# C++编程(6)

Tang Xiaosheng
北京邮电大学电信工程学院

# 第八章 名字空间和异常

- 模块化和界面
- 名字空间
- 异常
- 忠告

# 8.1 模块化和异常

- 一个大型系统是由多个模块组成的，编程中，模块可以被理解为是完成某项功能的一段代码
- 一个模块使用另一个模块时，并不需要知道被调用模块的所有细节，因此，需要将一个模块和它的界面区分开来(形式上的)，这是开发大型程序的有用技术
- 划分模块并不难，难的是提供跨过模块边界的安全、方便而有效的通信

- 程序的整个结构中分布着错误处理，编写时，特别要注意将由错误处理造成的模块之间的相互依赖减到最小
- C++提供的异常机制，可以用于降低检查、报告错误和处理错误之间的联系程度

# 8.2 名字空间

- 名字空间是一种描述逻辑分组的机制，如果有一些声明按照某种准则在逻辑上属于同一个集团，就可以将它们放入同一个名字空间
- 名字空间描述了程序的界面，一般在实现具体细节前进行，所以一般和函数的具体实现是分开的
- 成员可以在名字空间的定义里声明，而后再采用namespace-name::member-name形式去定义
- 不能在名字空间定义之外用加限定的语法形式为名字空间引进新成员
- 一般来说，程序中的每个声明都必须位于某个名字空间内，但是main()函数例外

# 理想情况下，一个名字空间应该

- 描述一个具有逻辑统一性的特征集合
- 不为用户提供对无关特征的访问
- 不给用户强加任何明显的记述负担

# 示例

```
namespace Parser{
    double expr(bool);
    double prim(bool get){/* … */}
    double term(bool get){/* … */}
    double expr(bool get){/* … */}
} //声明和实现同时进行

namespace Parser{
    double expr(bool);
    double prim(bool);
    double term(bool);
}//实现与界面分开
double Parser::expr(bool get){/* …*/}
double Parser::prim(bool get){/* …*/}
double Parser::term(bool get){/* …*/}
```

名字空间的成员必须采用如下记法形式引入：

```
namespace name {
    // 声明和定义
}

void Parser::logical(bool);
// Error, Parser中无logical，
// 也无法引入新名字logical
void Parser::prim(int);
// Error, prim要求bool参数
```

# 8.2.1 带限定词的名字

- 名字空间是作用域，普通的作用域规则也对名字空间成立
- 如果一个名字先前已经在本名字空间里或者其外围作用域里声明过，就可以直接使用。
- 也可以使用来自另外一个名字空间的名字，但需要该名字所属的名字空间作为限定词

# 带限定词的名字示例

```
namespace Lexer{  //定义一个新的namespace
    enum Token_value { NAME, NUMBER, END, MUL='*'};
    Token_value curr_tok;
    double number_value;
    string string_value;
    Token_value get_token() { /* … */}
}
            double Parser::term(bool get) // Parser作用域
            {
                double left=prim(get);  // 不需要限定词
                for(;;) switch(Lexer::curr_tok) { // Lexer
                    case Lexer::NAME:                 // Lexer
                        left *= prim(true);  // 不需要限定词
                    // …
                 } /* … */ }
```

# 8.2.2 使用声明

- 当编程中频繁使用另外一个名字空间中的变量时，可以通过使用声明语句来避免反复书写名字空间限定词

```
double Parser::prim(bool get)
{
    if(get) Lexer::get_token();
    switch(Lexer::curr_tok){
        case Lexer::NUMBER:
            Lexer::get_token();
        // …
    }
}
```

```
double Parser::prim(bool get)
{
    using Lexer::get_token;
    using Lexer::curr_tok;

    if(get) get_token();
    switch(curr_tok){
        case Lexer::NUMBER:
            get_token(); // …
            }}
```

- 也可以把有关的使用说明放在Parser名字空间的定义里

```
namespace Parser{
    double prim(bool);
    double term(bool);
    double expr(bool);

    using Lexer::get_token;
    using Lexer::curr_tok;
    using Error::error;
}
```

```
double Parser::term(bool get)
{
    double left = prim(get);
    for(;;)
    switch(curr_tok) {
        case Lexer::MUL:
            left *= prim(true);
            break;
        case Lexer::DIV:
            if(double d = prim(true))
            {left /= prim(true); break;}
            return error("divided by 0");
    /* … …  */ }}
```

# 8.2.3 使用指令

- 一个使用指令能把来自一个名字空间的所有名字都变成可用的

```
namespace Parser{
    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer;
    using namespace Error;
}
```

