用于大型程序的工具

异常处理

1. 栈展开：若对抛出异常的函数的调用语句位于一个try语句块内，则检查与该try块关联的catch子句。若找到了匹配的catch，就用该catch处理异常。否则，如果该try语句嵌套在其他try块中，则继续检查与外层try匹配的catch子句。如果仍然没有找到匹配的catch，则退出当前这个主调函数，继续在调用了刚刚退出的这个函数的其他函数中寻找，以此类推。
2. 栈展开过程中沿着嵌套函数的调用链不断查找，直到找到了与异常匹配的catch子句为止；或者也可能一直没有找到匹配的catch，则退出主函数后查找过程终止。
3. 假设找到了一个匹配的catch子句，则程序进入该子句并执行其中代码。执行完后找到与try块关联的最后一个catch子句之后的点，并从这里继续执行。若无找到匹配的catch子句，程序将调用标准库函数**terminate**，终止程序的执行。
4. 栈展开的过程中局部对象会被自动销毁。
5. 在搜寻catch语句的过程中，最终找到的catch是第一个与异常匹配的catch语句，所以在具有继承关系的异常处理中，要对catch语句的顺序进行组织，使得派生类异常的处理代码出现在基类异常的处理代码之前。
6. catch语句的匹配规则：
   - 允许从非常量向常量的类型转换，即一条非常量对象的throw语句可以匹配一个接受常量引用的catch语句
   - 允许从派生类向基类的类型转换
   - 数组被转换成指向数组类型的指针，函数被转换成指向该函数类型的指针

   除此之外，包括标准算数类型转换和类类型转换在内，其他所有转换规则都不能在匹配catch的过程中使用。
7. 有时单独一个catch语句不能完整地处理某个异常。在执行了某些校正操作之后，当前的catch可能会决定由调用链更上一层的函数接着处理异常。一条catch语句通过**重新抛出**的操作**将异常传递给另外一个catch语句**。这里的重新抛出仍然是一条throw语句，但**不包含任何表达式**。
8. 捕获所有异常的处理代码：**catch(...)**
9. 构造函数在进入函数体之前先执行初始化列表，因此在初始化列表抛出异常时构造函数的try还未生效，所以构造函数体内的catch语句无法处理构造函数初始化列表抛出的异常。因此引入**函数try语句块**。函数try语句块使得一组catch语句既能处理构造函数体（或析构函数体），也能处理构造函数的初始化过程（或析构函数的析构过程）。

```
class test{
    test()try:初始化列表{
        //函数体
    } catch(){} //异常捕获
}
```

10. 约定函数不会抛出异常：

```
//两者等价
void func() noexcept;
void func() throw();
```

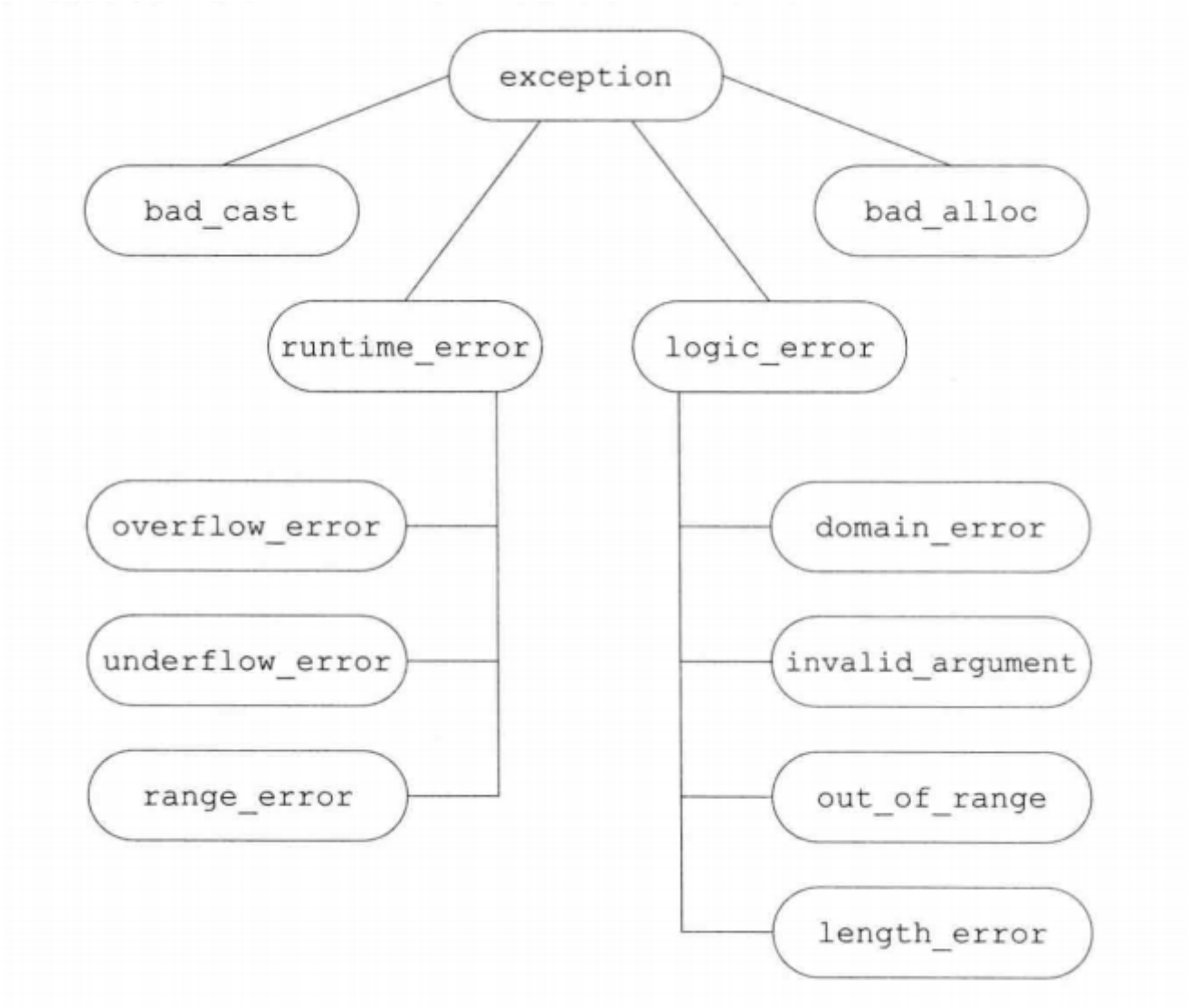
noexcept也可以作为运算符

```
void func() {}
void func1() {throw;}
void func2() noexcept { throw;}

int main(){
    cout<<noexcept(func())<<endl;
    cout<<noexcept(func1())<<endl;
    cout<<noexcept(func2())<<endl;
}

//输出为：0 0 1
```

