Module03-12
C++ 面向对象编程：异常处理

面向对象编程 - 异常处理

- 异常处理 (Exception Handling)：
  - 关于异常处理
  - 异常结组
  - 捕捉异常
  - 资源管理
  - 不是错误的异常
  - 异常描述
  - 未捕捉的异常
  - 关于异常的效率
  - 标准异常

异常处理 - 关于异常处理

- 异常处理机制
  - 异常处理是错误处理的一种方式
  - 实现错误处理与常规代码分离

异常处理 - 关于异常处理

- 异常处理语句块
  - 语句块的格式

```
try {
    /* … * /
} catch (exception-type) {
    /*  处理捕获的异常 */
}
```

异常处理 - 语句块

- 简单异常处理示例:

```
int Array::get(const int& index) {
    if (_size <= index || index < 0)
        throw IndexOutOfRange();

    return arr[index];
}

int main() {
    Array a(8);
    // Do some things...
    try {
        int n = a.get(12);
    } catch (IndexOutOfRange& e) {
        // Do some things...
    }
}
```

异常处理 - 异常结组

- 通过类层次简化 catch 工作

```
class DE1 { };
class DE2 { };

void f() {
    try {
        // do some things
    } catch (DE1 e) {
        // do some things
    } catch (DE2 e) {
        // do some things
    }
}
```

```
class E1 { };
class DE1: public E1 { };
class DE2: public E1 { };

void f() {
    try {
        // do some things
    } catch (E1 e) {
        // do some things
    }
}
```

异常处理 - 异常结组

- catch 基类类型的指针或引用

```
class E1 { };
class DE1: public E1 { };
class DE2: public E1 { };

void f() {
    try {
        // do some things
    } catch (E1* e) {
        // do some things
    }
}
```

```
class E1 { };
class DE1: public E1 { };
class DE2: public E1 { };

void f() {
    try {
        // do some things
    } catch (E1& e) {
        // do some things
    }
}
```

异常处理 - 捕获异常

- 捕获异常
  - 左边的代码中，什么时候到达 catch：
    1. 如果 X 是 E 的相同类型
    2. 如果 E 是 X 的无二义性的 public 基类
    3. 如果 E 和 X 是指针类型、且 [1] 或 [2] 对它们的引用的类型成立
    4. 如果 E 是引用类型、且 [1] 或 [2] 对 E 所引用的类型成立

```
void f() {
    try {
        throw X();
    } catch (E e) {
        // 何时到达这里？
    }
}
```

  - 关于异常对象
    - C++ 允许抛出任意类型（包括基本类型）的对象
    - 异常对象在抛出时需被复制，如果一个类型的对象不能被复制（如复制构造为 private ），则该类型的对象不应该被抛出

异常处理 - 异常捕获

- 重新抛出
  - 如果捕获一个异常后，异常处理器无法处理它、或只能部分处理，则可以重新抛出之
  - 示例：

```
void f() {
    try {
        // do some things

    } catch (E1 e) {
        // do some things
        throw; // 重新抛出捕获的原异常对象
    }
}
```

异常处理 - 异常捕获

- 捕获所有异常
  - 有些时候，由于某些异常没被捕获，导致应用程序意外终止，C++ 中有一种方式可以捕获任意类型的异常对象

```
void f() {
    try {
        // do some things
    }
    // 注意 (...) 类似函数中未定个数的参数列表，这里表示任意类型的异常
    catch (...) {
        // do some things
    }
}
```

异常处理 - 捕获异常

- 捕获所有异常（续）
  - 异常处理器的次序

```
void f() {
    try {
        // do some things
    } catch (DE1& e) {
        // 处理DE1异常
    } catch (E1& e) {
        // 处理E1异常
    }catch (...) {
        // 处理其它任何异常
    }
}
```

```
void f() {
    try {
        // do some things
    } catch (...) {
        // 处理其它任何异常
    } catch (E1& e) {
        // 不会到这里
    } catch (DE1& e) {
        // 不会到这里
    }
}
```

```
class E1 { };
class DE1: public E1 { };
```

异常处理 - 资源管理

- 资源获取即初始化 (Resource Acquisition Is Initialization - RAII)

```
void f1(const char* fname) {
    FILE* fp = fopen(fname, "r");
    //使用文件指针fp
    fclose(fp);
}

// 一种解决方案
void f2(const char* fname) {
    FILE* fp = fopen(fname, "r");
    try {
        //使用文件指针fp
    } catch (...) {
        fclose(fp);
        throw ;
    }
    //使用文件指针fp
    fclose(fp);
}
```

```
// 更简单且安全的方案
class FilePtr {
    FILE* f;
public:
    FilePtr(const char* name, const
char* mode) {
        f = fopen(name, mode);
    }
    // 赋值操作符、复制构造等
    ~FilePtr() { if (f)
fclose(f); }
    operator FILE*() { return f; }
};

void f3(const char* fname) {
    FilePtr f(fname, "r");
    // 使用文件指针f
    // 不管是正常退出、异常抛出，f的FILE*
资源正确销毁
}
```