# 8.2.4 多重界面

- 对于名字空间，编程者看到和用户看到的界面往往是没有必要相同的，用户看到的界面通常要比编程者所看到的简单的多
- 一般情况下，编程界面变化比较频繁
- 设计界面时要考虑两种界面，同时注意尽量避免编程界面的变化影响到用户界面

```
//给实现的界面
namespace Parser{
    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer;
    using namespace Error;
}

//给使用者的界面
namespace Parser{
    double expr(bool);
}
```

# 8.2.4.1 界面设计的各种选择

- 界面的作用就是尽可能减少程序不同部分之间的相互依赖，最小的界面将会使程序易于理解，有很好的数据隐蔽性质，容易修改，也编译的更快
- 对于上面描述的两种界面(用户和编程)，我们希望能做到：呈现一个易于理解的用户界面，该界面尽量少受编程界面的影响(至少表面看上去是这样)

# 界面设计的两种选择

```
//编程界面
namespace Parser{
    // …
    double expr(bool);
    // …
}
//用户界面
namespace Parser_interface{
    using Parser::expr;
}
```

看上去，用户界面还是很容易受到编程界面的伤害，原本希望能将其相互隔离

```
//编程界面不变
//用户界面
namespace Parser_interface{
    double expr(bool);
}

double Parser_interface::
    expr(bool get)
{
    return Parser::expr(get);
}
```

现在所有的依赖性都已最小化
当然，对于我们所面对的大部分问题而言，这种解决方案太过分了

# 8.2.5 避免名字冲突

- 名字空间就是为了表示逻辑结构
- 最简单的这类结构的应用就是为了分清楚一个人写的代码和另一个人写的代码
- 一般情况下，若只使用一个单独的名字空间，当从一些相互独立的部分组合起一个程序时，就可能遇到一些不必要的困难，比方说，同名函数或者变量的冲突问题

# 避免名字冲突示例

```
//my.h
char f(char);
int f(int);
class String{
 /* … */ };

//you.h
char f(char);
double f(double);
class String{
  /* … */ };
```

一般情况下，第三方很难同时使用my.h和you.h

解决方法：

```
namespace My{
    char f(char); int f(int);
    class String{/*…*/};
}
namespace You{
    char f(char); double f(double);
    class String{/*…*/};
}
```

使用时：

```
1 My::f(char), You::f(double)
2 using My::f; using You::String;
3 using namespace My; 或者
   using namespace You;
```

# 8.2.5.1 无名名字空间

- 有时，将一组声明包在一个名字空间内部就是为了避免可能的名字冲突
- 这样做的目的只是为了保持代码的局部性，而不是为了给用户提供界面
- 此时，我们可以使用无名名字空间

# 无名名字空间示例

```
#include “header.h”
namespace Mine{
    int a;
    voif f(){/*…*/}
    int g(){/*…*/}
}

//使用无名名字空间
#include “header.h”
namespace{
    int a;
    voif f(){/*…*/}
    int g(){/*…*/}
}
```

相当于：

```
#include “header.h”
namespace $$${
    int a;
    voif f(){/*…*/}
    int g(){/*…*/}
}
using namespace $$$;
//其中，$$$是在这个名字空间
//定义所在的作用域里具有唯一
//性的名字，在不同的编译单位
//里的无名名字空间也互不相同

//保证名称的私有性，避免冲突
```

# 8.2.6 名字查找

- 一个取T类型参数的函数常常与T类型本身定义在同一个名字空间里，因此，如果在使用一个函数的环境中无法找到它，我们就去查看它的参数所在的名字空间

```
namespace Chrono{
    class Date{/*…*/};
    bool operator==(const Date&,
        const std::string&);
    std::string
        format(const Date&);
}
```

```
void f(Chrono::Date d, int i){
    std::string s = format(d);
    //Chrono::format()
    std::string t = format(i);
    //Error:找不到使用int输入
    //参数的format()版本
}
```

# 名字空间示例

- 名字查找规则能够使程序员节省许多输入，同时也不必使用using指令污染名字空间，这个规则对于运算符的运算对象和模板参数特别有用
- 需要注意的是，名字空间本身必须在作用域里，函数也必须在它被寻找和使用之前声明

```
void f(Chrono::Date d, std::string s)
{
    if(d==s) { // …
    }
    else if(d == “August 4, 1914”) {  // …
    }
} //此函数最终会调用Chrono名字空间里的operator==
```

# 8.2.7 名字空间别名

- 用户给名字空间取很短的名字(例如namespace A)可能会出现冲突，但是如果给名字空间取很长的名字(例如namespace American_Telephone_and_Telegraph)又很不实用
- 可以通过为长名字提供较短的别名的方式解决此类问题

