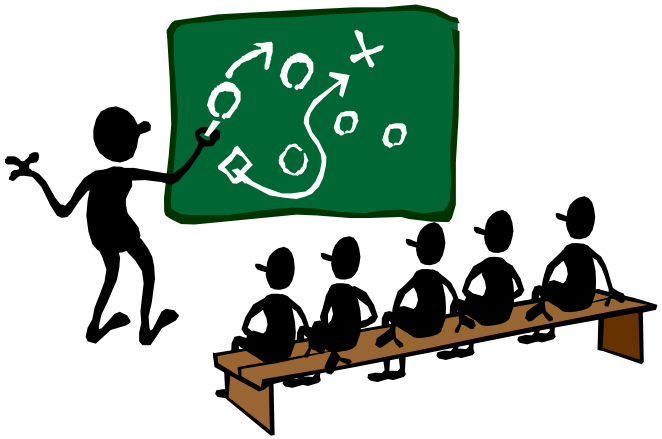
C++ Programming Language
Chapter 18 Exception Handling



Juinn-Dar Huang

Associate Professor

jdhuang@mail.nctu.edu.tw

June 2011

Learning Objectives

- The idea of exception handling
- Exception handling basics
- Try-throw-catch mechanism
- Exception, class hierarchy, and polymorphism
- Exception specification

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Introduction

- Typical approach for program development
  - write programs assuming things go well as planned
  - get regular things work
  - then take care of exceptional cases
- 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 Basics

- Meant to be used sparingly
- Difficult to teach with large examples
- Use simple toy example that would not normally use exception-handling
  - for teaching only

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Toy Example

- Imagine: people rarely run out of milk:

```
int donuts, milk; double dpg;
cout << "Enter number of donuts:";
cin >> donuts;
cout << "Enter number of glasses of milk:";
cin >> milk
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";
```

- Basic code assumes never running out of milk

Using if-else for Exception Handling

- Notice: If no milk ➔ divide-by-zero error!
- Program should accommodate unlikely situation of running out of milk
- Traditionally, use simple if-else structure

```
if (milk <= 0)
    cout << "Go buy some milk!\n";
else
{   /* regular flow here */ }
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Using Exception Handling

```
 9      try
10      {
11          cout << "Enter number of donuts:\n";
12          cin >> donuts;
13          cout << "Enter number of glasses of milk:\n";
14          cin >> milk;
15
16          if (milk <= 0)
17                  throw donuts;
18          dpg = donuts/static_cast<double>(milk);
19          cout << donuts << " donuts.\n"
20               << milk << " glasses of milk.\n"
21               << "You have " << dpg
22               << " donuts for each glass of milk.\n";
23      }
24      catch(int e)
25      {
26          cout << e << " donuts, and No Milk!\n"
27               << "Go buy some milk.\n";
28      }
```

exception detection & reporting

exception handling

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Discussion

- Code between in try block
  - Same code from ordinary version, except simpler if statement:
    ```
    if (milk <= 0)
       throw donuts;
    ```
  - much cleaner code
  - If “milk <= 0” ➔ do something exceptional
- The "something exceptional" is provided in catch block

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Try Blocks and Catch Blocks

- Try block
  - code that may throw exceptions
- Catch 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

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Try Blocks

- Basic method of exception-handling is try-throw-catch

- Try block

```
try
{
    Some_Code;
}
```

  - code for basic algorithm when all goes smoothly

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Throw

- Inside try-blocks, when something unusual happens:

```
try
{
        Normal_Code
        if (exception_happened)
                throw exception_object;
        More_Code
}
```

  - keyword **throw** followed by an exception object
  - called "throwing an exception"

Catch Blocks

- In C++, flow of control goes from try-block to catch-block after throwing an exception
  - try-block is exited and control passes to catch-block
- Executing catch-block called “catching the exception”
- Catch-block is called exception handler

Another Example (1/3)

```
struct Range_error {
    int d;
    Range_error(int dd) :d(dd) { }
};

char num_to_hexchar(int num) {
    if(num < 0 || num > 15)
        throw Range_error(num);
    if(num < 10)
        return '0' + num;
    else
        return 'A' + num - 10;
}
```

Know the control flow with and without exception

Another Example (2/3)

```
void g(int num) {
    // do something here
    try {
        char c = num_to_hexchar(num);
        // do something here
    }
    catch(Range_error)   {  // note: just type; without named object
        cerr << “Oops! There is a range error!” << endl;
    }
    // do something here
}
```

Know the control flow with and without exception

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Another Example (3/3)

```
void h(int num) {
    // do something here
    try {
        char c = num_to_hexchar(num);
        // do something here
    }
    catch(Range_error x)  {  // note: with named object
        cerr << “Range error!  Pass ” << x.d
            << “ to function num_to_hexchar.” << endl;
    }
    // do something here
}
```

Multiple Exceptions

- A try-block can throw multiple exceptions

```
class Exception_A { };
class Exception_B { };

void f() {
    // …
    try {
        // …
        if (cond_1) throw Exception_A();
        if (cond_2) throw Exception_B();
    }
    catch(Exception_A) {   // …   }
    catch(Exception_B) {   // …   }
    // …
}
```

Exception and Class Hierarchy

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

void f() {
    try {
        // …
    }
    catch(Overflow) {
        // handle Overflow or anything derived from Overflow
    }
    catch(MathError) {
        // handle any MathError that is Not Overflow
    }
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Exception and Polymorphism (1/2)