异常处理 - 资源管理

- 使用构造函数和析构函数
  - 局部非静态对象（自动对象），在程序进入其声明处初始化（构造），出了该对象所在的作用域自动销毁（析构），这就是 RAII 的基础支撑
  - 通过正确构造，一定能正确销毁（销毁工作一般由编译器隐式完成），成功构造后，即使随后发生异常，在 stack unwinding 过程中也会正确调用析构函数
  - 如果一个对象没能正确构造，那么其析构函数不会调用

资源管理 - auto_ptr

- auto_ptr
  - 自动指针， C++ 标准库的一个类模板
  - 支持 RAII 技术，在指针出了其作用域后指针销毁，指针指向的对象也被销毁，不需显式 delete（这与普通指针的性质不太一样）
  - 使用方式同普通指针
  - 具有破坏性复制语义：不可能同时有 2 个或更多指针指向同一个对象，由于该特性， auto_ptr 不能用于需要复制语义的场合，比如 STL 容器、函数按值传参、函数返回等
  - (DEMO)

异常处理 - 资源管理

- 构造函数中的异常
  - 由于构造函数无返回值，所以传统的编程方式无法方便传出错误信息
  - 异常可以解决上述问题，如果构造函数中出错，我们也可以抛出异常

异常处理 - 资源管理

- 析构函数中的异常
  - 析构函数被调用的 2 种情况：
    - 正常调用：自动对象出了其作用域、或动态对象指针被 delete
    - 在异常处理中被调用：在 stack unwinding 过程中，异常处理机制退出一个作用域，其中包含有析构函数的对象
  - 注意：对于上述第 2 种情况，绝不要在析构函数里抛出异常

异常处理 - 不是错误的异常

- throw 可以抛出任何类型的对象
  - throw 可以抛出任意类型的对象：自定义类型、基本类型的对象，而 catch 也可以捕获任意类型的对象
  - (DEMO)
  - 尽管如此，使用异常机制应该坚持“异常处理就是错误处理”的观点，将正常代码和错误处理代码分离

异常处理 - 异常的描述

- 函数的检查描述（异常清单）
  - 函数的异常清单

    `void f() throw (E1, E2) //` 表明该函数只可能抛出 `E1`、`E2` 异常
  - 一个函数如果没有声明异常清单，表明该函数有可能抛出任何异常
  - 如果一个函数声明的异常清单为 throw()，则表示该函数不会抛出异常
  - 如果一个函数抛出了其声明的异常清单以外的异常，则 std::unexpected() 被调用，进而调用 std::terminate() 结束程序

异常处理 - 异常的描述

- 对异常描述的检查
  - 如果一个函数的声明中包含了异常描述，那么这个函数的所有声明（还有定义）都必须有一个包含完全一致的异常描述，也即异常描述是函数声明的一部分
  - 对于派生类中覆盖基类的虚函数：
    - 异常描述必须至少与基类异常描述一样受限（派生类可以声明抛出更少的异常）

```
struct A {
    virtual void f1(); // 可抛出任何异常
    virtual void f2() throw (X, Y);
    virtual void f3() throw (X);
};

struct D: A {
    virtual void f1() throw (X); // OK
    virtual void f2() throw (X); // OK，抛出更少
    virtual void f3() throw (X, Y); // Error
};
```

异常处理 - 异常的描述

- 对异常描述的检查（续）
  - 函数指针的赋值
    - 可以将一个指向具有更受限的异常描述函数指针，赋值给另一个指向不那么受限的异常描述的函数指针，但反过来不可

```
void f();
void f1() throw (X);
void (*p1) () throw (X, Y) = &f1; // OK
void (*p2) () throw () = &f1; // Error
void (*p3) () throw (X) = &f1; // Error
```

  - 虽然异常描述是函数声明的一部分，但不是函数类型的一部分，所以 typedef 不能带异常描述

```
void func() {}
void func() throw() {} // Error, redefined
typedef void (*fp) () throw(X); // Error
```

异常处理 - 未捕获的异常

- std::terminate()
  - 如果一个抛出的异常没有被捕获，则 std::terminate() 函数被调用，而 std::terminate() 继而调用 abort() 终止程序
  - 如果希望捕获所有异常，除在各处捕获、处理异常外，可以在 main() 函数中添加一个捕获所有异常的处理器 `catch (…) {}`

异常处理 - 效率考量

- 异常处理的效率
  - 在不抛出异常的情况下，拥有异常处理的实现在运行期没有任何额外的开销
  - 在抛出异常的情况下，有 stack unwinding 动作，造成一定的额外的开销