# 名字空间别名

```
namespace ATT = American_Telephone_and_Telegraph;
ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";

namespace Lib = Foundation_library_v2r11;
//…

Lib::set s;
Lib::String s5 = "Sibelius";
//将来Foundation库的版本更新了，只需要修改别名
//Lib的初始化语句并重新编译即可，极大的简化了升
//级工作
//注意：程序中过多的使用别名也会引起混乱
```

# 8.2.8 名字空间组合

- 有时候我们需要从现存的界面出发组合出新的界面

```
namespace His_string{
    class String{/*…*/};
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
    void fill(char);
}
namespace Her_vector{
    template<class T> class Vector{/*…*/};
}
namespace My_lib{
    using namespace His_string;
    using namespace Her_vector;
    void myfct(String&);}
```

# 名字空间组合示例

```
void f() {
    My_lib::String s = “Byron”;
  //系统能够查到是
  // My_lib::His_string::String
}

using namespace My_lib;
void g(Vector<String>& vs)
{
    //…
    myfct(vs[5]);
    //…
}//Vector以及String都是My_lib
  //中可以查到的
```

仅当我们需要去定义什么东西时，才需要知道一个实体真正所在的名字空间

```
void My_lib::fill(char c){
    //…
} // Error
void His_string::fill(char c){
    //…
}// ok
void My_lib::myfct(String& v)
{
    // …
}// ok String是My_lib::String
//即 His_string::String
```

# 8.2.8.1 选择

- 有时我们只是想从一个名字空间里选用几个名字，可以通过写一个仅仅包含我们想要的声明的名字空间来达到这个目的

```
namespace His_string{ // 只有His_string的一部分
    class String{/*…*/};
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
} // 注意：此处的His_string是一个声明
```

- 但是，这种做法会引起混乱，任何对原来His_string的实现部分做的修改在这里无法体现出来(比方说：添加一个+的重载函数)

# 改进方案

- 通过using语句，使得从名字空间里选择一些特征的事变得更加明确

```
namespace My_string{
    using His_string::String;
    //如果String被改动，这里自然会有所变化
    using His_string::operator+;
    //一条using语句就可以将operator+的所有重载都包含进来
}
```

# 8.2.8.2 组合和选择

- 将组合(通过using指令)和选择(通过using声明)结合起来能产生更多的灵活性，这些都是真实世界的例子所需要的

```
namespace His_lib{
    class String{/*…*/};
    template<class T> class
        Vector{/*…*/};
}
namespace Her_lib{
    template<class T> class
        Vector{/*…*/};
    class String{/*…*/};
}
```

```
namespace My_lib{
    using namespace His_lib;
    using namespace Her_lib;
    using His_lib::String;
    //以偏向His_lib的方式解析
    //潜在的冲突
    using Her_lib::Vector;
    //同上
    template<class T> class
        list{/*…*/};
}
```

# 组合与选择

- 查看一个名字空间时，其中显式声明的名字(包括通过using指令声明的名字)优先于在其他作用域里的那些通过using指令才能访问的名字
- 例如：对于String和Vector名字冲突的解析将分别偏向于His_lib::String和Her_lib::Vector，此外，My_lib::List总会被使用，和Her_lib或者His_lib中是否提供了List没有任何关系

# 8.2.9 名字空间和老代码

- 现存有数以百万行计的C和C++代码，我们若想使用这些代码，必须通过名字空间缓和这种代码之间的名字冲突问题
- 幸运的是，对于C代码，我们可以继续使用它们（就像它们是在名字空间里定义的一样），但是，对于C++代码则无法做到这一点，只能尽可能减少由于设计时名字空间的引入给已有的C++程序带来的破坏

# 8.2.9.1 名字空间和C

```
#include <stdio.h>
int main()
{
    printf("Hello, world!\n");
}
```

```
//标准C中stdio.h可以如下定义
namespace std{
    int printf(const char* ...);
    //...
}
using namespace std;
```

```
//cstdio，给那些不希望一大批
//名字都能隐式地随意使用的人
namespace std{
    int printf(const char* ...);
    //...
}
```

```
//真正C++标准库中的stdio.h
#include <cstdio>
using std::printf;
//...
```

# 8.2.9.2 名字空间和重载

- 重载可以跨名字空间工作

```
//老的A.h
void f(int);

//老的B.h
void f(char);

//老的user.cpp
#include “A.h”
#include “B.h”
void g()
{
    f(‘a’); //调用B.h中的f()
}
```

```
//新的A.h
namespace A{
    void f(int);
}
```

```
//新的B.h
namespace B{
    void f(char);
}
```

```
//新的user.cpp
#include “A.h”
#include “B.h”
using namespace A;
using namespace B;
void g() {
    f(‘a’); //调用B.h中的f()
}
```