11. 若一个虚函数承诺不会抛出异常，则后续派生出来的虚函数也必须做出相同的承诺；与之相反，如果基类的虚函数允许抛出异常，则派生类的对应函数既可以允许抛出异常也可以不允许。
12. 异常类的层次：



类型exception仅仅定义了拷贝构造函数、拷贝赋值运算符、虚析构函数和一个名为what的虚成员。**what函数返回一个const char*，指向一个字符数组且确保不会抛出任何异常**。

命名空间

1. 定义命名空间：

```
namespace 空间名{
    //各种定义的数据
}   //不需要分号
```

命名空间的定义可以是不连续的，即对于一个命名空间的定义，可能是为已经存在的命名空间添加新成员，也可能是创建一个新的命名空间。

2. 作用域运算符同样可以用于全局作用域的成员，因为全局作用域是隐式的，所以它并没有名字，使用方法如此：“ ::成员 ”

3. 嵌套的命名空间使用作用域运算符：外部::内部::成员。C++11引入了一种新的嵌套命名空间，称为内联命名空间。内联命名空间中的名字可以被外层命名空间直接使用，无须在内联命名空间的名字前添加表示该命名空间的前缀，通过外层命名空间的名字就可以直接访问它。
4. 未命名的命名空间：指的是namespace后紧跟花括号起来的一系列声明语句。**未命名的命名空间中定义的变量拥有静态生命周期**：它们在第一次使用前创建，并且知道程序结束才销毁。一个未命名的命名空间可以在某个文件内不连续，但**不能跨越多个文件**。
5. **未命名的命名空间仅在特定的文件内部有效，其作用范围不会横跨多个不同文件**。
6. 为命名空间定义别名：namespace 别名=原名
7. **using声明语句一次只引入命名空间的一个成员**，有效范围从using声明的地方开始，一直到using声明所在的作用域结束为止，在此过程中，外层作用域的同名实体将被隐藏。**using指示语句一次性将命名空间中所有成员引入**，可能引起变量的二义性。
8. 实参相关的查找：当我们给函数传递一个类类型的对象时，**除了在常规的作用域查找外还会查找实参类所属的命名空间**。
9. 当一个另外的未声明的类或函数如果第一次出现在友元声明中，我们认为它是最近的外层命名空间的成员。

```
namespace A{
    class C{
        //f1和f2都属于命名空间A
        friend void f1();
        friend void f2(const C&);
    }
}
int main(){
    A::C cobj;
    f1();       //错误 A::f1未声明
    f2(cobj);   //正确 通过实参相关查找规则
}
```

多重继承和虚继承

1. 派生类的构造函数初始值列表中基类的构造顺序与派生列表中基类的出现顺序保持一致，与初始值列表中基类出现的顺序无关
2. 有多个基类的情况下，也可以令某个可访问基类的指针或引用直接洁指向一个派生类对象，编译器不会在派生类向基类的几种转换中进行比较和选择，它将这几种转换视为等价。
3. 虚继承的目的是令某个类做出声明，承诺愿意共享它的基类。其中共享的基类子对象称为虚基类。在此机制下，无论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象。指定虚基类的方式是在派生列表中添加关键字virtual
4. 虚派生只影响 从指定了虚基类的派生类中进一步派生出的类，不会影响派生类本身。
5. 在虚派生中，虚基类是由最低层的派生类初始化的。只要我们能够创建虚基类的派生对象，该派生类的构造函数就必须初始化它的虚基类。

6. 最低层派生类构造函数的初始值列表中，首先应该初始化虚基类，将下来再初始化直接基类。**虚基类总是先于非虚基类构造，这与它们在继承体系或者的次序和位置无关**。