第18章 用于大型程序的工具

18.1 异常处理

1. **异常处理（exception handing）** 机制允许程序中独立开发的部分能够在运行时就出现的问题进行通信并做出相应的处理。

18.1.1 抛出异常

1. 通过**抛出（throwing）** 一条表达式来**引发（raised）**一个异常。
2. 被抛出表达式的类型以及当前的调用链共同决定了哪段**处理代码（handling）** 将被用于处理该异常。
3. 当执行一个 `throw` 时，跟在 `throw` 后面的语句将不再执行，然后进行**栈展开（stack unwinding）**。栈展开过程沿着嵌套函数的调用链不断查找，直到找到了异常匹配的 `catch` 子句为止；或者也可能一直没找到匹配的 `catch` ，程序将调用标准库函数 `terminate` 终止程序的执行
4. 类对象分配的资源将由类的析构函数负责释放。因此，如果我们使用类来控制资源的分配，就能确保无论函数正常结束还是遭遇异常，资源都能被正确地释放。
5. 出于栈展开可能使用析构函数的考虑，析构函数不应该抛出不能被它自身处理的异常。如果析构函数需要执行某个可能抛出异常的操作，则该操作应该被放置在一个 `try` 语句块当中，并且在析构函数内部得到处理。
6. **异常对象（exception object）**是一种特殊的对象，编译器使用异常抛出表达式来对异常对象进行拷贝初始化。因此， `throw` 语句中的表达式必须拥有完全类型。当我们抛出表达式时，该表达式的静态编译时类型决定了异常对象的类型。
7. 抛出指针要求在任何对应的处理代码存在的地方，指针所指的对象都必须存在。

18.1.2 捕获异常

1. **catch子句（catch clause）** 中的 **异常声明（exception declaration）** 看起来像是只包含一个形参的函数形参列表。像在形参列表中一样，如果 `catch` 无须访问抛出的表达式的话，则我们可以忽略捕获形参的名字。
2. 通常情况下，如果 `catch` 接受的异常与某个继承体系有关，则最好将该 `catch` 的参数定义成引用类型。
3. 如果在多个 `catch` 语句的类型之间存在着继承关系，则我们应该把继承链最底端的类（most derived type）放在前面，而将继承链最顶端的类（least derived type）放在后面。
4. 一条 `catch` 语句通过**重新抛出（rethrowing）**的操作将异常传递给另一个 `catch` 语句。这里的重新抛出仍然是一条 `throw` 语句，只不过不包含任何表达式。

5. 为了一次性捕获所有异常，我们使用省略号作为异常声明，这样的处理代码称为**捕获所有异常（catch-all）** 的处理代码，形如 `catch(...)` 。
6. 如果 `catch(...)` 与其他几个 `catch` 语句一起出现，则 `catch(...)` 必须在最后的位置。出现在捕获所有异常语句后面的 `catch` 语句将永远不被匹配。

18.1.3 函数 `try` 语句块与构造函数

1. 要想处理**构造函数初始值**抛出的异常，我们必须将构造函数写成**函数try语句块**（也称为函数测试块，function try block）的形式。
2. 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），也能处理构造函数的初始化过程（或析构函数的析构过程）。

```
tempalte <typename T>
Blob<T>::Blob(std::initializer_list<T> il) try :
    data(std::make_shared<std::vector<T>>(il)) {
        // 函数载体
} catch(const std::bad_alloc& e) { handle_out_of_memory(e); }
```

3. 和其他函数调用一样，如果在参数初始化的过程中发生了异常，则该异常属于调用表达式的一部分，并将在调用者所在的上下文中处理。
4. 处理构造函数初始值异常的唯一方法是将构造函数写成函数 `try` 语句块。

18.1.4 `noexcept` 异常说明

1. c++11新标准中，可以通过提供**noexcept说明（noexcept specification）** 指定某个函数不会抛出异常。

```
void recoup(int) noexcept;  // 不会抛出异常
void alloc(int);            // 可能会抛出异常

void recoup(int) noexcept(true);    // 不会抛出异常
void alloc(int) noexcept(false);    // 可能抛出异常
```

2. `noexcept` 说明要么出现在该函数的所有声明语句和定义语句中，要么一次也不出现。
3. 编译器不会在编译时检查 `noexcept` 说明。如果一个函数在说明了 `noexcept` 的同时又含有 `throw` 语句或者调用了可能抛出异常的其他函数，编译器将顺利编译通过，并不会因为这种违反异常说明的情况而报错。

```
void f() noexcept(noexcept(g()));   // f和g的异常说明一致

// noexcept有两层含义：
// 当跟在函数参数列表后面时它是异常说明符；
// 而当作为noexcept异常说明的bool实参出现时，它是一个运算符。
```

4. 函数指针及该指针所指的函数必须具有一致的异常说明。
5. 如果一个虚函数承诺了它不会抛出异常，则后续派生出来的虚函数也必须做出同样的承诺；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，也可以不允许抛出异常。

18.2 命名空间

1. 多个库将名字放置在全局命名空间中将引发**命名空间污染（namespace pollution）**。
2. 命名空间（namespace）为防止名字冲突提供了更加可控的机制。

18.2.1 命名空间定义

1. 和其他名字一样，命名空间的名字也必须在定义它的作用域内保持唯一。命名空间既可以定义在全局作用域内，也可以定义在其他命名空间中，但是不能定义在函数或类的内部。
2. 命名空间作用域后无须分号。
3. 每个命名空间都是一个作用域。
4. 定义多个类型不相关的命名空间应该使用单独的文件分别表示每个类型（或关联类型构成的集合）。
5. 我们不把 `#include` 放在命名空间内部。如果我们这么做了，隐含的意思是把头文件中所有的名字定义成该命名空间的成员。
6. 模板特例化必须定义在原始模板所属的命名空间中。和其他命名空间名字类似，只要我们在命名空间中声明了特例化，就能在命名空间外部定义它了。
7. 全局作用域中定义的名字（既在所有类，函数及命名空间之外定义的名字）也就是定义在**全局命名空间（global namespace）** 中。全局作用域中定义的名字被隐式地添加到全局命名空间中。
8. 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，所以它没有名字：`::member_name` 表示全局命名空间中的一个成员。
9. **嵌套的命名空间**是指定义在其他命名空间中的命名空间。
10. c++11新标准引入了一种新的嵌套命名空间，称为**内联命名空间（inline namespace）**。
11. 内联命名空间中的名字可以被外层命名空间直接使用。也就是说，我们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可以直接访问它。

12. **未命名的命名空间（unnamed namespace）** 是指关键字namespace后紧跟花括号括起来的一些列声明语句。未命名的命名空间中定义的变量拥有静态的生命周期：它们在第一次使用前创建，并且直到程序结束才销毁。
13. 和其他命名空间不同，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的文件。
14. 定义在未命名的命名空间中的名字可以直接使用，毕竟我们找不到什么命名空间的名字来限定它们；同样的，我们也不能对未命名的命名空间的成员使用作用域运算符。
15. 一个未命名的命名空间也能嵌套在其他命名空间当中。此时，未命名的命名空间中的成员可以通过外层命名空间的名字来访问。
16. 在文件中进行静态声明的做法已经被c++标准取消了，现在的做法是使用未命名的命名空间。

