

C++: Exceptions

Miro Jurišić
meeroh@meeroh.org

Introduction

- Exceptions signal exceptional conditions in C++ programs
- Exceptions are objects
- Exceptions can't be ignored

Throwing an exception

- Don't allocate exceptions with new

```
if (bad things happened) {  
    throw MyException (args);  
}
```

Catching an exception

- Always catch exceptions by reference to avoid slicing
- You must not put a base class before a derived class in a catch chain
- Be careful not to throw a new exception when you want to rethrow

```
try {
    DoSomething ();
} catch (MyException& e) {    // Handle the exception
    throw std::exception ("My Exception"); // covert to a new exception
} catch (std::bad_alloc& e) { // out of memory
    DoLowMemoryWarning ();
    throw; // Rethrow the same exception
} catch (std::exception& e) { // other standard errors
} catch (...) {             // all other errors
}
```

Between a throw and a catch

- Exception object is created (neither on the stack nor on the heap)
- Stack is unwound to the nearest matching catch statement
- All automatic objects are destroyed during stack unwinding

```
void f ()
{
    std::string f1;
    throw std::exception ("oops");
}
```

```
void g ()
{
    std::string g1;
    try {
        std::string g2;
        f ();
    } catch (std::bad_alloc& e) {
    }
}
```

```
void h ()
{
    std::string h1;
    try {
        std::string h2;
        g ();
    } catch (...) {
    }
}
```

Using stack-based objects for resource management

- Objects acquire resources in constructor and during their lifetime
- Objects release resources in destructor
- Resources become exception safe

```
void Ugly ()
{
    try {
        char* s = new char [50];
        DoSomething ();
        delete [] s;
    } catch (...) {
        delete [] s;
        throw;
    }
}
```

```
void NotUgly ()
{
    CharArray s (new char [50]);
    DoSomething;
}
```

```
CharArray::CharArray (
    char*    inBuffer):
    mBuffer (inBuffer)
{
}
```

```
CharArray::~~CharArray ()
{
    delete [] mBuffer;
}
```

```
void Nice ()
{
    std::string (50); // or boost::scoped_array, etc
    DoSomething;
}
```

Exceptions in constructors

- Only fully created parts (members and bases) are destroyed

```
class Base {  
    string    b;  
};
```

```
class Derived: public Base {  
    string    d;  
};
```

```
Derived::Derived ():  
    Base (),  
    d ("foo")  
{  
    throw std::exception ("oops");  
}
```

Exception-safe assignment operator

- Implement your assignment operator in terms of swap

```
class MyClass {
    string    x;
    string    y;
};

MyClass&
MyClass::operator= (
    const MyClass&    inOriginal)
{
    x = inOriginal.x;
    y = inOriginal.y; // What if this throws?

    return *this;
}
```

```
MyClass&
MyClass::operator= (
    const MyClass&    inOriginal)
{
    MyClass temp (inOriginal);
    swap (*this, temp);
    return *this;
}

namespace std {
    void
    swap (
        MyClass&    inLeft,
        MyClass&    inRight)
    {
        std::swap (inLeft.x, inRight.x);
        std::swap (inLeft.y, inRight.y);
    }
}
```

Use standard classes and their friends

- `std::auto_ptr`
- `std::string`, `std::vector`
- `boost::scoped_ptr`, `boost::scoped_array`
- `boost::shared_ptr`, `boost::shared_array`

Double acquisition

- Beware of expression evaluation order when acquiring multiple resources
- Doing them one at a time is the easiest
- There is no smart pointer magic dust

`f (g1 (h1 ()), g2 (h2 ()))`

- h1 before g1
- h2 before g2
- g1 and g2 before f
- h1 and h2 in any order
- g1 and g2 in any order
- h2 can go before g1
- h1 can go before g2
- one possible order: h1, h2, g2, g1, f

```
f (new Class1 (), new Class2 ());
```

- g1 = Class1::Class1
- h1 = operator new
- g2 = Class2::Class2
- h2 = operator new
- Class 1 operator new, Class 2 operator new, Class2::Class2, Class1::Class1, f
- What happens if Class2 operator new throws?
- What happens if Class2::Class2 throws?

```
class MyClass {
    CharArray  mFirst;
    CharArray  mSecond;
};

MyClass::MyClass (
    char*      inFirst,
    char*      inSecond):
    mFirst (inFirst),
    mSecond (inSecond)
{
}

// Not safe
MyClass x (new char [10], new char [20]);
```

```
class MyClass {
    CharArray  mFirst;
    CharArray  mSecond;
};

MyClass::MyClass (
    CharArray&    inFirst,
    CharArray&    inSecond):
    mFirst (inFirst),
    mSecond (inSecond)
{
}

// Not safe
MyClass x (CharArray (new char [10]), CharArray& (new char [20]));
```

```
class MyClass {
    CharArray  mFirst;
    CharArray  mSecond;
};

MyClass::MyClass (
    CharArray&    inFirst,
    CharArray&    inSecond):
    mFirst (inFirst),
    mSecond (inSecond)
{
}

// Safe
CharArray a1 (new char [10]);
CharArray a2 (new char [20]);
MyClass x (a1, a2);
```