# 8.2.9.3 名字空间是开放的

- 可以通过多个名字空间声明给它加入名字
- 注意，当定义一个名字空间中已经声明过的成员时，安全的方法是采用A::语法格式

```
namespace A{
    int f();
    //A中现有f()
}

namespace A{
    int g();
    //A中现有f()和g()
}

void A::ff()
{   // ...
}
//Error:A中没有ff()

//不要使用如下方式
namespace A{
    void ff() {/*...*/};
} // 这样会引入一个新的函数ff()，而不是
// 实现已经声明过的f()函数，编译器不报错
```

# 8.3 异常

- 当一个程序是由一些相互分离的模块组成时，特别是当这些模块来自某些独立开发的库时，错误处理的工作就需要分成两个相互独立的部分
  - 一方报告那些无法在局部解决的错误
  - 另一方处理那些在其他地方检查出来的错误
- 异常(exception)机制是C++中用于将错误报告与错误处理分开的手段

# 8.3.1 抛出和捕捉

- 异常概念就是为了帮助处理错误的报告
- 基本思想是：如果函数发现了一个自己无法处理的问题，它就抛出(throw)一个异常，希望它的(直接或间接)调用者能够处理这个问题，如果一个函数想处理某个问题，它就可以说明自己要捕捉(catch)用于报告该种问题的异常

# 异常处理示例

```
struct Range_error{
    int i;
    Range_error(int ii)
    {i=ii;}
};
char to_char(int i)
{
    if(i<numeric_limits<char>::min()
    || numeric_limits<char>::max()<i)
        throw Range_error(i);
    return i;
}
```

```
void g(int i)
{
    try {
        char c = to_char(i);
        //...
    }
    catch(Range_error x){
        cerr << “oops\n”;
        cerr << x.i << endl;
    }
}
```

其中catch(/*...*/){/*...*/}称为异常处理器，它只能紧接着由try关键字作为前缀的块之后，或者紧接在另一个异常处理器之后

# 异常的处理过程

- 如果在一个try块中的任何代码(或者由那里调用的东西)抛出了一个异常，代码马上会从产生异常的地方跳转到try后面的第一个异常处理器
- 如果所抛出的异常属于某个处理器所描述的类型，这个处理器就被执行
- 如果try未抛出异常，这些异常处理器都将被忽略
- 如果抛出了一个异常，而又没有catch块捕捉它，整个程序就将终止

# 8.3.2 异常的辨识

- 典型情况下，一个程序中可能存在多种不同的运行时错误，可以将这些错误映射到一些具有不同名字的异常，为每一种异常取一个系统中唯一的名字是比较好的解决方法
- 处理器中不需要break语句
- 一个函数不必捕捉所有可能的异常
- 异常处理器也可以嵌套

# 异常的辨识示例

```
struct Zero_divide{};
struct Syntax_error{
    const char* p;
    Syntax_error(const char* q)
    {   p = q; }
};

try {
    //...
    expr(false);
    //...
}
```

```
catch(Syntax_error se){
    // 处理语法错误
}
catch(Zero_divide zd){
    // 处理除以0错误
}
//若expr执行过程中没有异常
//或者出现异常但是被捕捉到
//程序代码将执行到这里
```

# 8.3.3 计算器中的异常

- 利用异常处理机制来完成以前计算器程序中的错误处理
- 这样做的好处是减少原来代码中程序各个部分的耦合程度，主要是其他部分和错误处理部分的耦合程度
- 原来的那种方式不能很好的用到由分别开发的库组合而成的程序中
- 例子见课本p170和p171

# 8.3.3.1 其他的错误处理策略

- 状态变量是混乱和错误的一个常见根源，特别是允许它们大量出现并影响程序中较大的片断时
- 一般的，使处理错误的代码与“正常”代码分离是一种很好的策略
- 在导致错误的代码的同一个抽象层次上处理错误是非常危险的(完成错误处理的代码有可能又产生了引起错误处理的那个错误)
- 修改“正常”的代码，加上错误处理代码，所需要的工作量比增加单独的错误处理例程更多

# 8.4 忠告

- [1] 用名字空间表示逻辑结构
- [2] 将每个非局部的名字放到某个名字空间里，除了main()之外
- [3] 名字空间的设计应该让你能很方便地使用它，而又不会意外地访问了其他的无关名字空间
- [4] 避免对名字空间使用很短的名字
- [5] 如果需要，通过名字空间别名去缓和长名字空间名的影响
- [6] 避免给名字空间的用户添加太大的记法负担

# 忠告

- [7] 在定义名字空间的成员时使用namespace::member的形式
- [8] 只有在转换时，或者在局部作用域里，才用using namespace
- [9] 利用异常去松弛“错误”处理代码和正常代码之间的联系
- [10] 采用用户定义类型作为异常，不要使用内部类型
- [11] 当局部控制结构足以应付问题时，不要使用异常