异常处理 - 标准异常

- 标准异常类的一览

名字	抛出	头文件
标准异常（由语言抛出）		
bad_alloc	new	<new>
bad_cast	dynamic_cast	<typeinfo>
bad_typeid	typeid	<typeinfo>
bad_exception	异常描述	<exception>
标准异常（由标准库抛出）		
out_of_range	at() bitset<>::operator[]()	<stdexcept> <stdexcept>
invalid_argument	按位设置构造函数	<stdexcept>
overflow_error	bitset<>::to_ulong()	<stdexcept>
ios_base::failure	ios_base::clear()	<ios>

异常处理 - 标准异常

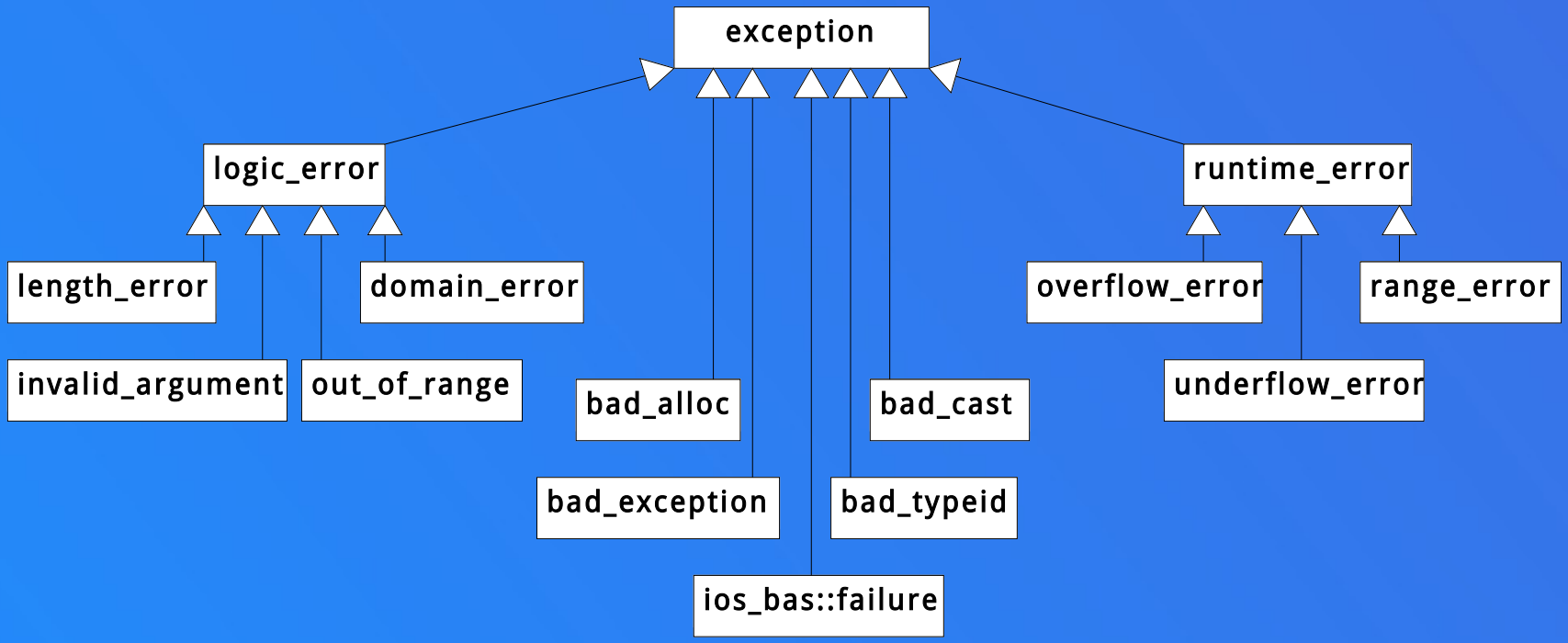
- 标准异常的基类 exception 类
  - class exception 在头文件 <exception> 里定义

```
class exception {
public:
    exception() throw () {
    }
    virtual ~exception() throw ();

    /** Returns a C-style character string describing the general
cause of the current error.  */
    virtual const char* what() const throw ();
};
```

异常处理 - 标准异常

- 标准异常的类层次结构



异常处理 - Bjarne's Advices

- Bjarne's Advices
  - 异常只用来做错误处理
  - 当更局部的控制机构能够应付时，不要使用异常
  - 采用“资源获取即初始化 (RAII)”技术来管理资源
  - 尽量少用 try 块，用“资源获取即初始化”技术，而不是显式使用异常处理器代码
  - 并不是每个函数都需要处理所有可能的错误
  - 避免从析构函数中抛出异常
  - 让 main() 捕获并报告所有的异常
  - 使正常处理代码和错误处理代码分离
  - 在构造函数中抛出异常前，先释放此构造函数中申请的所有资源

异常处理 - Bjarne's Advices

- Bjarne's Advices （续）
  - 注意通过 new 分配的内存在发生异常时没有被释放，而导致内存泄漏
  - 不要假定所有的异常是 std::exception 的子类
  - 库不应该单方面终止程序，应该抛出异常，由调用者去决定
  - 库不应该生成面向终端用户错误信息，应该抛出异常，由调用者去决定