EH & RTTI

异常处理(Exception Handling)

C++的 exception handling 有三个主要的子句组成：

- 一个 throw 子句。它在程序的某处丢出一个 exception，被丢出的 exception 可以是内建类 型，也可以是自定义类型。——抛出 exception 组件。
- 一个或多个 catch 子句。 每一个 catch 子句都是一个 exception handler。每个子句 可以处理一种类型(也包括其继承类)的 exception，在大括号中包含处理代码。——专治各 种不服组件。每一个 catch 子句都可以用来处理某种 exception。
- 一个 try 区段。用大括号包围一系列语句，这些语句有可能抛出 exception，从而引发 catch 子句的作用。——逮捕各种 exception 组件。

当一个 exception 被抛出后，控制权从函数调用中被释放，寻找一个吻合的 catch 子句，如 果各层调用都没有吻合的 catch 子句，`terminate()`将被调用。在控制权被放弃后，堆栈中 的每一个函数调用也被出栈，这个过程称为 unwinding the stack(关于 stack unwinding，可以参考《C++ Primer》第四版之 17.1.2 Stack Unwinding)，在每一个函数被出栈之前,其 局部变量会被摧毁。

异常抛出有可能带来一些问题，比方在一块内存的 lock 和 unlock 内存之间，或是在 new 和 delete 之间的代码抛出了异常，那么将导致本该进行的 unlock 或 delete 操作不能进行。解决 方法之一是：

```
void mumble( void *arena ) {
    Point *p;
    p = new Point;
    try {
        smLock( arena );
        // ...
    }
    catch (...) {
        smUnLock( arena );
        delete p;
        throw;
    }
    smUnLock( arena );
    delete p; }
```

在函数被出栈之前，先截住异常，在 unlock 和 delete 之后再将异常原样抛出。new expression 的调用不用包括在 try 块之内是因为，不论在 new operator 调用时还是构造函数调用时抛出异 常，都会在抛出异常之前释放已分配好的资源，所以不用再调用 delete 。

另一个办法是，将这些资源管理的问题，封装在一个类对象中，由析构函数释放资源，这样就 不需要对代码进行上面那样的处理——利用函数释放控制权之前会析构所有局部对象的原理。

在对单个对象构造过程中抛出异常，会只调用已经构造好的 base class object 或 member class object 的析构函数。同样的道理，适用于数组身上，如果在调用构造函数过程中抛出异常，那 么之前所有被构造好的元素的析构函数被调用，对于抛出异常的该元素，则遵循关于单个对象 构造的原则，然后释放已经分配好的内存。

只有在一个 catch 子句评估完毕并且知道它不会再抛出 exception 后，真正的 exception object 才会被释放。关于 catch 子句使用引用还是使用对象来捕获异常，省略。

执行期类型识别（Runtime Type Identification RTTI）

1. RTTI 只支持多态类，也就是说没有定义虚函数是的类是不能进行 RTTI 的。
2. 对指针进行 `dynamic_cast` 失败会返回 NULL ,而对引用的话，识别会抛出 `bad_cast exception`。
3. typeid 可以返回 `const type_info&`，用以获取类型信息。

关于 1 是因为 RTTI 的实现是通过 vptr 来获取存储在虚函数表中的 `type_info*` ，事实上为非多 态类提供 RTTI,也没有多大意义。 2 的原因在于指针可以被赋值为 0，以表示 no object，但是 引用不行。关于 3，虽然第一点指出 RTTI 只支持多态类，但 `typeid` 和 `type_info` 同样可用于 内建类型及所有非多态类。与多态类的差别在于，非多态类的 `type_info` 对象是静态取得(所 以不能叫“执行期类型识别”),而多态类的是在执行期获得。

参考：深度探索 C++对象模型