18.2.2 使用命名空间成员

1. **命名空间的别名（namespace alias）** 使得我们可以为命名空间的名字设定一个短得多的同义词。一个命名空间可以有好几个同义词或别名，所有别名都与命名空间原来的名字等价。
2. 一条 **using声明（using-declaration）** 语句一次只引入命名空间的一个成员： `using std::cout`
3. 一条 `using` 声明语句可以出现在全局作用域、局部作用域、命名空间作用域以及类的作用域中。
4. **using指示（using directive）** 和 `using` 声明类似的地方是，我们可以使用命名空间名字的简写形式： `using std` ； 和 `using` 声明不同的地方是，我们无法控制哪些名字是可见的，因为所有名字都是可见的。
5. `using` 指示使得某个特定的命名空间中所有的名字都可见，这样我们就无须再为它们添加任何前缀限定符了。
6. 如果我们提供了一个对 `std` 等命名空间的 `using` 指示而未做任何特殊控制的话，将重新引入由于使用了多个库而造成的名字冲突问题。
7. `using` 指示一般被看作是出现再最近的外层作用域中。
8. 头文件如果再其顶层作用域中含有 `using` 指示或 `using` 声明，则会将名字注入到所有包含了该头文件的文件中。头文件最多只能在它的函数或命名空间内使用 `using` 指示或 `using` 声明。
9. 在命名空间本身的实现文件中就可以使用 `using` 指示。

18.2.3 类、命名空间与作用域

1. 可以从函数的限定名推断出查找名字时检查作用域的次序，限定名以相反次序指出被查找的作用域。
2. 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，还会查找实参所属的命名空间。

18.2.4 重载与命名空间

1. `using` 声明语句声明的是一个名字，而非一个特定的函数。

2. 当我们为函数书写 `using` 声明时，该函数的所有版本都被引入到当前作用域中。
3. `using` 指示将命名空间的成员提升到外层作用域中，如果命名空间的某个函数与该命名空间所属作用域的函数同名，则命名空间的函数将被添加到重载集合中。
4. 如果存在多个 `using` 指示，则来自每个命名空间的名字都会成为候选函数集的一部分。

18.3 多重继承与虚继承

1. **多重继承（multiple inheritance）** 是指从多个直接基类中产生派生类的能力。多重继承的派生类继承了所有父类的属性。

18.3.1 多重继承

1. 基类的构造顺序与派生类列表中基类的出现顺序保持一致，而与派生类构造函数初始值列表中基类的顺序无关。
2. 在c++11新标准中，允许派生类从它的一个或几个基类中继承构造函数。但是如果从多个基类中继承了相同的构造函数（既形参列表完全相同）则程序将产生错误。
3. 如果一个类从它的多个基类中继承了相同的构造函数，则这个类必须为该构造函数定义它自己的版本。
4. 析构函数的调用顺序正好与构造函数相反。

18.3.2 类型转换与多个基类

1. 与只有一个基类的继承一样，对象、指针和引用的静态类型决定了我们能够使用哪些成员。

18.3.3 多重继承下的类作用域

1. 在多重继承的情况下，相同的查找过程在所有直接基类中同时进行。如果名字在多个基类中都被找到，则对该名字的使用将具有二义性。
2. 当一个类拥有多个基类时，有可能出现派生类从两个或更多基类中继承了同名成员的情况。此时，不加前缀限定符直接使用该名字将发生二义性。

18.3.4 虚继承

1. **虚继承（virtual inheritance）** 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（virtual base class）** 。在这种机制下，不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。

2. 使用虚继承的类层次是由一个人或一个项目组一次性设计完成的。对于一个独立开发的类来说，很少需要基类中的某一个是虚基类，况且新基类的开发者也无法改变已存在的类体系。
3. 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，它不会影响派生类本身。
4. 不论基类是不是虚基类，派生类对象都能被可访问基类的指针或引用操作。

```
// 关键字public 和 virtual 的顺序随意
class Raccoon : public virtual ZooAnimal { ... }
class Bear : virtual public ZooAnimal { ... }
```

18.3.5 构造函数与虚继承

1. 在虚派生中，虚基类是由最低层的派生类初始化的。
2. 含有虚基类的对象的构造顺序与一般的顺序稍有区别：首先使用提供给最低层派生类构造函数的初始值初始化该对象的虚基类子部分，接下来按照直接基类在派生类列表中出现的次序依次对其进行初始化。
3. 虚基类总是先于非虚基类构造，与它们在继承体系中的次序和位置无关。
4. 一个类可以有多个虚基类。此时，这些虚的子对象按照它们在派生列表中出现的顺序从左向右依次构造。