```
class MathError {
    // …
    virtual void debug_print() const { cerr << “Math Error\n”; }
};

class IntOverflow : public MathError {
    // …
    void debug_print() const { cerr << “Integer Overflow\n”; }
};

void f() {
    try {
        g();        // throw an IntOverflow exception
    }
    catch(const MathError m) {      // IntOverflow exception caught here
        m.debug_print();            // however, use MathError::debug_print()
    }                               // instead of IntOverflow::debug_print()
}
```

Exception and Polymorphism (2/2)

```
void f() {
    try {
        g();        // throw an IntOverflow exception
    }
    catch(const MathError& m) {     // IntOverflow exception caught here
        m.debug_print();            // use IntOverflow::debug_print() now
    }                               // because of polymorphism
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Catching Exceptions

```
void f() {
    try {  throw E();          /* E is a class with default ctor  */    }
    catch(H) {
        // when do we get here?
    }
}
```

- Rules
  - [1] if H is the same type as E
  - [2] if H is a public base of E
  - [3] if H and E are pointer types, and [1] or [2] holds for the types to which they refer
  - [4] if H is a reference, and [1] or [2] holds for the type to which H refers
- const prefix can be added to ensure the caught exception object will not be modified
  - very similar to function parameter

Re-Throw (1/2)

- If an exception handler cannot completely handle the error
  - it typically does what can be done locally then
  - throws the exception again ➔ re-throw!

```
void h() {
    try {  /* code that might throw Math Errors */ }
    catch(MathError) {
            if(can_handle_it_completely) {
                    // handle the error
                    return;
            } else {
                    // do what can be done here
                    throw;   // re-throw the same exception
            }
    }
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Re-Throw (2/2)

```
class MyExceptionB { };
class MyExceptionD : public MyExceptionB { };

void h() {
    try {   throw MyExceptionD();  }
    catch(MyExceptionB) {  cerr<< "h's catch\n";   throw;  } // re-throw
}

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


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Catch Every Exception

```
void f() {
    try {
            // do something
    }
    catch(...) {     // catch all exceptions here
            // cleanup for f()
            throw;   // re-throw
    }
}
```

- A technique for local cleanup and maintain invariants

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Order of Catch-Blocks

- Catch-blocks are evaluated **in order**
- A derived exception can be caught by handlers for more than one exception type
- That is, the order of catch-blocks are significant

```
class ExceptionB { };
class ExceptionD1 : public ExceptionB { };
class ExceptionD2 : public ExceptionB { };
void f() {
    try {  /* code might throw Exception B, D1, D2, and many others  */  }
    catch(ExceptionD1) { /* … */ }
    catch(ExceptionD2) { /* … */ }
    catch(ExceptionB) { /* … */ }
    catch(…) { /* … */ }
}  // only the order of D1 and D2 can be switched in this example
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Exception Specification (1/2)

- One can specify the set of exceptions that might be thrown as part of the function declaration

void f(int a) throw(X1, X2);

- Above specifies that f() may throw exceptions X1 and X2 ONLY (X1 and X2 are types)
- If unspecified exceptions are thrown
  - std::unexpected() is called
  - by default, std::unexpected calls std::terminate()
  - by default std::terminate calls std::abort() to end the program
- int f(); ➔ f() can throw any exceptions
- int g() throw(); ➔ g() cannot throw exceptions

Exception Specification (2/2)

```
void f() throw(X1, X2) {
    // do something
}
```

is equivalent to

```
void f()
try {
    // do something
}
catch(X1) { throw; }        // re-throw
catch(X2) { throw; }        // re-throw
catch(…) { std::unexpected(); } // unexpected won't return
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Exception Specification Checks (1/2)

- Exception specification must be consistent between function declaration and definition

```
int f() throw(X1);          // declaration part: f might throw X1 only
int f() {                   // definition part: error, inconsistent!
    // do something
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Exception Specification Checks (2/2)

- A virtual function can be overridden only by a function that has an exception-specification at least as restrictive as its own

```
class B {
public:
    virtual void f();                   // can throw any exceptions
    virtual void g() throw (X, Y);      // can throw X and Y (and derived types)
    virtual void h() throw(X);          // throw X only
};
class D : public B {
public:
    void f();                   // ok
    void g() throw(X);          // ok, only X is allowed, more restrictive
    void h() throw(X, Y);       // error, B::h can only throw X
}
```

Uncaught Exceptions

- Uncaught exception
  - an exception is thrown but not caught
- If there is an uncaught exception, std::terminate is invoked
  - as mentioned, std::terminate calls std::abort(), which ends the program, by default

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Exception Handling Is Error Handling

- Exception handling is nothing but another flow control mechanism
- One can use this mechanism for normal operations

```
void f(Queue<X>& q)
try {
    while(true) {
        X m = q.get();    // throw Empty if queue is empty
        // do thing in terms of m
    }
}
catch(Queue<X>::Empty) { return; }
```

- However, this kind of use is not encouraged
- Guideline: exception handling is error handling

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Dynamic Memory Allocation

- new operator throws std::bad_alloc exception if memory allocation fails

```
try {
        int *pointer = new int[100000000];
}
catch (bad_alloc)
{
        cout << "Ran out of memory!\n";
        // Can do other things here as well…
}
```

- In <new>, namespace std

Summary

- Understand exception handling in C++
  - separate normal code and exception handling code
  - separate exception detection and exception handling
- Try-throw-catch mechanism
- Multiple exceptions thrown in one try block
- Exception, class hierarchy, and polymorphism
- Re-throw, catching all exceptions, order of catch-blocks
- Unexpected and uncaught exceptions
- Exception handling is error handling
- Operator new throws std::bad_alloc if dynamic memory allocation fails

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw