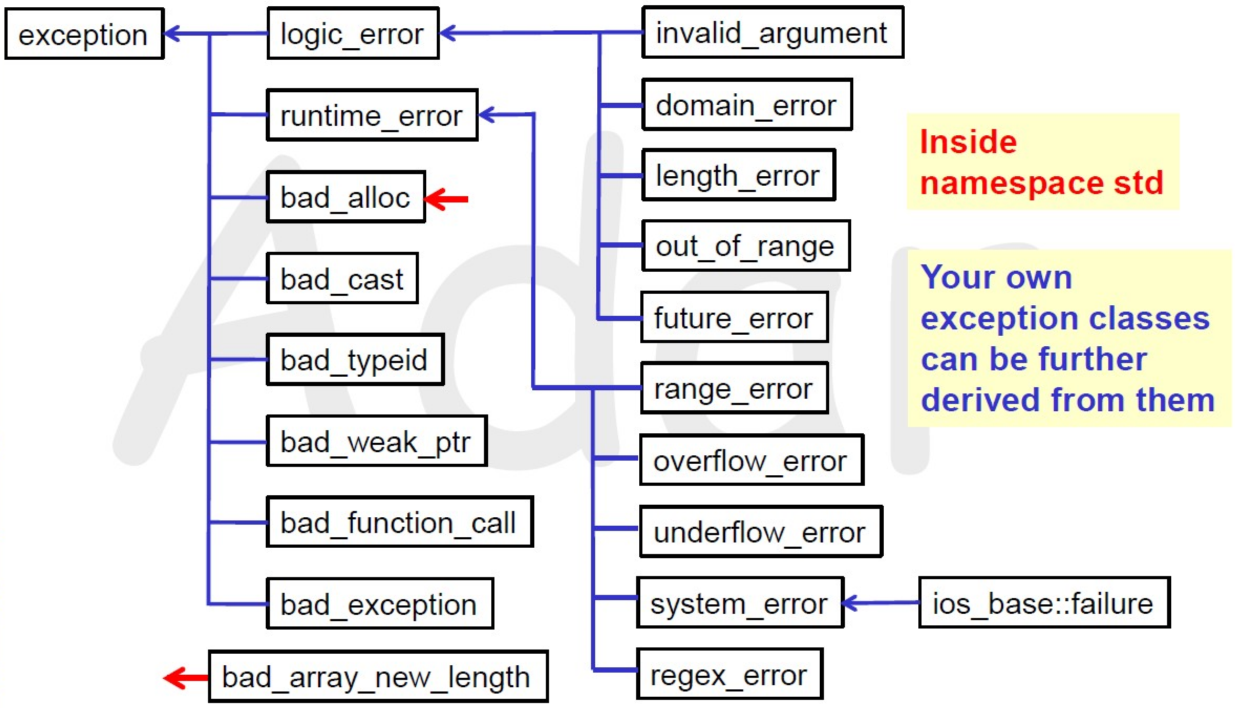
no parameter when rethrowing the same exception

Memory Allocation Error

- The new operator allocates memory for dynamic variables, ex: *NodePtr pointer = new Node;*
- What if there is no memory available?
  - Throw **std::bad_alloc** exception if allocation fails
  - Ex:

```
try
{
    NodePtr pointer = new Node;
}
catch(bad_alloc)
{
    cout << "Ran out of memory!";
    /* can do other things here as well …
}
```

Standard Exception Hierarchy (in C++11)



exception

logic_error

runtime_error

bad_alloc

bad_cast

bad_typeid

bad_weak_ptr

bad_function_call

bad_exception

bad_array_new_length

invalid_argument

domain_error

length_error

out_of_range

future_error

range_error

overflow_error

underflow_error

system_error

regex_error

ios_base::failure

Inside namespace std

Your own exception classes can be further derived from them

Courtesy: Prof. Jiunn-Dar Huang @ NYCUEE